



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithmic-Hardware Co-Design for Efficient Embedded Intelligence

From Arithmetic Optimization to Neuro-Inspired Computing

Faculty of Information Engineering, Informatics, and Statistics
Department of Information Engineering, Electronics and Telecommunications
PhD in Information and Communications Technologies

Marco Angioli

ID number 1773057

Advisor

Prof. Mauro Olivieri

Academic Year 2025/2026

Thesis defended on June 4, 2026
in front of a Board of Examiners composed by:
Prof. Stefano Buzzi
Prof. Riccardo Berta
Prof. Fulvio Schettino

**Algorithmic-Hardware Co-Design for Efficient Embedded Intelligence From
Arithmetic Optimization to Neuro-Inspired Computing**

Ph.D. Thesis. Sapienza University of Rome

© 2026 Marco Angioli. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: marco.angioli@uniroma1.it

*To my family,
for their endless love and support*

Abstract

The rapid growth of Artificial Intelligence (AI) has enabled unprecedented capabilities, but it is also driving a computational and energy *efficiency crisis* that affects the entire compute stack. This crisis is most acute at the edge, where over half of all new AI workloads are expected to be deployed before 2030. Here, embedded systems must sense, decide, and react to their environment under extreme power, memory, and real-time constraints—requirements that are fundamentally incompatible with the escalating resource demands of modern AI models. This dissertation addresses this crisis in the context of online decision-making at the edge by adopting contextual bandit algorithms as the central abstraction and pursuing *algorithmic-hardware co-design* to enable efficient, adaptive intelligence on resource-constrained devices.

The thesis progresses through three parts: from arithmetic and algorithmic optimization of linear models, to a neuro-inspired paradigm shift via hyperdimensional computing, and finally, to the design of dedicated hardware accelerators.

Part I addresses the dominant bottlenecks in linear bandit algorithms for embedded processors. Starting from the observation that integer division alone can account for up to 70% of execution time on these platforms, we introduce a novel data-dependent hardware division unit that reduces average latency by up to $20.65\times$. By integrating this unit into a vector accelerator and reformulating learning updates to avoid explicit matrix inversion, we achieve up to a $58\times$ speedup and a $50\times$ energy reduction, while reducing the computational complexity from cubic to quadratic.

Yet, even with these optimizations, the inherent limitations of dense linear algebra remain. Recognizing this, **Part II** changes the nature of the computation itself by adopting hyperdimensional computing (HDC), a neuro-inspired paradigm based on high-dimensional distributed representations. We introduce HD-CB, the first framework that models and solves contextual bandit problems directly in high-dimensional space using simple, highly parallel element-wise operations, and progressively develop a family of six variants spanning increasingly advanced learning rules and compact representations. Extensive evaluation on synthetic datasets and real-world benchmarks demonstrates that HD-CB is a strong candidate for hardware acceleration and online decision-making at the edge: it achieves faster convergence, exhibits linear complexity with problem size, and outperforms the 32-bit floating-point linear baseline using just 4 bits per component.

Part III bridges algorithms and silicon by designing a family of hardware accelerators for HDC. We first introduce a configurable coprocessor unit for binary HDC, integrated into a RISC-V core; we then extend hardware support to modular composite representations, demonstrating that this model achieves a unique balance among information capacity, classification accuracy, and hardware efficiency for embedded systems. Finally, we develop the first hardware-friendly accelerator for the Fourier Holographic Reduced Representations model and introduce AeneasHDC, an open-source framework that automates accelerator design-space exploration and reduces deployment time from weeks to minutes.

Together, these contributions demonstrate that algorithmic-hardware co-design, grounded in neuro-inspired representations, can make autonomous online decision-making at the edge a practical reality.

Contents

Acronyms	xviii
1 Introduction	1
1.1 Objectives and Research Questions	2
1.2 Summary of Contributions	5
1.3 Outline	9
1.4 Publications	11
2 Background	14
2.1 Artificial Intelligence	15
2.1.1 From Symbolic to Connectionist AI	16
2.1.2 Learning Paradigms	19
2.1.3 Contextual Bandit Problems	21
2.2 Embedded AI Systems and Hardware Devices	26
2.2.1 The Role of FPGAs in Embedded AI	27
2.2.2 The RISC-V Instruction Set Architecture	28
2.2.3 Klessydra Core Family	30
2.3 Optimization Strategies	31
2.3.1 Arithmetic-level Optimizations	32
2.3.2 Algorithmic Optimizations	36
2.3.3 Hardware Acceleration	38
2.3.4 From Brain Principles to Neuro-Inspired Algorithms and Hyperdimensional Computing	41
Part 1: Arithmetic and Algorithmic Optimizations	59
3 Design, Implementation and Evaluation of a New Variable Latency Integer Division Scheme	63
3.1 Integer Division Algorithms	65
3.2 Proposed Divider	68
3.3 Hardware Implementation	70
3.3.1 Baseline Version (VLNPD-Std)	70
3.3.2 High-Frequency Version (VLNPD-HF)	73
3.3.3 High-Performance Version (VLNPD-HP)	74
3.3.4 Limited Area Version (VLNPD-LA)	75
3.4 Results	76

3.4.1	Latency behavior analysis	76
3.4.2	Monte-Carlo Simulation Analysis	78
3.5	Implementation on FPGA	80
3.6	Implementation on ASIC	82
3.7	Performance analysis on benchmark programs	82
3.7.1	Benchmark set	82
3.7.2	Results	83
3.8	Summary of Findings	85
4	Exploring Variable Latency Dividers in Vector Accelerators	87
4.1	Context Analysis	88
4.2	Integration Through Hardware Reuse	90
4.3	Results	91
4.3.1	Monte Carlo Simulation Analysis	91
4.3.2	Benchmark Analysis	92
4.4	Summary of Findings	93
5	Algorithmic and Hardware Optimization of LinUCB for Embedded Systems	94
5.1	Algorithmic Optimizations	96
5.1.1	Optimization of the Disjoint LinUCB through the Sherman-Morrison Formula	96
5.1.2	Optimization of the Hybrid LinUCB through the Woodbury's Formula	98
5.1.3	Stability and Round-off Errors	101
5.2	Vector Computing Acceleration	105
5.2.1	Reference Embedded Vector Computing Platform	106
5.2.2	Mapping LinUCB algorithms on the vector accelerator	106
5.3	Results on full algorithm execution	108
5.3.1	Methodology	108
5.3.2	Algorithmic optimization impact on execution time	109
5.3.3	Algorithmic optimization impact on energy consumption	110
5.3.4	Vector computing acceleration impact on execution time	111
5.3.5	Vector computing acceleration impact on energy consumption	112
5.3.6	Discussion	112
5.4	Summary of Findings	113
Part 2: From Linear Algebra to Neuro-inspired Computing		114
6	From Covariance Matrices to Associative Memories	118
6.1	The Role of Precision Matrix in Linear Bandits	119
6.1.1	Linear Bandits without Precision Matrix	119
6.1.2	Linear Random Projection	120
6.2	Variant with FlyHash Algorithm	121
6.2.1	Nonlinear Encoding via FlyHash Algorithm	121
6.2.2	The A Matrix as an Auto-associative Memory	122

6.2.3	Reward Centering	123
6.3	From Ablation to Hyperdimensional Computing	124
7	HD-CB: The First Exploration of Hyperdimensional Computing for Contextual Bandits Problems	126
7.1	Proposed Approach	127
7.1.1	HD-CB Basics	127
7.1.2	HD-CB _{EPS}	128
7.1.3	HD-CB _{UNC1} : Integrating Contextual Confidence	129
7.1.4	HD-CB _{UNC2} : Controlling Updates with Confidence	130
7.1.5	HD-CB _{UNC3} : Compact Representation via Permutation	131
7.2	Results	133
7.2.1	Online Learning Capabilities	133
7.2.2	Off-Policy Evaluation	135
7.2.3	Evaluation on the MovieLens-100k Dataset	136
7.2.4	Convergence Time Analysis	136
7.2.5	Computational Complexity Analysis	137
7.2.6	Scalability Analysis	138
7.2.7	Memory Footprint Analysis	139
7.2.8	Implicit Exploration Bias	140
7.3	Summary of Findings	141
8	HD-CB_{BIN}: A Binary Neuro-Inspired Framework for Embedded Contextual Bandits	143
8.1	Proposed Approach	144
8.2	Online Learning and Parameter Analysis	147
8.2.1	Learning Capabilities	147
8.2.2	Impact of the Q Parameter	147
8.2.3	Comparative Analysis	148
8.3	Memory Requirements Analysis	148
8.4	Execution Time Analysis	150
8.4.1	Software Execution	150
8.4.2	Hardware Acceleration	151
8.5	Summary of Findings	152
9	Hybrid HD-CB: A High-Dimensional Approach to Contextual Bandit Problems in Hybrid Settings	154
9.1	Proposed Approach	155
9.1.1	Intuition	156
9.1.2	Algorithm description	156
9.2	Online Learning Performance	157
9.3	Evaluation on the MovieLens-100k Dataset	160
9.4	Hybrid HD-CB on Embedded Systems	161
9.5	Hardware Acceleration	162
9.6	Memory Requirements	163
9.7	Summary of Findings	163

10 Efficient Implementations of HD-CB	165
10.1 Matrix-Vector Reformulation of HD-CB	165
10.1.1 General principle	166
10.1.2 Extension to uncertainty-aware and hybrid variants	167
10.1.3 Computational analysis and performance demonstration	167
10.2 Memory-Optimized HD-CB via Permutations	168
10.2.1 Compressing confidence into the estimate memory	169
10.2.2 From per-action memories to a single global memory	169
10.2.3 Accuracy–memory–throughput trade-offs	172
10.3 Summary of Findings	173
11 From Heuristic Updates to Optimization-Based Learning	176
11.1 Probabilistic HD-CB via Integer Self-Organizing Maps-Inspired Bounded Stochastic Updates	177
11.1.1 Learning low-precision integer representations	178
11.1.2 Probabilistic HD-CB	179
11.1.3 Algorithm overview	179
11.1.4 Probabilistic update rule	180
11.1.5 Experimental Evaluation	181
11.2 Loss-Based HD-CB: From Heuristic Updates to Explicit Optimization	183
11.2.1 Implicit loss of the baseline update rule	185
11.2.2 Per-round loss definition	190
11.2.3 Gradient of the loss with respect to the representative	191
11.2.4 Simplifying via Sigmoid and Regularization	192
11.2.5 Experimental Evaluation	194
11.2.6 Loss-Probabilistic HD-CB	197
11.2.7 Comparison: Performance versus Representation Precision	199
11.3 A Neural Network Interpretation of HD-CB	201
11.3.1 Background: RVFL networks and extreme learning machines	201
11.3.2 The HD-CB Encoding Layer as a Frozen Random Nonlinear Projection	202
11.3.3 The Action Prototypes as Trainable Output Weights	203
11.4 Summary of Findings	205
Part 3: Hardware Acceleration and Co-Design	206
12 Configurable Hardware Acceleration for Hyperdimensional Com- puting Extension on RISC-V	210
12.1 Related Works	211
12.2 Hyperdimensional Coprocessor Unit	213
12.2.1 Architecture overview	213
12.2.2 Functional Units	217
12.3 HDCU configuration and programming	220
12.3.1 Hardware configuration	220
12.3.2 Software programming	220
12.3.3 HDCU Software Library	221

12.4	Hardware implementation	222
12.5	Performance Results	224
12.5.1	Impact on the Basic HDC Arithmetic Operation	224
12.5.2	Performance on real datasets	225
12.6	Discussion and Comparison	227
12.7	Summary of Findings	228
13	Efficient Hyperdimensional Computing with Modular Composite Representations	230
13.1	Modular Composite Representation	231
13.2	Capacity of hypervectors	233
13.3	Classification with HVs	234
13.4	Results	236
13.4.1	Capacity analysis	236
13.4.2	Classification performance	237
13.4.3	Hardware design for MCR	238
13.4.4	Performance of MCR-HDCU	246
13.5	Summary of Findings	251
14	A Hardware-Friendly Discretized Approach to FHRR	252
14.1	Fourier Holographic Reduced Representations	253
14.2	Residue Hyperdimensional Computing	254
14.3	Toward Hardware-Friendly FHRR	257
14.3.1	The Cartesian Formulation	257
14.3.2	The Phase Formulation	257
14.4	Discretizing the Unitary Circle for Modulo-Free Arithmetic	258
14.5	Discretized FHRR Accelerator Architecture	258
14.5.1	Binding and Unbinding Units	262
14.5.2	Bundling Unit and CORDIC-Based Normalization	262
14.5.3	Permutation Unit	264
14.5.4	Resonator Network	264
14.5.5	Preliminary Synthesis Results	267
14.6	Summary of Findings	268
15	Solving 3-SAT Problems Using Resonator Networks and FHRR	269
15.1	Complexity: P, NP, and NP-Complete Problems	270
15.1.1	Complexity Classes	271
15.1.2	Reductions and NP-Completeness	271
15.1.3	Subset Sum and 3-SAT as Canonical NP-Complete Problems	272
15.2	Residue Hyperdimensional Computing and Subset Sum	273
15.3	Encoding 3-SAT as Subset Sum	274
15.3.1	Step-by-Step Example: 3 Variables and 3 Clauses	274
15.4	Baseline Formulation	276
15.5	Optimization 1: enforcing variable consistency via codebooks and removing leading digits	279
15.6	Optimization 2: grouping codebooks to reduce the number of factors	280
15.7	Optimization 3: changing the numeric base	280

15.8 Combined optimizations	282
15.9 Summary of Findings	284
16 AeneasHDC: An Automatic Framework for Deploying Hyperdimensional Computing Models on FPGAs	285
16.1 Related Works	287
16.2 AeneasHDC	287
16.2.1 Explorable Design Space	289
16.2.2 Hardware Architecture	290
16.2.3 Automated Design-Space Exploration	292
16.3 Experiments	294
16.3.1 Single-Configuration Validation	294
16.3.2 Design-Space Exploration Results	296
16.4 Summary of Findings	298
17 Conclusions and Future Perspectives	300
17.1 Summary of Contributions	301
17.1.1 Part I: Arithmetic and Algorithmic Optimization	301
17.1.2 Part II: A Paradigm Shift to Neuro-Inspired Contextual Bandits	301
17.1.3 Part III: Hardware Acceleration and Co-Design	303
17.2 Limitations	304
17.2.1 Part I: Arithmetic and Algorithmic Optimizations	304
17.2.2 Part II: Neuro-Inspired Contextual Bandits	304
17.2.3 Part III: Hardware Acceleration and Co-Design	306
17.3 Future Work	306
17.3.1 Algorithmic Extensions	306
17.3.2 Hardware Evolution	306
17.4 Closing Remarks	307
Bibliography	309

List of Figures

1.1	High-level dissertation roadmap	11
2.1	Schematic of contextual bandits problems	21
2.2	Representative application domains of contextual bandits	24
2.3	From cloud-centric to edge AI and embedded constraints	26
2.4	Klessydra-T03 and PULPino architectural schematics	30
2.5	Conceptual landscape of hardware platforms comparing CPUs, GPUs, and specialized accelerators.	38
2.6	Analogy between the human brain and heterogeneous computing systems.	40
2.7	Number of publications on HDC over the last 50 years, based on Scopus.	46
2.8	Concentration of measure phenomenon between bipolar random HVs. Plot replicated from [1].	47
2.9	Overview of hypervector encoding techniques employed in HD/VSA for univariate and structured data.	51
2.10	Compositional encoding of a binary tree where each node is defined by its position in the hierarchy.	56
2.11	Architecture of a three-factor Resonator Network designed to factorize composite hypervectors.	57
3.1	n-bit Restoring Division Algorithm	66
3.2	Hardware implementation of the algorithm presented in [2]	67
3.3	Statistical analysis of "shift-only" consecutive steps in 32-bit integer divisions.	68
3.4	Example of an 8-bit integer division with the NPD. The R register is just shifted to the left for six subsequent steps.	69
3.5	Example of an 8-bit integer division with the proposed VLNPD. This approach allows skipping six steps.	69
3.6	Algorithmic description of the presented VLNPD.	71
3.7	Hardware implementation for the baseline proposed variable latency divider (VLNPD-Std)	72
3.8	Schematic of the dynamic barrel-shifter implemented by a set of three multiplexers.	72
3.9	Hardware scheme for the iteration Count unit.	72
3.10	Hardware implementation for the High-Frequency variant (VLNPD-HF)	74

3.11	Hardware implementation for the High-Performance variant (VLNPD-HP)	75
3.12	Example of an 8-bit integer division in the VLNPD-HP. The HP version performs a total of 7 steps in one clock cycle.	75
3.13	Trend of the VLNPD's average clock cycle against the difference in magnitude (as powers of two) between the operands.	77
3.14	Average clock cycles under varying bit width of operands.	78
3.15	Statistical distribution of division latency in 32-bit integer divisions for VLNPD variants.	79
3.16	Average execution time per division required in six different benchmarks as a function of the operating frequency	84
4.1	Simplified schematic for the hardware implementation of the VLNPD-Std	89
4.2	Speedup obtained by the variable-latency divider under SIMD execution.	93
5.1	Computational complexity trends for the standard Disjoint Algorithm and the proposed optimized version.	96
5.2	Stack Memory Usage for the standard Disjoint Algorithm and the proposed optimized version.	97
5.3	Computational complexity trends for the standard Hybrid Algorithm and the proposed optimized version.	100
5.4	Stack Memory Usage for the standard Hybrid Algorithm and the proposed optimized version.	101
5.5	Error growth in the Disjoint Linear UCB algorithm, measured by the Frobenius norm of the difference between $\mathbf{A}_{a_t}^{-1}$ (computed incrementally) and the same matrix obtained via full inversion.	103
5.6	Error growth in the Hybrid Linear UCB algorithm, measured by the Frobenius norm of the difference between $\mathbf{A}_0^{-1}$ (computed incrementally) and the same matrix obtained via full inversion.	104
5.7	Error growth in the Hybrid Linear UCB algorithm with periodic correction of the inverse matrices (every 5000 iterations).	104
5.8	Speedup attained by the Klessydra-T13 vector computing acceleration on the fundamental matrix computation kernels used in LinUCB algorithms	107
5.9	Algorithmic optimization impact on execution time.	110
5.10	Algorithmic optimization impact on energy consumption.	110
5.11	Impact of vector computing acceleration on execution time.	111
5.12	Impact of vector computing acceleration on energy consumption.	112
7.1	Illustration of the working principle of HD-CB. The collection of encoded hypervectors $\mathcal{X}_{t,a}$ for all actions a at time step t is denoted as $\mathcal{X}_t$, while the set of estimated payoffs $e_{t,a}$ is represented as $\mathbf{e}_t$	128
7.2	Schematic of HD-CB _{UNC1} . Each action a is associated with a confidence vector $\mathcal{B}_a$. The vectors $\mathcal{X}_t$, $\mathbf{e}_t$, $\mathbf{u}_t$, and $\mathbf{p}_t$ denote the context, estimated payoffs, uncertainties, and potentials for all actions at time t	130

7.3	Average reward and cumulative regret trends for HD-CB models as a function of the exploration parameters	134
7.4	Results obtained on the MovieLens-100k dataset by LinUCB and HD-CB models, varying the number of movies.	137
7.5	Convergence time as a function of α , comparing the LinUCB algorithms and the HD-CB _{UNC1} model.	137
7.6	Impact of hypervector size on average reward and execution time for HD-CB models.	138
7.7	Memory requirements comparison for LinUCB and HD-CB, across varying context feature vector sizes (d) and two HV sizes (D).	139
8.1	Average reward for HD-CB _{BIN} as a function of the exploration parameters	146
8.2	Impact of the Q -parameter on the learning performance of HD-CB _{BIN}	148
8.3	Memory requirements comparison for LinUCB, HD-CB, and HD-CB _{BIN} across varying context feature vector sizes (d) and HV sizes (D).	149
8.4	Execution time comparison for LinUCB and HD-CB variants across CPU, Raspberry Pi, and RISC-V.	150
8.5	Execution time comparison for LinUCB (standard and SM variants) and HD-CB _{BIN} , with and without hardware acceleration.	152
9.1	Average reward for HD-CB-Hyb as a function of the exploration parameters α	157
9.2	Speedup achieved by Hybrid HD-CB compared to Hybrid LinUCB on synthetic datasets with varying context (d) and action (f) feature vector sizes.	160
9.3	Execution time results for Hybrid HD-CB and Hybrid LinUCB (standard and SMW variants) evaluated on a general-purpose CPU and the embedded Klessydra core.	162
9.4	Execution time results (in clock cycles) for Hybrid HD-CB and Hybrid LinUCB (standard and SM variants) with hardware acceleration using the HDCU and VCU, respectively. The experiments varied the action feature vector size (f) while keeping $d = 8$ and $N = 8$ fixed.	163
9.5	Comparison of memory requirements for HD-CB Hyb _{BIN} and Hybrid LinUCB	164
10.1	Learning behavior of HD-CB _{UNC2} with confidence multiplexed into the estimate memory via permutation.	170
10.2	Learning behavior of HD-CB _{EPS} with one single global memory	171
10.3	Effect of hypervector dimensionality on the regret penalty induced by the compact single-memory formulation.	175
10.4	Memory footprint versus number of actions for baseline HD-CB variants and proposed single-memory formulations.	175
11.1	Clipping function used to bound integer-valued memories.	179
11.2	Memory footprint (KiB, log scale) vs. context dimensionality d for baseline HD-CB, HD-CB _{BIN} and HD-CB _{PROB}	183

11.3	Per-round prediction loss of Loss-HD-CB _{EPS} on a synthetic contextual bandit dataset ($N = 10, d = 5, D = 1000$). The raw loss (grey) and its moving average (green, window 50) show clear convergence from approximately 0.04 to a residual level below 0.005, confirming that the update rule effectively minimizes the squared-error objective. . .	196
11.4	Comprehensive comparison of all HD-CB variants: average reward (OPE, averaged across all (N, d) configurations) versus bits per element ($\log_2$ scale).	200
11.5	Graphical correspondence between RVFL and HD-CB	203
12.1	Integration of the HDCU in the Klessydra-T03 core.	214
12.2	High-level schematic of the HDCU microarchitecture showing functional units and scratchpad memories.	216
12.3	Bundling Unit with M set to 4	217
12.4	Permutation Principle Scheme applied to a 128-bit HV processed using a SIMD of 32	218
12.5	Maximum achieved (a) speedup and (b) energy reduction factor for each benchmark and stage	227
13.1	Redrawn illustration from [3] showing the possible discretized values in $\mathbb{Z}_{16}$ for a single component of an MCR HV with $r = 16$	232
13.2	Decoding performance and information capacity analysis for MCR compared to BSC, MAP-I, and FHRR.	237
13.3	Average accuracy across 123 classification datasets for 1-bit BSC, MAP-I4, MCR-4 and MAP-C32, using HVs of size $D = 1024$. MCR outperforms BSC and MAP-I4 and approaches MAP-C32.	238
13.4	Scatter plot comparing classification accuracy of MAP-I4 (x -axis) against MCR-4 (y -axis) across 123 classification datasets.	239
13.5	Average classification accuracy across 123 classification datasets for MCR-4 with different D compared to BSC ($D = 1024, b = 1$).	240
13.6	High-level schematic of the MCR-HDCU microarchitecture showing functional units and scratchpad memories.	242
13.7	Architecture of the MCR Distance Unit.	243
13.8	Architecture of the MCR Superposition Unit.	244
13.9	Overview of the real datasets selected for the MCR evaluation	248
13.10	Execution time of 7 selected datasets on the baseline Klessydra-T03 core compared to the MCR-HDCU accelerator with different SIMD $\in \{8, 16, 32, 64\}$	249
14.1	Periodicity of the RHC similarity kernel for a single modulus ($m = 5$).	255
14.2	RHC similarity kernel for combined moduli ($m_1 = 3, m_2 = 5$).	255
14.3	Architecture of the proposed accelerator for FHRR, featuring dedicated memories and specialized functional units for each basic FHRR operation connected via a shared bus to an intermediate buffer, enabling efficient chaining of HDC operations.	259
14.4	Architecture of the FHRR Encoding Unit.	260
14.5	Architecture of the FHRR Similarity Unit.	261

14.6 Architecture of the FHRR Bundling Unit.	263
14.7 Architecture of the Resonator Network Unit.	265
16.1 Overview of the Aeneas Hypercompute framework	288
16.2 Reconfigurable hardware parallelism in AeneasHDC	291
16.3 Architecture of the AeneasHDC design-space exploration engine. . .	292
16.4 Example of design space exploration in AeneasHDC	297
16.5 Cumulative average score progression during design-space for two representative scenarios.	298

List of Tables

2.1	Microcontroller units (MCUs) comparison, ordered by clock frequency [4].	28
2.2	Latency (clock cycles), maximum operating frequency $f_{\max}$ (MHz), and FPGA resource usage for addition, multiplication, and division in INT32, 32-bit fixed-point, and FP32 on the Xilinx Zynq UltraScale+ ZCU106 (EK-U1-ZCU106-G) device.	34
2.3	Overview of selected HDC/VSA models, their vector spaces and core operations (adapted and extended from [5]). The models highlighted in bold are those that will be treated in detail in this thesis.	49
3.1	Average latency of integer dividers for uniformly distributed numbers, including the ones required by the hardware implementation	79
3.2	Operating frequency, execution time, area and energy consumption results obtained on Xilinx Virtex UltraScale+ VCU118 board	81
3.3	Operating frequency, execution time, area and energy consumption results on Synopsys Fusion Compiler in GF22FDX technology. . . .	81
4.1	Operating frequency, latency and area results obtained on Xilinx Virtex UltraScale+ VCU118 board.	90
4.2	Hardware Resources for the VLNPD, varying the Bit-width	91
4.3	Vector Division Instructions in Klessydra Instruction Set	91
4.4	Average Division Latency as a Function of SIMD for the VLNPD-Std and different Radix- n Implementations	92
5.1	Asymptotic computational complexity in the original Disjoint and Hybrid Algorithms and in the proposed optimized versions	100
5.2	Performance results for the Disjoint Algorithm in the Std. and Opt. versions on the examined execution platforms	109
5.3	Performance results for the Hybrid Algorithm in the Std. and Opt. versions on the examined execution platforms	109
6.1	Impact of omitting the matrix inverse step on policy value. The forward product $A_a b_a$ consistently underperforms the ridge regression solution $A_a^{-1} b_a$ by approximately -9% . Results are averaged over 8 configurations ($N \in \{5, 10, 15, 20\}$, $d \in \{10, 15, 20\}$), 10 datasets $\times$ 5 simulation runs per configuration. All policies use $\epsilon = 0.05$	120

6.2	Policy value comparison: linear random projection ($D = 4096$) via a dense Gaussian matrix without the precision matrix versus the linear baselines. The projected policy performs identically to the unprojected baseline without the the precision matrix, and both trail the full-inversion reference by approximately -9% . Increasing D to 8192 results in no change. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.	120
6.3	Policy value comparison: FlyHash encoding with unscaled rewards versus linear random projection, both without the precision matrix. FlyHash achieves $+5\%$ average improvement over linear projection, demonstrating that the nonlinear encoding alone provides a measurable benefit. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.	122
6.4	Effect of omitting the covariance matrix A_a in the variant with FlyHash . The estimator with only b_a vector matches the $A_a b_a$ variant, confirming that A_a does not contribute useful information. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.	124
6.5	Effect of reward centering ($r' = r - 0.5$) on the b_a -only FlyHash estimator. Centering further closes the gap to the standard linear bandit estimator, from $\approx -4\%$ to $\approx -2\%$. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.	124
7.1	Off-policy evaluation for HD-CB models and traditional CB algorithms on synthetic datasets with binary rewards, with varying number of actions N and context feature vector sizes d . Average rewards $\pm$ std. deviation were calculated for each configuration across 50 datasets using OBP.	134
7.2	Computational complexity of contextual bandit algorithms	138
8.1	Performance Comparison of LinUCB, Real-Valued HD-CB, and HD-CB _{BIN} Across Context Sizes (d) and Number of Actions (N).	149
8.2	Accelerator Configuration and Resource Utilization	151
9.1	Hybrid bandit: per-round execution time (ms) and average regret (best α per model). Binary Reward.	159
9.2	Hybrid bandit: per-round execution time (ms) and average regret (best α per model). Continuous Reward.	159
9.3	Movielens: Average model-based regret for Hybrid LinUCB and Hybrid HD-CB as the number of movies N increases.	161
10.1	Per-round execution time (ms) across devices (means only). HD-CB is implemented in the original version (Loop) and Matrix.	168
10.2	Hybrid bandit: average per-round execution time (ms) and average regret (best α per model). Binary Reward.	174
10.3	Average per-round execution time (ms) across devices.	174

11.1	Off-policy evaluation results on the OBP library. The table compares the real-valued variant against two variants with low-precision representations. Best values among these variants are underlined; global bests are in bold.	182
11.2	Off-policy evaluation results comparing LinEPS, HD-CB _{REAL} , and the estimated Loss variants. The best performing method in each row is highlighted in bold.	197
11.3	Off-policy evaluation results comparing LinEPS, HD-CB _{EPS} , and all proposed variants. Columns report average reward for each (N, d) configuration. The best performing method in each row is highlighted in bold.	201
12.1	ISE for HDC Operations	214
12.2	HDCU Configuration Parameters	220
12.3	Hardware Requirements of the Klessydra-T03 and the HDCU with $\text{SIMD} = 32$; $f = 220$ MHz; SPMs = 2 kB	222
12.4	Hardware Requirements of HDCU for Different SIMD Configurations	223
12.5	Execution time for each operation (in μs)	223
12.6	Energy consumption for each operation (in μJ)	224
12.7	Execution time in μs per task and configuration. Configurations achieving the highest speedup over software execution are highlighted in bold.	226
12.8	Energy Consumption in μJ per Task and Configuration. Configurations achieving the highest energy reduction factor over software execution are highlighted in bold.	226
12.9	Comparison of HDC Architectures and Feature Support	228
13.1	MCR-HDCU Configuration Parameters	242
13.2	Hardware requirements of MCR-HDCU for different SIMD.	245
13.3	Execution Time in μs and Energy Consumption in μJ per arithmetic operation in MCR	246
13.4	Execution time in μs comparing software and SIMD hardware realizations across the selected datasets	250
13.5	Execution time (μs) and Energy consumption (μJ) for BSC ($D = 1024$) and MCR-4 ($D = 64$), across datasets and SIMD configurations	250
14.1	Post-Implementation Resource Utilization on ZCU106 (XCZU7EV) for varying SIMD widths ($D = 1024$, $Q_\phi = 10$, Q16.16).	268
15.1	Baseline success rate of the resonator network on 3-SAT instances using the Subset Sum reduction of Section 15.3 and 2-entry (pick / not-pick) codebooks for each subset element (base-10 representation, $D = 10,000$).	278
15.2	Success rate with dedicated vs grouped codebooks. Grouping reduces the number of factors and generally improves performance, especially for intermediate problem sizes.	281

15.3	Representative configurations and success rates for 3-SAT instances with $n_{\text{vars}} = 20$ and $n_{\text{clauses}} = 30$ under different combinations of grouping, numeric representation, and activation rule (all at $D = 10,000$).	282
15.4	Approximate search-space size M for selected 3-SAT configurations under the considered mapping.	284
16.1	Explorable design space in the AeneasHDC framework. Parameters are grouped into model architecture, encoding, training, and inference categories.	289
16.2	Summary of HDC model configurations and datasets for the single-configuration validation experiments.	294
16.3	AeneasHDC: Test configurations	295
16.4	CARDIO classification: comparison between the manually configured model (Test 1) and the best configuration identified by the area-constrained DSE (budget: 10 000 LUTs, 10 000 FFs).	297

List of Acronyms

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BLAS	Basic Linear Algebra Subprograms
BRAM	Block Random Access Memory
BSC	Binary Spatter Codes
CB	Contextual Bandits
CLZ	Count Leading Zeros
CNN	Convolutional Neural Network
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
CSR	Control and Status Register
CTR	Click-Through Rate
DL	Deep Learning
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSE	Design-Space Exploration
DSP	Digital Signal Processor
ELM	Extreme Learning Machine
EMG	Electromyography
FF	Flip-Flop
FFT	Fast Fourier Transform
FHRR	Fourier Holographic Reduced Representations
FLOP	Floating Point Operation
FP32	32-bit Floating Point

FPE	Fractional Power Encoding
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GCC	GNU Compiler Collection
GDPR	General Data Protection Regulation
GEMM	General Matrix Multiply
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HD-CB	Hyperdimensional Contextual Bandits
HDC	Hyperdimensional Computing
HDCU	Hyperdimensional Coprocessor Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HRR	Holographic Reduced Representations
HV	Hypervector
IC	Integrated Circuit
IoT	Internet of Things
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
kNN	k-Nearest Neighbors
LinEPS	Linear Epsilon-Greedy
LinUCB	Linear Upper Confidence Bound
LUT	Look-Up Table
LVQ	Learning Vector Quantization
MAC	Multiply-Accumulate
MAP	Multiply-Add-Permute
MCR	Modular Composite Representations
MCU	Microcontroller Unit

MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSB	Most Significant Bit
MSE	Mean Squared Error
NMI	Normalized Mutual Information
NPD	Non-Performing Divider
NPU	Neural Processing Unit
OBP	Open Bandit Pipeline
OPE	Off-Policy Evaluation
RAM	Random Access Memory
RHC	Residue Hyperdimensional Computing
RISC	Reduced Instruction Set Computing
RL	Reinforcement Learning
RNS	Residue Number System
RTL	Register-Transfer Level
RVFL	Random Vector Functional Link
SDM	Sparse Distributed Memory
SGD	Stochastic Gradient Descent
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SMW	Sherman–Morrison
SMW	Sherman–Morrison–Woodbury
SNN	Spiking Neural Network
SoC	System-on-Chip
SPM	Scratchpad Memory
SRAM	Static Random Access Memory
STDP	Spike-Timing-Dependent Plasticity
TOPS	Tera Operations Per Second

UCB	Upper Confidence Bound
VCU	Vector Coprocessor Unit
VHDL	VHSIC Hardware Description Language
VLNPD	Variable-Latency Non-Performing Divider
VSA	Vector Symbolic Architectures
WTA	Winner-Take-All

List of Symbols

Symbol	Description
General Notation	
$\mathbb{R}$	Set of real numbers
$\mathbb{Z}$	Set of integers
$\mathbb{C}$	Set of complex numbers
$\mathbb{B}$	Set of binary values $\{0, 1\}$
t	Time step or iteration index
T	Total number of time steps (horizon)
$\mathcal{O}(\cdot)$	Big-O notation for asymptotic computational complexity
$\ \cdot\ $	Euclidean norm (L_2 norm)
$^\top$	Transpose operator
$\sigma(\cdot)$	Sigmoid function
Contextual Bandits (LinUCB / LinEPS)	
N	Number of available actions (arms)
a	Action index, $a \in \{1, \dots, N\}$
a_t	Action selected at time step t
a_t^*	Optimal action in hindsight for context $\mathbf{x}_t$
d	Dimensionality of the context feature vector
f	Dimensionality of the action feature vector (Hybrid setting)
$\mathbf{x}_{t,a}$	Context feature vector for action a at time t ($\in \mathbb{R}^d$)
$\mathbf{u}_a$	Action feature vector for action a ($\in \mathbb{R}^f$)
$r_{t,a}$	Reward received for selecting action a at time t
$\hat{r}_{t,a}$	Estimated (predicted) reward for action a at time t
R_T	Cumulative regret over T rounds
$\mathbf{A}_a$	Covariance matrix for action a ($\in \mathbb{R}^{d \times d}$)
$\mathbf{b}_a$	Bias vector for action a ($\in \mathbb{R}^d$)
$\hat{\boldsymbol{\theta}}_a$	Estimated parameter vector (weights) for action a
α	Exploration parameter controlling the confidence interval width
ϵ	Exploration probability in ϵ -greedy strategies
$p_{t,a}$	Potential (estimated payoff + exploration bonus) for action a at time t
$s_{t,a}$	Uncertainty term for action a at time t
$\mathbf{A}_0$	Shared covariance matrix in Hybrid LinUCB
$\mathbf{b}_0$	Shared bias vector in Hybrid LinUCB
$\boldsymbol{\beta}$	Shared parameter vector in Hybrid LinUCB
$\mathbf{B}_a$	Cross-term matrix for action a in Hybrid LinUCB ($\in \mathbb{R}^{d \times k}$)
$\mathbf{z}_{t,a}$	Shared feature vector for action a at time t
k	Dimensionality of the shared feature vector in Hybrid LinUCB

Hyperdimensional Computing (HDC/VSA)

Symbol	Description
D	Dimensionality of the hypervector space ($D \gg d$)
$\mathcal{H}$	The hyperspace (e.g., $\{0, 1\}^D$, $\{-1, +1\}^D$, or $\mathbb{C}^D$)
$\oplus$	Superposition (Bundling) operator
$\otimes$	Binding operator
$\oslash$	Unbinding (inverse of binding)
$\odot$	Element-wise (Hadamard) product
$\rho(\cdot)$	Permutation operator (typically cyclic shift)
$\rho^k(\cdot)$	k -fold application of permutation
$\delta(\cdot, \cdot)$	Similarity or distance metric (e.g., Cosine, Hamming)
ϕ	Phase angle in FHRR representations
Q	Bit-width parameter
L	Number of quantization levels

HD-CB Framework

$\mathcal{X}_{t,a}$	Encoded context hypervector for action a at time t
$\mathcal{A}_a$	Representative (prototype) hypervector for action a
$\mathcal{B}_a$	Confidence hypervector for action a (HD-CB _{UNC} variants)
$\mathbf{A}$	Prototype matrix $[\mathcal{A}_1, \dots, \mathcal{A}_N]^\top \in \mathbb{R}^{N \times D}$
$\mathbf{B}$	Confidence matrix $[\mathcal{B}_1, \dots, \mathcal{B}_N]^\top \in \mathbb{R}^{N \times D}$ (HD-CB _{UNC} variants)
$R_{t,a}$	Reward-encoded hypervector for action a at time t
$e_{t,a}$	Estimated payoff (similarity score) for action a at time t
$c_{t,a}$	Confidence value for action a at time t
$u_{t,a}$	Uncertainty value for action a at time t ($= 1 - c_{t,a}$)
β	Smoothing factor for exponential moving average (HD-CB _{UNC1})

Probabilistic HD-CB

κ	Saturation threshold bounding components to $[-\kappa, +\kappa]$
α_t	Time-decaying base update probability
α_0	Initial update probability
P_t	Effective update probability at time t
$f_\kappa(\cdot)$	Saturation function: $\max(-\kappa, \min(\cdot, +\kappa))$

Loss-Based HD-CB

L_t	Per-round squared-error loss at time t
g_t	Gradient scalar factor at time t
η	Learning rate
λ	L_2 regularization coefficient
G_t	Prediction error: $r_{t,a_t} - \hat{r}_{t,a_t}$ (Loss-Probabilistic variant)
ν	Normalization constant ($= D\kappa/2$) for clipped-ReLU

RVFL / Neural Network Interpretation

$\mathbf{W}_{\text{in}}$	Random, frozen input weight matrix ($\in \mathbb{R}^{D \times n}$)
$\mathbf{W}_{\text{out}}$	Trainable output weight matrix ($\in \mathbb{R}^{C \times D}$)
$\mathbf{H}$	Hidden-layer activation matrix ($\in \mathbb{R}^{M \times D}$)

Symbol	Description
$\mathbf{h}$	Hidden-layer representation vector ($\in \mathbb{R}^D$)
$\mathbf{C}$	Codebook matrix containing atomic hypervectors
Φ	Random projection matrix or encoding map

VLNPD (Variable-Latency Divider)

R	Remainder register (64-bit)
D_{div}	Divisor register (32-bit)
shift_amount	Number of iterations to skip via dynamic shift
$\text{leading_}R$	Number of leading zeros in the remainder register
$\text{leading_}D$	Number of leading zeros in the divisor

Chapter 1

Introduction

The real voyage of discovery consists
not in seeking new landscapes, but
in having new eyes.

Marcel Proust

For millennia, humans have been fascinated by the workings of the brain and by the dream of replicating its abilities to perceive, remember, imagine, and decide. Across cultures and eras, thinkers, writers, philosophers, and mathematicians have imagined and described artificial beings endowed with human-like intelligence, planting the seeds for what would eventually become the scientific pursuit of Artificial Intelligence (AI) [6–9]. Over the past two decades, AI has undergone a profound renaissance, evolving from a scientific curiosity into a ubiquitous general-purpose technology deployed across critical sectors and everyday services [10–15].

Yet, this success hides a stark reality: the AI revolution has come at a computational and energy cost that today is becoming physically and economically unsustainable. An emerging *efficiency crisis* now spans the entire computing spectrum, from massive foundation models in cloud data centers to lightweight classifiers on battery-powered sensors [16, 17].

At one end of this spectrum, the computational demands of frontier models have grown at an unprecedented pace, doubling roughly every 2 months since 2020 [18]. Flagship models now demand roughly ten billion times more computation than their predecessors from a decade ago [19], driving ever-larger GPU clusters and an unsustainable projected 2× increase in energy consumption between 2024 and 2030 [20], shifting the bottleneck in data centers from raw throughput to power delivery and cooling.

At the opposite end of the spectrum, scaling hardware is not an option: computation must operate within tight budgets of power, memory, latency, and cost. This is the domain of the Internet of Things (IoT): a global network of small, connected devices equipped with sensors, actuators, and modest processing capabilities [21, 22]. These devices continuously collect data from the physical world, transmit it over networks, and process it in real time. Over the past decade, the IoT has evolved into a global-scale infrastructure where connected devices already outnumber the global population and produce roughly 75 zettabytes of data annually as of 2025 [23], with

a projected 39 billion endpoints by 2030 [24].

In this scenario, the traditional cloud-centric model, in which raw data are streamed to centralized data centers for processing, is increasingly impractical [25]. Transmitting such massive volumes of data would overwhelm network infrastructure and incur prohibitive costs in latency, bandwidth, and energy. Time-critical applications like autonomous driving require millisecond-scale responses, yet round-trip latency to cloud servers can exceed 200 ms. Furthermore, transmitting sensitive information—faces, voices, or biomedical signals—raises serious privacy and regulatory concerns [26]. Intel estimated that a single autonomous vehicle could generate on the order of 4 TB of sensor data per day, making continuous upload and storage in the cloud economically prohibitive [27].

In recent years, these constraints have driven an urgent paradigm shift toward Edge AI, where intelligence is pushed directly to the data source [28]. Studies consistently show that local AI processing can reduce total system energy by up to 90% and reduce environmental footprints by more than 80% [29] compared to cloud-based inference [30, 31]. By 2030, edge inference is projected to account for approximately 50% of all enterprise AI inference workloads [32], with global spending on edge computing solutions already estimated at \$261 billion as of 2025 [33].

Edge AI is no longer a feature—it is a necessity to realize smart, autonomous units capable of reacting to the environment and making decisions without relying on cloud connectivity. However, this shift creates a fundamental tension between the growing algorithmic demand and hardware reality: edge devices operate with constrained processing power, kilobyte-scale memory, and strict battery budgets [34, 35]. Running even modest AI algorithms on these platforms can exceed available resources or drain batteries within hours [36].

Taken together, these trends define the core problem addressed by this thesis. The exponential growth of AI capabilities is colliding with the physical limits of current computing architectures across the entire stack; enabling the next generation of pervasive intelligence requires a fundamental rethinking of how learning algorithms are designed for—and with—hardware [35]. The primary levers for future AI progress are no longer aggressive device scaling, but rather hardware-algorithm co-design strategies that extract maximum computation per joule and per bit.

Evidence of this path is already visible in the roadmaps of industry leaders. While NVIDIA [37] reports energy efficiency improvements of nearly three orders of magnitude over the past decade, a decomposition of these gains reveals a striking reality: traditional semiconductor scaling contributed only approximately $2.5\times$. The remaining roughly $400\times$ was achieved through architectural and algorithmic innovation—domain-specific design, tight hardware–software co-optimization, and the exploration of new computing paradigms.

1.1 Objectives and Research Questions

It is within this technological and conceptual landscape that the present thesis is situated. This dissertation targets Edge AI systems that must be not only intelligent but also autonomous and responsive, adopting Contextual Bandits (CB) as its primary algorithmic case study for enabling agents to learn from local interactions

and evolve continuously. CB are online sequential decision-making problems under partial feedback that underpin many adaptive services and provide a lightweight alternative to full reinforcement learning when future contexts are independent of chosen actions. Their ability to update policies from logged interactions has led to growing adoption in edge scenarios such as personalized recommendation, adaptive resource allocation, and autonomous systems [38–40]. However, achieving this real-time adaptability imposes a computational overhead that is frequently overlooked in theoretical models but becomes the limiting factor in practice. Making online decision-making viable on resource-constrained embedded platforms, therefore, poses fundamental computational challenges, rendering contextual bandits an ideal testbed for studying the AI efficiency crisis on the edge.

This dissertation addresses this challenge by exploring the intersection of algorithmic theory and hardware implementation. Specifically, it posits that bridging the gap between AI complexity and edge constraints requires moving beyond isolated optimizations. Instead, this work advances three complementary directions:

1. *Algorithmic Optimization*: developing and implementing techniques to reduce computational complexity and memory footprints without sacrificing accuracy.
2. *Hardware Specialization*: designing dedicated arithmetic units and accelerators that target favorable area–latency–energy trade-offs.
3. *Neuro-inspired Computing*: reformulating learning algorithms through brain-inspired paradigms to leverage inherently parallel and hardware-friendly representations.

Crucially, this thesis contends that the largest efficiency gains arise not from any single direction in isolation, but from their joint optimization: a principle of *full-stack co-design* that runs throughout this work. Accordingly, the central research problem is formulated not merely as an optimization task, but as an investigation into the limits of embedded intelligence:

? Research Question

Can the computational bottlenecks of online decision-making algorithms on embedded platforms be overcome through hardware-algorithm co-design and neuro-inspired representations? More broadly, can this approach decouple adaptive intelligence from brute-force hardware scaling?

To address this problem, the research trajectory follows a progressive narrowing of the efficiency gap, structured into three parts corresponding to three distinct levels of intervention. Contextual bandits serve as the common thread that connects the three parts, providing a consistent and demanding workload against which each layer of optimization is evaluated. However, each part also yields contributions of independent value that extend well beyond the bandit domain. Each part addresses a specific subsidiary research question:

- **Part I: Optimization of Conventional Models.** Starting from the observation that division and matrix inversion dominate the cost of standard linear CB algorithms, this part investigates the limits of conventional optimization:

? Research Question

RQ1: How far can arithmetic, algorithmic, and microarchitectural co-design push the efficiency of classical linear bandit algorithms?

In response, it proposes cross-layer optimizations—including custom arithmetic units and approximated inversions—that yield order-of-magnitude improvements in latency and energy.

- **Part II: The Neuro-inspired Shift.** Yet, even with the optimizations proposed in Part I, the inherent limitations and quadratic scaling of dense linear algebra remain, hindering deployment on extreme edge devices. Recognizing this barrier, Part II marks a paradigm shift and asks:

? Research Question

RQ2: Instead of further optimizing the execution of classical kernels, can we change the nature of the computation itself to bypass the quadratic scaling limit?

To answer this, we reformulate contextual bandit learning using neuro-inspired Hyperdimensional Computing (HDC), achieving linear complexity and hardware-friendly operations while maintaining competitive decision quality.

- **Part III: Dedicated Hardware Design.** The final part closes the loop by recognizing that the full potential of these new algorithms can only be realized when the underlying hardware is tailored to their unique properties, such as massive parallelism and low-precision arithmetic. Accordingly, this part asks:

? Research Question

RQ3: What efficiency gains can be unlocked by translating neuro-inspired learning algorithms into dedicated hardware accelerators?

and designs dedicated accelerators and automated deployment tools, unlocking the final tier of efficiency required for autonomous edge intelligence. Importantly, while the hardware accelerators developed in this part are motivated by and benchmarked against contextual bandit tasks, their architectures are deliberately general and can accelerate any HDC learning pipelines.

Each part is structured around these targeted research questions, making explicit the bottleneck addressed, the methodology employed, and the contributions achieved.

Notably, while this thesis targets constrained embedded platforms, its core principles generalize beyond the edge: they address the inefficiencies of CB and thus extend naturally to applications in high-performance data centers, where energy density is a primary bottleneck. Moreover, the novel arithmetic units and HDC hardware accelerators presented in Parts I and III extend to a broader range of AI tasks.

1.2 Summary of Contributions

To address the research problem defined above, this dissertation makes the following contributions, organized according to the three-part structure:

Part I: Arithmetic and Algorithmic Optimization

- **Contribution 1: Data-Aware Arithmetic Optimization.** We introduce the Variable Latency Non-Performing Divider (VLNPD), a novel data-dependent division algorithm that dynamically adjusts execution time by exploiting relationships among input operands. Through early termination and iteration skipping, VLNPD significantly reduces average latency and energy consumption while maintaining a compact implementation suitable for AI applications on embedded platforms. We present four hardware variants offering different trade-offs between latency, area, and energy efficiency (Chapter 3), and demonstrate integration into a RISC-V vector coprocessor, proving that exploiting operand statistics can significantly reduce the latency of division-intensive workloads (Chapter 4). Together, these contributions reduce the impact of the division bottleneck in linear bandit workloads from 70% of the total execution time to approximately 12%.
- **Contribution 2: Optimized Linear Contextual Bandits (Chapter 5).** We propose optimized implementations of Linear Bandits that replace standard matrix inversions with incremental Sherman–Morrison–Woodbury updates, reducing computational complexity scaling from cubic to quadratic with the context size. We assess the impact of this optimization on different embedded platforms, and analyze its impact on the algorithms’ stability. Combined with vectorized kernels and the hardware optimizations of the preceding chapters, this co-design achieves up to $58\times$ speedup and $50\times$ energy reduction compared to standard baselines.

Part II: Neuro-Inspired Contextual Bandits

- **Contribution 3: Ablation of Covariance-Matrix Dependence in Linear Bandits (Chapter 6).** We investigate what role the precision matrix plays in the linear bandit estimator and whether it can be bypassed by encoding contexts in a high-dimensional space. Controlled ablations across three regimes—low-dimensional features, dense Gaussian projection, and biologically-inspired sparse projection with k -Winners-Take-All—show that dimensionality expansion alone is insufficient, that nonlinearity is the critical ingredient, and that under nonlinear encoding the covariance matrix reduces to a scaled reward vector plus cross-talk noise, and can be omitted entirely. These findings establish the theoretical foundation and empirical evidence for the transition to hyperdimensional computing.
- **Contribution 4: Hyperdimensional Contextual Bandits (HD-CB) (Chapter 7).** We introduce HD-CB, the first framework that models and solves contextual bandit learning directly in high-dimensional space via lightweight

component-wise operations. We detail this approach in four variants tailored to different exploration strategies and resource constraints. This reformulation achieves linear time complexity, faster convergence, and a smaller memory footprint compared to linear baselines, making the approach particularly suitable for hardware acceleration and deployment on resource-constrained platforms. We analyze its performance in terms of average reward, execution time, convergence speed, and memory footprint, comparing all proposed schemes with traditional CB algorithms using synthetic datasets and real benchmarks.

- **Contribution 5: Binary Hyperdimensional Bandits (HD-CB_{BIN}) (Chapter 8).** We specialize the HD-CB framework for extreme efficiency with HD-CB_{BIN}, a binary variant that minimizes memory footprint for deployment on low-end microcontrollers. By restricting hypervectors to binary representations, this variant enables lightweight bitwise operations that are particularly efficient on resource-constrained hardware. When both HD-CB_{BIN} and linear bandits are accelerated on dedicated hardware, HD-CB_{BIN} achieves up to $141\times$ speedup, with the advantage growing with the context dimensionality.
- **Contribution 6: Hybrid Hyperdimensional Bandits (HD-CB_{Hybrid}) (Chapter 9).** We extend the HD-CB paradigm to hybrid contextual bandit settings, demonstrating how shared feature representations can be learned in hyperspace to model complex context-action interactions. The linear scaling of Hybrid HD-CB contrasts sharply with the $O(d^3 f^3)$ cost of standard Hybrid LinUCB. When both algorithms are accelerated on hardware, Hybrid HD-CB achieves up to $48.9\times$ speedup, with the advantage growing as the action feature dimension increases.
- **Contribution 7: Memory-Optimized HD-CB Variants (Chapter 10)** We derive memory-optimized HD-CB formulations that reduce storage requirements from one hypervector per action to a single global memory vector by exploiting permutation-based channel multiplexing and label-based superposition. This compression makes HD-CB deployment feasible on embedded platforms where storing many separate hypervectors would exceed on-chip memory.
- **Contribution 8: Probabilistic Hyperdimensional Bandits (HD-CB_{Prob}) (Chapter 11).** We develop HD-CB_{Prob}, a low-precision variant that addresses the unbounded growth of action hypervectors in standard HD-CB without the information loss of periodic binarization. This approach bounds the learning by design to a predefined precision by introducing a probabilistic update rule. At each time step, only a randomly selected subset of vector components is updated, enabling the use of low-precision, saturated integer counters by controlling the learning rate with time-decaying update probabilities. This allows the agent to retain historical information with high-fidelity using low-precision components. As a result, overflow is avoided, update costs decrease over time, and learning no longer relies on periodic binarization steps that may severely degrade the usefulness of learned representations.

- **Contribution 9: Loss-Based Hyperdimensional Bandits (HD-CB_{Loss}) (Chapter 12).** We first derive the implicit loss function that the baseline HD-CB update rule minimizes, revealing a per-component linear loss and an implicit ensemble structure where each hypervector position operates as an independent weak learner at a distinct reward threshold. Diagnosing the limitations of this implicit objective, we then formulate HD-CB_{Loss}, a variant that replaces heuristic prototype updates with explicit minimization of a differentiable loss function over hyperdimensional representations, enabling gradient-based learning while preserving the computational simplicity of HDC operations. We further combine the loss-based update direction with the bounded stochastic mechanics of Probabilistic HD-CB, yielding HD-CB_{Loss-Prob}—a variant that retains error-driven learning within fixed-precision integer arithmetic suitable for extreme edge deployment. Building on these formulations, we finally map the entire HD-CB pipeline onto a Random Vector Functional Link network.

Part III: Hardware Realization

- **Contribution 10: Hyperdimensional Coprocessor Unit for Binary HDC (HDCU, Chapter 12).** We introduce the Hyperdimensional Coprocessor Unit (HDCU), a highly flexible hardware accelerator for binary hyperdimensional computing integrated into the execution stage of the Klessydra-T03 RISC-V core. The architecture features optimized functional units for bundling, binding, permutation, and similarity, with dedicated scratchpad memories for low-latency, high-bandwidth access. Crucially, the design is configurable at synthesis time in data parallelism, supported operations, and memory size, while allowing the hypervector dimension to be set at runtime. A custom RISC-V Instruction Set Extension, fully integrated into the GCC toolchain, exposes the accelerator to software via simple intrinsics, enabling programmers to accelerate diverse learning tasks without hardware expertise. To broaden portability beyond the Klessydra ecosystem, we also developed an AXI4-compliant standalone version that preserves the core functional units while replacing the ISA-coupled interface with standard AXI4 datapaths, enabling seamless integration into any AXI-capable SoC.
- **Contribution 11: Hardware Acceleration for Modular Composite Representations (MCR-HDCU, Chapter 13).** Motivated by the representational limits of binary hypervectors, we explored Modular Composite Representations (MCR), a higher-capacity model within the hyperdimensional computing family, analyzing its feasibility for resource-constrained platforms. Our contribution is twofold: first, we reposition MCR in the literature, demonstrating through large-scale experiments that it provides a more favorable trade-off than binary HDC: higher information capacity and classification accuracy, with reduced memory overhead. Second, we introduce MCR-HDCU, the first dedicated accelerator for this model, showing that modular arithmetic maps naturally to digital logic through implicit binary overflow, avoiding the cost of explicit modulo operators. At equivalent accuracy, MCR-HDCU achieves faster execution and lower energy consumption than binary models

accelerated with the original HDCU.

- Contribution 12: Discretized Accelerator for Fourier Holographic Reduced Representations (Chapter 14).** We address the most expressive model in the family of hyperdimensional computing: the Fourier Holographic Reduced Representations (FHRR). While FHRR’s complex-valued, unit-modulus hypervectors offer superior representational capacity, their reliance on high-precision complex arithmetic has historically precluded efficient hardware implementation and adoption on embedded platforms. We propose a hardware-friendly discretization strategy that quantizes phases on the unit circle into uniformly spaced angles, enabling modular arithmetic without explicit modulo operators—the same insight that enabled efficient MCR. Building on this, we introduce the first dedicated FHRR accelerator, featuring functional units for all primitive operations as well as the first hardware implementation of a Resonator Network.
- Contribution 13: Vector-Symbolic Reasoning for NP-Complete Problems (Chapter 15).** To validate HDC’s capacity for rigorous symbolic computation beyond learning tasks, we apply FHRR and Resonator Networks to solve NP-complete 3-SAT problems. We exploit a classical polynomial-time reduction from 3-SAT to Subset Sum, encode the resulting integers using FHRR, and leverage resonator dynamics to search for satisfying assignments. The results demonstrate that resonator networks can model and solve combinatorial logic problems, while also revealing fundamental capacity limits that inform future algorithm and hardware design. This proof-of-concept validates the broader applicability of the FHRR accelerator for symbolic reasoning workloads.
- Contribution 14: Automated Co-Design Framework (AeneasHDC, Chapter 16).** We present AeneasHDC, an open-source end-to-end framework that bridges the gap between high-level HDC model specifications and low-level FPGA implementations. The framework supports five VSA families, multiple encoding techniques, configurable quantization strategies, and four learning tasks providing a rich configurable workflow. Given user specifications through an intuitive GUI, AeneasHDC automatically generates both high-level software implementations and synthesizable hardware accelerators through HLS. To navigate the vast design space, AeneasHDC incorporates a Bayesian-guided design-space exploration engine that jointly optimizes accuracy and hardware metrics under user-defined constraints. In area-constrained experiments, the automated exploration discovered configurations that simultaneously improved accuracy by +1.63 percentage points while reducing LUT usage by 63%, FF usage by 58%, dynamic power by 75%, and increasing inference throughput by $3.5\times$ compared to manually selected baselines. By substantially reducing accelerator development time from weeks to minutes, AeneasHDC enables rapid design-space exploration for edge intelligence applications without requiring expert hardware knowledge.

1.3 Outline

The remainder of this dissertation is organized into three parts that correspond to three distinct levels of intervention on the embedded AI stack.

Chapter 2 establishes the conceptual and technological foundations. It introduces the main AI paradigms with a focus on contextual bandits, reviews the constraints of embedded and edge platforms (with emphasis on RISC-V-based architectures), and surveys key approaches to efficiency improvement across the stack, from arithmetic-level optimizations to hardware acceleration and neuro-inspired computing.

Part I: Arithmetic and Algorithmic Optimizations of Linear Bandits

Starting from the observation that division and matrix inversion are the dominant bottlenecks, this part proposes a set of cross-layer optimizations for the linear bandits.

- Chapter 3 dismantles the arithmetic bottleneck of division operations in linear bandits by introducing the VLNPD, a data-dependent division unit that drastically reduces execution time and energy consumption by exploiting operand statistics.
- Chapter 4 integrates the VLNPD into a RISC-V Vector Coprocessor, demonstrating the advantages of variable latency division under SIMD execution.
- Chapter 5 shifts to the algorithmic level, presenting an optimized implementation of linear bandits that replaces explicit matrix inversion with incremental Sherman-Morrison-Woodbury updates. Co-designed with the vector support developed in the preceding chapters, this formulation significantly reduces execution time, memory footprint, and energy consumption.

Part II: From Linear Algebra to Neuro-inspired Computing Despite the optimizations in Part I, the inherent quadratic complexity of dense linear algebra remains a limiting factor for extreme edge deployment. Part II marks a paradigm shift: instead of further optimizing how fast we execute linear models, we change the nature of the computation itself. This part replaces real-valued vectors and matrices with the neuro-inspired hyperdimensional computing, leveraging high-dimensional, distributed representations to achieve learning rules that are inherently parallel, robust, and computationally lightweight.

- Chapter 6 asks whether the precision matrix in linear bandits can be bypassed entirely by encoding contexts in a high-dimensional space. A systematic ablation study shows that nonlinear high-dimensional encoding is the critical ingredient, providing theoretical motivation for the hyperdimensional framework that follows.
- Chapter 7 establishes the theoretical framework by introducing HD-CB, the first model to solve contextual bandit problems directly in high-dimensional space using component-wise operations.
- Chapter 8 specializes the framework for extreme efficiency with HD-CB_{BIN}, a binary variant that minimizes memory footprint and maximizes arithmetic simplicity for low-end microcontrollers.

- Chapter 9 extends the paradigm to Hybrid HD-CB, demonstrating how shared feature representations can be learned in hyperspace to model complex context-action interactions.
- Chapter 10 presents efficient implementations of HD-CB. A matrix-form reformulation exposes data parallelism for BLAS-capable backends, and permutation-based memory compression reduces storage from one or more per-action to a single global memory vector, making HD-CB deployment feasible on memory-constrained platforms.
- Chapter 11 advances the learning foundations of HD-CB. First, probabilistic updates with bounded integer counters replace unbounded accumulation, enabling deployment at 3–4 bit precision. Second, the implicit loss function underlying the baseline update rule is derived, and its limitations are identified. Third, a loss-based formulation grounds HD-CB in explicit optimization, improving accuracy by up to 3 percentage points. The loss and probabilistic mechanisms are then combined into Loss-Probabilistic HD-CB. Finally, a formal mapping onto Random Vector Functional Link networks connects HD-CB to established neural network theory.

Part III: Hardware Acceleration and Co-Design While Part II demonstrates the algorithmic efficiency of HDC, fully exploiting its intrinsic parallelism requires dedicated hardware support. Part III shifts the focus from algorithms to implementation, presenting a family of domain-specific accelerators and automated tools designed to bridge the gap between algorithmic promise and practical deployment.

- Chapter 12 presents the HDCU, a configurable RISC-V extension designed to accelerate binary HDC operations and enable real-time learning.
- Chapter 13 investigates MCR as a higher-capacity alternative to binary hypervectors, repositioning this model through large-scale experiments on 123 classification datasets and introducing the first dedicated hardware accelerator (MCR-HDCU).
- Chapter 14 develops a discretized, hardware-friendly accelerator for FHRR, enabling complex symbolic processing in hardware.
- Chapter 15 demonstrates the symbolic reasoning capabilities of the proposed hardware by applying FHRR and Resonator Networks to solve combinatorial optimization problems (3-SAT).
- Chapter 16 introduces AeneasHDC, an open-source framework that automates the generation of synthesizable FPGA accelerators from high-level HDC model specifications, streamlining the path from research to hardware realization.

Chapter 17 concludes the dissertation, summarizing the main findings and outlining future research directions for efficient, autonomous edge intelligence.

To help the reader navigate the dissertation’s progressive arc—from optimizing conventional models, through a neuro-inspired paradigm shift, to dedicated hardware realization—Fig. 1.1 provides a high-level roadmap. The three parts correspond to

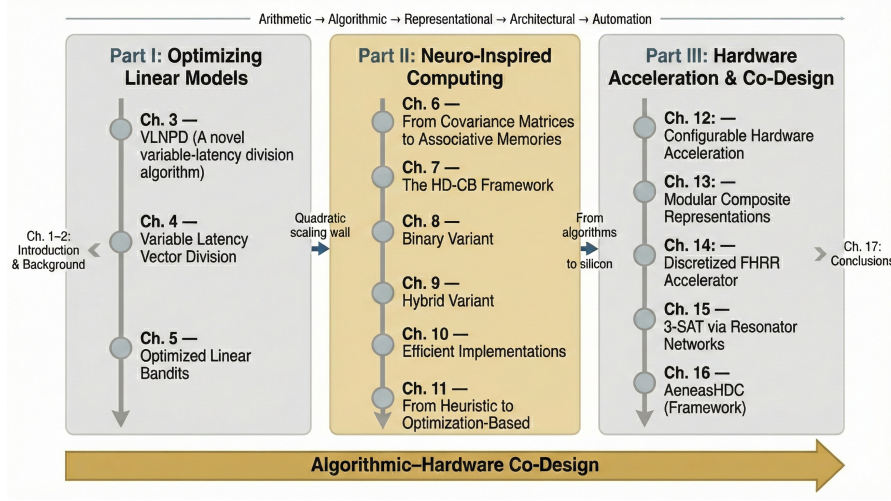


Figure 1.1 High-level dissertation roadmap showing the three-part progression from arithmetic and algorithmic optimization of linear models (Part I), through neuro-inspired reformulation via Hyperdimensional Computing (Part II), to dedicated hardware acceleration and automated co-design (Part III). The top axis indicates the escalating scope of intervention; the bottom arrow represents the algorithmic-hardware co-design principle that underpins the entire work.

three escalating levels of intervention, while the continuous co-design thread running across the bottom reflects the unifying principle that the most substantial efficiency gains emerge from the joint optimization of algorithms and hardware.

1.4 Publications

Most of the material presented in this dissertation is based on research published during my PhD journey. This section lists the corresponding publications, organized by status, and indicates the chapter(s) to which each manuscript is most closely related.

Published

- **Paper 1:** *Design, Implementation and Evaluation of a New Variable Latency Integer Division Scheme*. M. Angioli, M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, S. Jamili, M. Olivieri. *IEEE Transactions on Computers*, 73(7):1767–1779, 2024. Related: Chapter 3.
- **Paper 2:** *Exploring Variable Latency Dividers in Vector Hardware Accelerators*. M. Angioli, M. Barbirotta, A. Cheikh, A. Mastrandrea, M. Olivieri. *PRIME 2024*, pp. 1–4, 2024. Related: Chapter 4.
- **Paper 3:** *Efficient implementation of LinearUCB through algorithmic improvements and vector computing acceleration for embedded learning systems*. M. Angioli, M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, M. Olivieri.

ACM Transactions on Embedded Computing Systems, 2025. Related: Chapter 5.

- **Paper 4:** *HD-CB: The First Exploration of Hyperdimensional Computing for Contextual Bandits Problems*. M. Angioli, A. Rosato, M. Barbirotta, R. Martino, F. Menichelli, M. Olivieri. *IEEE Open Journal of the Computer Society*, 2025. Related: Chapter 7.
- **Paper 5:** *HD-CB_{BIN}: A Lightweight Approach for Contextual Bandit Learning in Real-Time Applications*. M. Angioli, A. Rosato, M. Barbirotta, R. Martino, A. Marcelli, A. Mastrandrea, M. Olivieri. *IEEE International Joint Conference On Neural Networks 2025*. Related: Chapter 8.
- **Paper 6:** *Configurable Hardware Acceleration for Hyperdimensional Computing Extension on RISC-V*. R. Martino, M. Angioli, A. Rosato, M. Barbirotta, A. Cheikh, M. Olivieri. *IEEE Transactions on Computers*, 2025. Related: Chapter 12.
- **Paper 7:** *A General-Purpose AXI Plug-and-Play Hyperdimensional Computing Accelerator*. R. Martino, M. Pisani, M. Angioli, A. Mastrandrea, A. Rosato, M. Olivieri. *Electronics (MDPI)*, 15(2):489, 2026. Related: Chapter 12.
- **Paper 8:** *AeneasHDC: An Automatic Framework for Deploying Hyperdimensional Computing Models on FPGAs*. M. Angioli, S. Jamili, M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, A. Rosato, M. Olivieri. *IEEE International Joint Conference On Neural Networks 2024*. Related: Chapter 16.

Under Review or In Preparation

- **Paper 9:** *Hybrid HD-CB: Contextual Bandits in Hybrid Action Spaces via Hyperdimensional Computing*. M. Angioli et al. *In preparation*. Related: Chapter 9.
- **Paper 10:** *Contextual Bandits for Resource-Constrained Devices using Probabilistic Learning*. M. Angioli, K. Johansson, A. Loutfi, D. Kleyko. *Submitted to IEEE International Joint Conference on Neural Networks 2026*. Related: Chapter 11.
- **Paper 11:** *Loss-Based HD-CB: From Heuristic Updates to Explicit Optimization*. M. Angioli et al. *In preparation*. Related: Chapter 11.
- **Paper 12:** *Efficient Hyperdimensional Computing with Modular Composite Representations*. M. Angioli, C. J. Kymn, A. Rosato, A. Loutfi, M. Olivieri, D. Kleyko. *Submitted to IEEE Emerging Topics in Computational Intelligence*. Related: Chapter 13.
- **Paper 13:** *A Hardware-Friendly Discretized Accelerator for FHRR*. M. Angioli et al. *In preparation*. Related: Chapter 14.

Other Research Contributions

During my PhD, I also contributed to the following research works. While they fall outside the main scope of this dissertation and are not included in the presented material, they contributed to the breadth of my research experience in embedded systems, fault tolerance, and biomedical applications:

- *Automatic hardware accelerators reconfiguration through LinearUCB algorithms on a RISC-V processor.* M. Angioli, M. Barbirotta, A. Mastrandrea, S. Jamili, M. Olivieri. *PRIME 2023*, pp. 169–172, 2023.
- *Dynamic triple modular redundancy in interleaved hardware threads: an alternative solution to lockstep multi-cores for fault-tolerant systems.* M. Barbirotta, F. Menichelli, A. Cheikh, A. Mastrandrea, M. Angioli, M. Olivieri. *IEEE Access*, Vol. 12, pp. 95720–95735, 2024.
- *A Universal Hardware Emulator for Verification IPs on FPGA: A Novel and Low-Cost Approach.* S. Jamili, A. Mastrandrea, A. Cheikh, M. Barbirotta, F. Menichelli, M. Angioli, M. Olivieri. *International Conference on Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES 2023)*, Lecture Notes in Electrical Engineering, Springer, 2024.
- *Heterogeneous tightly-coupled dual core architecture against single event effects.* M. Barbirotta, F. Menichelli, A. Mastrandrea, A. Cheikh, M. Angioli, S. Jamili, M. Olivieri. *International Conference on Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES 2023)*, 2024.
- *Special Session: SE-UVM, an Integrated Simulation Environment for Single Event Induced Failures Characterization and its Application to the CV32E40P Processor.* M. Barbirotta, M. Angioli, A. Mastrandrea, F. Menichelli, A. Cheikh, M. Olivieri. *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct. 2024.
- *Alternative Implementations of Error Correcting Codes in Register Files of a RISC-V Safety Critical Processor.* M. Barbirotta, M. Angioli, A. Mastrandrea, A. Cheikh, F. Menichelli, M. Olivieri. *International Conference on Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES 2024)*. Published in *Lecture Notes in Electrical Engineering*, Springer, 2025, pp. 56–64.
- *Fault tolerant voting circuits: A Dual-Modular-Redundancy approach for Single-Event-Transient mitigation.* M. Barbirotta, M. Angioli, A. Mastrandrea, F. Menichelli, M. Pisani, M. Olivieri. *Microprocessors and Microsystems*, Vol. 119, 2025.
- *A Novel Machine Learning Framework for Drowsiness Detection Using an Electrostatic Wearable Sensor and Hyperdimensional Computing.* L. Ferri, M. Gazzanti Pugliese di Cotrone, M. Angioli, M. Balsi, et al. *8th IEEE International Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0&IoT)*, 2025.

Chapter 2

Background

"The joy of discovery is certainly the liveliest that the mind of man can ever feel."

Claude Bernard

Artificial Intelligence involves developing parametric mathematical models capable of learning from data, adapting to experience, and solving tasks across a vast spectrum of applications. As discussed in the previous chapter, the exponential growth in computational power and data availability has pushed AI to unprecedented capabilities, but also to increasingly unsustainable demands for computation, energy, and memory—demands that are particularly acute for embedded platforms. Understanding how to make AI more efficient, therefore, requires revisiting its theoretical foundations, computational models, and hardware implementations, and examining how these layers interact. Only by looking across the entire computing stack—from learning algorithms and representations down to arithmetic units and microarchitectures—it is possible to identify where efficiency can be gained.

This chapter provides the conceptual groundwork for the contributions presented in this thesis. It is organized into three main sections, each addressing a different aspect of the theoretical and technological background underlying this research.

The first section introduces the field of Artificial Intelligence in its broadest sense, encompassing perception, reasoning, learning, and action as the fundamental capabilities of intelligent systems. It distinguishes between the two main historical paradigms of AI—*symbolic*, based on explicit rule manipulation and logical inference, and *connectionist*, which relies on data-driven models that learn directly from examples. Within this context, the section discusses the three major learning paradigms in AI—supervised, unsupervised, and reinforcement learning—with particular emphasis on the latter, which formalizes how agents can learn to make optimal decisions through interaction with their environment. Special attention is given to the *contextual bandit* framework, a simplified yet powerful subclass of reinforcement learning that captures the essential trade-off between exploration and exploitation and constitutes one of the central algorithmic foundations of this thesis.

The second section shifts the focus toward the hardware substrate on which AI algorithms are deployed, with particular emphasis on resource-constrained embedded

and edge systems, outlining the main characteristics and limitations of these devices that make the execution of modern AI models particularly challenging. The discussion also introduces the *RISC-V* instruction set architecture, an open, modular, and extensible platform that serves as the hardware testbed for many of the experimental developments in this work.

Subsequently, the third section focuses on the key methodologies for improving computational efficiency in AI, which represent the main thematic axes of this thesis. It first examines *arithmetic-level optimizations*, aimed at accelerating fundamental operations that dominate the computational cost of many learning algorithms. The discussion then moves to *algorithmic optimizations*, which reformulate mathematical models, learning processes, and data representations to reduce computational complexity, energy consumption, and memory footprint, while preserving accuracy and stability. Next, it reviews the principles of *hardware acceleration*, describing how the design of specialized architectures can drastically improve performance-per-watt by exploiting the regularity and parallelism of AI workloads. Finally, it introduces *neuro-inspired paradigms*, including hyperdimensional computing, which draws inspiration from the structure and operation of biological neural systems to achieve extreme energy efficiency through distributed, low-precision, and massively parallel computation.

Together, these perspectives define the multidimensional optimization space in which this thesis is situated. They provide the theoretical and technological background for understanding how algorithmic innovation, architectural design, and neuro-inspired computation can converge toward a new generation of energy-efficient, high-performance AI systems capable of operating not only in powerful data centers but also on resource-constrained embedded and edge devices.

2.1 Artificial Intelligence

Among the many questions that have accompanied scientific thought, one of the most persistent is whether the ability to think can be understood, formalized, and ultimately reproduced in artificial systems. Intelligence, in this sense, is not a single faculty but an ensemble of capabilities: sensing the world, forming internal models, learning from interaction, and using these models to guide action toward desired outcomes. AI studies how to design computational systems that exhibit such behaviors. As noted in historical surveys of the field [41], AI has been characterized through multiple, partially overlapping lenses over the years. Early definitions emphasized the replication of tasks requiring human intelligence [42], while others framed AI as an engineering discipline focused on building intelligent machines and programs [43]. A widely adopted formulation is the agent-centric view of Russell and Norvig, which defines AI as the study of agents that perceive their environment and take actions to achieve goals [8].

These definitions emphasize different aspects, but regardless of which is preferred, any complete AI system must integrate several fundamental functional components: *perception*, *reasoning*, *learning*, and *action*. Perception refers to the ability to sense and interpret the environment (e.g., through computer vision or speech recognition). Reasoning encompasses the use of internal models to infer new information and

make decisions. Learning allows the system to improve its performance over time by adapting parameters or representations based on data and feedback, rather than relying solely on hand-crafted rules. Action denotes the capacity to affect the external world, either through physical actuators (as in robotics) or through virtual outputs such as recommendations, control signals, or decisions. These components form a closed loop: perception provides input, reasoning and learning determine what should be done, and action modifies the environment, which in turn generates new perceptual input. Implementing this continuous closed loop under strict resource constraints is the primary challenge addressed in edge AI and this thesis.

Over the decades, different answers have been proposed to the question of how these capabilities should be realized in concrete models. Some approaches treat intelligence as rule-based symbolic manipulation over explicit structures; others view it as an emergent property of large networks of simple units learning from data. The remainder of this section traces this evolution, from classical symbolic AI to modern connectionist approaches, and sets the stage for the neuro-inspired and hybrid paradigms that play a central role in this thesis.

2.1.1 From Symbolic to Connectionist AI

Historically, the AI field has been characterized by two dominant methodological paradigms: *symbolic* AI and *connectionist* AI.

Symbolic AI. The symbolic (or “classical”) approach, which dominated the early decades of research, assumes that intelligence arises from manipulating symbols (objects, logical expressions, rules, ontologies) [44, 45]. In this paradigm, an intelligent system reasons by applying explicitly defined rules to symbol structures, deriving conclusions in a way analogous to formal logic. Individual symbols can stand for features, relations, physical objects, scenes, or abstract classes, and more complex expressions are constructed by combining simpler ones. Symbolic representations thus have a rich combinatorial structure: a finite alphabet of basic symbols and composition rules can generate an unbounded number of expressions. A vivid example is natural language, where a large number of words is composed from a small alphabet of letters, and in turn a finite vocabulary can be arranged into an effectively infinite number of meaningful sentences [45]. The theoretical foundation of symbolic AI can be traced to the Physical Symbol System Hypothesis proposed by Newell and Simon (1976) [44], which claims: “A physical symbol system has the necessary and sufficient means for general intelligent action”. This implies two things: first, that human cognition itself is a form of symbol manipulation (necessity); and second, that any physical system capable of manipulating symbols appropriately can, in principle, exhibit intelligent behavior (sufficiency). The philosophical roots of this view extend back centuries. Thomas Hobbes argued that reasoning was “nothing more than reckoning” [46], Leibniz envisioned a universal logical calculus to represent and manipulate all human knowledge [47], David Hume attempted to reduce perception to “impressions” and “ideas” as the atomic elements of experience [48], and Immanuel Kant proposed that human experience is structured by a priori forms and concepts that the mind imposes on sensory input [49]. These thinkers anticipated many of the key assumptions of symbolic AI: that knowledge can be represented

in structured form and manipulated through rule-governed operations on symbols. Early AI research in the 1950s–1980s was dominated by symbolic methods, leading to expert systems that encoded human knowledge as explicit if–then rules (e.g., for medical diagnostics or algebraic solving). While Symbolic AI achieved some success in these domains, its limitations soon became apparent. Similarity between symbols is typically all-or-none: two identical symbols are maximally similar, whereas distinct symbols are treated as completely dissimilar, even if they are semantically related [45]. To compare or combine symbolic structures one must follow pointers, traverse trees or graphs, and match elements explicitly, which leads to complex and largely sequential algorithms for search, pattern matching, and reasoning [44]. As the number of symbols and relations grows, these operations quickly become computationally expensive and scale poorly. In practice, symbolic AI required extensive manual knowledge engineering and remained brittle in the face of uncertainty, noise, and the combinatorial explosion of real-world complexity (a tension often summarized as the *symbol grounding problem* [45]). Moreover, conventional symbolic implementations assume highly reliable hardware, since even a small error in a symbolic data structure can corrupt an entire proof or plan—an assumption [50], that sits uneasily with the noisy, unreliable components found in biological brains [51]. By the 1980s, these conceptual and practical challenges contributed to the so-called “AI winter” and motivated the search for alternative, more robust and brain-plausible approaches, paving the way for connectionist AI.

Connectionist AI. The *connectionist* approach draws inspiration from the structure of the brain and seeks to model intelligence using large networks of simple, neuron-like processing units connected by weighted links, i.e., artificial neural networks [52]. Instead of relying on explicitly encoded rules, these systems learn from data by adjusting connection weights, so that intelligent behavior emerges from the collective dynamics of many simple units rather than from hand-crafted symbolic programs. Within this paradigm, different representational schemes have been explored. *Localist* representations assign a single dedicated unit to each concept, much like the so-called “grandmother cells” in the brain [53, 54]. This makes them easy to interpret but closely mirrors the drawbacks of symbolic representations: first, constructing complex computationally structured objects requires introducing new units and connections for every combination; second, distinct concepts are typically represented by disjoint units (neurons), so there is no immediate notion of graded similarity between them [53, 55]. To address these issues, modern neural networks rely on *distributed* representations [52, 56], related to the idea of *holographic* [53, 55–57]. In these representations, information is modeled as distributed over many neurons: each object is encoded as a pattern of activity over many units, and each unit participates in the representation of many objects. Mathematically, the state of a set of neurons (of finite size) is modeled as a vector whose components correspond to neuron activations (each element represents the state of a particular neuron). This supports fully distributed representations of objects at many levels of complexity, from simple features or single items to complex scenes built from many parts [52, 57]. Crucially, in distributed representations, a single component of the vector has no clear meaning on its own; it only makes sense as part of the

full activity pattern. In other words, individual units in a distributed representation do not correspond to specific, easily interpretable concepts [56]. Similar objects give rise to similar activation patterns and can be compared efficiently via vector operations such as dot products or cosine similarity [52, 56]. Distributed codes enjoy several appealing properties: they can store a very large number of distinct concepts in a fixed-dimensional space; they provide a rich semantic basis because features are shared across many concepts; they are robust to noise and hardware unreliability; and they allow a unified processing pipeline in which both simple and composite entities are manipulated as vectors using well-understood linear-algebraic tools [53, 55, 57]. Historically, the connectionist paradigm has deep roots in neuroscience and cognitive psychology. Foundational models include the McCulloch–Pitts neuron (1943) [58], which proposed a binary computational abstraction of a neuron, and Rosenblatt’s perceptron (1958) [59], an early learning system for binary classification. After an initial wave of enthusiasm, the limitations of simple perceptrons—most notably their inability to solve non-linearly separable problems—contributed to a decline of interest in the 1970s [60]. The field was revived in the 1980s with the rediscovery of back-propagation and the rise of parallel distributed processing, which explicitly embraced distributed, subsymbolic representations and learning from experience [61, 62]. At the same time, a number of conceptual challenges were articulated. Critics such as Fodor and Pylyshyn [63] argued that early connectionist models struggled with compositionality, productivity, and systematicity—that is, with representing complex structures built from simpler parts and with explaining why certain patterns of generalization (e.g., understanding a new sentence built from familiar words) come so naturally to humans. Jackendoff [1, 64] later reformulated these concerns as the binding problem (how to link features to the correct objects), the "problem of two" (how to represent multiple instances of the same type), and the problem of variables (how to represent roles in relational structures) [65]. For example, a model should distinguish between "the little star above the big star" and "the big star above the little star," even though both descriptions involve the same words and objects; or, if it understands "the dog chased the cat," it should systematically generalize to "the cat chased the dog" by swapping the roles of the same entities. The superposition of distributed representations further complicates these issues, since encoding multiple items in overlapping patterns can make it difficult to recover individual components, such as when combining patterns for "a–b" and "b–c" yields an activation that no longer cleanly distinguishes which pairings were originally present [66, 67]. Despite these issues, the connectionist approach underwent a decisive resurgence in the 2010s, driven by massive computational resources, large datasets, and advances in architecture design. Deep learning demonstrated that large networks can automatically learn hierarchical distributed representations, powering breakthroughs in perception, language, and decision-making—while still leaving open questions about how to reconcile their strengths with the structured, compositional aspects of human cognition [68, 69].

Historically, researchers often viewed connectionist methods as fundamentally opposed to older symbolic methods, with the two camps debating whether neural networks would supersede symbolic reasoning or vice versa. For example, proponents of connectionism questioned whether high-level symbolic manipulation was necessary to achieve intelligence, while advocates of symbolic AI criticized neural networks

for lacking clear interpretability or logical guarantees [63]. This dichotomy led to periods where one approach dominated and the other receded. Today, however, it is widely accepted that symbolic and connectionist approaches are complementary rather than mutually exclusive. Symbolic AI provides tools for logical reasoning, structured knowledge representation, and interpretability, whereas connectionist AI offers scalable learning from raw data, robustness to noise, and the ability to generalize from examples [53, 55]. Modern AI systems increasingly blend both paradigms—for instance, integrating neural modules for perception with symbolic components for reasoning, planning, or rule-following—giving rise to neuro-symbolic architectures that combine the strengths of both views [57, 65].

Importantly, this thesis builds on this hybrid philosophy by exploring hyperdimensional computing, a brain-inspired paradigm introduced in Section 2.3.4 and itself a fusion of symbolic and connectionist principles. HDC operates on high-dimensional distributed representations that encode information as patterns across many components, akin to neural networks [56, 57]. Yet, it also supports symbolic-like arithmetic, enabling structured reasoning in a connectionist substrate [65]. This dual nature allows HDC to capture structure and flexibility simultaneously, offering a promising path toward efficient, interpretable, and hardware-friendly learning models.

2.1.2 Learning Paradigms

Within the broader field of AI, the subfield devoted to designing systems that can automatically improve their performance through experience is known as *Machine Learning* (ML). As formally defined by Mitchell (1997) [70], “a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .” ML systems adapt by detecting statistical regularities in data and tuning internal parameters through optimization.

Broadly speaking, three major learning paradigms are recognized in ML: *supervised learning*, *unsupervised learning*, and *reinforcement learning*, each defined by a distinct type of feedback and learning objective.

Supervised learning. In supervised learning, the model is provided with a dataset of input–output pairs, where each input $\mathbf{x}_i \in \mathbb{R}^d$ is associated with a corresponding target output or label y_i . The objective is to learn a function $f : \mathbb{R}^d \rightarrow \mathcal{Y}$ that generalizes well to new, unseen data. This paradigm is applicable to both classification problems, where $\mathcal{Y}$ is a finite set of discrete classes (e.g., recognizing handwritten digits), and regression problems, where $\mathcal{Y} \subseteq \mathbb{R}$ represents continuous quantities (e.g., predicting housing prices). Learning proceeds by minimizing a loss function $\mathcal{L}(f(\mathbf{x}_i), y_i)$ over the dataset, typically via gradient-based optimization. Supervised learning powers many of the most successful AI systems in production today, including those used in image recognition, speech transcription, language translation, and medical diagnosis. Its effectiveness, however, depends on the availability of large and well-labeled datasets, which are often expensive or difficult to obtain.

Unsupervised learning. In unsupervised learning, the system is presented only with input data, without any associated labels or outputs. The goal is to infer structure or regularity within the data itself, such as identifying groups of similar samples (clustering), discovering latent variables (dimensionality reduction), or detecting anomalies. Algorithms such as k -means clustering, principal component analysis (PCA), autoencoders, and Gaussian mixture models exemplify this setting. Because it does not rely on labeled data, unsupervised learning is particularly valuable in domains where manual annotation is infeasible or subjective. Furthermore, unsupervised representations often serve as useful initialization or pretraining for downstream supervised tasks.

Reinforcement learning. Reinforcement learning (RL) deals with sequential decision-making in interactive environments. In contrast to supervised learning, where correct input–output pairs are explicitly provided, an RL *agent* must learn from *evaluative* feedback obtained through trial-and-error interaction with its environment [40]. At each time step t , the agent observes a state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$ according to a policy $\pi(a \mid s)$, receives a scalar reward $r_{t+1} \in \mathbb{R}$, and transitions to a new state s_{t+1} . The objective is to find a policy that maximizes the expected *return*, typically defined as the discounted sum of future rewards

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma \in [0, 1]$ is a discount factor trading off immediate and delayed consequences. Problems of this kind are commonly formalized as *Markov decision processes* (MDPs), specified by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{P}$ encodes the transition dynamics and $\mathcal{R}$ the expected one-step reward. In this framework, learning proceeds by improving the policy $\pi(a \mid s)$ —for example, by estimating value functions that predict expected return and updating the policy to favor actions with higher value. A central challenge is the *exploration–exploitation tradeoff*: the agent must explore the environment to discover new rewarding strategies while exploiting its current knowledge to maximize long-term returns. RL has been successfully applied to complex domains such as robotics, resource management, game playing, and recommendation systems.

In this thesis, all three learning paradigms will appear both in algorithmic analysis and in empirical evaluation. Specifically, the hardware architectures developed in this work will be benchmarked across classification, clustering, and sequential decision tasks. However, the specific learning rules adopted for each case are beyond the scope of this conceptual chapter and will be introduced in detail in the corresponding technical sections.

From an algorithmic standpoint, this work concentrates particularly on a subclass of reinforcement learning problems known as *contextual bandits* (CB). CB captures the exploration–exploitation trade-off and is widely used when reinforcement learning is impractical due to the absence of dependence between future contexts and chosen actions. This class of problems will be introduced in the next section.

2.1.3 Contextual Bandit Problems

Contextual Multi-Armed Bandits, or simply Contextual Bandits, are a class of sequential decision-making problems where an agent interacts with the environment to learn and adapt in real-time through a reward-driven process [38]. Given a finite set of N possible actions (or "arms"), at each time step t , the agent interacts with the environment observing a context vector $x_{t,a} \in \mathbb{R}^d$ for each action a , selects an action a_t , and receives a reward (or payoff) r_{t,a_t} , as depicted in Figure 2.1. This reward is then used to update the model parameters, enabling it to adapt and gather information to predict the relationship between the observed context and the reward associated with each action [71]. When the context is shared across actions, we use the notation x_t (i.e., $x_{t,a} = x_t$ for all a).

Differently from RL scenarios, where actions influence subsequent states and the objective is to maximize cumulative rewards over multiple state transitions, in CB problems, the selected action does not affect the next state, and the algorithm aims to maximize the immediate rewards at each iteration [38, 71–73].

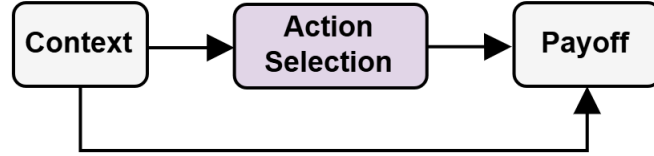


Figure 2.1 Schematic representation of contextual bandits problems. At each iteration, the algorithms observe a context, pick an action and receive a reward related to the chosen option.

Linear Hypothesis A widely studied branch of algorithms for modeling CB problems, which will be considered in this work, assumes a linear relationship between the reward and the context [74, 75]. Under this *linear hypothesis*, the agent maintains ridge-regression statistics to estimate, at each round t , the expected reward of each action a given its context vector $\mathbf{x}_{t,a} \in \mathbb{R}^d$. These algorithms can be broadly categorized by how they model this relationship (the *Disjoint* vs. *Hybrid* formulation) and by the adopted exploration strategy.

Modeling Formulations Mathematically, these algorithms can model actions independently or assume they share common underlying features.

The *Disjoint* formulation represents the simplest approach. Here, each action a is associated with an unknown weight vector $\boldsymbol{\theta}_a \in \mathbb{R}^d$ such that $\mathbb{E}[r_{t,a} \mid \mathbf{x}_{t,a}] = \mathbf{x}_{t,a}^\top \boldsymbol{\theta}_a$, assuming sub-Gaussian noise. The parameter $\boldsymbol{\theta}_a$ is estimated online via ridge regression by maintaining per-action statistics: a covariance matrix $\mathbf{A}_a \in \mathbb{R}^{d \times d}$ and a bias vector $\mathbf{b}_a \in \mathbb{R}^d$. At each time step, the agent observes the context $\mathbf{x}_{t,a}$ and estimates the expected reward $\hat{r}_{t,a}$ for each arm:

$$\hat{\boldsymbol{\theta}}_a = \mathbf{A}_a^{-1} \mathbf{b}_a, \quad \hat{r}_{t,a} = \mathbf{x}_{t,a}^\top \hat{\boldsymbol{\theta}}_a. \quad (2.1)$$

Upon taking the selected action a_t (chosen by the exploration strategy) and observing reward r_{t,a_t} , the statistics for a_t are updated:

$$\mathbf{A}_{a_t} = \mathbf{A}_{a_t} + \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top, \quad \mathbf{b}_{a_t} = \mathbf{b}_{a_t} + r_{t,a_t} \mathbf{x}_{t,a_t}. \quad (2.2)$$

The *Hybrid* formulation is more complex but powerful when actions share similarities. Here, each action a is associated with an action feature vector $\mathbf{u}_a \in \mathbb{R}^f$, and the system utilizes a shared feature vector $\mathbf{z}_{t,a} \in \mathbb{R}^k$ (typically the outer product of context and action features) to model interactions. The model assumes that the expected reward decomposes into two linear components: a *global* effect shared across the entire action space, and a *local* effect specific to the individual arm:

$$\mathbb{E}[r_{t,a} \mid \mathbf{x}_{t,a}] = \underbrace{\mathbf{z}_{t,a}^\top \boldsymbol{\beta}}_{\text{Global Term}} + \underbrace{\mathbf{x}_{t,a}^\top \boldsymbol{\theta}_a}_{\text{Local Term}}. \quad (2.3)$$

where $\boldsymbol{\beta}$ is a shared unknown parameter vector common to all arms, and $\boldsymbol{\theta}_a$ is the unknown parameter vector specific to action a .

At each time step, $\boldsymbol{\theta}_a$ and $\boldsymbol{\beta}$ are estimated via ridge regression by maintaining global shared ridge-regression statistics $\mathbf{A}_0 \in \mathbb{R}^{k \times k}$ and $\mathbf{b}_0 \in \mathbb{R}^k$, as well as action-specific statistics $\mathbf{A}_a \in \mathbb{R}^{d \times d}$, $\mathbf{B}_a \in \mathbb{R}^{d \times k}$, and $\mathbf{b}_a \in \mathbb{R}^d$.

At each time step, the agent estimates the expected reward using:

$$\boldsymbol{\beta} = \mathbf{A}_0^{-1} \mathbf{b}_0, \quad \boldsymbol{\theta}_a = \mathbf{A}_a^{-1} (\mathbf{b}_a - \mathbf{B}_a \boldsymbol{\beta}), \quad \hat{r}_{t,a} = \mathbf{z}_{t,a}^\top \boldsymbol{\beta} + \mathbf{x}_{t,a}^\top \boldsymbol{\theta}_a. \quad (2.4)$$

After selecting an action a_t , both the shared and specific matrices are updated, as per (2.5), (2.6) and (2.7). Equations (2.5) and (2.7) handle the updating of shared matrices, while (2.6) is specifically dedicated to updating the matrices associated with the selected action.

$$\begin{aligned} \mathbf{A}_0 &= \mathbf{A}_0 + \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t} \\ \mathbf{b}_0 &= \mathbf{b}_0 + \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{b}_{a_t} \end{aligned} \quad (2.5)$$

$$\begin{aligned} \mathbf{A}_{a_t} &= \mathbf{A}_{a_t} + \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top \\ \mathbf{B}_{a_t} &= \mathbf{B}_{a_t} + \mathbf{x}_{t,a_t} \mathbf{z}_{t,a_t}^\top \\ \mathbf{b}_{a_t} &= \mathbf{b}_{a_t} + r_{t,a_t} \mathbf{x}_{t,a_t} \end{aligned} \quad (2.6)$$

$$\begin{aligned} \mathbf{A}_0 &= \mathbf{A}_0 + \mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top - \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t} \\ \mathbf{b}_0 &= \mathbf{b}_0 + r_{t,a_t} \mathbf{z}_{t,a_t} - \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{b}_{a_t} \end{aligned} \quad (2.7)$$

The Hybrid algorithm imposes considerably higher memory requirements and computational demands, potentially presenting challenges for its direct deployment in resource-constrained embedded environments. However, including shared matrices in the Hybrid model allows it to generalize information across actions, significantly accelerating convergence in scenarios with correlated arms [74].

Exploration Strategies Regardless of the modeling formulation, an exploration strategy is required to select the action a_t based on the estimated rewards $\hat{r}_{t,a}$. This balances exploring less-known options with exploiting previously gathered information, ensuring the model can adapt and improve over time while avoiding

premature convergence to suboptimal solutions. Two widely used strategies in CB algorithms are the epsilon-greedy and the Upper Confidence Bound (UCB), which, when integrated into linear algorithms, are referred to as LinEPS and LinUCB, respectively [38, 74, 76, 77].

In the epsilon-greedy technique (LinEPS), at each time step, with probability $1 - \epsilon$, the model selects the action with the highest estimated reward ($\arg \max_a \hat{r}_{t,a}$), while in the remaining cases, it selects a random action. This strategy ensures continuous exploration over time but does not account for the model's confidence, potentially leading to inefficient choices.

The UCB exploration strategy [74] (LinUCB), on the other hand, incorporates a confidence interval to each estimate, favoring actions with either high payoff or high uncertainty. This results in a principled trade-off between exploration and exploitation that adapts dynamically over time. For the Disjoint case, the uncertainty term is $s_{t,a} = \mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}$ and the expected reward becomes:

$$p_{t,a} = (\mathbf{A}_a^{-1} \mathbf{b}_a)^\top \mathbf{x}_{t,a} + \alpha \sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}}. \quad (2.8)$$

For the Hybrid case, the uncertainty term accounts for shared correlations:

$$s_{t,a} = \mathbf{z}_{t,a}^\top \mathbf{A}_0^{-1} \mathbf{z}_{t,a} - 2 \mathbf{z}_{t,a}^\top \mathbf{A}_0^{-1} \mathbf{B}_a^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a} + \mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a} + \mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{B}_a \mathbf{A}_0^{-1} \mathbf{B}_a^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a} \quad (2.9)$$

and the expected reward becomes:

$$p_{t,a} = \mathbf{z}_{t,a}^\top \boldsymbol{\beta} + \mathbf{x}_{t,a}^\top \boldsymbol{\theta}_a + \alpha \sqrt{s_{t,a}}. \quad (2.10)$$

Numerous other exploration strategies have been proposed in the literature, including Thompson Sampling, posterior-sampling methods, and neural-network-based approaches [78–81]. However, these are beyond the scope of this work, and the interested reader is referred to [38] for a comprehensive survey.

Notably, unlike RL algorithms such as Q-learning [38, 72], which update state-action values with scalar adjustments, linear CB algorithms heavily rely on computationally intensive matrix operations—particularly the inversion of large matrices, whose complexity scales cubically with the problem size. These requirements create a significant bottleneck for deployment on embedded systems. The matrix updates involve non-trivial data dependencies that are difficult to parallelize [82], resulting in high memory bandwidth consumption and energy overhead. These challenges often conflict with the strict resource constraints of edge devices, directly motivating the architectural optimizations and hardware-aware strategies introduced in this thesis.

Performance Metrics The performance of contextual bandit algorithms is typically evaluated using two key metrics: *average reward* and *cumulative regret*. The average reward reflects the model's ability to consistently select high-reward actions over time. In contrast, cumulative regret quantifies the total difference between the rewards the algorithm actually obtained and the rewards it could have obtained by always selecting the best possible action for each context. Formally, the regret over T rounds is defined as:

$$R_T = \sum_{t=1}^T \left(r_{t,a_t^*} - r_{t,a_t} \right).$$

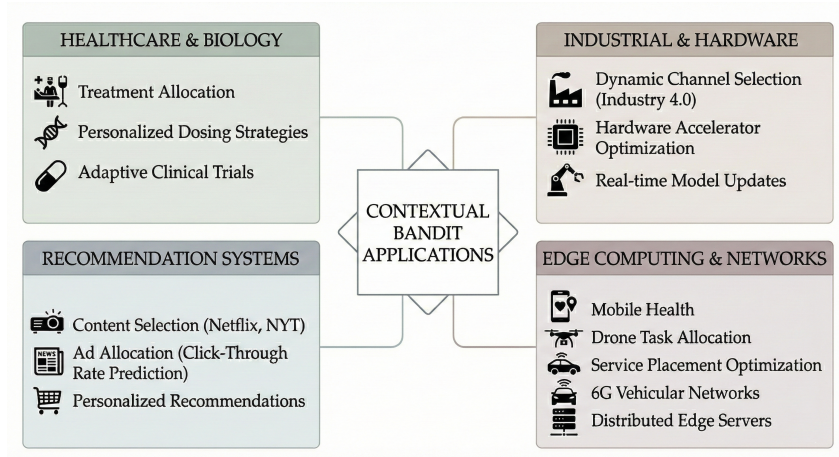


Figure 2.2 Representative application domains of contextual bandit algorithms. Examples span healthcare and biology (e.g., treatment allocation, adaptive clinical trials), recommendation systems (content selection, ad allocation, personalized recommendations), industrial and hardware settings (dynamic channel selection, accelerator optimization, real-time model updates), and edge computing and networks (mobile health, task allocation, service placement, vehicular and distributed edge networks).

where a_t^* is the optimal action in hindsight for context $\mathbf{x}_t$, and a_t is the action selected by the algorithm. A well-designed contextual bandit algorithm aims to minimize this regret over time, converging toward optimal behavior as it gathers more information. Throughout this thesis, these metrics will serve as the primary criteria for assessing the efficiency and learning performance of the proposed algorithms and hardware implementations.

Applications CB algorithms have demonstrated their versatility across a wide range of applications, as summarized in Figure 2.2. In healthcare and biology, these algorithms have been applied to improve treatment allocation in clinical trials [83] and propose personalized dosing strategies [84]. In recommendation systems, they found notable applications in personalizing article selection or tailoring movie recommendations. A canonical example is click-through rate prediction and content selection in web banners: each time a user visits a page, the algorithm observes contextual features (e.g., user demographics, time of day, device type) and selects one among multiple candidate ads to display. The reward is binary—whether the user clicks the ad or not—and only feedback for the chosen ad is observed. Another impactful use case is in *budget-constrained ad allocation*, where CBs select ads to maximize engagement while respecting per-campaign or per-user budget constraints. For instance, in real-time bidding environments, advertisers must decide which ad to show and how much to bid, given budget constraints and uncertain user behavior. Here, the contextual bandit framework enables adaptive exploration while ensuring efficient use of advertising resources [38, 85]. From a practical viewpoint, in Microsoft [86], The New York Times [87] and Yahoo [74] recommendation systems, CB algorithms have been widely utilized to enhance personalized content and news article delivery while Netflix [88] uses them to improve

movie recommendations, demonstrating the ability to learn user preferences over time, enhancing the experience and engagement.

In industrial environments, they have been successfully employed for dynamic channel selection in Industry 4.0 wireless networks [89], as well as for efficient data annotation and real-time model updates in Industrial Cyber-Physical Systems [90]. Furthermore, recent advancements have shown the potential of CB algorithms in enhancing hardware efficiency. The study in [71] employs these algorithms to automatically select the optimal configuration of a general-purpose hardware accelerator according to the workload and reconfigure the architecture at run-time.

Recently, CB algorithms have also emerged in edge computing environments for localized data processing and real-time decision-making. The work in [85] proposes using CB algorithms directly executed on mobile phones or wearable devices for mobile health, personalizing interventions based on user context such as GPS location, calendar busyness, and heart rate. This approach ensures interventions can occur even in network issues or failures, providing robust and timely support. It also addresses privacy concerns by keeping sensitive health data localized on the user's device. In [39], authors employ CB algorithms on resource-constrained nanodrones to dynamically allocate inference tasks between edge and cloud, based on predicted network delays. This approach demonstrates improved adaptability in task allocation, enabling efficient computation offloading and optimized edge computing in fluctuating network conditions. The work in [91] proposes a CB approach to optimize the placement of services on edge nodes, enhancing response times and service quality for users. Their algorithm runs directly on edge devices and dynamically rents computing resources based on spatial and temporal demand patterns, adapting to changes in user needs across different times and places. [92] proposes a CB algorithm to split tasks across multiple edge devices, selecting the most reliable ones in real-time. This ensures stable, deadline-driven performance even in fluctuating network conditions, making it ideal for large-scale, distributed edge applications. In the context of emerging networks, authors in [93] utilize CB to optimize intelligent and secure radio environments in 6G vehicular-aided heterogeneous networks, demonstrating their potential to enhance the efficiency and security of these advanced networks. In the domain of assistive robotics, human-in-the-loop CB-based approaches have been proposed to personalize robot-assisted feeding, achieving the trade-off between acquisition success and the querying workload [94].

The diverse challenges and application domains of CB algorithms highlight the critical need to develop new approaches that enhance accuracy, reduce execution and convergence times, minimize computational complexity, and lower energy consumption [82, 85].

In this thesis, contextual bandits play a central role as a theoretical and experimental testbed for studying the trade-off between algorithmic accuracy, computational cost, and energy efficiency. They provide a controlled yet expressive framework to evaluate how algorithmic reformulations, arithmetic optimizations, and hardware acceleration can jointly enable intelligent decision-making under strict resource constraints.

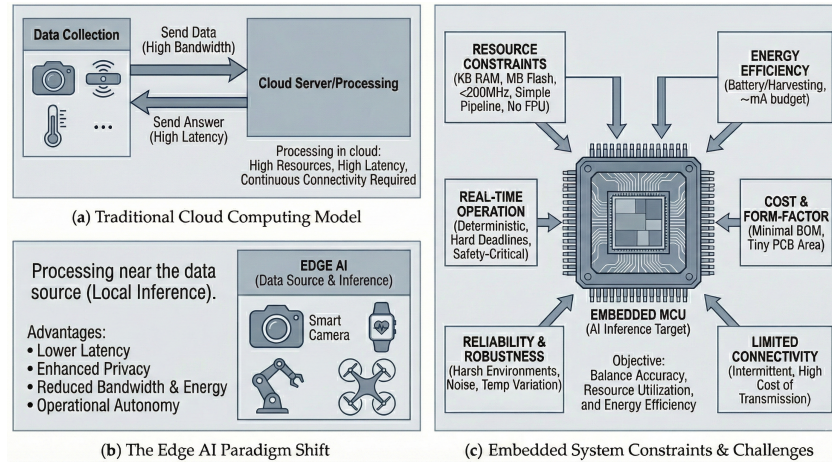


Figure 2.3 From cloud-centric AI to edge intelligence and embedded constraints. (a) Traditional cloud computing model, where sensor data are streamed to a remote server for processing, incurring high bandwidth usage and latency and requiring continuous connectivity. (b) Edge-AI paradigm shift, in which inference is performed near the data source, reducing latency and bandwidth, improving privacy, and enabling more autonomous operation. (c) Key constraints and challenges of embedded microcontroller-class platforms targeted for on-device AI, including tight memory and compute budgets, stringent energy-efficiency requirements, real-time operation, reliability and robustness, cost and form-factor limitations, and intermittent or expensive connectivity.

2.2 Embedded AI Systems and Hardware Devices

The previous sections introduced the basic concepts of AI and the main learning paradigms. Yet, abstract algorithms ultimately run on concrete hardware substrates. In the context of edge AI considered in this thesis—characterized by resource-constrained, real-time, and low-power scenarios—the properties of the hardware backend are as critical as the choice of the learning algorithm itself.

In these environments, the relevant design question is no longer just "What is the most performant model?", but rather: *"What is the best compromise between accuracy, resource utilization, and energy efficiency under tight constraints?"* Understanding where and how AI workloads are executed is therefore essential to designing practical, efficient solutions.

An *embedded system* is defined as a specialized computing platform designed to perform a dedicated function, often as part of a larger electrical or mechanical system [95]. This thesis specifically targets the class of *resource-scarce* edge devices, which Branco et al. [4] characterize as operating below roughly 200 MHz, with less than 3 MB of on-chip Flash, under 500 KB of (S)RAM, and average current consumption below 20 mA. As illustrated in Table 2.1, these specifications reflect the vast majority of microcontrollers deployed as IoT end-nodes, including wearable medical devices, wireless sensor nodes, and low-cost industrial controllers.

Deploying AI on such constrained substrates presents a set of specific challenges, summarized in Figure 2.3:

- **Resource constraints:** Unlike cloud servers, these platforms typically rely on low-power microcontrollers with very simple microarchitectures: in-order

pipelines, narrow datapaths, and frequently *no* hardware floating-point unit (FPU) or vector extensions, forcing ML workloads to rely on fixed-point arithmetic or software-emulated floating point. These tight bounds fundamentally restrict the size, precision, and architectural complexity of AI models that can be executed locally.

- **Energy efficiency:** With current budgets often limited to a few milliamps (or relying on harvesting), energy efficiency is the primary design objective. Since data movement often dominates the energy cost of computation, this strongly favors computation near the sensor, compact models, and memory access patterns that minimize both on-chip traffic and wireless transmissions.
- **Real-time operation:** Embedded systems often close control loops with the physical environment (e.g., automotive, robotics, medical monitoring), requiring fast and responsive algorithms.
- **Cost and form-factor constraints:** Large-scale deployments impose stringent limits on bill-of-materials cost and packaging area. This pushes designs toward simple microcontrollers, minimal external memory, and limited I/O, often precluding the use of high-end processors or large off-chip DRAM and further constraining the feasible AI workload.
- **Reliability and robustness:** Deployed in harsh or uncontrolled environments (such as industrial plants, vehicles, outdoor infrastructure, or on-body/in-body medical devices) embedded systems must guarantee long-term stability and tolerance to noise, temperature variations, interference, and partial failures. AI components must operate reliably under such conditions, with limited possibilities for frequent retraining, recalibration, or manual intervention in the field.
- **Limited connectivity and bandwidth:** Many edge nodes experience intermittent, low-bandwidth, or energy-expensive connectivity. In scenarios such as remote sensing, agricultural monitoring, or battery-powered and duty-cycled devices, continuous offloading to the cloud is infeasible. This reinforces the need for local, stand-alone inference capable of operating correctly even when back-end connectivity is unavailable, and encourages designs in which models can be autonomous and adaptable to changing environments.

By 2026, edge inference is projected to account for approximately 70% of all AI inference deployments [96], with over 50% of new enterprise AI deployments in the United States already involving edge processing as of 2025 [33]. In this landscape, moving computation closer to the data source is not merely an optimization but a necessity. This strongly motivates the *algorithm-hardware co-design* approach adopted in this thesis, which explicitly embraces device-level constraints rather than treating the hardware as an afterthought.

2.2.1 The Role of FPGAs in Embedded AI

Field-Programmable Gate Arrays (FPGAs) are increasingly emerging as a key enabler for flexible and energy-efficient intelligent embedded computing.

Table 2.1 Microcontroller units (MCUs) comparison, ordered by clock frequency [4].

Name	Clock Frequency	Flash	(S)RAM	Power
Arduino Uno (ATMega128P)	16 MHz	32 KB	2 KB	12 mA [97]
Arduino Mega (ATMega2560)	16 MHz	256 KB	8 KB	6 mA [98]
Arduino Nano (ATMega2560)	16 MHz	26–32 KB	1–2 KB	6 mA [98]
STM32L0 (Cortex-M0)	32 MHz	192 KB	20 KB	7 mA [99]
Arduino MKR1000 (Cortex-M0)	48 MHz	256 KB	32 KB	4 mA [100]
Arduino Due (Cortex-M3)	84 MHz	512 KB	96 KB	50 mA [101]
STM32F2 (Cortex-M3)	120 MHz	1 MB	128 KB	21 mA [102]
STM32F4 (Cortex-M4)	180 MHz	2 MB	384 KB	50 mA [103]
Raspberry Pi A+	700 MHz	SD Card	256 MB	80 mA [104]
Raspberry Pi Zero	1 GHz	SD Card	512 MB	80 mA [104]
Raspberry Pi 3B	1.2 GHz	SD Card	1 GB	260 mA [104]

An FPGA is a reconfigurable integrated circuit composed of an array of programmable logic blocks (implementing Boolean operations via Lookup Tables (LUTs) and Flip-Flops) interconnected by a configurable routing fabric. Modern devices augment this fabric with heterogeneous hard macros, such as Digital Signal Processors (DSP) and block RAM (BRAM). This architecture allows developers to tailor the hardware datapath precisely to the computational characteristics of AI workloads by describing the hardware logic using Hardware Description Languages (HDLs), which is then synthesized and deployed as a bitstream to configure the FPGA’s underlying fabric. Unlike CPUs or GPUs, which rely on fixed instruction sets and standard data types, FPGAs enable bit-level customization, achieving high performance and energy efficiency [105, 106].

This combination of flexibility and fine-grained parallelism makes FPGAs particularly attractive for *edge AI*, where real-time responsiveness and strict power budgets must coexist. They support massive concurrency for mapping computational kernels (e.g., matrix multiplications, distance metrics) while leveraging distributed on-chip memory to minimize energy-expensive off-chip data movement [4, 105].

FPGA-based implementations can achieve $10\times$ – $100\times$ improvements in throughput and energy efficiency compared to microcontroller-based systems, enabling real-time execution of tasks ranging from computer vision to radar-based activity recognition [4]. Despite these advantages, FPGAs face limitations, including lower clock frequencies than standard processors and a complex development cycle requiring expertise in hardware description languages and timing closure [105].

In the context of this thesis, FPGAs serve a dual role: first, as a flexible validation platform where algorithmic and architectural choices can be iteratively optimized before ASIC commitment; and second, as a target architecture for the custom accelerators and hybrid systems proposed in the following chapters.

2.2.2 The RISC-V Instruction Set Architecture

The growing demand for efficient and application-tailored embedded intelligence has accelerated the adoption of open and extensible microprocessor architectures. Among these, the *RISC-V* instruction set architecture (ISA) has emerged as a natural foundation for domain-specific system-on-chip designs in edge and IoT scenarios.

Originating at the University of California, Berkeley, in 2010, RISC-V follows the Reduced Instruction Set Computing (RISC) philosophy but is modular, extensible, and open by design [107]. Technically, it consists of a small base integer set (e.g., RV32I for 32-bit embedded cores, or RV64I for 64-bit systems) that can be combined with standardized optional extensions for integer multiplication and division (M), atomic instructions (A), compressed encoding (C), single- and double-precision floating point (F/D), vector operations (V), and others. This modularity allows implementers to tailor cores to specific application domains, balancing cost, performance, and energy efficiency.

Crucially, unlike proprietary ISAs (e.g., ARM), which typically require licensing agreements and vendor-specific support for instruction customization, RISC-V is specified as an open standard under permissive licenses, allowing anyone to implement, modify, or extend the architecture without royalties. The ISA explicitly reserves opcode space for non-standard, user-defined *Instruction Set Extensions* (ISEs), enabling designers to introduce custom instructions and tightly coupled accelerators while remaining compliant with the base architecture.

This mechanism enables low-latency coupling between the core and domain-specific accelerators, as will be discussed in Section 12, resulting as a perfect candidate for edge AI and specialized embedded workloads. User-defined instructions can act as specialized arithmetic or logical operations that control accelerators, directly accessing registers and control signals and thereby reducing instruction overhead and memory traffic compared to external coprocessor or memory-mapped interfaces [107–109]. Several works have demonstrated RISC-V-based accelerators and ISEs for neural networks and machine learning, achieving significant reductions in latency and energy consumption with relatively small architectural modifications [108–111].

This openness has had a profound impact on both academia and industry. By eliminating licensing barriers, RISC-V has become a catalyst for innovation in custom processors, hardware–software co-design, and domain-specific computing, fundamentally changing how embedded and AI-oriented processors are developed. Instead of designing monolithic, fixed-function chips, engineers can adopt a modular and iterative approach: starting from a simple baseline RISC-V core and progressively enhancing it with custom instructions or accelerators optimized for specific workloads (e.g., convolutions, vector similarity, cryptography, etc.). Because RISC-V cores are open and standardized, these extensions can be shared, reused, and validated across different implementations. As a result, a vibrant ecosystem of open-source cores, tools, and development frameworks has emerged, making RISC-V one of the fastest-growing computing standards in history [112].

The combination of open hardware and customizable instructions provides a fertile ground for research in energy-efficient and domain-specific architectures, which is central to this thesis. In the following, we examine one such architecture: the *Klessydra* family of RISC-V processors, designed at Sapienza University of Rome, which embodies these principles through a lightweight, multi-threaded microarchitecture optimized for embedded and real-time systems.

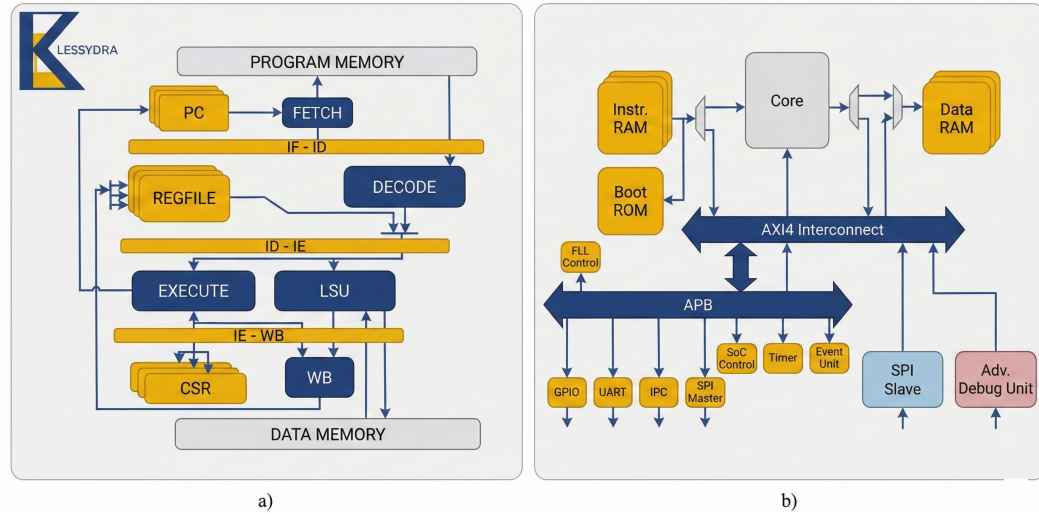


Figure 2.4 Architectural schematics: (a) Internal microarchitecture of the Klessydra-T03 core; (b) The PULPino SoC platform. Klessydra is designed to be pin- and bus-compatible with PULPino, serving as a drop-in replacement for the default core.

2.2.3 Klessydra Core Family

The *Klessydra* family constitutes a suite of open-source, RISC-V-compliant processing cores tailored for low-power IoT environments [113, 114]. Implemented as *softcores*—i.e., synthesizable RTL descriptions mapped to FPGAs. This extensibility makes Klessydra an ideal platform for rapid prototyping, hardware–software co-design, and the tight integration of custom accelerators.

Architecturally, Klessydra cores are fully compatible with the RISC-V RV32I base integer ISA, and selected configurations also support standard extensions such as integer multiplication/division (M) and atomic primitives (A) for synchronization [113]. They are designed to be pin- and bus-compatible with the PULPino [115] open-source SoC platform, enabling straightforward integration with existing open-source peripherals, memory subsystems, and toolchains.

The design philosophy of Klessydra is intentionally lightweight: the cores avoid complex features such as deep pipelines, branch prediction, speculative execution, or large cache hierarchies, favoring instead deterministic behavior, low area, and low power consumption. This aligns with the constraints outlined in the previous sections, where predictable latency, robustness, and tight energy budgets are prioritized over peak single-thread performance.

The *Klessydra-T03* is the baseline member of the family: a parametric, interleaved multi-threaded (IMT) core specifically targeting computation- and control-intensive embedded workloads. Its microarchitecture is organized around a simple four-stage in-order pipeline that fetches and executes instructions from multiple hardware threads in a round-robin fashion. By interleaving independent threads cycle-by-cycle, the core effectively hides control and memory latencies without resorting to speculation or out-of-order execution. With three hardware threads scheduled in an interleaved manner, the pipeline can operate stall-free for many workloads, achieving high utilization with minimal control overhead [113]. To guarantee correctness,

each hardware thread is associated with its own register file and control and status registers (CSRs).

A defining feature of the Klessydra architecture is its suitability for *tight coupling* with domain-specific accelerators. Leveraging the extensibility of the RISC-V ISA and the structured microarchitecture of the T03, custom ISEs and co-processors can be integrated directly into the execution pipeline. This enables the low-latency offloading of compute-intensive kernels while preserving a compact control core. Previous studies have leveraged this capability to implement vector accelerators for convolutional filters [114] and fault-tolerant variants for safety-critical applications [116].

In the context of this thesis, Klessydra-T03 (Figure 2.4) is adopted as the reference core for implementing and evaluating custom hardware accelerators. However, the proposed architectures and co-design methodologies are not limited to this specific instance; they are intrinsically portable to any RISC-V-compliant processor.

2.3 Optimization Strategies

As the previous sections outlined, the increasing computational demands of modern AI algorithms are leading to an *AI efficiency crisis*, in which performance requirements are colliding with the physical and economic limits of current hardware. This challenge spans the entire computing spectrum: from massive foundation models that consume megawatts of power in centralized data centers, to lightweight classifiers deployed on embedded and battery-powered devices where every operation counts.

Addressing this growing disparity requires a comprehensive, cross-layer optimization strategy that spans the entire computing stack. From the low-level arithmetic operations at the heart of learning algorithms to algorithmic simplifications, specialized hardware architectures, and even paradigm-level reformulations inspired by biological computation, each layer offers a distinct set of levers for improving execution time and reducing computational complexity, energy consumption, and memory overhead. Crucially, these levers are not independent: for instance, algorithmic quantization schemes constrain the required numerical precision, which in turn shapes the design of arithmetic units and custom accelerators, while hardware capabilities feed back into which algorithmic approximations are practical in deployed systems.

This section introduces the four main axes of optimization that underpin the methodology of this thesis. Section 2.3.1 explores *arithmetic-level optimizations*, focusing on how approximating or accelerating core operations such as division can significantly reduce computational bottlenecks without sacrificing accuracy. Section 2.3.2 presents *algorithmic optimizations*, in which learning rules and model structures are reformulated to cut down the number of required operations, reduce precision, lower memory accesses, and improve numerical stability under resource constraints. Section 2.3.3 discusses *hardware acceleration*, highlighting how domain-specific architectures, instruction-set extensions, and co-processors can be co-designed with algorithms to exploit parallelism and data reuse for faster execution time and lower energy consumption. Finally, Section 2.3.4 introduces *neuro-inspired paradigms*, with a focus on hyperdimensional computing, which takes inspiration from the

robustness, parallelism, and efficiency of the brain to design learning systems that are both efficient and scalable.

These optimization directions are not mutually exclusive; rather, their synergy offers a path toward scalable, sustainable AI, from cloud to edge.

2.3.1 Arithmetic-level Optimizations

At the foundation of every AI algorithm lies numerical arithmetic. From classical statistical models to modern deep neural networks, large language models, and emerging neuro-inspired approaches, AI workloads ultimately reduce to long sequences of additions, multiplications, and divisions. While high-level algorithmic design determines *which* operations are required, the efficiency of these low-level primitives has a profound impact on overall execution time, silicon area, and energy consumption, especially in resource-constrained settings. In this subsection, we therefore focus on how the choice of *datatypes and precision*, the *type of operations*, and the *hardware backbone* on which they run jointly determine the practical cost of computation.

Datatypes and precision. The first ingredient is how numerical quantities are represented in memory and along the datapath. Several families of data types are commonly used in AI workloads, each inducing a different trade-off between dynamic range, precision, hardware complexity, and energy consumption. We adopt the conventional notation where integer formats are denoted as, e.g., INT32 or INT8 to indicate signed integers with 32 or 8 bits, respectively, and floating-point formats are denoted as FP32, FP16, and so on, to indicate IEEE 754 single precision (32 bits) and half precision (16 bits).

The simplest representation is provided by signed integer types, which encode whole numbers on a uniform grid: each bit pattern corresponds to an exact integer value, and there is no notion of fractional part. We denote INTFP_{bit} as the number of bits used to represent these integer values (e.g., INT8, INT32). Integer arithmetic units are simple and fast, and integer addition and multiplication are typically among the least expensive arithmetic operations in a processor’s datapath. However, pure integer precision is insufficient for many AI models, where parameters such as neural network weights, activations, or reinforcement learning value estimates must represent real-valued quantities and evolve through small incremental updates.

Floating-point formats defined by the IEEE 754 standard address this limitation by introducing a normalized mantissa and an exponent. A real number x is encoded as

$$x = (-1)^s \times m \times 2^e,$$

where s is the sign bit, m is a normalized mantissa in a fixed range, and e is a biased exponent. The bitfields (s, e, m) are laid out contiguously in the word. We denote FP_{bit} as the total number of bits comprising the floating-point format (e.g., FP32, FP16). This representation provides a very large dynamic range, since scaling by powers of two is absorbed by the exponent, and it offers roughly uniform relative precision across several orders of magnitude. At the same time, supporting normalization, exponent alignment, rounding, and special values (NaNs,

$\pm\infty$, subnormals) requires complex FPUs with long datapaths and nontrivial control logic. These units occupy significant silicon area and incur higher latency and energy per operation compared to their integer counterparts.

Fixed-point formats emerge as a middle ground: they allow the representation of fractional quantities while retaining much of the simplicity of integer formats. Fixed-point representations store real numbers as signed integers with an implicit binary point. A fixed-point number in $Qm.n$ format uses one bit for the sign, m integer bits, and n fractional bits. Internally, the hardware simply stores a two's-complement integer I ; the associated real value is obtained by placing the binary point after the n least significant bits, i.e.

$$x = I \cdot 2^{-n}.$$

For example, in a 16-bit $Q7.8$ format ($n = 8$), the integer $I = 256$ encodes the real value $x = 256 \cdot 2^{-8} = 1.0$, while $I = 384$ encodes $x = 1.5$. Fixed-point arithmetic can be implemented using standard integer adders and multipliers, plus simple shifts to adjust the scale, without any need for normalization or exponent handling. This yields units that are smaller, faster, and more energy-efficient than FPUs, at the cost of a limited dynamic range and the need to manage scaling carefully. Plain signed integers correspond to the special case $n = 0$, where no fractional bits are available.

As will be discussed in detail in the next subsection on algorithmic optimizations, in many cases the real-valued elements of AI models can be quantized into low-precision integer formats, such as INT8 or INT4, with negligible impact on accuracy. These quantized formats can be viewed as a generalization of fixed point tailored to learned models. At the extreme end of this spectrum, binary $\{0, 1\}$, bipolar $\{-1, +1\}$, and ternary $\{-1, 0, +1\}$ representations can be employed, drastically simplifying arithmetic operations and reducing memory footprint. In such regimes, expensive operations can be replaced by bitwise logic, sign flips, and small lookup tables, improving hardware efficiency.

The choice of datatype therefore maps directly to distinct hardware structures with different cost profiles. Table 2.2 substantiates this point by reporting latency, maximum operating frequency ($f_{\max}$), and hardware resource usage on FPGA for three fundamental arithmetic operations in representative INT32, 32-bit fixed-point, and FP32 formats. The INT32 results confirm the expected efficiency of integer arithmetic: the adder achieves single-cycle latency with minimal logic cost (32 LUTs) and an $f_{\max}$ of 810 MHz, while the multiplier also completes in one cycle at 800 MHz by exploiting dedicated DSP slices in modern FPGAs. The 32-bit fixed-point results closely track the INT32 baseline for addition and multiplication, consistent with the fact that fixed-point arithmetic largely reuses the same underlying integer datapath. The slight increase in LUT size reported for fixed-point multiplication reflects minor scale-management logic rather than a fundamentally different multiplier structure. By contrast, FP32 arithmetic is intrinsically more complex. In Table 2.2, the FP32 data refer to the *maximum-latency* configurations available in the Xilinx/AMD Floating-Point IP [117]; lower latencies can be obtained for these operations, but at the cost of reduced operating frequency and increased hardware requirements. Under these settings, the FP32 adder requires 12 clock cycles to achieve an $f_{\max}$ of 620 MHz and uses 198 LUTs, reflecting the additional control and datapath structures needed

Table 2.2 Latency (clock cycles), maximum operating frequency $f_{\max}$ (MHz), and FPGA resource usage for addition, multiplication, and division in INT32, 32-bit fixed-point, and FP32 on the Xilinx Zynq UltraScale+ ZCU106 (EK-U1-ZCU106-G) device. INT32 and 32-bit fixed-point values are from synthesis results. For FP32, the latency values (ADD=12, MUL=9, DIV=29) refer to the *maximum-latency* configurations available in the Xilinx/AMD Floating-Point IP [117]; lower latencies can be achieved for these operations at the cost of reduced operating frequency and increased hardware requirements.

Operation	INT32				Fixed-Point 32				Float32 (IEEE 754)			
	Lat	$f_{\max}$ (MHz)	LUT	DSP	Lat	$f_{\max}$ (MHz)	LUT	DSP	Lat	$f_{\max}$ (MHz)	LUT	DSP
ADD	1	810	32	0	1	800	32	0	12	620	198	2
MUL	1	800	47	4	1	780–800	50–60	4	9	635	111	2
DIV	32	600	136	0	32–64	550–600	136–259	0	29	524	187	0

for exponent alignment, normalization, rounding, and special-case handling. A similar trend is observed for multiplication: the FP32 multiplier requires 9 cycles and more than doubles the LUT usage compared to the integer implementation (111 vs. 47), due to separate management of sign, exponent, and mantissa components.

The energy footprint of arithmetic operations varies accordingly: a FP32 addition consumes roughly an order of magnitude more energy than a fixed-point addition, and a floating-point multiplication or division may cost $20\text{--}100\times$ more [118]. Moving from 32-bit floating point to 16-bit floating point or 8-bit fixed-point arithmetic, thanks to algorithmic optimizations, often yields per-operation energy savings of $5\text{--}20\times$ [119, 120].

Operation types and their latency. Beyond the datatype, the nature of the operations themselves dictates their hardware cost. Adders and multipliers admit highly parallel implementations that decompose computation into concurrent sub-operations, enabling short critical paths and high clock frequencies. *Division*, in contrast, is fundamentally sequential. Lacking the associativity and commutativity that facilitate parallelization [121, 122], hardware dividers typically generate the quotient digit-by-digit via iterative refinement. This inherent data dependency prevents fine-grained parallelism, making division significantly slower and leads either to expensive hardware implementations or to severe performance bottlenecks [123, 124].

This structural asymmetry is evident in Table 2.2: while INT32 addition and multiplication complete in a single cycle, division requires tens of cycles. Consequently, the cost of an algorithm is not determined merely by the *count* of operations, but by the *mix*: an implementation rich in divisions will be disproportionately slower and more energy-consuming. The same reasoning extends to other cost asymmetries: for FP32, addition is cheaper than multiplication, which in turn is cheaper than division.

From general-purpose units to application-aware arithmetic. The discussion above highlights that both datatypes and operation types crucially affect the concrete cost of arithmetic. It is therefore important to clearly understand and analyze the *hardware backbone* on which the algorithms are executed.

High-end CPUs and GPUs typically feature dedicated FPUs and vector extensions

that mask the complexity of floating-point arithmetic operations, together with large off-chip DRAM to store model parameters and activations. At the other end of the spectrum, many microcontrollers and embedded cores only provide an integer ALU. On such platforms, floating-point operations must be emulated in software, and division in particular may require hundreds or thousands of cycles, turning seemingly innocuous normalization steps into major latency and energy bottlenecks. In addition, on-chip memories are typically measured in tens or hundreds of kilobytes, and access to external storage—if available at all—dominates the energy budget. Under these constraints, both the number of arithmetic operations and the volume and precision of memory accesses become critical design factors.

An implementation that relies heavily on FP32 division and large tensors may be acceptable on a server-class GPU with ample DRAM bandwidth, but completely impractical on a microcontroller without an FPU and with only a few hundred kilobytes of SRAM. Conversely, an algorithm reformulated to use fixed-point INT16 arithmetic, quantized weights and activations, and a small number of divisions might fit comfortably within the real-time and power constraints of an embedded system and its limited on-chip memory.

Therefore, for embedded platforms, arithmetic optimization is not merely a refinement but an *enabler* of real-time execution.

A powerful strategy to overcome these problems, when designing domain-specific accelerators, as will be discussed in Section 2.3.3, is to abandon rigid general-purpose constraints in favor of *application-aware arithmetic* [118, 120]. This opens two main avenues for optimization:

1. *Approximate Computing*: This approach deliberately trades precision for efficiency. Since many AI tasks are inherently error-resilient, complex circuits can be replaced with simplified logic that truncates carry chains or uses lookup-based approximations [125]. For instance, input-aware multipliers [126] and dividers [127] can achieve order-of-magnitude energy savings with negligible accuracy loss by adapting to the statistical distribution of ML workloads.

2. *Variable-Latency Arithmetic*: A complementary approach—central to this thesis—is to abandon the worst-case timing model. Instead of fixing latency to the slowest possible input case, *variable-latency units* allow the operation time to depend on the operands, terminating early when sufficient precision is reached, or when some condition is satisfied. Variable latency arithmetic units have been studied for decades, covering addition [128–130], multiplication [131], division [122], and more complex operations [132]. Such adaptive arithmetic units significantly reduce average latency and energy with negligible hardware overhead and are particularly attractive for AI inference and embedded workloads, where operand statistics—such as narrow dynamic ranges or repetitive patterns—enable frequent early completion.

Driven by these observations, this thesis frequently adopts reduced numerical formats and designs dedicated arithmetic units tailored to the target workload and precision requirements. Furthermore, we propose an innovative variable-latency division algorithm, and detail its hardware implementation across four architectural variants, demonstrating how operand-dependent execution can substantially reduce latency and energy without compromising correctness.

2.3.2 Algorithmic Optimizations

While arithmetic-level techniques optimize the execution of individual operations, *algorithmic optimizations* operate at a higher level, reshaping the computational structure of a model to reduce the number, precision, and complexity of operations. The goal is to design models and learning rules that are inherently lighter, requiring fewer parameters, arithmetic operations, memory traffic, and energy, while preserving accuracy, robustness, and stability.

Algorithmic choices matter across the entire AI stack, from large-scale training to embedded inference; however, their impact is most pronounced in resource-constrained settings, where modest reductions in computation or data movement can translate into substantial energy and latency gains [118, 133]. To reason about these improvements, we first introduce the notion of computational complexity that will be used throughout the thesis.

Number of operations and asymptotic complexity. The simplest and most abstract view of computational cost ignores hardware details and simply counts the number of primitive operations an algorithm performs as a function of the input size. Let $T(N)$ denote the number of elementary arithmetic operations required to process an input of size N . In asymptotic analysis, we focus on how $T(N)$ scales as N grows, rather than on its exact value.

We will adopt the standard *Big-O notation* throughout this thesis. Formally, we say that $T(N)$ is $O(f(N))$ if there exist constants $c > 0$ and N_0 such that

$$T(N) \leq c f(N) \quad \text{for all } N \geq N_0.$$

Intuitively, this means that, for sufficiently large inputs, $T(N)$ grows no faster than a constant multiple of $f(N)$. Different asymptotic classes correspond to qualitatively different scaling behaviors. For example, a linear-time algorithm with $T(N) = O(N)$ remains affordable as N increases, while a cubic algorithm with $T(N) = O(N^3)$ quickly becomes prohibitive.

For instance, computing the dot product between two D -dimensional vectors requires D multiplications and $D - 1$ additions, so its cost is $O(D)$. A dense matrix-matrix multiplication between two $N \times N$ matrices involves N^2 dot products of length N , leading to a cubic complexity $O(N^3)$. By contrast, searching for an element in a *sorted* array using binary search halves the search interval at each step and therefore needs only $O(\log N)$ comparisons. These asymptotic classes are crucial to understand which algorithms remain tractable as the problem size grows. However, Big- O notation does not distinguish between different datatypes or different operation types: an addition, a multiplication, and a division all count as one “operation”. In practice, as discussed in Section 2.3.1, these hidden constants depend strongly on the arithmetic precision and on the mix of operations involved; algorithmic optimizations aim first to reduce $T(N)$, and then to make its constituent operations as cheap as possible on the target hardware.

Families of algorithmic techniques. Algorithmic optimizations generally fall into two categories. The first family comprises *compression-oriented methods*, which

preserve the overall functional form of a model while making its representation more compact, reducing the number of parameters, arithmetic operations, and memory transfers. The second one comprises *reformulation methods*, which change *what* is computed by rewriting training or inference procedures to lower asymptotic complexity. In what follows, we briefly review representative techniques from both families and clarify how they interact with the hardware considerations introduced in Section 2.3.1.

- *Model Compression Techniques.* The literature offers a rich taxonomy of compression methods. *Pruning* [134–136] exploits the over-parameterization of modern networks by removing redundant weights or entire structures (unstructured vs. structured pruning). *Low-rank factorization* approximates dense tensors as products of lower-dimensional factors, reducing both parameters and arithmetic intensity [137, 138]. *Knowledge distillation* transfers the behavior of a large *teacher* model to a compact *student*, often improving accuracy at a fixed model budget [139]. Within this broader landscape, in this thesis, we focus on *quantization*. Unlike previous methods that alter network topology, quantization reduces the numerical precision of the parameters (e.g., from FP32 to INT8/INT4), to decrease model size and memory bandwidth and to enable cheaper arithmetic [119, 140]. Crucially, quantization is tightly coupled to hardware co-design: once bitwidths and numerical formats are fixed, they directly constrain datapath widths and guide the implementation of arithmetic units. Accordingly, the quantization strategies introduced here provide the explicit link between algorithmic layer and arithmetic- and architecture-level optimizations developed in Sections 2.3.1 and 2.3.3.
- *Algorithmic Reformulation.* Beyond compression, *reformulation* methods reduce cost by rewriting *how* a target computation is carried out, often changing the operation mix and/or improving asymptotic scaling while preserving (or closely approximating) the original solution. In deep learning, representative examples include efficient architectures such as MobileNet and Xception [141, 142] which replace standard convolutions with depthwise separable ones, as well as linearized Transformer attention mechanisms that mitigate the quadratic cost of standard self-attention [143]. Classical algebraic reformulations are also widely used for convolutions: FFT-based approaches transform computations into the frequency domain to speed up large kernels [144, 145], while Winograd algorithms reduce the number of multiplications for small kernels at the cost of cheaper additions [146].

In the context of this thesis, we apply this reformulation principle to enable efficient *online learning*. Many adaptive estimation and learning procedures require inverting a covariance matrix at every time step. This operation typically entails cubic computational complexity, making it prohibitively expensive for real-time edge execution [147, 148]. By exploiting rank-one update identities such as the Sherman-Morrison-Woodbury formula [149, 150], the inverse can be updated incrementally using rank-one corrections. This replaces explicit inversion with matrix–vector operations that are substantially more amenable to vectorization and low-power hardware implementation.

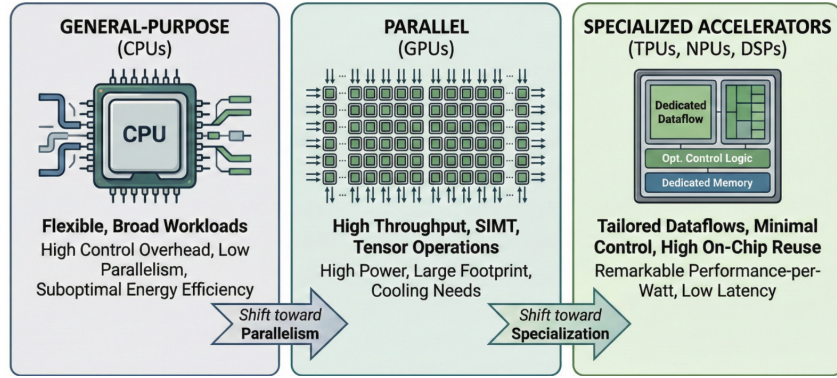


Figure 2.5 Conceptual landscape of hardware platforms. General-purpose CPUs offer maximum flexibility but suffer from high control overhead, limited parallelism, and suboptimal energy efficiency. GPUs increase throughput via massively parallel SIMT execution, at the cost of high power consumption, large silicon area, and substantial cooling needs. Specialized accelerators (e.g., TPUs, NPUs, DSPs) push further toward tailored dataflows, minimal control, and high on-chip data reuse, achieving markedly higher performance-per-watt and lower latency for targeted workloads.

These examples illustrate how changing *what* is computed—by exploiting algebraic identities, problem structure, or approximate representations—can deliver substantial reductions in execution time, memory traffic, and energy consumption, independently of low-level arithmetic optimizations. In this thesis, we adopt this reformulation-oriented perspective while explicitly leveraging quantization where it aligns with hardware efficiency. In particular, we reformulate learning algorithms to eliminate costly matrix inversions, and quantize high-dimensional representations within the HDC paradigm to enable efficient inference on custom accelerators. In this way, algorithmic redesign and precision reduction jointly enable the proposed learning methods to meet the stringent performance, memory, and energy requirements of modern edge and low-power AI systems.

2.3.3 Hardware Acceleration

As the computational demands of modern AI models continue to grow, the limitations of conventional general-purpose hardware have become increasingly pronounced, necessitating a shift toward specialized architectures.

Central Processing Units (CPUs) are designed to execute a broad range of workloads. They feature deep control pipelines, complex branch prediction, and hierarchical caching mechanisms aimed at optimizing performance across general-purpose tasks. However, this generality introduces significant inefficiencies for AI workloads, which are dominated by dense, regular, and highly parallel computations. These patterns are poorly matched to scalar or superscalar execution, leading to low arithmetic intensity, high memory traffic, and suboptimal energy efficiency [133, 151]. Graphics Processing Units (GPUs) emerged as a natural first step toward hardware acceleration. Originally designed for rendering tasks, GPUs consist of hundreds or thousands of lightweight arithmetic cores organized in a Single Instruction, Multiple Thread (SIMT) fashion. Their high degree of parallelism and large memory

bandwidth make them well suited for matrix and tensor operations, which dominate neural network training and inference. For this reason, GPUs have become the workhorse of deep learning, enabling orders-of-magnitude speedups compared to CPUs. For instance, training throughput on GPUs is typically $20\text{--}100\times$ higher than on high-end CPUs, and their performance-per-watt on dense linear algebra can be up to $10\times$ superior [133, 151]. These benefits, however, come at a cost: high power consumption, substantial cooling requirements, and considerable silicon area [30, 31]. A single NVIDIA A100 GPU can draw up to 400 W under load, and large training clusters comprising tens of thousands of such devices can reach multi-megawatt power levels [152].

These drawbacks are even more critical in edge and embedded systems, where energy, thermal, and size constraints are far more severe. The high static power consumption and physical footprint of standard GPUs often preclude their deployment in mobile, wearable, or battery-operated devices. Likewise, embedded CPUs, even when scaled down for low-power operation, typically lack the parallel throughput required to support real-time, on-device AI inference. This creates a gap where neither traditional CPUs nor GPUs can satisfy the strict requirements of modern IoT applications.

These limitations have driven the development and adoption of *dedicated hardware accelerators*: specialized processing units tailored to the dataflow and arithmetic of specific algorithms. Unlike CPUs or GPUs, accelerators eliminate general-purpose overhead in favor of fixed-function datapaths, maximized on-chip data reuse, and fine-grained parallelism. The efficiency gains are remarkable, with domain-specific accelerators typically achieving $10\text{--}100\times$ better performance-per-watt than general-purpose GPUs [153]. In the data center, architectures like Google’s TPUv4 deliver roughly $1.2\text{--}1.7\times$ higher throughput and nearly $2\times$ better energy efficiency than contemporary A100 GPUs [154]. At the edge the Google’s Coral Edge TPU performs 4 TOPS at only 2 W; ARM’s Ethos-U55 targets milliwatt-range efficiency and with $3\text{--}13.5$ TOPS/W [155, 156]; and the Hailo-8 accelerator delivers 26 TOPS with a typical power consumption of roughly 2.5 W [157]. Similarly, Apple’s Neural Engine delivers up to 35 TOPS (INT8) in the A17 Pro SoC [158] and approximately 16 TOPS in the M2 chip [159].

These designs share common traits: reduced-precision arithmetic, customized memory hierarchies, and dataflows explicitly matched to the target workload. As will be discussed later in this thesis, the advantages of dedicated design become even more pronounced when dealing with the unconventional arithmetic and dataflows of neuro-inspired algorithms.

These architectural trends have driven a broader transition toward heterogeneous computing [17]. Much like a biological brain, where different regions are responsible for distinct functions, a heterogeneous system couples a general-purpose host processor with multiple specialized accelerators (Figure 2.6). The CPU manages high-level control and scheduling, while accelerators execute compute-intensive kernels in parallel [159, 160].

Designing Custom Accelerators Designing a dedicated hardware accelerator is a multidisciplinary process that begins with a deep understanding of the target

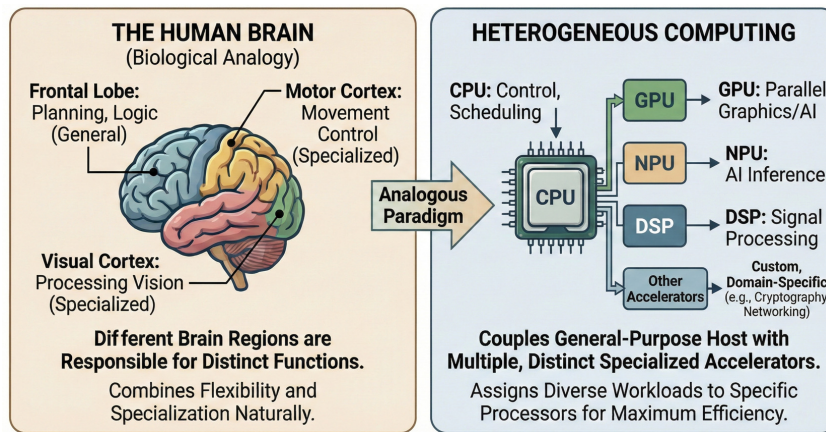


Figure 2.6 Analogy between the human brain and heterogeneous computing. In the brain, distinct cortical regions (frontal lobe, motor cortex, visual cortex, *etc.*) are specialized for different functions while remaining part of a unified system. Similarly, heterogeneous computing couples a general-purpose CPU, responsible for control and scheduling, with multiple domain-specific accelerators (GPU, NPU, DSP, and other custom engines) that execute specialized workloads, combining flexibility with high efficiency.

algorithm and its computational characteristics. The designer must carefully analyze the algorithmic structure to identify bottlenecks such as arithmetic hotspots, memory access patterns, or control dependencies, and formulate a hardware architecture that addresses these inefficiencies while satisfying application-specific constraints on latency, area, and energy consumption.

This process typically involves:

- Decomposing the algorithm into its core computational kernels;
- Mapping these kernels to custom datapaths, optimized for parallelism, data reuse, or approximate arithmetic;
- Implementing the architecture using a hardware description language (HDL), such as SystemVerilog or VHDL;
- Validating the design via simulation and functional testing;
- Synthesizing and deploying the accelerator on a reconfigurable FPGA platform for rapid prototyping;
- Eventually targeting ASIC implementation for maximum efficiency, once the design is finalized.

A critical aspect of accelerator design is deciding how the accelerator interfaces with the main processor. In a *loosely coupled* configuration, the accelerator is accessed through a memory-mapped interface or external peripheral bus (e.g., AXI, AMBA), requiring explicit software instructions for data movement and synchronization. This approach offers design flexibility and modularity but introduces higher latency and control overhead. In contrast, a *tightly coupled* accelerator is integrated directly into the processor pipeline, acting as a co-processor and controlling it via custom

ISEs. This allows the host CPU to invoke the accelerator via a single instruction, with direct register access and minimal overhead. Tightly coupled designs enable fine-grained, low-latency and low-power offloading. Open architectures like RISC-V have been instrumental in democratizing this tight integration [107, 114].

Notably, the effectiveness of a hardware accelerator is not determined solely by its architecture, but also by the algorithm it implements. This is where the algorithmic optimizations introduced in the previous section play a pivotal role. Techniques such as pruning, quantization, and reformulation reduce the arithmetic intensity, memory footprint, and control complexity of the model, leading to more efficient hardware implementation. In essence, simplifying the algorithm simplifies the hardware. Hardware–algorithm co-design represents one of the most powerful levers for overcoming the AI efficiency crisis. This thesis contributes to this line of research by exploring the design of custom accelerators and arithmetic units optimized for learning on embedded platforms.

2.3.4 From Brain Principles to Neuro-Inspired Algorithms and Hyperdimensional Computing

The previous sections discussed optimization strategies that can be adopted across the entire computing stack—from arithmetic and algorithmic improvements to hardware acceleration—to reduce the memory footprint, execution time, and energy consumption of AI workloads, enabling their deployment on edge devices. Yet even with such advances, a profound problem remains: AI was conceived to emulate aspects of natural intelligence and the human brain, but its implementations have steadily evolved into systems that diverge significantly from how biological brains actually compute. Contemporary learning systems excel at statistical pattern recognition and achieve superhuman performance on specific benchmarks, but they typically require enormous labeled datasets, heavy supervision, and remain brittle to distribution shifts, adversarial perturbations, and continual learning scenarios [161–163].

In contrast, biological brains operate with massively parallel circuits built from noisy, unreliable components, yet still achieve remarkable robustness, adaptability, and generalization across a wide range of tasks and environments [164, 165]. The human brain performs approximately 10^{14} – 10^{15} synaptic operations per second while consuming less than 20 W of power [166–168]. This fundamental discrepancy has led researchers to reconsider what makes the brain such an effective computational system—from its encoding schemes and learning rules to its architectural organization and energy constraints—and how these principles might be reengineered into neuro-inspired computing paradigms. This section surveys the major computing principles of biological brains—precision and noise tolerance, massive parallelism, fault tolerance, local learning, architectural organization, spatiotemporal coding, memory-compute co-location, energy efficiency, and rapid generalization — contrasts them with modern AI algorithms and introduces neuromorphic computing paradigms, with a particular focus on hyperdimensional computing, the central theme of this thesis.

Brains, AI, and Conventional Hardware: A Comparative View

The biological brain is an extraordinarily efficient and adaptive computational system. Understanding how it differs from today’s AI models and hardware architectures is key to motivating neuro-inspired approaches [169–171]. These differences were first systematically noted by John von Neumann, who, in his seminal work *The Computer and the Brain* [172], highlighted the fundamental divergence between serial, deterministic digital computers and the statistical, massively parallel behavior of biological systems.

Scale and Parallelism. Scale and parallelism in the brain are immense. The human brain contains approximately 10^{11} neurons and 10^{14} – 10^{15} synapses, all operating concurrently. There is no centralized processor; cognition emerges from billions of neurons and thousands of distributed, interacting circuits working in parallel [173, 174]. Even a simple act like gazing at a street scene engages countless micro-computations across visual areas simultaneously. Modern accelerators also exploit parallelism, but typically through dense and highly regular operations on structured data layouts. By contrast, neural tissue features sparse, irregular, and strongly recurrent connectivity: each neuron interacts with thousands of others in overlapping, non-uniform patterns. This mismatch makes neural-style computation difficult to map efficiently onto von Neumann systems optimized for regular control flow and deterministic memory accesses.

Stochasticity, Redundancy, and Robustness. Neural computation proceeds with stochastic, low-precision signals. Chemical synapses release neurotransmitter vesicles probabilistically in response to spikes, and ion channels in neuronal membranes open and close randomly [51]. Yet, the brain achieves remarkable reliability through population-level representations and massive redundancy. Information is holographically distributed across large ensembles, so the failure or perturbation of individual elements rarely causes catastrophic loss of function [175, 176]. In essence, the brain exemplifies *graceful degradation*: if a neuron fails or a lesion occurs, the system adapts rather than crashing. Stroke patients, for instance, often regain skills through neural re-routing. In contrast, digital hardware typically relies on high-precision, deterministic computations and operates under rigid reliability assumptions, where even small faults can lead to failure [177, 178].

Furthermore, in the brain, noise and variability are not merely tolerated but actively harnessed. Unlike classical AI models, where small adversarial perturbations can easily derail performance [162, 179], random fluctuations in neural activity support exploration during learning and help escape suboptimal behavioral policies [180, 181]. Trial-to-trial variability allows animals to explore different movement strategies without explicit retraining, much like a child experimenting with different grips to tie shoelaces. A person can read a street sign at dusk, recognize a friend’s face in a crowd, or speak over a poor phone connection, and the brain still extracts meaning; when reading scrambled text, it effortlessly reconstructs the correct words.

Local and Continuous Learning. A fundamental difference also lies in how learning occurs. Modern deep learning relies on global, offline optimization, where error gradients are computed across the entire network and updates are applied in batches. Once trained, the model is typically frozen. By contrast, learning in the brain is *local, continuous, and distributed*. Synaptic strengths are adjusted via

mechanisms like Spike-Timing-Dependent Plasticity, using only local signals available at the synapse [182]. This plasticity is modulated by neuromodulatory signals (e.g., dopamine) that encode reward or surprise, enabling reinforcement learning without the need for global synchronization or labeled datasets [183, 184]. Consequently, the brain learns asynchronously and continuously throughout life [163, 185], avoiding the catastrophic forgetting that challenges modern AI.

Data Efficiency. Biological learning is also remarkably data-efficient. A child might see a giraffe once and immediately recognize others (one-shot learning), whereas conventional AI requires thousands of labeled examples per class. Brains excel at generalization from limited data by leveraging prior knowledge, hierarchical structure, and rich internal models built through unsupervised experience [186, 187]. For example, infants can infer intuitive physical rules simply through exploratory play [188]. In contrast, when exposed to unfamiliar environments or novel tasks, conventional AI models often generalize poorly and require costly retraining or fine-tuning [189]. The brain, conversely, is a self-organizing system shaped by lifelong experience. From acquiring a native language without explicit labels to adapting gait after injury, neural circuits are continuously refined by plasticity mechanisms that operate through sparse, local updates rather than global parameter sweeps.

Recurrence and Feedback. Architecturally, the brain differs fundamentally from the feedforward structures common in deep learning. The connectome exhibits dense, recurrent connectivity with extensive feedback loops and lateral interactions [165, 190]. Virtually every cortical area that projects to another receives reciprocal projections. These loops facilitate bidirectional inference and predictive coding, enabling the integration of memory, context, and perception into unified cognitive episodes. This contrasts with current AI systems, which, despite recent advances in attention mechanisms, often struggle to match the flexible, top-down disambiguation capabilities of the biological substrate [191, 192].

Temporal Dynamics and Event-Driven Processing. Temporal dynamics play a central role in neural computation. Unlike digital systems that operate synchronously under a global clock, brain activity is inherently asynchronous and continuous. Information is conveyed not just by firing rates, but by the precise timing of spikes, neuronal synchrony, and phase relationships of oscillations [180, 193]. This temporal coding strategy enables the brain to perform computation in a spatiotemporal and event-driven fashion: neurons integrate inputs over time and emit spikes only when meaningful changes occur. We do not “poll” the environment at gigahertz frequencies; instead, signal processing is triggered precisely when perceptual inputs or internal states warrant it [173, 194]. This sparse, low-latency signaling allows the brain to process information at tens of meters per second while maintaining extraordinary energy efficiency.

Co-location of Memory and Computation. Perhaps the most significant structural divergence lies in the memory hierarchy. The classic von Neumann architecture strictly separates the CPU (compute) from RAM (storage), creating a bottleneck where vast energy is consumed moving data back and forth—the so-called “memory wall”. In the brain, memory and computation are co-located: each synapse stores a weight (the strength of the connection) and simultaneously participates in the computation by modulating signal flow. The “data” resides exactly where the processing occurs [195, 196]. This tight integration is the primary driver of

the brain’s efficiency. The adult human brain operates on a power budget of less than 20 W [166, 168] while supporting continuous learning and perception. In stark contrast, training large artificial networks involves massive data movement, often consuming megawatt-hours of energy [197, 198].

The Neuromorphic Promise. Neuro-inspired, or *neuromorphic*, computing seeks to bridge this gap by adopting these biological principles—event-driven dynamics, sparsity, distributed computation, and local plasticity—into hardware and algorithm design [169, 194, 199]. The goal is to achieve the superior energy efficiency, robustness, and fault tolerance of biological systems. For a comprehensive overview of this landscape, including current hardware architectures and scalability challenges, the reader is referred to recent surveys [199–201].

Within this evolving context, this work focuses on *Hyperdimensional Computing*, a promising neuro-inspired paradigm rooted in the observation that the brain represents and manipulates information through high-dimensional, distributed activity patterns. HDC encodes data as distributed vectors in thousands of dimensions and performs operations using simple, noise-tolerant algebra. These representations naturally exhibit desirable properties of biological cognition—such as robustness to noise, compositionality, and associative recall—while supporting efficient and parallelizable implementations on both conventional and neuromorphic hardware. The following section presents the foundations of HDC, outlines its core properties, and provides the groundwork for the algorithms and hardware proposed in this thesis.

Hyperdimensional Computing/Vector Symbolic Architectures

Section 2.1.1 introduced the two main approaches of artificial intelligence: symbolic and connectionist. The symbolic approach represents information through symbols and solves problems or infers new knowledge by manipulating them. In the connectionist approach, on the other hand, information is processed by a network of simple computational units, often called neurons, connected by weighted links whose strengths are learned from data. This section introduces Hyperdimensional Computing, also known as Vector Symbolic Architectures (HDC/VSA), a neuro-inspired computing approach that originated at the intersection of these two paradigms, combining the flexibility of connectionist models with structured vector transformations [65, 164].

The origins of HDC/VSA can be traced back to Pentti Kanerva’s foundational work on sparse distributed memory, motivated by psychological and neuroscientific observations of human memory [164, 202]. Kanerva was fascinated by the fact that remembering is an effortless, flexible, and approximate task. When we see a staircase, for example, we immediately recognize it as a member of a familiar category, regardless of whether it is spiral or straight, wooden or concrete. A single perceptual instance automatically evokes the concept “staircase” alongside an open-ended set of expectations (connecting floors, steps, railings). This content-addressable retrieval happens for countless categories without requiring exact indexing.

Kanerva also noted that once a concept has been activated, related memories tend to surface spontaneously. Thinking of a staircase may trigger memories of houses, episodes of moving furniture, images from movies, or even metaphors involving “climbing the career ladder.” A snippet of melody can recall an entire song and the

circumstances in which we heard it; the smell of a particular food can suddenly bring back a childhood scene. This *spreading activation* is accompanied by characteristic errors: we occasionally misattribute details or merge similar episodes. Human memory is thus both robust and fallible, able to generalize across instances while preserving specific details, and degrading gracefully rather than catastrophically under noise or damage.

Kanerva asked what architecture could account for this constellation of phenomena: content addressability, similarity-based interference, and graceful degradation. His answer was *Sparse Distributed Memory* (SDM) [202]. In SDM, both addresses and stored items are high-dimensional binary patterns. Memory is modeled as a large set of fixed locations, each assigned a random high-dimensional address. To store a pattern, its address activates all locations whose addresses lie within a certain Hamming radius, and the pattern’s bits are accumulated in their internal counters. To recall, the memory is probed with a (possibly noisy) cue: all locations whose addresses are close to the cue are activated, their counters are read, and a pattern is reconstructed by majority vote. This mechanism naturally reproduces the qualitative properties of biological memory: retrieval from noisy cues is possible due to redundancy, while similar patterns activate overlapping locations, leading to natural interference and generalization.

From this starting point, Kanerva formulated the broader framework of hyperdimensional computing [164]. The central insight is that these desirable properties arise not from specific hardware, but from the mathematical properties of high-dimensional spaces, distributed representations, and the algebraic operations performed on them.

HDC/VSA represents information using high-dimensional vectors, known as hypervectors (HVs), where information is *distributed*: each component contributes equally and independently to the overall representation [1]. These HVs are combined through simple vector arithmetic and compared using a similarity measure [1, 5]. This distributed and holistic representation confers many crucial properties to HDC/VSA. First, it provides *robustness to noise and faults*, since even when a large fraction of components are corrupted, the overall representation can still be reliably recognized [203, 204]. Second, it offers *inherent parallelism*, because all HV components can be processed independently, making HDC/VSA particularly well-suited for efficient hardware implementations [205–212]. Within this framework, the same principles that made SDM a good qualitative model of human memory—content addressability, robustness, similarity-based retrieval, and graceful degradation—become the building blocks of a general computing paradigm in which symbolic structures (sets, sequences, records) can be represented and manipulated as hypervectors.

For a long time, however, HDC/VSA remained a relatively niche topic, mostly confined to cognitive modeling and theoretical neuroscience. However, over roughly the last decade, the situation has begun to change. A combination of factors, including sustained dissemination efforts by the HDC/VSA community, convergence of related lines of work, successful engineering deployments in machine learning [213, 214] and cognitive architectures [215], as well as the publication of comprehensive foundational surveys [1, 216, 217], has led to a rapid growth of interest. As illustrated in Fig. 2.7, the number of HDC-related publications has increased almost exponentially over the last fifty years, with a particularly steep rise in the last decade. HDC/VSA has emerged as a compelling alternative for deployment in resource-constrained scenarios,

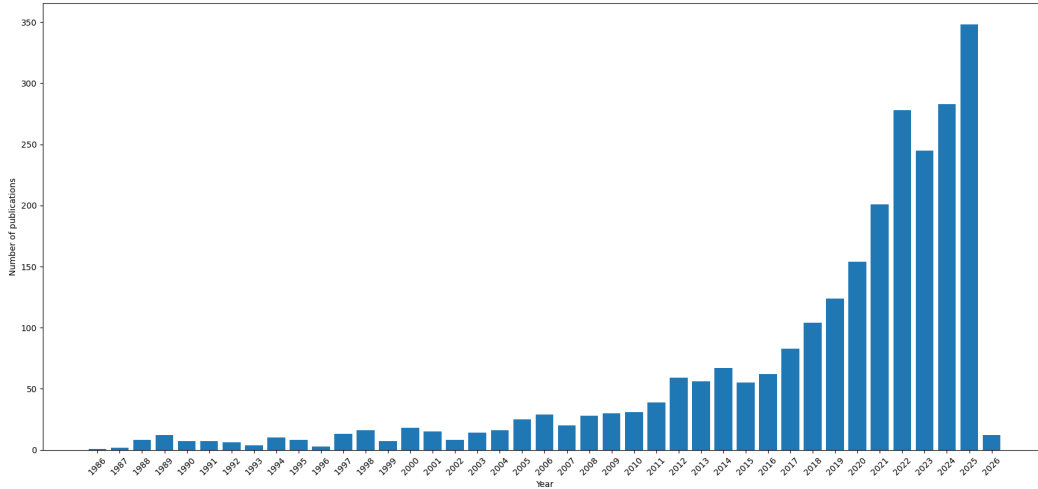


Figure 2.7 Number of publications on HDC over the last 50 years, based on Scopus.

digital hardware accelerators, and neuromorphic substrates [217], and has been successfully applied to classification [216, 218], regression [219, 220], clustering [221, 222], and reinforcement learning [223–225].

The following paragraphs introduce the basics of HDC/VSA: we first explain how independent concepts can be mapped into a high-dimensional space, then introduce the HDC/VSA arithmetic, show how to encode complex compositional structures, and finally discuss learning rules based on HDC/VSA.

Atomic Representations At the core of HDC/VSA lies the observation that fundamental concepts, symbols, or entities can be mapped to high-dimensional random hypervectors that, thanks to the geometry of high-dimensional spaces, are with high probability nearly orthogonal, i.e., linearly independent [164]. This property is counterintuitive if we reason by analogy with low-dimensional Euclidean spaces. In such spaces, standard linear algebra dictates that there exist at most D mutually *exactly* orthogonal unit vectors forming an orthonormal basis. In symbolic and localist representations, this limitation is explicit: each symbol is typically assigned its own dedicated basis direction or neuron, so the dimensionality of the representation space must grow linearly with the number of symbols. Introducing a new symbol, therefore, requires increasing the dimensionality and, in practice, modifying the surrounding architecture (e.g., enlarging embedding matrices).

HDC/VSA operates in a different regime by assigning concepts to random hypervectors that are *quasi-orthogonal*. While exact orthogonality implies an angle of exactly 90° , quasi-orthogonality implies the angle lies within a small interval around 90° [1]. In high-dimensional spaces, independently sampled random hypervectors are quasi-orthogonal with overwhelming probability [1]. Formally, if the components of $\mathbf{a}$ and $\mathbf{b}$ are i.i.d. with zero mean and finite variance and the vectors are normalized, then $\langle \mathbf{a}, \mathbf{b} \rangle$ has expectation 0 and variance $O(1/D)$. Hence, the probability that $|\langle \mathbf{a}, \mathbf{b} \rangle|$ exceeds a small fixed threshold τ decays exponentially with D . Consequently, the algebraic behavior of quasi-orthogonal directions increasingly approximates that of exactly orthogonal ones. A visual illustration of this phenomenon is provided in

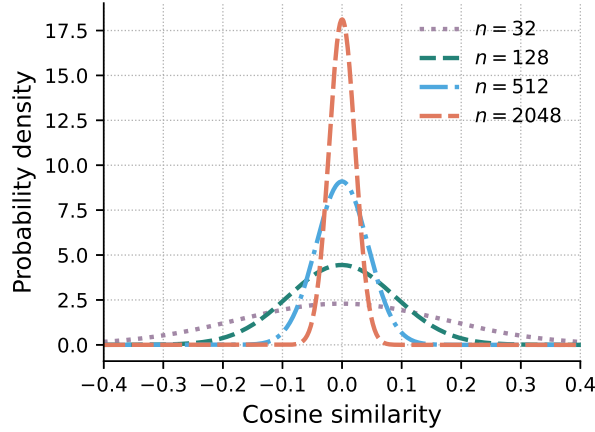


Figure 2.8 Concentration of measure phenomenon between bipolar random HVs. Plot replicated from [1].

Fig. 2.8, where pairwise similarities concentrate tightly around zero.

Thus, while there are only D exactly orthogonal basis vectors, there exist astronomically many random hypervectors that are all mutually quasi-orthogonal, and their number grows *exponentially* with D even if the admissible interval around 90° is extremely narrow [1]. This phenomenon, often termed the *blessing of dimensionality* [1, 226], underpins atomic representations in HDC/VSA. New concepts can be introduced simply by sampling additional random hypervectors of the same fixed dimensionality D ; with overwhelming probability, these new hypervectors remain quasi-orthogonal to all previously defined ones. In doing so, HDC/VSA decouples the number of representable items from the dimensionality of the representation space: capacity grows combinatorially in D , while the dimensionality itself remains fixed.

HDC/VSA arithmetic Numerous HDC/VSA models have been proposed in the literature, yet they all support the same basic primitives [5, 218]. The atomic HVs representing the basic concepts, introduced in the previous paragraph, can be combined and manipulated using a small set of operations:

- *Superposition* (+): combines multiple HVs into a single composite structure. In analogy to the simultaneous activation of several neural patterns, superposition is typically modeled as (weighted) addition,

$$\mathbf{s} = \mathbf{a} + \mathbf{b} + \mathbf{c},$$

and is often also referred to as *bundling*. After superposition, the composite HV remains similar to each of its components and can therefore be used as a holistic representation of a set or multiset of items. However, the similarity to each constituent decreases as more HVs are superimposed. This can be seen as an unstructured, additive form of similarity preservation [1]. Importantly, superposition is commutative and associative, so the result does not depend on the order in which the components are combined. Since this aggregation

can produce components outside the admissible dynamic range, a subsequent *normalization* step is often applied to project the HV back into its original domain,

$$\mathbf{s} = f(\mathbf{a} + \mathbf{b} + \mathbf{c}),$$

where $f(\cdot)$ is a function that limits the range of the components of $\mathbf{s}$ (e.g., a majority rule, sign function, clipping, or renormalization).

- *Binding* ($\circ$): due to its commutative and associative properties, superposition alone leads to the problem of *superposition catastrophe* introduced in Section 2.1.1, i.e., during recursive application of superposition, information about combinations of the initial objects (e.g., grouping) is lost since, for example:

$$(\mathbf{a} + \mathbf{b}) + (\mathbf{c} + \mathbf{d}) = (\mathbf{a} + \mathbf{c}) + (\mathbf{b} + \mathbf{d}) = \dots = \mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d}.$$

Binding addresses this problem by associating arguments (e.g., key–value pairs) into a new HV that is dissimilar to the inputs. In algebraic terms, binding is realized as a bilinear, multiplication-like operator (e.g., componentwise multiplication, XOR, or circular convolution). Binding preserves *structured similarity*: if $\mathbf{a}$ is similar to $\mathbf{a}'$ and $\mathbf{b}$ is similar to $\mathbf{b}'$, then $\mathbf{a} \circ \mathbf{b}$ is similar to $\mathbf{a}' \circ \mathbf{b}'$. An inverse operation, *unbinding* ($\oslash$), which is either exact or approximate depending on the specific model [5], allows for recovering the original arguments:

$$\mathbf{a} \oslash (\mathbf{a} \circ \mathbf{b}) \approx \mathbf{b},$$

- *Permutation* (ρ): reorders the components of an HV to produce a dissimilar one, and is used to represent order or role information, such as in sequences. The permutation operator ρ is typically chosen to be a fixed, invertible mapping on indices (e.g., a cyclic shift), so that repeated application ρ^k corresponds to shifting by k positions. As with binding, permutation preserves structured similarity: if $\mathbf{a}$ is similar to $\mathbf{a}'$, then $\rho(\mathbf{a})$ is similar to $\rho(\mathbf{a}')$. This operation is exploited, for example, when encoding ordered tuples or sequences as

$$\mathbf{s} = \rho^0(\mathbf{a}_1) + \rho^1(\mathbf{a}_2) + \rho^2(\mathbf{a}_3) + \dots,$$

where different powers of ρ encode different positions.

Finally, a *similarity* measure (or inversely, a *distance* metric) quantifies the proximity of two HVs in high-dimensional space, serving as the basic mechanism for recognition and retrieval. Together, these four primitives form the algebra of HDC/VSA, enabling the composition of complex information *without increasing dimensionality*.

The implementation of these operations varies depending on the characteristics of the vector space $\mathcal{H}$ in which HVs are defined [5]. The components of HVs can be of different *data types*, with each choice giving rise to a distinct HDC/VSA model defined over a specific $\mathcal{H}$. For instance, the binary spatter codes (BSC) model [227] operates with binary HVs ($\mathcal{H} = \{0, 1\}^D$), while the multiply–add–permute (MAP) [228] model exists in several variants: MAP-B with binary/bipolar values ($\mathcal{H} = \{\pm 1\}^D$), MAP-I with integers ($\mathcal{H} = \mathbb{Z}^D$), and MAP-C with real numbers ($\mathcal{H} = \mathbb{R}^D$). Holographic reduced representations (HRR) and Fourier holographic reduced representations

Table 2.3 Overview of selected HDC/VSA models, their vector spaces and core operations (adapted and extended from [5]). The models highlighted in bold are those that will be treated in detail in this thesis.

Model	Vector space $\mathbf{X}$	Atomic init. x_i	Similarity	Bundling	Binding / Unbinding	Reference
MAP-C	$\mathbb{R}^D$	$\mathcal{U}(-1, 1)$	Cosine	Elem. sum	Elem. mult.	[228]
MAP-I	$\mathbb{Z}^D$	$\mathcal{B}(0.5) \cdot 2 - 1$	Cosine	Elem. sum	Elem. mult.	[228]
MAP-B	$\{-1, 1\}^D$	$\mathcal{B}(0.5) \cdot 2 - 1$	Cosine	Elem. sum + threshold	Elem. mult.	[228]
BSC	$\{0, 1\}^D$	Bernoulli(0.5)	Hamming	Elem. sum + threshold	XOR	[227]
HRR	$\mathbb{R}^D$	$\mathcal{N}(0, 1/D)$	Cosine	Elem. sum + norm	circ. conv.	[57]
VTB	$\mathbb{R}^D$	$\mathcal{N}(0, 1/D)$	Cosine	Elem. sum + norm	VTB	[233]
MBAT	$\mathbb{R}^D$	$\mathcal{N}(0, 1/D)$	Cosine	Elem. sum + norm	Matrix multipl.	[234]
FHRR	$\mathbb{C}^D$	$e^{i\phi_i}, \phi_i \sim \mathcal{U}(-\pi, \pi]$	Angle distance	Angles of elem. addition	Elem. angle addition	[235]
MCR	$\mathbb{Z}_K^D$	$\mathcal{U}\{0, \dots, K-1\}$	modular Manhattan	Vector sum + norm.	Elem. modular add (mod k)	[3]

(FHRR) [57] employ real ($\mathcal{H} = \mathbb{R}^D$) and complex numbers ($\mathcal{H} = \mathbb{C}^D$), respectively. This variety of models points to the trade-off between information capacity and computational complexity, enabling adaptation to different application domains and hardware platforms. For example, BSC is often favored in resource-constrained scenarios [229] and in-memory computing [230], whereas complex-valued FHRR is attractive for neuromorphic hardware [210, 212, 231, 232]. Table 2.3 summarizes, for all the HDC/VSA models in the literature, their vector spaces $\mathcal{H}$, initialization schemes, and core operations; the models highlighted in bold are those that will be treated in detail in this thesis.

Compositional Structures and Learning Tasks

The previous sections described how independent concepts can be mapped into a high-dimensional space using random atomic hypervectors and introduced the HDC/VSA arithmetic that allows these hypervectors to be combined. A natural next question is how these operations can be composed to build *compositional* representations, i.e., hypervectors that encode sets, records, sequences, and other structured objects, and how such representations can be used to support and solve learning tasks. In this section, I first review common encoding techniques that map raw data into the high-dimensional space, and then briefly outline how these encodings are used in learning tasks [1, 216, 218]. The proposed organization of this Section mirrors the one in [218].

Univariate data We first consider *univariate* variables, i.e., single features that take either categorical values (e.g., color $\in \{\text{red}, \text{green}, \text{blue}\}$, day of week, device ID) or numerical values (e.g., temperature in $^\circ\text{C}$, heart rate in BPM, voltage in V). Categorical variables can be treated as independent concepts or symbols of the problem at hand, and thus mapped to random hypervectors drawn from the chosen high-dimensional space [218]. Thanks to the blessing of dimensionality, as discussed above, these hypervectors are quasi-orthogonal with high probability and effectively realize a high-dimensional Kronecker-delta code for discrete symbols (e.g., RED, GREEN, BLUE are all mutually quasi-orthogonal and can be safely superposed with other codes).

On the other hand, when dealing with numerical variables, purely random codes are insufficient because we would like to reflect the inherent order and metric structure of the values in the high-dimensional space (e.g., 21°C should be closer to

22 °C than to 35 °C). To do that, in HDC/VSA the variable range is discretized into L levels and each level $\ell \in \{0, \dots, L-1\}$ is assigned a distinct level hypervector $\mathbf{h}_\ell$ constructed such that adjacent levels yield more similar hypervectors than distant ones [218]. This can be done adopting different techniques, depicted in Figure 2.9

- *Thermometer*: in this technique [236, 237], the first level hypervector is initialized with all 0s or -1 s and the subsequent ones are obtained by progressively flipping a specific number of elements. Specifically, the algorithm flips $D/(L-1)$ continuous components in each subsequent hypervector. For example, for $L = 5$ and a binary hypervector of dimension D , level ℓ can be encoded by activating approximately $\ell D/L$ components in a fixed order, i.e., $\mathbf{h}_0$ has almost no active bits, $\mathbf{h}_1$ has a small prefix of ones, and so on, until $\mathbf{h}_4$ has almost all bits set. The Hamming weight increases monotonically with the level, and any two levels satisfy $\text{supp}(\mathbf{h}_\ell) \subset \text{supp}(\mathbf{h}_{\ell+1})$. This yields a similarity profile that decreases smoothly with the distance $|\ell - \ell'|$ and is particularly simple to implement in hardware.
- *Fixed-density (“float”)*: this technique [237] provides a constant number of active components in its hypervectors. A sliding window of active components of fixed size (the “float”) is shifted across dimensions as the level increases. Concretely, one chooses a window of width $w \ll D$ and encodes level ℓ by placing the window starting at position p_ℓ (e.g., $p_\ell = \lfloor \ell D/L \rfloor$ modulo D). Thus, each $\mathbf{h}_\ell$ has exactly w active bits, but their positions differ with ℓ . Neighboring levels have overlapping windows and therefore share many active components, while distant levels have little or no overlap. Compared to thermometer codes, the distance is constant, which can be advantageous for some similarity measures and for hardware implementations with fixed switching activity [218].
- *Random flipping*: in this technique [237–239], the first level hypervector is initialized at random, $\mathbf{h}_0$. Then, for each subsequent level $\ell + 1$, $\mathbf{h}_{\ell+1}$ is generated by flipping a fixed number r of components in $\mathbf{h}_\ell$. If the flipped indices are chosen to be disjoint over a range of levels, the overlap between adjacent levels is high, while the overlap between levels that are many steps apart decreases approximately linearly in the number of steps. This scheme yields fine control over the similarity decay as a function of $|\ell - \ell'|$ and has been successfully used, for example, for biosignal processing and semantic inference [238, 239].
- *Random selection*: this strategy [218, 237, 240] begins by sampling two endpoint hypervectors, $\mathbf{h}_{\min}$ and $\mathbf{h}_{\max}$, that encode the minimum and maximum of the value range. In addition, a *selection* hypervector ϕ is drawn once, with components independently sampled from a uniform distribution in $[0, 1]$ [240]. For each level ℓ , a scalar threshold $\tau_\ell \in [0, 1]$ is defined (typically increasing with ℓ), and the level hypervector $\mathbf{h}_\ell$ is obtained by comparing ϕ to τ_ℓ component-wise: if $\phi_d \geq \tau_\ell$ the d -th component is copied from $\mathbf{h}_{\max}$, otherwise it is copied from $\mathbf{h}_{\min}$. Reusing the same ϕ across all levels makes the lower levels remain closer to $\mathbf{h}_{\min}$, the higher ones closer to $\mathbf{h}_{\max}$, and intermediate levels mixtures of the two, yielding a smooth interpolation of similarities along the level

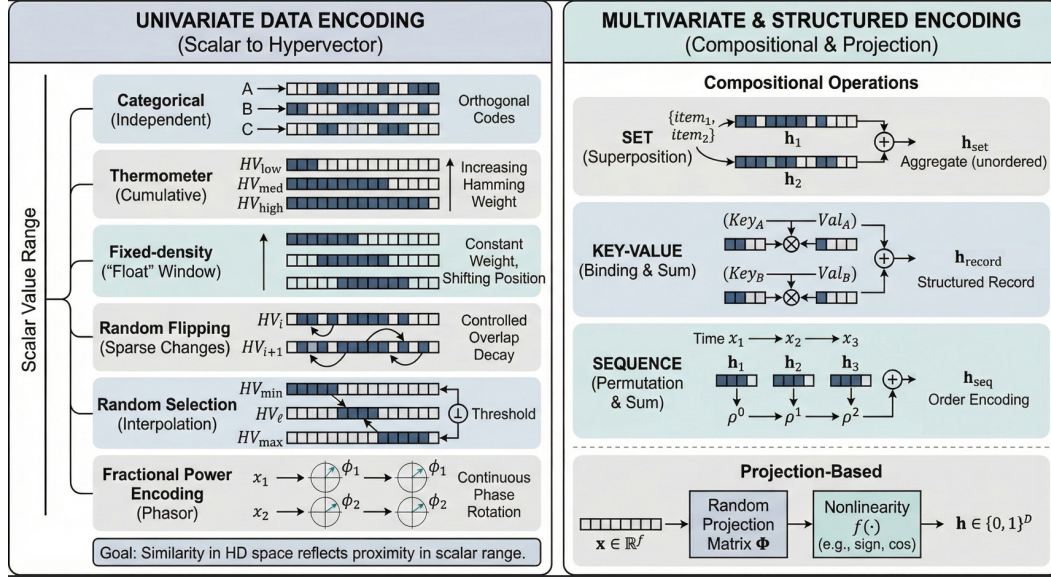


Figure 2.9 Overview of hypervector encoding techniques employed in HDC/VSA. The left panel illustrates diverse strategies for mapping scalar values (categorical or discrete/continuous numerical) into high-dimensional space. The fundamental goal for numerical data is to ensure that the similarity between resulting hypervectors reflects the proximity of their corresponding scalar values in the input range. The right panel demonstrates methods for representing complex data structures. *Compositional operations* utilize the standard arithmetic operations of superposition (+), binding ($\otimes$), and permutation (ρ) to construct sets, structured key-value records, and ordered sequences. Below that, *projection-based* encoding shows how a dense input vector $x \in \mathbb{R}^f$ is mapped to a binary hypervector h via multiplication with a random projection matrix Φ followed by a component-wise nonlinearity $f(\cdot)$.

index (approximately linear when τ_ℓ is chosen accordingly [218]). Because the choice at each component is stochastic rather than deterministic, this encoding introduces variability in the resulting level hypervectors while requiring storage only for the two endpoints and the selector ϕ , and it can be applied to any hypervector representation (binary, bipolar, real-valued, etc.).

In all these techniques, the design goal is to map scalar values into the high-dimensional space so that the similarity between hypervectors reflects proximity on the original numerical scale, while still producing quasi-orthogonal codes for values that are far apart. This makes it possible, for example, to represent a patient's heart rate or a sensor's temperature reading as a hypervector that both respects the metric structure of the variable and can be seamlessly combined with other hypervectors using the HDC/VSA algebra.

These constructions can be understood as discrete approximations of simple kernels (e.g., triangular kernels). More recent approaches derive level hypervectors directly from a target similarity matrix or via learning [241, 242].

All the approaches discussed above rely on an explicit quantization of the variable range. The FHRR model also offers the *fractional power encoding* (FPE) technique [57, 220]. This encoding starts by initializing a complex-valued basis

hypervector $\mathbf{g}$ on the unit circle (phasor representation), whose component d can be written as $g_d = e^{i\phi_d}$ with phase $\phi_d \in (-\pi, \pi]$. A real value x is then encoded via component-wise exponentiation of the phases of $\mathbf{g}$, yielding

$$\mathbf{g}^x = (e^{ix\phi_1}, \dots, e^{ix\phi_D}),$$

which can be interpreted as repeatedly “self-binding” the basis hypervector by a fractional power. The resulting phasor codes induce continuous similarity profiles as a function of the difference $x - x'$ (e.g., approximating sinc- or Gaussian-shaped kernels, depending on how the phases ϕ_d are sampled) [220]. An important property of these codes is that they naturally capture *circular* structure: level hypervectors for scalar values x and x' differ only by a rotation of their phases, so the similarity depends on $x - x'$ modulo the period. This yields correlation structures well-suited for cyclic variables such as angles, phases, or time-of-day, and has been exploited in several works on encoding circular data and periodic phenomena in HDC/VSA [237, 240, 243].

Encodings for multivariate and structured data. Moving from single variables to multivariate inputs, HDC/VSA distinguishes between two broad families of encoders [218]: *compositional* encodings that are built directly from basis hypervectors and HDC/VSA operations, and *projection-based* encodings that rely on random linear mappings followed by a nonlinearity.

In compositional encodings, structured data are assembled using superposition, binding, and permutation:

- *Set encodings* represent an (unordered) collection of items $\{x_1, \dots, x_n\}$ by superposing their hypervectors,

$$\mathbf{h}_{\text{set}} = f(\mathbf{h}(x_1) + \dots + \mathbf{h}(x_n)),$$

where $\mathbf{h}(x_i)$ denotes the hypervector obtained by encoding each item x_i using the univariate schemes introduced above (e.g., random hypervectors for categorical variables, correlated level hypervectors or FPE for numerical ones), and $f(\cdot)$ performs normalization. A concrete example is an event-based sensor node that, within a given time window, observes a set of active events (e.g., MOTION, DOOR_OPEN, LIGHT_ON). Each event type is first encoded as a categorical hypervector, and the state of the node in that window is encoded as the superposition of the hypervectors of all events that occurred. This representation supports approximate membership queries (e.g., “was MOTION present in this window?”) and can be used directly for downstream tasks such as activity or context classification [1, 218].

- *Key-value (record-based) encodings* extend this idea to tabular or record-like data. Each feature index (or role) k_i is assigned a key hypervector $\mathbf{k}_i$, each value v_i is encoded as a value hypervector $\mathbf{v}_i$, and the whole record is represented as

$$\mathbf{h}_{\text{kv}} = f(\mathbf{k}_1 \circ \mathbf{v}_1 + \dots + \mathbf{k}_n \circ \mathbf{v}_n),$$

where binding associates each feature with its value (creating the key-values pairs) and superposition aggregates across features [1, 218]. For instance, a

patient record with attributes AGE, BLOOD_PRESSURE, and HEART_RATE can be encoded by binding the role hypervectors for each attribute with their corresponding scalar encodings (e.g., based on correlated level hypervectors) and summing them. This family of encodings has been shown to perform particularly well for tabular data [218].

- *Sequence and n -gram based encodings* exploit permutation to capture order. Specifically, the i -th element of a sequence is first encoded into a hypervector $\mathbf{h}(x_i)$ using the univariate schemes introduced above (e.g., categorical or numerical encoders), and then “placed” at position i by applying a permutation operator ρ^{i-1} . The whole sequence of length L is represented by:

$$\mathbf{h}_{\text{seq}} = f(\rho^0(\mathbf{h}(x_1)) + \rho^1(\mathbf{h}(x_2)) + \cdots + \rho^{L-1}(\mathbf{h}(x_L))),$$

where $f(\cdot)$ denotes normalization. Each ρ^{i-1} acts as a position-dependent “role”: the same item encoded at different positions produces different hypervectors, and the superposition aggregates information across the entire sequence.

A first example is sentences in natural language processing. Each word is encoded as a categorical hypervector. To obtain a sentence hypervector, the word hypervectors are permuted according to their position in the sentence (using ρ^{i-1}) and then superposed. Sentences in different languages or with different word orders yield different sequence hypervectors, while sentences that share many words in similar positions produce similar hypervectors, which can be exploited for tasks such as language identification or topic classification [218].

A second example is an EMG-based hand-gesture recognition system [239]. The EMG signal is segmented into short time windows, and within each window, each sample x_i (e.g., the amplitude of the EMG at time index i) is encoded into a hypervector using a numerical encoder (e.g., correlated level hypervectors). To represent the temporal structure, the permutation ρ^{i-1} for index i is applied, and all position-encoded samples in the window are then superposed to obtain a single hypervector $\mathbf{h}_{\text{window}}$. Different gestures (e.g., fist, pinch, open hand) produce characteristic EMG patterns, which translate into distinct window hypervectors that can be then classified. In this way, the encoding captures both the values of the EMG signal and their temporal order inside the window.

More complex structures (trees, graphs, hierarchical records) can be constructed recursively using the same primitives, by first encoding substructures and then treating their hypervectors as new atomic units [1, 216]. This recursive compositionality allows HDC/VSA to represent arbitrarily rich symbolic structures while keeping the dimensionality fixed.

Projection-based encoders, on the other hand, map the original d -dimension feature vector $\mathbf{x} \in \mathbb{R}^d$ into the high-dimensional space via a random linear transformation followed by a (possibly nonlinear) discretization [244]. This construction leverages Johnson–Lindenstrauss-type results to approximately preserve distances [245]. A standard example is the *thresholded random projection* [244], with $\Phi \in \mathbb{R}^{D \times d}$ sampled from a normal distribution.

$$\mathbf{h} = \text{sign}(\Phi \mathbf{x}),$$

Another example is the *Random Fourier features* method [246], that is also based on the random projection but uses a periodic function as the nonlinearity, to approximate shift-invariant kernels, such as Gaussian or sinc kernels, within the HDC/VSA framework.

$$\mathbf{h} = \sqrt{2}\cos(\Phi\mathbf{x} + b),$$

Learning tasks. Once an input x has been mapped or encoded into the high-dimensional space as $\mathbf{h}(x) \in \mathcal{H}$, learning tasks can be performed [218]. HDC/VSA has been successfully applied in the literature for modeling and solving classification [216, 218], regression [219, 220], clustering [221, 222], and reinforcement learning problems [223–225]. In all these cases, the HDC/VSA pipeline can be viewed as a two-stage process: (i) an encoding stage that constructs $\mathbf{h}(x)$ using the univariate and compositional schemes described above, and (ii) a training procedure and decision rule operating in the high-dimensional space. This section briefly introduces the basic principles for classification, regression, and clustering, leaving the detailed description of the learning rules and algorithms adopted in this thesis to the dedicated chapters.

Classification is a supervised learning problem in which the model must predict categorical class labels of new input patterns based on past observations. For a multiclass problem with classes $\mathcal{C} = \{1, \dots, C\}$ and training pairs (x_n, y_n) , each class c is associated with a prototype hypervector $\mathbf{p}_c$. In the simplest baseline, prototypes are formed by superposing the encoded training examples of each class and optionally normalizing:

$$\mathbf{p}_c = f\left(\sum_{n:y_n=c} \mathbf{h}(x_n)\right), \quad c \in \mathcal{C}.$$

At inference time, a new input x is encoded into $\mathbf{h}(x)$ and assigned to the class whose prototype is most similar under the chosen distance or similarity measure,

$$\hat{y} = \arg \max_{c \in \mathcal{C}} \delta(\mathbf{h}(x), \mathbf{p}_c).$$

Several refinements have been proposed to iteratively adjust the prototypes [218]. The simplest one is a perceptron-like update: after classifying a training sample x_n with predicted label $\hat{y}_n$, if $\hat{y}_n \neq y_n$ the prototypes are updated as

$$\begin{cases} \mathbf{p}_{y_n} \leftarrow f(\mathbf{p}_{y_n} + \eta \mathbf{h}(x_n)), \\ \mathbf{p}_{\hat{y}_n} \leftarrow f(\mathbf{p}_{\hat{y}_n} - \eta \mathbf{h}(x_n)), \end{cases}$$

where $\eta > 0$ is a learning-rate parameter. This prototype shaping can be performed in a single pass or over multiple epochs, and can be combined with clipping or sparsification to control prototype growth. Other techniques include iterative prototype updates, learning-rate-controlled updates analogous to stochastic gradient descent, discriminative prototype shaping, and compressed or multi-prototype representations [218]. From a functional perspective, these classifiers implement linear decision rules in the hypervector space and can be interpreted as approximate kernel methods, where the encoder implicitly defines the kernel [220, 244]. This

important connection clarifies why HDC/VSA can achieve competitive performance on a broad range of classification benchmarks and suggests that more sophisticated linear classifiers can also be used on top of HDC/VSA encodings [218].

Regression aims at predicting a real-valued output y from an encoded input $\mathbf{h}(x)$. A common HDC/VSA formulation [219] maintains a model hypervector $\mathbf{w} \in \mathcal{H}$ and uses a linear prediction

$$\hat{y} = \langle \mathbf{w}, \mathbf{h}(x) \rangle,$$

followed by an online update of $\mathbf{w}$ based on the prediction error $e = y - \hat{y}$,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta e \mathbf{h}(x),$$

with learning rate $\eta > 0$. This is directly analogous to linear regression in a randomized feature space defined by the encoder and can be implemented in streaming or batch fashion.

Clustering is an unsupervised learning task in which input patterns are grouped into K clusters based on similarity. In HDC/VSA, each cluster $k \in \{1, \dots, K\}$ is represented by a centroid hypervector $\mathbf{c}_k$ [221, 222, 247]. Given encoded data $\mathbf{h}(x_n)$, an assignment step associates each point with the closest centroid in similarity, and an update step revises the centroids by superposition, yielding a high-dimensional analogue of K -means.

In all these cases, the same core ingredients recur: an encoder that maps data to hypervectors, linear updates in the high-dimensional space, and similarity-based retrieval or assignment.

Factorization of Composite Representations in HDC

Apart from learning tasks, forming composite representations in high-dimensional spaces has many advantages, as it enables encoding complex structured data—such as sequences, graphs, and trees—into single hypervectors, in superposition. Given such a composite HV, it is often desirable to *decode* the underlying structure by recovering the constituent HVs. This procedure is variously referred to as *recovery*, *decoding*, *parsing*, or *factorization* in the HDC/VSA literature [1]. In general, decoding assumes access to an *item memory* (or codebook) storing the relevant atomic or composite HVs and to knowledge of how the composite HV was constructed (which operations, in which order). The following discussion revisits decoding techniques, starting from simple unbinding and clean-up, and arriving at resonator networks as an efficient solution to more challenging factorization problems.

Recovery and clean-up in simple composites. Consider the simple case in which we have four basic symbols $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$ and we map them to random HVs, creating an item memory (or dictionary) for the problem. Using binding and bundling to combine these symbols, we can form the composite $\mathbf{s}$:

$$\mathbf{s} = \mathbf{a} \circ \mathbf{b} + \mathbf{c} \circ \mathbf{d},$$

To decode such composites and recover the item coupled with a specific cue, we can apply the *unbinding* operation between $\mathbf{s}$ and that cue. Unbinding with respect to $\mathbf{a}$

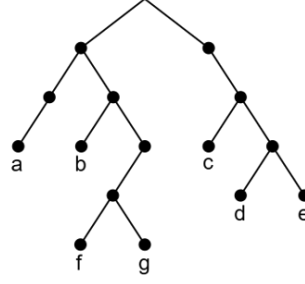


Figure 2.10 Compositional encoding of a binary tree. Each node or leaf (e.g., $a, b, \dots, g$) is defined by its unique position in the hierarchy, which is encoded by binding a base vector with a sequence of role hypervectors (e.g., L for Left, R for Right) representing the path from the root. The representation of the entire tree is formed by superposing all constituent node HVs.

(to retrieve its partner **b**) yields:

$$\mathbf{s} \circ \mathbf{a} = \mathbf{b} + \mathbf{a} \circ (\mathbf{c} \circ \mathbf{d}) = \mathbf{b} + \boldsymbol{\nu},$$

where $\boldsymbol{\nu}$ is a quasi-orthogonal noise term originating from cross-talk with the other bound pair. A *clean-up* step then compares this noisy result with all HVs in the item memory and returns the most similar index. For limited-size compositional structures, this procedure can recover the original factors with formal guarantees [1].

From recovery to factorization. This clean-up mechanism can also be applied to more complex examples, such as the tree encoding shown in Fig. 2.10.

To encode a tree in the high-dimensional space, each leaf node $a, b, \dots, g$ is represented by a random hypervector $BV_a, BV_b, \dots, BV_g$. The two possible directions at each node (left and right) are encoded with random role vectors $\mathbf{L}$ and $\mathbf{R}$. The position within the tree is then encoded using permutations, with each level of depth applying an additional permutation. For instance, if the path to node g is denoted by the sequence $\{\mathbf{L}, \mathbf{L}, \mathbf{R}, \mathbf{L}, \mathbf{L}\}$, its representation is:

$$HV_g = BV_g \circ \rho^0(\mathbf{L}) \circ \rho^1(\mathbf{L}) \circ \rho^2(\mathbf{R}) \circ \rho^3(\mathbf{L}) \circ \rho^4(\mathbf{L}),$$

After encoding all the nodes, the entire tree structure is superimposed into one hypervector:

$$\text{tree} = HV_a + HV_b + \dots + HV_g.$$

If we want to determine which node lies along a specific path, say $\{\mathbf{L}, \mathbf{R}, \mathbf{L}\}$, we can retrieve the corresponding node by unbinding the related role hypervectors from the composite tree vector:

$$\text{tree} \oslash (\rho^0(\mathbf{L}) \circ \rho^1(\mathbf{R}) \circ \rho^2(\mathbf{L})) \approx BV_b + \boldsymbol{\nu},$$

followed by a clean-up over the item memory of node HVs $\{BV_a, \dots, BV_g\}$:

$$\underset{i}{\operatorname{argmax}} \text{ similarity}(BV_i, \text{tree} \oslash (\rho^0(\mathbf{L}) \circ \rho^1(\mathbf{R}) \circ \rho^2(\mathbf{L}))).$$

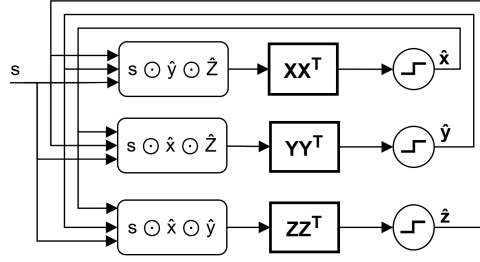


Figure 2.11 Architecture of a three-factor Resonator Network designed to factorize a composite hypervector $s \approx x_{i^*} \circ y_{j^*} \circ z_{k^*}$. The system maintains state estimates $\hat{x}, \hat{y}, \hat{z}$ for each factor. At each iteration, a “guess” for one factor is generated by unbinding the current estimates of the others from the input s (left blocks). This noisy guess is then cleaned up using an associative memory containing the valid codebook vectors (center blocks $XX^\top, YY^\top, ZZ^\top$) and passed through a nonlinearity to refine the estimate (right blocks). The loop continues until the system converges to a stable solution.

A more difficult question, however, is the inverse one: given the global tree HV and the HV of a node, how can we recover the *path* to that node? If we unbind the leaf from the tree representation, we obtain

$$\mathbf{s} = \text{tree} \odot \text{BV}_b \approx \rho^0(\mathbf{L}) \circ \rho^1(\mathbf{R}) \circ \rho^2(\mathbf{L}) + \nu,$$

that is, a product of several role vectors (successive left/right decisions at different depths). This composite is not close to any single HV stored in the item memory; instead, it lives in the space of all *combinations* of left/right/stop decisions along a path. A naive clean-up strategy would require an explicit codebook containing every possible path up to the maximum depth of the tree, and recovery would amount to a nearest-neighbour search over this exponentially large set.

This illustrates the *factorization problem*: given a composite vector formed as a product of multiple codevectors,

$$\mathbf{s} = \mathbf{x}_{i^*} \circ \mathbf{y}_{j^*} \circ \mathbf{z}_{k^*},$$

we want to recover the individual factors $\mathbf{x}_{i^*}, \mathbf{y}_{j^*}, \mathbf{z}_{k^*}$ without enumerating all possible tuples from the corresponding codebooks [248]. Similar factorization problems arise in visual scene parsing and other tasks where several attributes (e.g., object, color, position) are bound and then superimposed [248].

Resonator Network: decoding by factorization. *Resonator networks* [248, 249] offer a principled, iterative method to solve this problem and decode composite hypervectors into their original factors. A resonator is a nonlinear dynamical system that repeatedly refines guesses of the individual factors (e.g., each element in a path), exploiting the properties of HDC/VSA to search in superposition and test many potential factorizations in parallel, until a set of factors is found that agrees with the input. Solutions emerge as stable fixed points of the network dynamics, which is why they are said to “resonate” with the input.

Figure 2.11 illustrates the architecture of a resonator network for factorizing a composite HV

$$\mathbf{s} = \mathbf{x}_i^* \circ \mathbf{y}_j^* \circ \mathbf{z}_k^*,$$

where the three factors are sampled from codebooks $X = \{\mathbf{x}_1, \dots, \mathbf{x}_M\}$, $Y = \{\mathbf{y}_1, \dots, \mathbf{y}_M\}$, and $Z = \{\mathbf{z}_1, \dots, \mathbf{z}_M\}$.

The network maintains current estimates for each factor, $\hat{\mathbf{x}}(t)$, $\hat{\mathbf{y}}(t)$, and $\hat{\mathbf{z}}(t)$. In the figure, each row corresponds to one of these estimates. On a given row, the left block performs an *inference step*: it binds the input $\mathbf{s}$ with the current estimates of the other two factors (e.g., $\mathbf{s} \oslash (\hat{\mathbf{y}} \circ \hat{\mathbf{z}})$) to produce a noisy guess of the remaining factor. Because binding distributes over addition, the product $\hat{\mathbf{y}} \circ \hat{\mathbf{z}}$ compactly represents all pairwise combinations of candidates from Y and Z , so this inference step effectively evaluates many potential factorizations at once.

The central block, labeled $XX^\top$, $YY^\top$, or $ZZ^\top$, implements a clean-up memory built from the corresponding codebook, projecting the noisy guess onto the span of valid items and computing similarity scores. Finally, the rightmost block applies a nonlinearity $g(\cdot)$ (typically an element-wise threshold to ± 1) that prevents runaway positive feedback and returns a refined HV estimate [248, 249]. Iterating these updates makes the whole system “resonate” toward a configuration of factors that is consistent with $\mathbf{s}$.

Formally, the updates for the three-factor case can be written as

$$\hat{\mathbf{x}}(t+1) = g(XX^\top (\mathbf{s} \circ \hat{\mathbf{y}}(t) \circ \hat{\mathbf{z}}(t))), \quad (2.11)$$

$$\hat{\mathbf{y}}(t+1) = g(YY^\top (\mathbf{s} \circ \hat{\mathbf{x}}(t) \circ \hat{\mathbf{z}}(t))), \quad (2.12)$$

$$\hat{\mathbf{z}}(t+1) = g(ZZ^\top (\mathbf{s} \circ \hat{\mathbf{x}}(t) \circ \hat{\mathbf{y}}(t))), \quad (2.13)$$

where the initial estimates are typically set to the superposition of all candidates in each codebook (e.g., $\hat{\mathbf{x}}(0) = \sum_i \mathbf{x}_i$).

Example: recovering paths in a tree. In the tree example, we want to recover which path led to node b . Our input is the composite

$$\mathbf{s} = \text{tree} \circ \text{BV}_b,$$

which, up to noise, equals the product of the role vectors along the path to b : $\mathbf{s} \approx \rho^0(\mathbf{L}) \circ \rho^1(\mathbf{R}) \circ \rho^2(\mathbf{L})$. We treat each decision at a different depth as a separate factor. For a depth-3 path, we introduce three codebooks:

$$X = [\rho^0(\mathbf{L}), \rho^0(\mathbf{R}), \rho^0(\mathbf{S})], \quad Y = [\rho^1(\mathbf{L}), \rho^1(\mathbf{R}), \rho^1(\mathbf{S})], \quad Z = [\rho^2(\mathbf{L}), \rho^2(\mathbf{R}), \rho^2(\mathbf{S})],$$

where $\mathbf{S}$ encodes the “stop” decision (no further movement from that level).

We instantiate three estimators $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$ with initial states given by the superposition of all options at each level, e.g. $\hat{\mathbf{x}}(0) = \rho^0(\mathbf{L}) + \rho^0(\mathbf{R}) + \rho^0(\mathbf{S})$, and similarly for $\hat{\mathbf{y}}(0)$ and $\hat{\mathbf{z}}(0)$.

Running the resonator updates, the three estimators gradually move from representing all possible paths in superposition to a consistent sequence of moves, effectively recovering the path without having to enumerate all possible paths in memory [248].

Resonator networks thus generalize simple unbinding + clean-up to deeper, multi-factor structures: they use the algebra of HDC/VSA to search large combinatorial spaces in superposition and rely on clean-up memories to settle on one coherent factorization. This makes them a powerful tool for decoding holistic high-dimensional representations of complex data structures [248].

Part 1
Arithmetic and Algorithmic
Optimizations

Overview

As established in Chapter 2, Contextual Bandits provide a principled framework for adaptive decision-making under partial feedback. Their ability to adapt policies online from logged interactions makes them suitable for a wide range of real-world systems, leading to their increasing adoption in edge applications [38, 39] to make embedded platforms not only intelligent but also autonomous and adaptive. Among the many algorithmic variants, the Linear Bandits family remains one of the most widely deployed due to its theoretical guarantees and practical effectiveness. However, deploying these algorithms on resource-constrained embedded platforms and achieving real-time adaptability reveals a computational overhead that is frequently overlooked in theoretical models but becomes the limiting factor in practice.

As analyzed in Chapter 2, linear bandits rely on the linearity assumption between the context and the reward, employing ridge regression procedures to assess the expected reward for each action and determine the optimal one. The key point of this process is matrix inversion, a computationally intensive operation whose complexity grows cubically with the size of the matrix. The Disjoint linear CB requires inverting N matrices of dimensions $d \times d$ at each step, for an asymptotic complexity equal to $\mathcal{O}(d^3)$. The Hybrid variant is even more demanding, requiring the inversion of a large $k \times k$ and N $d \times d$ matrices, asymptotically growing as $\mathcal{O}(k^3) = \mathcal{O}(f^3 d^3)$. This cubic scaling, common to both algorithmic variants, imposes a heavy computational load that becomes the limiting factor for real-time applications where rapid decision-making is crucial.

In particular, on embedded cores the matrix inversion cost is completely dominated by the division operation. A preliminary profiling study conducted during the early stages of this research [71] quantified the severity of this bottleneck on a representative embedded platform. The Disjoint LinUCB algorithm was executed on the Klessydra-T13 RISC-V processor—a single-issue, in-order core without floating-point support—using a context dimension $d = 8$, $N = 8$ arms, 32-bit fixed-point arithmetic, and fully static memory allocation (i.e., no dynamic heap usage). These parameters represent a small practical setting for a linear bandit deployment; real-world scenarios with richer context representations or larger action spaces would only amplify the computational load. Even in this minimal configuration, cycle-accurate profiling revealed that integer division alone accounts for up to 70% of total execution cycles. The root cause is structural: on the Klessydra-T13, as on many embedded cores targeting area- and power-constrained applications, integer division is implemented through a non-pipelined, fixed-latency restoring algorithm that executes in 32 clock cycles regardless of the operand values, while addition and multiplication complete in a single cycle. As a result, the vast majority of execution time is concentrated in

a single arithmetic unit. At the arithmetic level, the performance of linear bandits on this class of processors is therefore *divider-bound*: no amount of loop-level optimization can overcome the bottleneck as long as the underlying division hardware remains the limiting factor.

Compounding this computational bottleneck is the quadratic memory overhead imposed by the covariance matrices— $\mathcal{O}(d^2)$ per arm for Disjoint, $\mathcal{O}(k^2) + N \cdot \mathcal{O}(d^2)$ for Hybrid—which quickly becomes prohibitive on embedded platforms with kilobyte-scale memory budgets.

These findings define the trajectory of Part I, which systematically dismantles these bottlenecks through cross-layer optimization using a bottom-up approach.

- **Chapter 3** addresses the arithmetic bottleneck identified above—and, more generally, that of any edge AI workload whose execution time is dominated by division operations—by introducing the VLNPD, a novel data-dependent division unit that exploits operand statistics to minimize latency and energy consumption while incurring negligible hardware overhead.
- **Chapter 4** integrates the VLNPD into a RISC-V Vector Coprocessor, demonstrating that variable-latency division yields significant benefits under SIMD execution.
- **Chapter 5** addresses the algorithmic scaling by proposing a reformulated, optimized version of linear bandits that replaces explicit matrix inversion with lightweight incremental Sherman-Morrison-Woodbury updates. By combining these algorithmic optimizations with a vector accelerator and the hardware optimizations proposed in the previous chapters, we demonstrate the advantages of a hardware-software co-design on linear bandits for embedded platforms.

Together, these contributions yield order-of-magnitude improvements in execution time and energy consumption. Yet, as Part II will show, the inherent quadratic scaling of dense linear algebra ultimately motivates a more radical shift, exploring neuro-inspired computing paradigms that bypass the linear-algebra bottleneck entirely to model and solve CB problems.

Chapter 3

Design, Implementation and Evaluation of a New Variable Latency Integer Division Scheme

Chapter based on Paper 1 [250]: Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Saeid Jamili, and Mauro Olivieri. Design, implementation and evaluation of a new variable latency integer division scheme. IEEE Transactions on Computers, 73(7):1767–1779, 2024. doi: 10.1109/TC.2024.3386060

"The art of being wise is the art of knowing what to overlook."

William James

The preliminary analysis in Part 1 Overview identified integer division as the dominant arithmetic bottleneck in the execution of linear bandit algorithms on embedded processors, accounting for up to 70% of total execution time. This penalty is a direct consequence of conventional embedded design choices, which prioritize area-efficient, fixed-latency dividers. Under these constraints, it is clear that neither algorithmic refinements nor vectorization can fully offset the cost of matrix inversion, because the underlying bottleneck will ultimately be the divider. This motivates a closer examination of division itself.

Integer division is one of the fundamental operations in computer arithmetic, used in a wide range of applications such as digital signal processing [121], random number generation [251], cryptography [252], artificial intelligence [121, 122, 253], matrix factorization [254, 255], and image processing [256, 257].

Compared to addition and multiplication, division is inherently slower due to the absence of associative and commutative properties that do not allow factorization and parallelization [2, 121–123, 258], resulting in an expensive hardware implementation or severe performance bottleneck for many applications [123, 124]. The implementation of dedicated integer division units is sometimes avoided and replaced by alternative methods, emulating integer divisions with floating-point dividers [2], yet requiring significantly large area and power consumption [259] that make them unsuitable for FPGA implementation, or Integrated Circuit (IC) microarchitectures with limited hardware resources. In these contexts, low-hardware-cost dividers with

a fixed execution latency are often used [260, 261], possibly resulting in performance limitation for those embedded applications where computational speed is critical such as automotive, video processing, and industrial control.

These observations motivate the first research question of this dissertation:

? Research Question

Can we exploit input-dependent properties to design an integer division unit for AI on resource-constrained processors that remains compact while achieving lower average latency and energy?

Variable latency arithmetic units have been studied for decades, covering addition [128–130], multiplication [131], division [122], and more complex operations [132]. Variable latency represents a valid alternative to fixed-latency when the average execution time of the target application, resulting from the average latency and the sustainable clock speed, is significantly shorter than in a fixed latency implementation, with negligible hardware overhead.

This chapter is mainly based on Paper 2 [250] and introduces a variable latency data-dependent integer divider that dynamically identifies and skips redundant iterations that would otherwise produce only left shifts, leveraging the number of leading zeros in the divisor and in the evolving partial remainder. By enabling early termination and iteration skipping, our design significantly improves the average convergence time and power consumption, outperforming existing fixed [262] and variable latency alternatives [2, 122, 263, 264] in the literature, maintaining hardware requirements of the latter. These properties make the presented approach perfect for low-power embedded applications where the execution time is critical, making it suitable for on-the-edge machine learning and edge computing applications [71, 114, 265, 266].

The contributions of the proposed study are as follows:

- Introducing a novel variable latency integer division algorithm derived from the classic non-performing restoring technique;
- Proposing an efficient baseline hardware implementation of the algorithm, based on high-speed Count Leading Zeros (CLZ) units and a single-cycle barrel shifter with the reuse of the same register to store the remainder and the quotient;
- Detailing the hardware architectures of the algorithm in four different variants, specifically designed for targeting different application contexts;
- Comparing all the proposed hardware schemes with the reference designs in literature [2, 264], in terms of average latency, hardware cost, operating frequency and energy consumption on FPGA and ASIC, demonstrating lower latency and energy consumption per division compared to the reference designs;
- Reporting, to the best of our knowledge, the first energy consumption analysis and ASIC synthesis of variable latency dividers in the literature;

- Showing and discussing a comprehensive evaluation of the performance on six benchmark applications.

The rest of the Chapter is organized as follows: Section 3.1 reviews division techniques and analyzes the restoring algorithm that forms the basis of our work. Section 3.2 presents the adopted methodology and the proposed novel restoring algorithm. In Section 3.3, we describe the basic hardware implementation scheme for the proposed algorithm, and three alternative implementation schemes optimized for latency, clock frequency, and hardware resources, respectively. Section 3.4 analyzes the average latency as a dependence of the input operands, by Monte Carlo functional simulation. Section 3.5 reports synthesis and implementation results of all the versions of the algorithm on FPGA using Vivado 2022.2 while Section 3.6 reports ASIC synthesis results. In Section 3.7, we test the performance of each of the proposed dividers using six real application benchmarks, comparing the overall performance with state-of-the-art variable-latency dividers and fixed-latency radix- n dividers. Finally, Section 3.8 discusses the main outcomes of the work.

3.1 Integer Division Algorithms

Integer division involves a dividend A and a divisor B , resulting in two integer values, namely the remainder W and the quotient Q , which are always less or equal to the divisor B and the dividend A , respectively, and satisfy (3.1).

$$A = B \cdot Q + W \quad (3.1)$$

Like all the division algorithms used for comparison with this work [2, 122, 262, 264], we target unsigned integer division. In the case of signed operands, the signs of the results are determined separately from the division operation, according to (3.2).

$$s_Q = s_A \text{ XOR } s_B \quad ; \quad s_W = s_A \quad (3.2)$$

Integer divisions can be implemented in hardware by different algorithms, depending on the design requirements such as computing speed and available resources [123]. The algorithms can be broadly classified into fixed and variable-latency schemes.

In a fixed-latency divider, a constant number of quotient bits is computed in each iteration of the algorithm, starting from the most significant bit (MSB). Like the classic pen-and-paper technique, the algorithm looks for the largest multiple of the divisor that can be subtracted from a partial remainder. The number of iterations of a fixed-latency divider depends on the radix used to represent the dividend and divisor, and not on their values. Due to the minimal hardware requirements, a widely used fixed-latency divider is radix-2, with one quotient bit computed at each iteration, thus taking n clock cycles to perform an n -bit division. Other typical values for the radix are 4, 8, and 16, which imply a constant division latency of $n/2$, $n/3$, and $n/4$ clock cycles, respectively. However, in resource-constrained designs, in both FPGAs and ASICs, the hardware complexity typically limits the implementation to radix-2 and 4 [123][2].

A common way to implement a radix-2 fixed-latency divider is the Restoring Division algorithm [262, 267], which we detail here as a useful basis for discussing

```

1: Input  $\rightarrow$ 
2:   Divider :  $A \in [0, 2^n - 1]$ 
3:   Divisor :  $D \in [0, 2^n - 1]$ 
4:   Partial Remainder :  $W = A$ 
5:   Quotient :  $Q = 0$ 
6:   count = 0
7:
8: Procedure  $\rightarrow$ 
9: while count <  $n$ 
10:    $W = (W \ll 1) - D$ 
11:   if  $W > 0$  then
12:      $Q_{n-1-count} = 1$ 
13:   else
14:      $Q_{n-1-count} = 0$ 
15:      $W = W + D$  # Restoration step
16:   end if
17:   count ++
18: end while

```

Figure 3.1 n -bit Restoring Division Algorithm

the proposed novel schemes. The method is presented in Fig. 3.1. To perform a division between n -bit integer numbers, this algorithm uses two n -bits registers, Q and W , to respectively store the quotient and the partial remainder. W is initially set equal to the dividend. Then, at each iteration, a tentative remainder is computed by left-shifting the W register by one bit and subtracting the divisor. If the tentative remainder is non-negative, the new quotient bit is set to one. Otherwise, the quotient bit is set to zero, and the divisor is added back to W . When the iteration count reaches n , the division finishes with the final quotient stored in the Q register and the remainder stored in W . Notably, in hardware implementations, the registers Q and W are commonly concatenated into a single $2n$ -bits register R , a technique we will refer to in the following.

The main drawback of the approach is the restoration step, which slows the execution. Several solutions have been proposed to avoid this procedure. In the Non-Restoring algorithm [262], the recovery step is omitted in exchange for additional operations and more complexity. In the Non-Performing Divider (NPD) [262], the tentative remainder is stored only when the subtraction result is non-negative. Thus, the addition in the restoring procedure is unnecessary because the R register is updated only when required. However, these two solutions do not change the total latency of the operation, which is fixed and equal to n clock cycles.

Conversely, variable-latency dividers can determine a variable number of quotient bits per iteration, so that the required cycle count for completing the division is not fixed but depends on the input data values. Notably, differently from approximation-based dividers [268], these solutions produce an exact result in a variable number of clock cycles. In [122], three variable-latency integer dividers are proposed starting from the restoring division algorithm. In the first, the subtraction is skipped until the MSB of the dividend reaches the MSW of the remainder. In the second, the

In [2], a similar approach is used to realize a variable-latency integer divider, denoted as Quick-Div. The algorithm was specifically designed for FPGA-based soft processors and compared in detail with fixed-latency radix- n dividers. Similar to [264], this approach performs dynamic shifting of the divisor but exploits a highly optimized hardware design depicted in Fig. 3.2 that uses a Count Leading Zeros (CLZs) technique and splits the shift into two steps. During the first iteration, the divisor is left-shifted by its number of leading zeros, whereas in the remaining ones, it is right-shifted by the number of leading zeros in the remainder. This approach maximizes the operating frequency and allows using a 32-bit register to store the divisor, without losing bits during the shift procedure. The division is complete when the divisor is greater than the current partial remainder. Notably, this requires an additional clock cycle to set the completion signal at the end of the division. In our solution, we avoid this by using an internal counter, which is dynamically incremented without increasing the critical path. Additionally, we use a single 64-bit register for storing the remainder and the quotient on which we perform the dynamic shift without losing bits. The Quick-Div algorithm allows an average number of clock cycles per 32-bit integer division of 1.69, assuming uniform operand distribution. However, the hardware implementation requires a fixed additional latency of two clock cycles: one for storing input signals in registers and one for performing the first shifting iteration, for an average of 3.69. The performance is tested on many application benchmarks, and the implementation results report 365 LUTs, 129 FFs and an operating frequency of 426 MHz, making this architecture the reference design for the performance comparison of the proposed work.

3.2 Proposed Divider

As summarized in Section 3.1, the fixed-latency NPD [262] represents an optimized version of the Restoring divider in which the tentative remainder is stored only in the case of positive subtractions. Whenever the difference between the MSW of $R \ll 1$ and the divisor is negative, the previous value of R simply shifts one position to the left. As a result, several iterations will only perform a left shift of the R register,

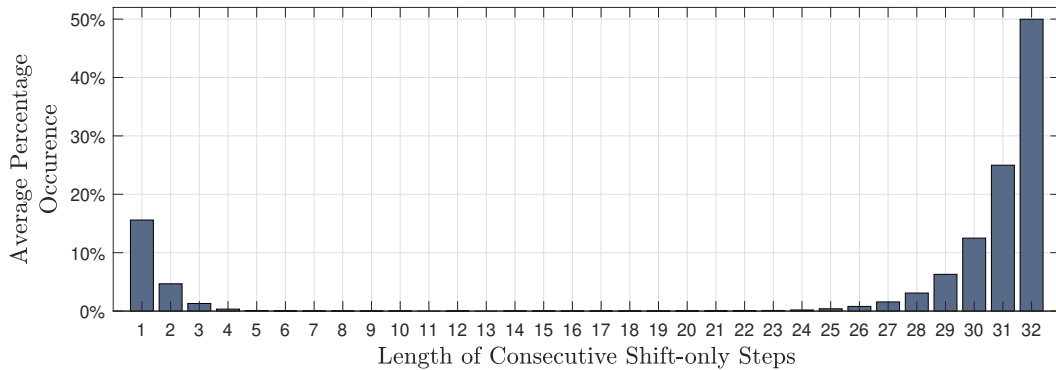


Figure 3.3 Statistical analysis of "shift-only" consecutive steps in 32-bit integer divisions. For more than 98% of the divisions, the consecutive shifts are between 27 and 31, underlying the importance and impact of a dynamic shift mechanism on the NPD.

Dividend A: 0000 0110

Divisor B: 0000 0010

Step 1:

R << 1: 0000 0000 0000 1000 –
 B: 0000 0010 =
 diff: 1111 1110 → R = R << 1
 ⋮

Step 7:

R << 1: 0000 0010 0000 0000 –
 B: 0000 0010 =
 diff: 0000 0000 → R(63:32) = diff, R(0)=1

Step 8:

R << 1: 0000 0000 0000 0010 –
 B: 0000 0010 =
 diff: 1111 1110 → R = R << 1

Result : 0000 0000, 0000 0010
 Remainder Quotient

Figure 3.4 Example of an 8-bit integer division with the NPD. The R register is just shifted to the left for six subsequent steps.

Step 1:

Leading Zeros R: 13

Leading Zeros B: 6

R << (13-6): 0000 0010 0000 0100 –
 B: 0000 0010 =
 diff: 0000 0000 → R(63:32) = diff, R(0)=1

Step 2:

R << 1: 0000 0000 0000 0010 –
 B: 0000 0010 =
 diff: 1111 1110 → R = R << 1

Figure 3.5 Example of an 8-bit integer division with the proposed VLNPD. This approach allows skipping six steps.

while the subtraction result is not used. Fig. 3.3 shows the statistical distribution of consecutive "shift-only" iterations within a 32-bit division using the NPD. In more than 98% of the cases, the division performs 27 to 32 *consecutive* steps in which the R register is just shifted to the left.

The variable latency data-dependent divider proposed in this work is based on the NPD in conjunction to exploiting the relationship between the number of leading zeros in the divisor and in the partial remainder, to dynamically detect and skip the consecutive iterations that would perform only a left shift.

Fig. 3.6 depicts the step-by-step overview of the presented division technique. At each iteration, the CLZ function computes the position of the most significant non-zero bit for the R register and the divisor D . The difference between these two quantities represents the number of division steps in which the subtraction result is certainly negative, and it is used to shift the R register dynamically by

shift_amount bit positions. The term -1 in the calculation of *shift_amount* accounts for a classical division step, which is always performed after the dynamic shift. If *shift_amount* > 0 , an equivalent amount of quotient bits is evaluated in a single step, skipping the same number of "shift-only" steps. The iteration counter is updated accordingly. Note that when *shift_amount* results greater than the remaining division iterations, i.e. $32 - 1 - \text{count}$, the actual shift amount is limited accordingly.

Fig. 3.4 and 3.5 illustrate an example of 8-bit integer division performed with the standard NPD algorithm and the proposed technique, respectively. In the first step of the original algorithm (Fig. 3.4), the difference between the 8 most significant bits of $R \ll 1$ and the divisor D is computed and results to be negative, so R is left-shifted by one bit position. The same situation occurs for the following six iterations. The whole procedure consists of 8 iterations, with subtraction results used in only 2 of them.

In the first step of the proposed algorithm (Fig. 3.5), R and D have 13 and 6 leading zeros, respectively, which means that for $(13 - 6) - 1 = 6$ iterations the difference between the shifted remainder and the divisor will be negative. Thus, R is left-shifted by 6 bit positions and the classic division iteration is executed, obtaining a non-negative difference between the 8 most significant bits of $R \ll 1$ and D . The subtraction result is stored in the 32 most significant bit of R , and the least significant bit R_0 is set to one. In the second step, $\text{count}=7$ and only the classic division step is executed. The whole procedure consists of 2 iterations.

While the presented algorithm can be extended to high-precision computations, from 64 to 1024 bits, the proposed work focuses on medium or low-precision embedded applications, and we go into the details of hardware synthesis of the 32-bit implementation.

In the following, we will refer to the proposed division algorithm and its hardware implementation variants as Variable Latency Non-Performing Divider (VLNPD).

3.3 Hardware Implementation

3.3.1 Baseline Version (VLNPD-Std)

The schematic diagram in Fig. 3.7 depicts the baseline hardware implementation (VLNPD-Std) for the proposed algorithm, with the critical path highlighted in red. The hardware unit has two 32-bit input data signals, an enable signal *Div_enable*, a clock signal, and a reset signal. An output signal *Division_completed* flags that the division has been completed, and the result (quotient and remainder) is available on the output of register R . The architecture includes a 64-bit and a 32-bit registers for R and for the divisor D , respectively, a 6-bit register in the iteration count logic, and a 1-bit output register sampling the *Division_completed* flag, for a total of 103 sequential logic elements, which is exactly the same amount as for the implementation of the NPD. The division starts when *Div_enable* is set, the input operands are stored in the registers, and the counter is set to zero. At each iteration, corresponding to one clock cycle, the value stored in the R register is updated according to Fig. 3.6. A 32-bit CLZ unit is used to find the number of leading zeros in the divisor D , while a 64-bit one is used for the R register. The difference

between the number of leading zeros defines how many bit positions the R register should be dynamically left-shifted, according to Fig. 3.6. When $shift_amount$ is greater than the remaining division steps, i.e. $32 - 1 - counter$, it is accordingly limited, and a control signal named $limited_shift$ is set. Conversely, no dynamic shift is performed when this difference is less than or equal to zero. Finally, the classic division step is implemented by subtraction between the 32 most significant bits of the shifted R register and the divisor.

The effectiveness of the proposed division scheme relies on the availability of a

```

1: Input →
2:   Divider :  $A \in [0, 2^{32} - 1]$ 
3:   Divisor :  $D \in [0, 2^{32} - 1]$ 
4:    $R_{(63 : 32)} = 0 \times 0000$ 
5:    $R_{(31 : 0)} = A$ 
6:   count = 0
7:   division_complete = 0
8:
9: Procedure →
10: while division_complete == 0
11:   leading_D = CLZ( $D$ )  $\in [0, 31]$  # D Leading Zeros
12:   leading_R = CLZ( $R$ )  $\in [0, 63]$  # R Leading Zeros
13:
14:   # Dynamic Shift:
15:   shift_amount = leading_R - leading_D - 1
16:   if shift_amount > 0
17:     if shift_amount > 31 - count
18:       shift_amount = 31 - count
19:     end if
20:      $R = R \ll shift\_amount$ 
21:     count = count + shift_amount
22:   end if
23:
24:   # Classic division step:
25:   difference =  $R_{(62 : 31)} - D$ 
26:   if (difference < 0)
27:      $R = R \ll 1$ 
28:   else
29:      $R_{(63 : 32)} = difference$ 
30:      $R_0 = '1'$ 
31:   end if
32:
33:   if (++count == 32)
34:     division_complete = 1
35:   end if
36: end while

```

Figure 3.6 Algorithmic description of the presented VLNPD.

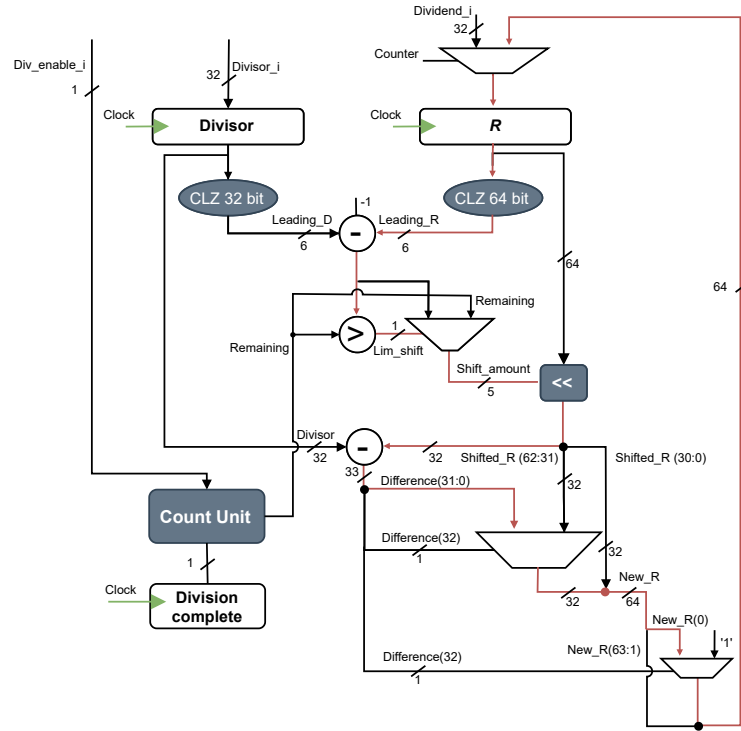


Figure 3.7 Basic hardware implementation for the proposed variable latency divider (VLNPD-Std). The difference between the leading zeros is used to set the dynamic shift. D is subtracted from the output of the dynamic shifter, and the result is used to update R .

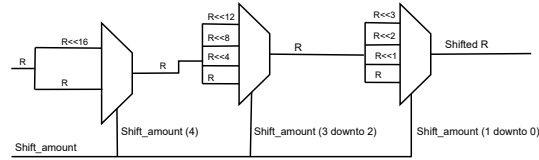


Figure 3.8 Schematic of the dynamic barrel-shifter implemented by a set of three multiplexers.

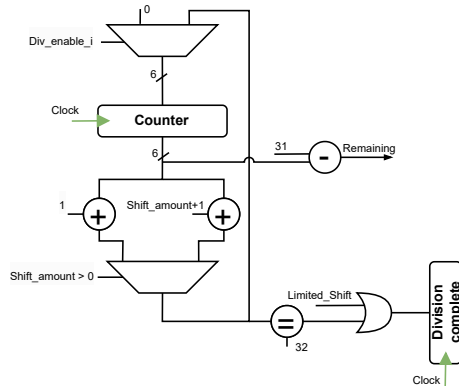


Figure 3.9 Hardware scheme for the iteration Count unit.

fast single-cycle barrel shifter and single-cycle CLZ units, in order to avoid increasing the clock cycle time. The CLZ units were implemented according to [269], featuring a high-speed design tailored for small hardware overhead on FPGAs. The best-performing barrel shifter implementation for the chosen 32-bit operand length was found to be the multiplexing structure shown in Fig. 3.8, taking a 64-bit signal as the input and performing any left-shift between 1 and 31 bit positions. For 64-bit division implementations and above, a better implementation could be a multi-level architecture, as explored in [270]. The Count Unit detailed in Fig. 3.9 updates the iteration count, calculates the remaining steps and sets the *Division_completed* flag bit. If a dynamic shift is performed, the count is updated by adding *shift_amount* + 1 to the previous value; otherwise, it is incremented by 1. When the updated count value reaches 32, or the *limited_shift* control signal is '1', the division ends and the *Division_completed* flag is set. The final remainder and the quotient are stored in the 32 most significant bits and the 32 less significant bits of the *R* register, respectively.

Notably, unlike the architectures presented in [2] and [264], in the proposed implementation, the output control logic uses the updated value of the internal counter to detect the division completion, rather than checking the condition *remainder* < *divisor*. This allows for avoiding an additional clock cycle to set the completion signal without increasing the critical path.

3.3.2 High-Frequency Version (VLNPD-HF)

Since the basic architecture has the CLZ units and the barrel shifter on its register-to-register critical path, a first optimization can be obtained by splitting that combinational path into two different clock cycles for a higher operating frequency, paying in terms of the average latency. In the first cycle, the classic division step and CLZ of the subtraction result are executed, while in the second clock cycle, the dynamic shift is performed, aligning the operands and making them ready again for the next step. Compared to the VLNPD-Std, this solution requires an additional clock cycle every time the operands are not aligned.

The actual hardware implementation is shown in Fig. 3.10. With respect to the baseline implementation, the *R* register has been moved after the calculation of the dynamic shift, a 6-bit register called *CLZ_Remainder* has been inserted to contain the 64-bit CLZ output, a 5-bit one named *CLZ_Divisor* contains the 32-bit CLZ output and a 1-bit register called *shift_en* has been introduced to disable the shifter every time the operands have been already aligned in the previous clock cycle. Whenever *shift_amount* > 0 and *shift_en* = 0, the dynamic shift of *R* is carried out, and the output of the shifter is directly written back into the *R* register instead of being used in the subtractor. At the next iteration, the most significant '1' of register *R* is already aligned with that of the divisor, and the divisor is directly subtracted from the 32 most significant bits of *R*. Note that if *shift_amount* > (32 - *count*), the shifter takes *R* << 1 as input to perform up to 32 iterations in one clock cycle.

Notably, this approach makes it possible to move the remainder's leading zeros count in a different clock cycle with respect to the dynamic shift, decreasing the maximum number of division steps per clock cycle from *shift_amount*+1 to *shift_amount*,

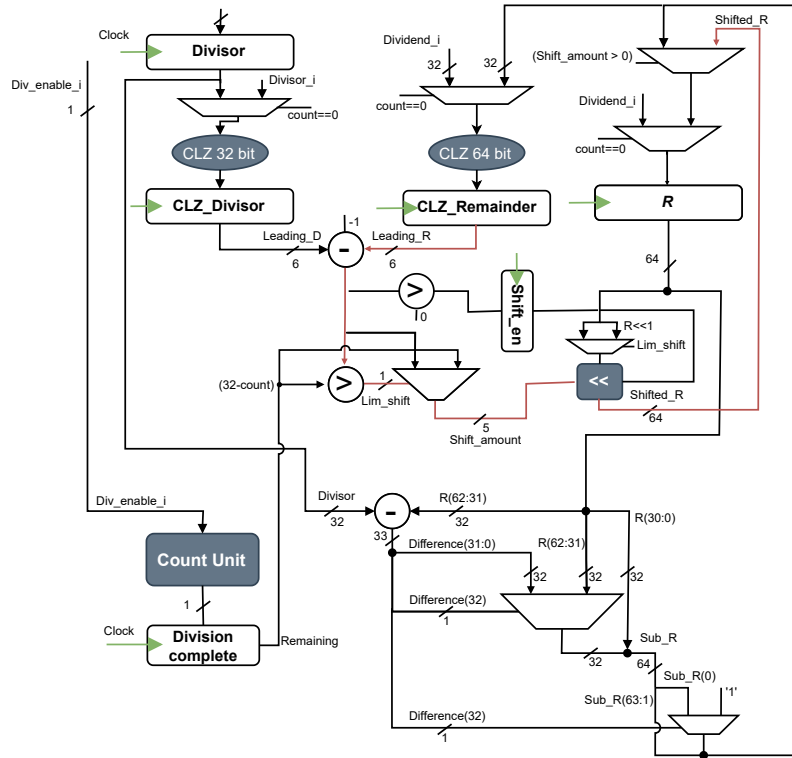


Figure 3.10 Hardware architecture for the High-Frequency variant (VLNPD-HF) of the proposed variable latency divider. The critical path is split as highlighted with red lines at the cost of increased average latency.

splitting the previous critical path and creating the new one highlighted in Fig. 3.10.

3.3.3 High-Performance Version (VLNPD-HP)

In the described VLNPD-Std variable latency divider, the dynamic shift of the R register is done to align its most significant one with that of the divisor. Then, the classic division step is performed, and the sign of the result is used to choose the new partial remainder value. However, it is possible to observe that whether the result of the subtraction is negative after a dynamic shift, at the next clock cycle, the new $R \ll 1$ will certainly be greater than the divisor, the dynamic shift will not be performed, and the results of the subtraction will always be greater than zero. When this condition occurs, R can be directly assigned to the result of the subtraction between $R \ll (\text{shift_amount} + 1)$ and the divisor, saving one clock cycle. Fig. 3.12 depicts this technique on an 8-bit integer division example.

From the hardware point of view, this version is depicted in Fig. 3.11 in which it is possible to highlight that it requires two parallel subtractions at each clock cycle and additional control logic to select which result chooses as new R register. The additional logic results in a higher area occupation and lower frequency (due to the longer critical path length) but allows for a significant reduction of the average latency per division, as will be described in Section 3.4.

positions to the left dynamically at each iteration of the algorithm and is represented by a 5-bit control signal (`shift_amount`). The implementation of this unit is optimized by a series of three multiplexers, as shown in Fig. 3.8, but despite that, it requires 151 LUTs. An alternative divider version with significantly lower area occupation paid in performance can be obtained by limiting the possible shifting range. Observing the statistical distribution of the average number of subsequent shift-only steps in a 32-bit division (Fig. 3.3), it is possible to note that this number appears to be gathered in the range and 24-31. Therefore, reducing the required area for the dynamic shifter is possible by limiting the shift in this range. The performance deviation with this solution depends heavily on the application domain, as will be discussed in Section 3.7, but from a statistical point of view, this solution allows a significant reduction in the hardware resources in exchange for a small deviation in the division average latency.

3.4 Results

This section analyzes the performance in terms of clock cycle latency for each variant of the presented VLNPD, to verify the effectiveness of the proposed approach. Subsection 3.4.1 analyzes the properties of the latency as a function of the input data, showing how it changes with the amplitude ratio between the operands. In subsection 3.4.2, Monte Carlo simulations for 4, 6, 8, 12, 16, and 32-bit integer divisions evaluate the average latency of the proposed dividers for uniformly distributed input values, for direct comparison with the other dividers available in the literature.

3.4.1 Latency behavior analysis

All the VLNPD implementation variants have data-dependent latency, meaning that the number of clock cycles required to complete a division depends on the input data values. Here we discuss some particular cases to understand the expected behavior of the VLNPD:

- *dividend < divisor*: If the dividend is smaller than the divisor, the correct result is the quotient equals zero, and the remainder equals the dividend. However, it is unnecessary to check this condition, as the number of leading zeros in the dividend will certainly be less than in the divisor. In the first iteration, the dividend is placed in the LSW of the R register, the shift amount is always greater than or equal to 32, and the shift limiting mechanism is enabled, shifting *R* left by 31 positions and subtracting the divisor. This last step always returns a negative result, resulting in an additional shift of *R*. In this way, in a single clock cycle, the dividend is shifted from the LSW to the MSW, becoming the remainder of the division. On the other hand, the quotient in the LSW, is zero;
- *dividend = divisor*: when the dividend and divisor are equal, the difference between the leading zeros is exactly 32. As in the previous case, these divisions automatically end in one clock cycle with the difference that the result of the final subtraction will be non-negative and equal to zero. Consequently, the quotient is set to 1 and the remainder to zero, which gives the correct result;

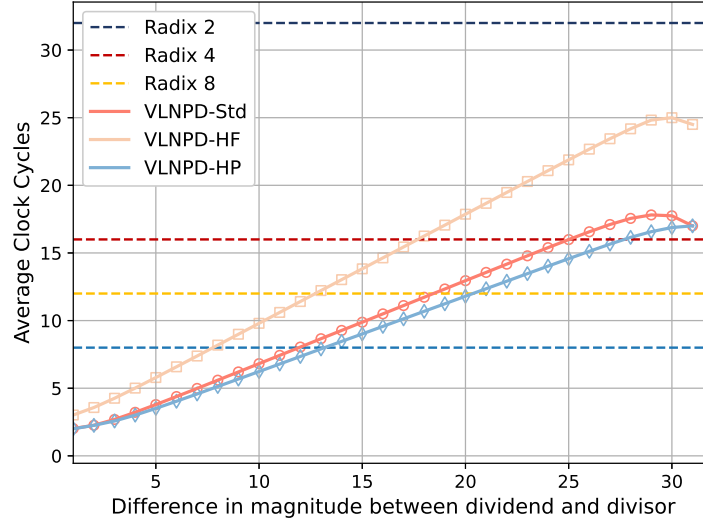


Figure 3.13 Trend of the VLNPD's average clock cycle against the difference in magnitude (as powers of two) between the operands.

- $dividend = 1$: the special case of the dividend equal to 1 may fall into one of the two previous cases, and both are completed in one clock cycle, as described;
- $divisor = 1$: this case is the most critical for variable latency algorithms since this division requires as many clock cycles as the number of '1's in the dividend. The latency for these cases ranges from 1 (when dividend equals 1) to 32 (when dividend equals $2^{32} - 1$). However, it is important to note that, unlike other architectures in the literature, we can solve this situation by checking whether the number of leading zeros in the divisor equals 31; still, we did not implement that to avoid adding extra hardware overhead. By adding this extra condition to the `division_complete` flag, divisions by one are completed in one clock cycle. The produced result is automatically the correct one since, in the first iteration, the quotient is equal to the dividend and the remainder is equal to zero;
- $\log_2(dividend) = \log_2(divisor)$: if the base-2 logarithm of the dividend is equal to that of the divisor, the operands have the same n number of leading zeros. At the first iteration, R has $n + 32$ leading zeros, and it is, then, dynamically left-shifted by $(n + 32) - n - 1 = 31$ positions. The most significant ones are aligned in this way, and the final subtraction gives the correct result of the operation. In this case, the difference is always positive (because $dividend > divisor$ and $\log_2(dividend) = \log_2(divisor)$): the quotient is set to 1, and the result of the subtraction is the new remainder;
- $(dividend/divisor) \bmod 2 = 0$: if the dividend is equal to the divisor multiplied by any power of two, the proposed algorithm performs integer division in two clock cycles. In the first one, the leading ones are aligned, the result of the subtraction is zero, and the quotient bit is set to one. After this operation, all subsequent division steps consist only of a left shift of the R register and

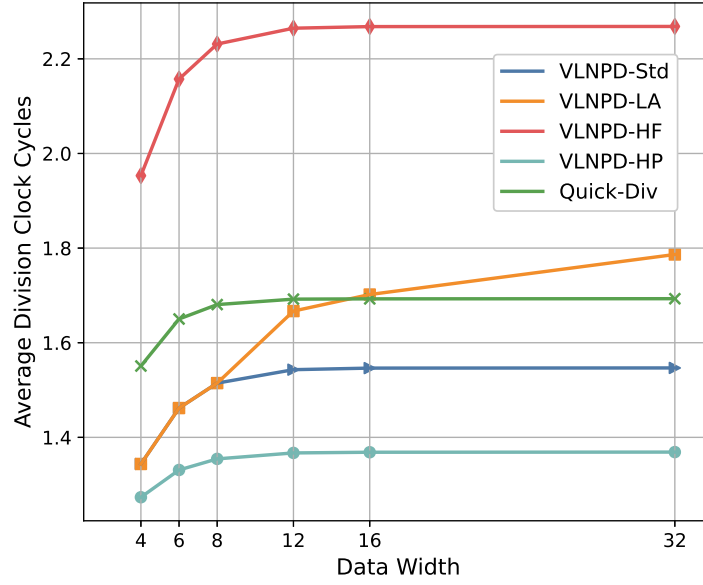


Figure 3.14 Average clock cycles under varying bit width of operands.

can be dynamically performed in the second clock cycle. Note that this case also includes the divisions where both the dividend and the divisor are powers of two.

Fig. 3.13 shows how the average number of cycles required by the proposed algorithm varies as a function of the distance (in powers of 2) between the dividend and the divisor. For distances between 0 and 2^{13} , the clock cycles required by the VLNPD-Std algorithm are always less than or equal to 8, the latency expected from a radix-16 divisor. After this value, the number of cycles remains less than 12 (latency of a radix-8) up to a distance of 2^{19} . The VLNPD-Std variable latency divider is also better than a radix-4 one for distances up to 2^{25} . The number of clock cycles as a function of the distance is lower in the VLNPD-HP version and higher in the VLNPD-HF one, for the reasons outlined in Section 3.3.

3.4.2 Monte-Carlo Simulation Analysis

To compare and evaluate the performance between the proposed algorithm and the fixed latency solutions, we tested more than 10^{10} dividend and divisor pairs through Monte-Carlo simulations in the case of 4, 6, 8, 12, 16 and 32 bit operand width. Fig. 3.14 shows the results obtained for each version. The results were obtained by simulating until the first and second decimal places of the average latency remained constant over 10 billion iterations. In all the implementation variants the increase in the average latency becomes negligible for bit-widths greater than 12, except for the case of VLNPD-LA in which the limited range of the shifter causes a steeper increase, yet remaining below 1.8 cycles. The VLNPD-Std version of the proposed divider presented in Section 3.2 requires an average number of clock cycles converging at a value of 1.55. This average latency is achieved thanks to the presented dynamic shifting method. In detail, Fig. 3.15 reports the clock cycle distribution required

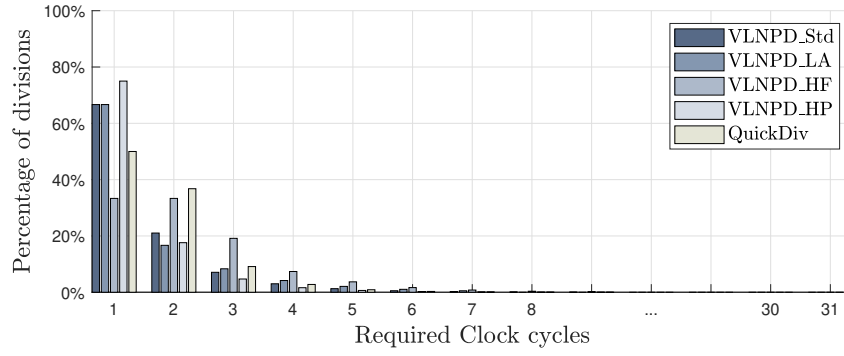


Figure 3.15 Statistical distribution of division latency in 32-bit integer divisions. VLNPD-Std: 66.7% of divisions completed in 1 clock cycle, 21% in 2 cycles, 7% in 3 clock cycles. VLNPD-HP: 75% in 1 clock cycle.

Table 3.1 Average latency of integer dividers for uniformly distributed numbers, including the ones required by the hardware implementation

	Average Latency
NPD	33.000
VLNPD-Std	2.546
VLNPD-LA	2.786
VLNPD-HF	3.268
VLNPD-HP	2.370
Quick-Div [2]	3.693
Priority Encoders [264]	3.730

by a 32-bit integer division performed with the VLNPD-Std algorithm. Overall, 66.67% of the divisions are completed in one clock cycle. The 21% and the 7% of divisions are completed in two and three clock cycles, respectively, while the number of divisions requiring more than five clock cycles is less than 1%. The optimization technique described in Section 3.3.3 allows a lower average number of clock cycles for the VLNPD-HP version, which converges at 1.36. In this case, thanks to the additional step that can be performed at each dynamic shift, 75% of the divisions are completed in one clock cycle. In contrast, the average latency increases in the VLNPD-HF divider, reaching a value of 2.27, in exchange for a higher operating frequency. As explained in Section 3.3.2, this is due to the additional cycle at each dynamic shift, which allows the splitting of the critical path. With the VLNPD-LA version, the number of clock cycles statistically slightly increases and reaches 1.78. As mentioned before, the performance of this version strongly depends on the application, as will be detailed in Section 3.7. Finally, Fig. 3.14 also shows the performance of Quick-Div[2], which exhibits to an average number of clock cycles of 1.69 and completes 50% of divisions in one clock cycle.

All the results reported in Fig. 3.14 refer to the algorithmic-level performance, assuming the operands are already available in the local input registers and without

considering any additional latency overhead related to the implementation. For the architecture in [2], in fact, one more cycle is required to split the shift into two steps and increase the operating frequency. The average number of clock cycles, including the ones required by the hardware implementation, are summarized in Table 3.1 for all the VLNPD variants and the designs in [2], and [264].

Notably, in VLNPD schemes, the latency might be further reduced by avoiding one clock cycle to load the operands in internal registers, introducing a multiplexer to directly pass the dividend at the input of the shifter and the CLZ unit. In the present analysis, we did not implement this further improvement to limit the hardware overhead and to assume the same operation setup for all the compared dividers.

3.5 Implementation on FPGA

We synthesized and implemented the VLNPD divider, in all its variants, on the Xilinx Virtex UltraScale+ VCU118 board (XCVU9P-L2FLGA2104E) using Vivado 2022.2. We also replicated the design in [264], for which the data available in the literature are related to a Virtex-7 FPGA board. Table 3.2 reports the results of the implementations in terms of hardware resources, i.e. Look-Up Tables (LUTs) and Flip-Flops (FFs), operating frequency, average execution time per division at the maximum operating frequency, and average energy consumption per division.

The average execution time per 32-bit division is computed according to (3.3) and is reported in Table 3.2.

$$Execution_time = \frac{average_latency}{frequency} \quad (3.3)$$

The average dynamic energy consumption per division and average static energy consumption per division is calculated according to (3.4) and (3.5).

$$E_dynamic = \frac{P_{dynamic}}{frequency} * average_latency \quad (3.4)$$

$$E_static = \frac{P_{static}}{frequency} * LUT\% * average_latency \quad (3.5)$$

where $P_{dynamic}$ and P_{static} are the dynamic and static power data, respectively, which have been obtained by the Vivado power estimation based on the switching activity trace file extracted from the gate-level simulation of actual division operations; $LUT\%$ is the percentage of the total LUTs of the device occupied by the division unit. For the design in [2], the energy results were not available in the literature, and therefore, they were produced by replicating the microarchitecture on the same target FPGA.

The original NPD shows an operating frequency of 800 MHz and a very low area occupancy. It requires an average execution time of 41.25 ns and an average dynamic energy of 2969.01 pJ due to its fixed latency of 33 cycles.

The VLNPD designs significantly improve the execution time and the average dynamic energy per division, at the cost of more hardware resources. The VLNPD-Std version requires 385 LUTs and 103 FFs with an operating frequency of 300 MHz .

Table 3.2 Operating frequency, execution time, area and energy consumption results obtained on Xilinx Virtex UltraScale+ VCU118 board

	Operating Frequency [MHz]	Number of LUTs	Number of FFs	Average Dynamic Energy per Division [pJ]	Average Static Energy per Division [pJ]	Average Execution Time per division [ns]
NPD	800	136	103	2969.010	10.189	41.250
VLNPD-Std	300	385	103	178.270	6.290	8.489
VLNPD-LA	315	290	103	195.056	4.370	8.846
VLNPD-HF	460	463	115	326.846	7.020	7.105
VLNPD-HP	268	438	103	165.820	8.733	8.839
Quick-Div [2]	426	365	129	332.391	6.424	8.670
Priority Encoders [264]	286	446	103	261.100	13.162	13.321

Table 3.3 Operating frequency, execution time, area and energy consumption results on Synopsys Fusion Compiler in GF22FDX technology.

	Operating Frequency [GHz]	Total cell Area [μm^2]	Average Dynamic Energy per Division [pJ]	Average Static Energy per Division [pJ]	Average Time per division [ns]
NPD	1.66	310.97	13.464	1.171	19.800
VLNPD-Std	1.00	727.10	3.081	0.686	2.546
VLNPD-LA	1.06	619.27	2.111	0.484	2.451
VLNPD-HF	1.40	750.80	2.735	0.665	2.334
VLNPD-HP	0.98	893.30	3.375	1.008	2.407
Quick-Div [2]	1.22	722.78	3.690	0.775	3.026

Despite the lower frequency, thanks to an average of 2.55 cycles per division, it exhibits an average execution time of 8.489 *ns*, providing a speedup of $4.86\times$ over the original algorithm and $1.02\times$ compared to [2]. The latency difference also affects the average dynamic energy, which in the case of the VLNPD-Std equals 178.270 *pJ*, 93.99% less than the original division algorithm and 46.37% less than [2]. These values are also lower than all the compared variable latency architectures, making the proposed VLNPD the most performing in terms of the average latency, average execution time, and average static and dynamic energy.

The VLNPD-HF version described in Section 3.3.2 allows the operating frequency to be increased by 53.33% over the VLNPD-Std version, resulting in a value of 460 *MHz*. This version requires 20.25% more LUTs, 12 more FFs, and a higher average number of clock cycles (Table 3.1). Nevertheless, this architecture has the highest frequency and the lowest execution time of all the compared variable latency dividers. In fact, the VLNPD-HF version exhibits an average execution time speedup of $5.80\times$ over the original algorithm, $1.20\times$ compared to the VLNPD-Std version and $1.22\times$ over [2].

The VLNPD-LA design described in Section 3.3.4 reduces the number of LUTs by 24.67%, making this version the one with the smallest area occupation and the lowest static energy per operation.

Finally, although the VLNPD-HP version has the lowest operating frequency and the highest hardware resource utilization, it has the lowest average latency per division, providing significantly better performance.

3.6 Implementation on ASIC

We further evaluated the ASIC implementation of the proposed VLNPD dividers by synthesizing them with Synopsys Fusion Compiler on GlobalFoundries 22FDX (GF22FDX) technology. Table 3.3 contains the comparison between the proposed dividers and the existing reference dividers, again in terms of area, dynamic and static energy consumption per division operation, maximum operating frequency in typical process corner, and resulting average time per operation. Also, for the ASIC implementation, energy consumption data were obtained by the power calculator tool based on switching activity trace files extracted from gate-level simulations of real operations.

With respect to the original NPD, the four VLNPD dividers proposed in this work provide an average execution time speedup that ranges from $7.77\times$ for the VLNPD-Std version to $8.48\times$ for the VLNPD-HF version. Correspondingly, the average energy per division operation is reduced by 77.12% for the VLNPD-Std implementation up to 84.32% for the VLNPD-LA implementation.

To compare with the reference variable-latency design Quick-Div, we replicated and synthesized the design reported in [2] on the target technology. The execution time speed-up obtained by the VLNPD dividers ranges from $1.18\times$ to $1.29\times$, while the average energy per division operation is reduced by 16.50% up to 42.79%.

The VLNPD-Std version of the variable latency divider shows an operating frequency of 1 GHz (40% less than the original algorithm), which is increased up to 1.40 GHz (only 15.66% less than the original algorithm and 12.83% higher than the reference design Quick-div) by the VLNPD-HF architecture. The area occupation range from $619.27 \mu m^2$ for the VLNPD-LA design to $893.30 \mu m^2$ in the VLNPD-HP occupying less area than the reference design Quick-div. Overall, the VLNPD-HF Variable Latency Divider results to be the best version for ASIC implementation.

3.7 Performance analysis on benchmark programs

Since the actual speed of variable latency dividers is data-dependent, it is relevant to explore the performance of the proposed design as well as other reference designs in the execution of real-world computation kernels. We implemented the execution of six representative benchmarks in C++ using the target division algorithms. Also, since it is very likely that in a system-on-chip architecture, the critical path that imposes the clock frequency is not in a small sequential divider, we compared the actual performance of the dividers in a continuous range of frequencies from 100 MHz up to the maximum operating frequency specific to each divider. The range of considered clock frequencies is representative of an FPGA implementation; equivalent results may be obtained for frequency ranges related to ASIC implementation.

3.7.1 Benchmark set

Details on the six benchmark routines adopted for the analysis are the following:

RNG: pseudo-random number generation function. At each iteration, a new random number, X_{n+1} , is calculated starting from the previous one, according to (3.6). Like the `minstd_rand` functions in C++, we used $m = 2^{31} - 1$, $a = 16807$ and

$c = 0$. In this benchmark, 23% of the executed instructions are divisions.

$$X_{n+1} = (aX_n + c) \bmod m \quad (3.6)$$

SQRT: Newton-Raphson method used to find the approximate value of the square root of a number x according to (3.7). The iterative method starts with an initial guess g and uses it to improve the estimation, g' , until the desired accuracy is achieved. In this routine, 26% of the operations are divisions.

$$g' = 0.5 * (g + x/g) \quad (3.7)$$

PRIME: function that checks whether an integer input n is a prime number. If at least one of the divisions between n and every possible divisor in the range $[0, \sqrt{n}]$ has a remainder equal to zero, the number is not prime. In this benchmark, 25% of instructions are divisions.

RSA: function for decrypting text files using RSA technique (3.8), [271]. The private key is indicated with d , and the modulus is indicated with n . The percentage of divisions in this benchmark depends on n ; on average, circa 13% are divisions.

$$m = (c^d \bmod n) \quad (3.8)$$

LU_FACT: function for performing the LU factorization of fixed-point matrices. This technique is used for efficiently solving linear systems and inverting matrices in AI tasks such as regressions and recommendation systems, and is the method adopted in this thesis to perform the matrix inversions required by linear contextual bandits (Chapter 5). For 3×3 matrices, in this benchmark 16% of the operations are divisions.

IMDIV: pixel-wise image division used to report the fractional change or ratio for each pixel. In this benchmark, 33% of operations are divisions.

3.7.2 Results

The results of the analysis are expressed as the average division absolute execution time when running the benchmark routines, in the examined clock frequency range. Fig. 3.16 summarizes the obtained data. The general most relevant outcome of the analysis is that - in all the benchmarks - the proposed VLNPD dividers can obtain the same or better average execution time at a frequency lower than the other compared dividers. This is particularly relevant as it would allow running the entire system-on-chip architecture - in which the divider is to be integrated - at a lower frequency while maintaining the same division operation performance, in the view of low power consumption. Other specific details on the results are discussed below. In the RNG benchmark, the VLNPD-Std version of our variable latency divider shows an average number of clock cycles of 2.49, providing a speedup equal to $13.25\times$ over the original algorithm and $1.40\times$ over Quick-div. The low average is due to how this benchmark works. Considering that in (3.6), m is equal to $2^{31} - 1$, this function performs two types of division:

1. $dividend \leq m$: as described in Section 3.4.1, the proposed algorithm always completes these divisions in one clock cycle;

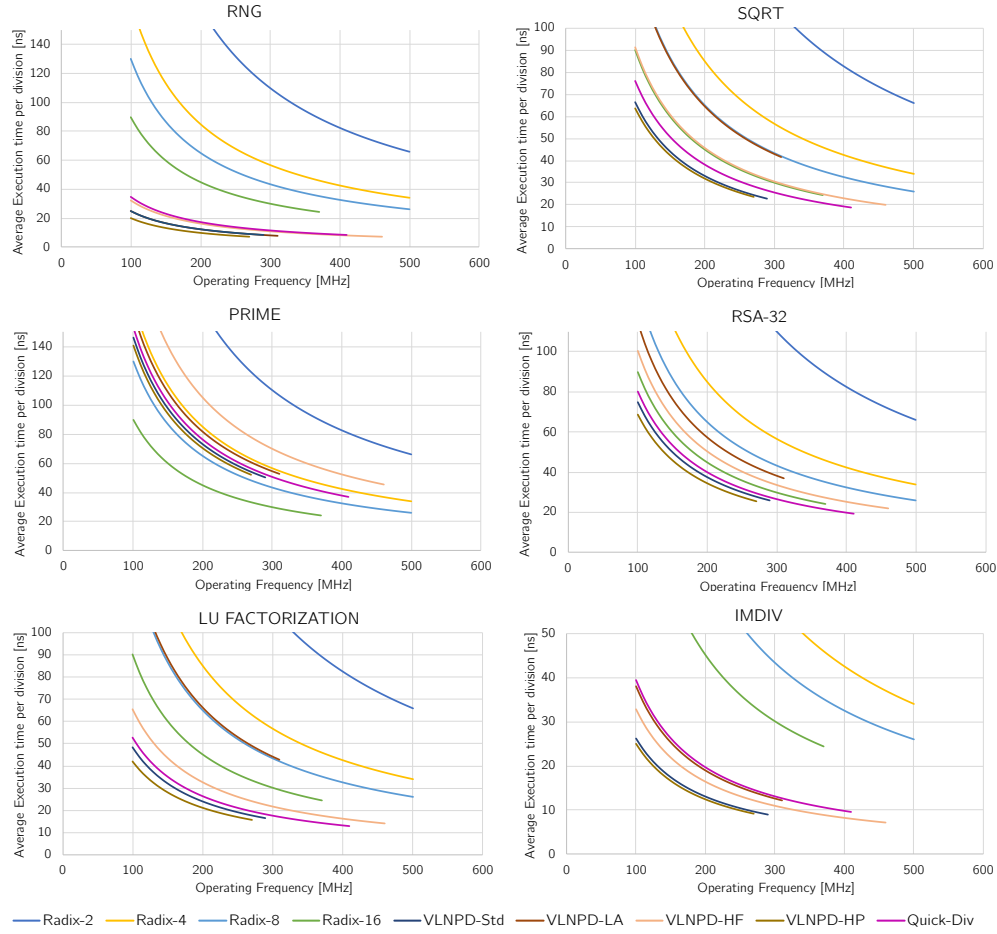


Figure 3.16 Average execution time per division required in six different benchmarks as a function of the operating frequency

2. $2^{31} < \text{dividend} < 2^{32}$: in this case, the distance (in powers of two) between the dividend and the divisor can be at most 1, and these divisions require two clock cycles (Fig. 3.13). Such a range for the dividend means that the first performed dynamic shift equals $32 - 1 - 1 = 30$. The subtraction result will certainly be negative and another shift will be required. However, this is exactly the case optimized in the VLNPD-HP version that, in this benchmark takes an average of 1 clock cycle.

To this quantity, an additional clock cycle must be added due to the hardware implementation, as described in Section 3.4. This discussion also explains why the VLNPD-LA exhibits the same performance of the VLNPD-Std in this benchmark.

In the SQRT benchmark, the latency increases with the difference in magnitude between input operands, reaching its maximum in the last iteration when the estimation g' matches the actual number's square root. In this case, the following condition holds: $\log_2(g') = \frac{1}{2}\log_2(x)$, meaning that the maximum latency according to Fig. 3.13, will be equal to 10 clock cycles. On average, the VLNPD-Std version requires 6.65 clock cycles for this benchmark, providing a speedup equal to $4.96\times$

compared to the NPD and $1.15\times$ over Quick-div.

The PRIME benchmark is the worst for variable-latency dividers since the dividend is fixed and the divisor varies in $[2, \sqrt{n}]$. Also, the procedure is interrupted on the first division with a non-zero remainder, meaning that, in most cases, only a small part of the available range is tested. For this benchmark, the VLNPD-Std presents an average clock cycle of 14.61, providing a speedup of $2.26\times$ and $1.06\times$ over NPD and Quick-div, respectively. Note that this example shows how the workload strongly impacts the performance of the VLNPD-LA version that is almost never capable of performing the dynamic shift.

In the RSA benchmark, no particular observations are required since the latency depends on the distance of the powers of c and the modulus n . The VLNPD-Std presents an average number of clock cycles equal to 7.49, with a speedup equal to $4.41\times$ compared to NPD and $1.07\times$ over Quick-div.

In the LU_FACT routine, the elements of the lower triangular matrix are computed through the division by the corresponding diagonal elements of the upper triangular one. In this benchmark, the VLNPD-Std requires 4.82 clock cycles, with a speedup equal to $6.84\times$ relative to the NPD and $1.1\times$ over Quick-div.

Finally, in the IMDIV program, the division is usually performed between two pixels with similar values. The VLNPD-Std has an average of 2.62 cycles and a speedup of $12.60\times$ compared to NPD and $1.50\times$ over Quick-div.

3.8 Summary of Findings

The presented variable latency data-dependent division exploits the relationship between the leading zeros in the divisor and the partial remainder to reduce the average execution time and energy consumption. These features make it suited for low-power embedded applications with high-speed requirements.

We presented the algorithm, its hardware implementation and a detailed performance analysis to evaluate its effectiveness. Experimental results show that the approach achieves an average of 1.55 clock cycles per 32-bit integer division, providing a speedup of $20.65\times$ over the starting NPD algorithm, and in the range of $2.26\times$ up to $13.25\times$ when applied to real benchmarks, depending on the input data.

We illustrated four implementation versions of the presented divider, and we synthesized all of them on a Xilinx Virtex UltraScale+ VCU118 FPGA, as well as on ASIC GF22FDX technology. The FPGA VLNPD-Std version achieved the lowest execution time per division in the literature with an average of 8.489 *ns*, also reaching the lowest average dynamic energy consumption, with a reduction of 93.99% compared to the original algorithm and 46.37% over the reference design. The same properties were observed on ASIC in GF22FDX technology, with an execution time of 2.546 *ns* and dynamic and static energy consumption of 3.081 *pJ* and 0.686 *pJ*, respectively. The VLNPD-HF version improved the operating frequency by 53.33% on FPGA and 40% on ASIC, leading to a further reduction in the average execution time compared to the VLNPD-Std version, with 7.105 *ns* on FPGA and 2.334 *ns* on ASIC despite a higher average latency in clock cycles. The best performance in terms of latency was achieved by the VLNPD-HP version with an average of 1.36 clock cycles and the 75% of divisions completed in just one iteration. In contrast, the

VLNPD-LA version reduced the area occupation by 26.67%, leaving performance highly dependent on the target application.

The availability of different versions allow selecting the implementation according to the system requirements. Overall, the VLNPD-Std is the most versatile hardware implementation of the algorithm that balances operating frequency, power consumption and hardware requirements, offering a general excellent alternative to fixed-latency solutions in embedded systems. The VLNPD-HF version should be preferred when the system performance is limited by the operating frequency of the variable latency arithmetic units. The VLNPD-HP results particularly suited when clock frequency is imposed by other parts of the system, so that minimizing the clock cycle count per division is primary. Finally, the VLNPD-LA version is recommended in scenarios with very strict hardware resource constraints and the workload profiling can be used to tune the shifter range.

Chapter 4

Exploring Variable Latency Dividers in Vector Accelerators

Chapter based on Paper 2 [272]: Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, and Mauro Olivieri. Exploring variable latency dividers in vector hardware accelerators. In 2024 19th Conference on Ph. D Research in Microelectronics and Electronics (PRIME), volume 1, pages 1–4, Larnaca, Cyprus, 2024. IEEE, Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/PRIME61930.2024.10559734

"Great things are not done by
impulse, but by a series of small
things brought together."

Vincent Van Gogh

The development of the VLNPD divider in the previous chapter marked a significant milestone in this thesis, offering a novel solution to accelerate integer division in low-power, real-time systems. Designed to dynamically skip redundant cycles by analyzing partial remainders and leading zero patterns, the VLNPD demonstrated that division, often considered prohibitively expensive for embedded AI, can be substantially optimized in both average latency and energy consumption.

Motivated by the success of variable-latency division in the scalar execution context, this chapter explores its potential in the parallel execution model of vector accelerators. Vector hardware accelerators are specialized computing units, usually organized into multiple vector *lanes*, each equipped with arithmetic functional units that enable the parallel execution of the same instruction on different data in a Single Instruction Multiple Data (SIMD) fashion. These accelerators are widely adopted in embedded systems [273, 274] when the area, power, and energy requirements of GPUs are prohibitive. Among the basic operations supported by vector accelerators, integer division emerges as the slowest and most challenging to implement in hardware: its long and input-dependent latency can complicate lane synchronization, scheduling, and throughput predictability—potentially limiting the performance gains that vectorization is meant to deliver. Yet division remains essential in many embedded workloads, from matrix factorization to normalization and fixed-point post-processing, making its efficient realization in vector hardware critical.

These considerations motivate the research question guiding this chapter:

? Research Question

How do variable-latency integer dividers behave when integrated into vector accelerators, in terms of throughput, latency, area overhead, and the resulting performance-efficiency trade-offs?

To answer this question, we integrate the VLNPD into the Klessydra-T13 Vector Coprocessor Unit (VCU) [114, 273], optimizing its implementation through strategic hardware reuse and extending the instruction set to support vector divisions. The contributions of this chapter are twofold:

- presenting, to the best of our knowledge, the first investigation into variable-latency division within vector hardware accelerators;
- illustrating an optimized integration strategy via hardware reuse that supports multiple data widths with minimal area overhead.

As we will show, this integration not only yields substantial speedups on general benchmarks, but also dramatically reduces the contribution of division to the overall execution time of linear bandit workloads—from 70% on the scalar core to approximately 12% in the vectorized implementation—clearing the path for the algorithmic optimizations of Chapter 5.

The remainder of the chapter is organized as follows. Section 4.1 introduces the target platform and motivates the selection of the VLNPD variant for integration. Section 4.2 describes the proposed ISA extension and hardware reuse strategy. Section 4.3 evaluates performance through Monte Carlo simulations and real-world benchmarks. Finally, Section 4.4 summarizes the main outcomes.

4.1 Context Analysis

The Klessydra-T13 is a parametric RISC-V core that executes three hardware threads in an Interleaved Multi-threading fashion, incorporating a highly reconfigurable VCU with dedicated Scratchpad memories (SPMs) for high-performance vector [275, 276]. The VCU can be configured in terms of Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP) to meet the computational demands of specific applications [71]. A custom instruction set extension, integrated into the RISC-V GCC compiler toolchain, supports key arithmetic operations on vectors.

Critically for this work, the Klessydra-T13 VCU natively supports multiple data widths—8, 16, and 32 bits—enabling the system to execute either one 32-bit, two 16-bit, or four 8-bit arithmetic operations per vector lane. This flexibility imposes a key constraint on divider integration: supporting all data widths naively would require dedicated division units for each configuration, significantly increasing area overhead.

Chapter 3 introduced the VLNPD algorithm and demonstrated its superiority over fixed-latency and competing variable-latency dividers in scalar contexts. However, integrating a division unit into a vector accelerator introduces additional constraints: the divider must be efficiently replicable across multiple lanes while maintaining acceptable latency and not entering the critical path.

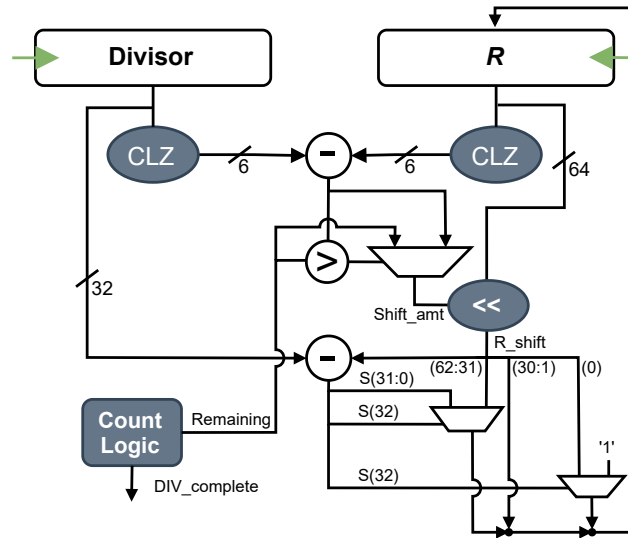


Figure 4.1 Hardware implementation of the VLNPD-Std. The difference between the leading zeros is used to perform the dynamic shift. The *Divisor* is subtracted from the output of the dynamic shifter, and the result updates R .

To identify the most suitable implementation, we analyzed the four VLNPD variants from Chapter 3 alongside classical radix- n dividers and state-of-the-art variable-latency alternatives. For latency evaluation, we performed Monte Carlo simulations over 10 billion uniformly distributed operand pairs. For area and frequency, we synthesized each architecture on the Xilinx Virtex UltraScale+ VCU118 using Vivado 2022.2 with default settings.

Table 4.1 summarizes the results. All the dividers operate safely above the 145 MHz operating frequency of the Klessydra-T13 core [114], ensuring their integration does not increase the critical path.

Radix-2 and Radix-4 dividers offer minimal hardware demands (136 and 240 LUTs) but their fixed latencies of 33 and 17 cycles make them not ideal for accelerators designed to minimize the execution time of target applications. Radix-8 and Radix-16 reduce latency to 12 and 9 cycles but at significantly increased area (503 and 710 LUTs), posing challenges for efficient replication across vector lanes.

Variable-latency solutions offer an interesting trade-off between resources and latency. The VLNPD-LA model offers the lowest area requirements, but its application-specific nature [250] could limit the performance of a variable-latency approach in a general-purpose vector hardware accelerator. The VLNPD-HF and Quick-Div models excel in optimizing operating frequency, achieving impressive values of 460 MHz and 426 MHz, respectively. However, this focus misaligns with our primary objectives, since the core’s clock frequency is predetermined by other units, as previously discussed. The VLNPD-HP divider stands out for achieving the lowest latency in the literature for 32-bit integer division, averaging 2.37 clock cycles, but it requires 438 LUTs. Finally, the VLNPD-Std emerges as the most versatile hardware implementation of the VLNPD algorithm, presenting a balance between latency and hardware demands with an average latency of 2.54 clock cycles and 385 LUTs. For these reasons, VLNPD-Std was selected for integration into the Klessydra-T13 VCU.

Table 4.1 Operating frequency, latency and area results obtained on Xilinx Virtex Ultra-Scale+ VCU118 board.

	Operating Frequency [MHz]	Number of LUTs	Average Latency [Clock Cycles]
Radix-2	800	136	33
Radix-4	650	240	17
Radix-8	470	503	12
Radix-16	330	710	9
VLNPD-Std	300	385	2.546
VLNPD-LA	315	290	2.786
VLNPD-HF	460	463	3.268
VLNPD-HP	268	438	2.370
Quick-Div [2]	426	365	3.693
Priority Encoders [264]	286	446	3.730

4.2 Integration Through Hardware Reuse

Fig. 4.1 describes the implemented dedicated division functional unit within the accelerator, replicated over the vector lanes. To handle the variable latency between the operations across the vector lanes, we developed a synchronization mechanism that ensures the overall latency for vector divisions is determined by the slowest operation in the SIMD set.

As introduced in section 4.1, since the VCU supports a versatile range of data widths, including 8, 16, and 32 bits, supporting vector divisions for varying data widths should require dedicated units for each lane significantly increasing the area overhead. Given the inherent complexity of integer division, which cannot be factorized like other arithmetic operations, the integration of the VLNPD-Std architecture plays a pivotal role by allowing for the strategic reuse of hardware resources. The VLNPD rely on the high-speed hierarchical CLZ units proposed in [269], which combines multiple 8-bit CLZ instances—four for 32-bit operations and eight for 64-bit operations— enabling their reuse across divisions of 8, 16, and 32-bit data widths. Additionally, all the registers in the design can be reused (*R* and *Divisor*), except for the counters, while the rest of the logic should be replicated.

Table 4.2 shows the synthesized hardware resources required for the VLNPD-Std architecture at different data widths, including the Look-Up Tables (LUTs) and Flip-Flops (FFs). In this Table, The *Expected Total* indicates the resources that would be required if four 8-bit, two 16-bit, and one 32-bit VLNPDs were instantiated independently. In contrast, the *Total* reveals the actual resources utilization. The strategic reuse of hardware resources allows for saving about 340 LUTs and 230 FFs for each lane, resulting in a significant reduction of the hardware overhead as the SIMD grows.

To support vector integer divisions, we introduced new instructions (in Table 4.3) into the Klessydra-T13’s RISC-V custom instruction set extension presented in [114]. These instructions, integrated into the GCC compiler toolchain, facilitate the

Table 4.2 Hardware Resources for the VLNPD, varying the Bit-width

Data Width	#LUTs	#FFs
8	105	35
16	190	61
32	385	103
Expected Total	1185	365
Total	845	135

division operations between vectors located in the SPMs, the operations between a vector and a scalar, and the computation of remainders, all performed in a SIMD fashion.

Table 4.3 Vector Division Instructions in Klessydra Instruction Set

Assembly Syntax	Short description
kvdiv (rd), (rs1), (rs2)	Division between two vectors located in SPMs.
ksvdivsc (rd), (rs1), (rs2)	Division between a vector in SPM and a scalar in SPM.
ksvdivrf (rd), (rs1), (rs2)	Division between a vector in SPM and a scalar in a register.
kvrem (rd), (rs1), (rs2)	Remainder calculation for division between two vectors in SPMs.
ksvremsc (rd), (rs1), (rs2)	Remainder calculation for division between a vector in SPM and a scalar in SPM.
ksvremrf (rd), (rs1), (rs2)	Remainder calculation for division between a vector in SPM and a scalar in a register.

4.3 Results

To validate the proposed vector variable divider and verify its performance, we conducted a comprehensive analysis focusing on the average division latency as a function of the SIMD and the efficiency in two real-case computation kernels.

4.3.1 Monte Carlo Simulation Analysis

Monte Carlo simulations allow us to evaluate the average number of clock cycles per division in the used divider and the traditional fixed latency solutions. We performed simulations for SIMD equal to 1, 2, 4, and 8, testing more than 10^{10} 32-bit dividend and divisor pairs, simulating until the first and second decimal places of the average latency remained constant over 10 billion iterations. Table 4.4 summarizes the algorithmic level performance of the selected VLNPD architecture against radix- n solutions, assuming the operands are already available in the local input registers.

The vector divider with the VLNPD-Std architecture exhibits a latency ranging from 1.546 clock cycles with SIMD equal to 1 down to 0.389 as SIMD increases to 8, representing a substantial speedup over traditional fixed-latency solutions. Specifically, when compared against the radix-2, the speedup ranges from $20.7\times$ to $10.3\times$, as the SIMD increases. For radix-4, the improvement spans from $10.3\times$ to

$5.1\times$ and from $7.11\times$ to $3.53\times$ when using radix-8. Notably, even when compared to the radix-16 implementation, which is the fastest but most complex architecture among fixed-latency designs, the VLNPD-Std achieves a speedup that ranges from $5.17\times$ to $2.57\times$ while requiring approximately half the hardware resources per lane, as detailed in Section 4.1. From this discussion, it becomes clear that the total latency in radix- n dividers decreases linearly with increasing SIMD width, while in variable latency solutions, the reduction is non-linear and depends on the slowest division operation executed in parallel.

4.3.2 Benchmark Analysis

To demonstrate the efficiency of the proposed approach in real application scenarios, we selected the matrix inversion with LU factorization (MATINV) and the image division (IMDIV) benchmarks. The first is directly representative of the matrix inversion kernel at the core of linear contextual bandits, while the latter is a division-intensive benchmark. We used a SIMD value equal to 4 and tested three different configurations: the sequential execution on the core without hardware acceleration, the operation with the original VCU but no vector division, and the execution with the upgraded VCU incorporating the new vector divider. Specifically, in the case of the upgraded VCU, we contrasted the newly adopted variable latency solution for vector division against traditional radix- n approaches. Fig. 4.2 illustrates the achieved results, presenting the speedups for the MATINV and IMDIV benchmarks on the left and right y-axes, respectively. For the MATINV benchmark with 4×4 matrix, the speedup increases from $1.05\times$ using the baseline VCU to $3.82\times$ with the inclusion of the variable latency vector division based on the VLNPD-Std architecture. The same trend can be observed when increasing the matrix size. The obtained speedup grows from $1.34\times$ to $4.87\times$ for 8×8 matrices, from $1.99\times$ to $7.38\times$ for 16×16 matrices and, from $3.15\times$ to $11.08\times$ for 32×32 ones. Notably, the adoption of the VLNPD-Std architecture for vector division consistently outperforms all the traditional radix- n solutions.

These results have a direct implication for the linear bandit workload that motivates this dissertation. As reported in the Part I Overview, cycle-accurate profiling of Disjoint LinUCB on the scalar Klessydra-T13 core with $d = 8$ and $N = 8$ arms revealed that integer division alone accounts for up to 70% of total execution cycles. The 8×8 MATINV benchmark captures precisely this dominant kernel—the per-arm covariance matrix inversion. With the VLNPD-enabled VCU operating at SIMD = 4, the $4.87\times$ speedup on this kernel compresses the division impact from approximately 70% to roughly 12% of total execution time. In other

Table 4.4 Average Division Latency as a Function of SIMD for the VLNPD-Std and different Radix- n Implementations

SIMD	Radix-2	Radix-4	Radix-8	Radix-16	VLNPD-Std
1	32	16	11	8	1.546
2	16	8	5.5	4	0.773
4	8	4	2.75	2	0.573
8	4	2	1.375	1	0.389

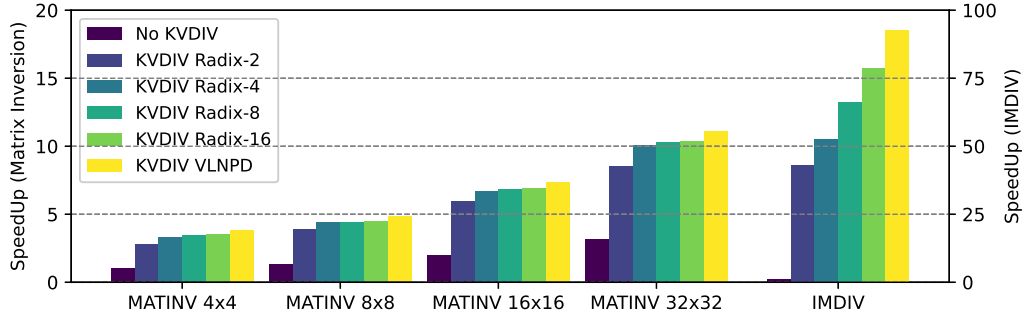


Figure 4.2 SpeedUp on two real-world benchmarks obtained using the KVDIV instruction. The SIMD is set to 4.

words, the combined effect of vectorization and variable-latency division reduces the division bottleneck by nearly $6\times$. This reduction clears the path for the algorithmic optimizations of Chapter 5: once division no longer gates execution, the remaining cubic scaling of matrix inversion itself becomes the next target.

The benefit of a variable latency approach is even more evident in the division-intensive IMDIV benchmark, where it provides a speedup equal to $91\times$ over the software execution and $1.76\times$ over the radix-4.

4.4 Summary of Findings

This Chapter proposed employing a variable-latency division approach in vector hardware accelerators to balance performance and hardware overhead. Using the VCU of the Klessydra T13 core as a case study, we evaluated different division designs from the literature, selecting and integrating the VLNPD approach. We detailed the implementation process of vector divisions, emphasizing our approach to hardware reuse and extending the RISC-V instruction set with new custom instructions. We validated the proposed approach with Monte Carlo simulations and two real benchmarks, showing that variable-latency vector dividers can be a viable way to improve the division time while being resource-efficient.

As shown in the benchmark analysis of Section 4.3.2, the combined effect of vectorization and variable-latency division reduces the division share from the 70% measured on the scalar core to approximately 12%, clearing the path for the algorithmic optimizations presented in the next chapter.

Chapter 5

Algorithmic and Hardware Optimization of LinUCB for Embedded Systems

Chapter based on Paper 3 [82]: Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Efficient implementation of linearucb through algorithmic improvements and vector computing acceleration for embedded learning systems. ACM Transactions on Embedded Computing Systems, 24(4):1–23, 2025

"Constraints are the friends of creativity."

Anonymous

The previous chapters addressed the key *arithmetic* bottleneck in embedded linear contextual bandits: Chapter 3 introduced the VLNPD to accelerate integer division, and Chapter 4 integrated it into the Klessydra-T13 Vector Coprocessor, reducing the contribution of division from 70% to approximately 12% of total execution time in representative linear-algebra kernels. However, even with faster division, the scaling of linear bandits remains dominated by repeated manipulations of covariance matrices, whose inversion at every decision step scales cubically with the context dimension ($\mathcal{O}(d^3)$) and demands quadratic memory ($\mathcal{O}(d^2)$). On embedded platforms with kilobyte-scale memory and real-time constraints, this overhead becomes the primary obstacle to online learning.

These considerations motivate the research question guiding this chapter:

Research Question

Can the Disjoint and Hybrid LinUCB algorithms be reformulated to eliminate explicit matrix inversion, and co-designed with the vector support developed in the previous chapters, to achieve real-time online decision-making on resource-constrained embedded platforms?

This chapter, mainly based on Paper 4 [82], addresses this challenge, introducing algorithmic and hardware optimization strategies, which can be applied independently or combined, to reduce complexity and enhance the execution time and energy

Algorithm 1 Optimized Disjoint LinUCB

```

1: Input:  $\alpha \in \mathbb{R}_+$ 
2: for  $t = 1, 2, \dots, T$  do
3:   Observe the context for all arms  $a \in \mathcal{A}_t$ :  $\mathbf{x}_{t,a} \in \mathbb{R}^d$ 
4:   for all  $a \in \mathcal{A}_t$  do
5:     if  $t = 1$  then
6:        $\mathbf{A}_a^{-1} = \mathbf{I}_d$ 
7:        $\mathbf{b}_a = \mathbf{0}_{d \times 1}$ 
8:     end if
9:      $\hat{\boldsymbol{\theta}}_a = \mathbf{A}_a^{-1} \mathbf{b}_a$ 
10:     $p_{t,a} = \hat{\boldsymbol{\theta}}_a^\top \mathbf{x}_{t,a} + \alpha \sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}}$ 
11:  end for
12:  Choose  $a_t = \arg \max_{a \in \mathcal{A}_t} p_{t,a}$ 
13:  Observe a real-valued payoff  $r_t$ 
14:   $\mathbf{A}_{a_t}^{-1} = \mathbf{A}_{a_t}^{-1} - \frac{\mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1}}{1 + \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t}}$ 
15:   $\mathbf{b}_{a_t} = \mathbf{b}_{a_t} + r_t \mathbf{x}_{t,a_t}$ 
16: end for

```

efficiency of LinUCB Contextual Bandits algorithms [74].

On the algorithmic side, the resource-intensive process of updating and inverting the matrix underlying the learning process of these models is replaced by a more efficient, incremental update mechanism based on the Sherman-Morrison-Woodbury formula [149]. While this principle has been adopted in previous works to implement the Disjoint LinUCB algorithm [76, 277, 278], in this chapter, we extend this technique to a more complex Hybrid variant and thoroughly assess the impact of the incremental update on the complexity, execution time, memory requirements, energy consumption and numeric stability of the LinUCB algorithms across different embedded platforms.

On the other hand, looking at the hardware perspective, we map the reformulated algorithms onto the Klessydra-T13 Vector Coprocessor equipped with the VLNPD-based vector divider introduced in Chapter 4, leveraging custom vector instructions to accelerate the matrix-vector operations that now dominate execution time.

While many efforts in the literature have explored applications of CB algorithms and improved their accuracy, no previous study has investigated approaches to reduce the computational complexity of LinUCB for deployment on edge devices with low-power and real-time requirements. This work aims to fill that gap.

The rest of the chapter is organized as follows. Section 5.1 details the proposed algorithmic reformulation and discusses the achieved complexity reduction and numerical stability considerations. Section 5.2 describes the mapping of the optimized algorithms onto the Klessydra-T13 vector accelerator. Section 5.3 presents experimental results on execution time, memory footprint, and energy consumption. Finally, Section 5.4 summarizes the main outcomes.

5.1 Algorithmic Optimizations

As discussed in the Part I Overview, the matrix inversion at the core of both Disjoint and Hybrid LinUCB imposes a cubic computational complexity ($\mathcal{O}(d^3)$ for the first and $\mathcal{O}(f^3 d^3)$ for the latter) that represents the dominant algorithmic bottleneck for embedded deployment. To mitigate this problem, the approach in [74] proposes to update the original matrices at each step while computing and caching the inverses periodically instead of in real-time. However, this approach may limit online learning capabilities due to less frequent model updates, requires more memory and still necessitates longer iterations at regular intervals.

Given that matrix inversion constitutes a bottleneck affecting the scalability and efficiency of LinUCB algorithms, this section describes and explores mathematical strategies that can yield the same results without relying on this operation. First, we discuss the modifications applied to the simpler Disjoint LinUCB algorithm, where the same principle has been adopted in previous works [76, 277–280], and then we extend the approach to the more complex Hybrid model. For each variant, we thoroughly investigate the impact of the proposed techniques on the computational complexity, execution time, energy consumption, and memory requirements, which are essential aspects for applications on embedded systems, while varying the problem parameters.

5.1.1 Optimization of the Disjoint LinUCB through the Sherman-Morrison Formula

In the Disjoint algorithm, the $\mathbf{A}_a$ matrices are utilized in their inverted form, $\mathbf{A}_a^{-1}$, during action selection and subsequently adjusted in the normal form during the update phase based on the received reward. However, the process of updating and

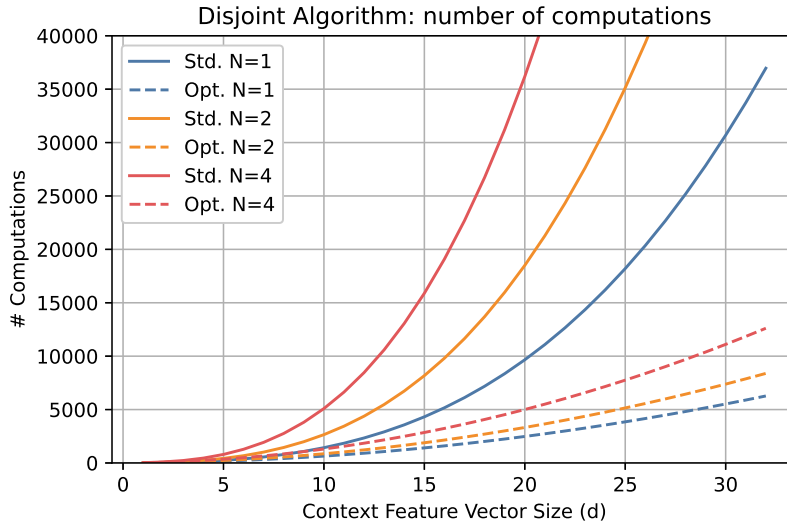


Figure 5.1 Computational complexity trends for the standard Disjoint Algorithm and the proposed optimized version.

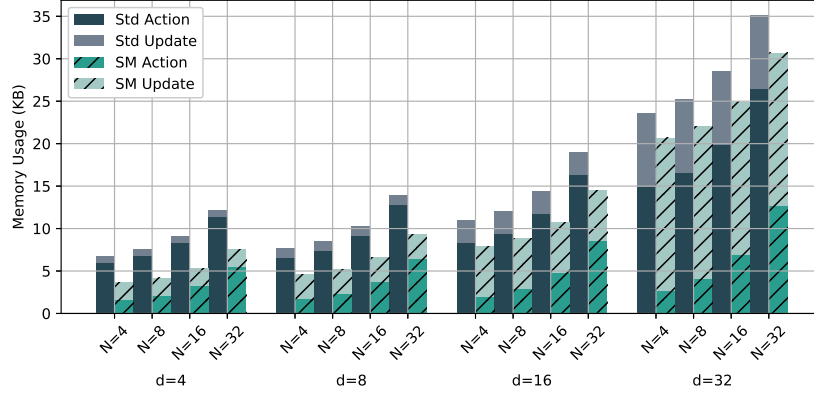


Figure 5.2 Stack Memory Usage for the standard Disjoint Algorithm and the proposed optimized version.

then inverting the matrix in each iteration introduces computational inefficiencies, especially as the matrix size increases.

Considering that $\mathbf{A}_a$ undergoes incremental updates through a rank-one perturbation, $\mathbf{x}_t \mathbf{x}_t^\top$, it is well-suited for the application of the Sherman-Morrison (SM) theorem [149]. The theorem represents an efficient method for updating the inverse of a matrix, declaring that if $\mathbf{A} \in \mathbb{R}^{n \times n}$ is an invertible square matrix with inverse $\mathbf{A}^{-1}$, and it undergoes a rank-one update $\mathbf{u} \mathbf{v}^\top$, with $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, then the updated inverse can be efficiently computed by leveraging and updating the previous one, as demonstrated in (5.1).

$$(\mathbf{A} + \mathbf{u} \mathbf{v}^\top)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1} \mathbf{u} \mathbf{v}^\top \mathbf{A}^{-1}}{1 + \mathbf{v}^\top \mathbf{A}^{-1} \mathbf{u}} \quad (5.1)$$

The update rule in (2.2) modified with the Sherman-Morrison formula for the Disjoint algorithm is shown in (5.2), while Algorithm 1 presents the complete optimized algorithm.

$$(\mathbf{A}_{a_t} + \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top)^{-1} = \mathbf{A}_{a_t}^{-1} - \frac{\mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1}}{1 + \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t}} \quad (5.2)$$

This approach enables the direct storage of the inverse matrix $\mathbf{A}_a^{-1}$, reducing the stack memory requirements for intermediate variables, and completely eliminates the matrix inversion operation, significantly reducing the computational overhead by saving the inversion of $N[d \times d]$ matrices at each time step, at the cost of a minimal increase in the complexity of the update phase.

Fig. 5.1 illustrates the computational complexity trends of the Disjoint algorithm varying N and d parameters, comparing the standard and optimized versions. The proposed optimization reduces the asymptotic complexity of the Disjoint algorithm from $\mathcal{O}(d^3)$ to $\mathcal{O}(d^2)$.

Fig. 5.2 depicts the stack memory usage, obtained through a Python memory profiler, of both the traditional and optimized versions during the action selection and update phases, with d and N varying from 4 to 32. The standard algorithm

consumes up to 35.13 KB (26.47 KB during action selection and 8.66 KB during the update phase), whereas the optimized version requires 30.70 KB (12.68 KB for action selection and 18.02 KB for the update phase). This corresponds to a reduction of 4.43 KB, or approximately 12.6% in total stack memory usage. This analysis also reveals that while the optimization significantly reduces memory usage in the action selection phase (by about 52%), it also shifts stack memory usage to the update phase.

5.1.2 Optimization of the Hybrid LinUCB through the Woodbury's Formula

In this section, the optimization strategies applied to the Disjoint algorithm are generalized and extended to the more complex Hybrid LinUCB model.

Similar to the Disjoint variant, each action in the Hybrid algorithm is associated with a $d \times d$ matrix $\mathbf{A}_a$, used in its inverted form during the action selection phase and incrementally updated via rank-one perturbations, as per (2.6). Consequently, the Sherman-Morrison formula is again applicable, allowing to directly store $\mathbf{A}_a^{-1}$. However, although this approach improves computational efficiency by removing $N[d \times d]$ matrix inversions at each iteration, the overall performance gain for this model remains modest. The computational complexity is, in fact, predominantly driven by the inversion of the large $k \times k$ shared matrix, $\mathbf{A}_0$. To illustrate, when considering $d = 8$ and $f = 16$, the inversion of $\mathbf{A}_0$ requires approximately 2,097,152 operations per iteration, which is significantly higher than the relatively minor cost of 512 operations for each $\mathbf{A}_0$ inversion. Consequently, it is crucial to explore whether a similar optimization strategy can be applied to $\mathbf{A}_0$, aiming to substantially accelerate the algorithm.

The $\mathbf{A}_0$ matrix is always used in its inverted form during the action choice, and it is then incrementally updated according to Equations (2.5) and (2.6) from Chapter 2, to which in the following we will refer as first and second update step, respectively.

In the first one, the Hybrid algorithm modifies $\mathbf{A}_0$ by incorporating the term $\mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}$, which represents a complex adjustment that goes beyond the normal rank-one perturbation, entailing the multiplication of three matrices. To efficiently address this more complex update scenario, we apply a generalized version of the Sherman-Morrison formula, known as the Woodbury matrix identity [150]. This formula is useful when the matrix update process involves an incremental rank- k perturbation. Given an invertible matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ with its inverse $\mathbf{A}^{-1}$, and matrices $\mathbf{U} \in \mathbb{R}^{n \times k}$, $\mathbf{C} \in \mathbb{R}^{k \times k}$, and $\mathbf{V} \in \mathbb{R}^{k \times n}$, the inverse of the updated matrix after applying a perturbation $\mathbf{UCV}$ can be computed by (5.3).

$$(\mathbf{A} + \mathbf{UCV})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{C}^{-1} + \mathbf{VA}^{-1}\mathbf{U})^{-1}\mathbf{VA}^{-1} \quad (5.3)$$

Applying the equation to the Hybrid algorithm makes it possible to modify the first update step in (2.5) as shown in (5.4).

$$(\mathbf{A}_0 + \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t})^{-1} = \mathbf{A}_0^{-1} - \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top \cdot (\mathbf{A}_{a_t} + \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top)^{-1} \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \quad (5.4)$$

Notably, $\mathbf{A}_{a_t}$ required in this equation can be derived by inverting $\mathbf{A}_{a_t}^{-1}$, consistent with the traditional algorithm, where $\mathbf{A}_{a_t}^{-1}$ is calculated through inversion during the update phase.

Algorithm 2 Optimized Hybrid LinUCB

```

1: Input:  $\alpha \in \mathbb{R}_+$ 
2:  $\mathbf{A}_0^{-1} = \mathbf{I}_k$ 
3:  $\mathbf{b}_0 = \mathbf{0}_k$ 
4: for  $t = 1, 2, \dots, T$  do
5:   Observe the context for all arms  $a \in \mathcal{A}_t$ :  $(\mathbf{z}_{t,a}, \mathbf{x}_{t,a}) \in \mathbb{R}^{k+d}$ 
6:    $\hat{\beta} = \mathbf{A}_0^{-1} \mathbf{b}_0$ 
7:   for all  $a \in \mathcal{A}_t$  do
8:     if  $t = 1$  then
9:        $\mathbf{A}_a^{-1} = \mathbf{I}_d$ 
10:       $\mathbf{B}_a = \mathbf{0}_{d \times k}$ 
11:       $\mathbf{b}_a = \mathbf{0}_{d \times 1}$ 
12:    end if
13:     $\hat{\theta}_a = \mathbf{A}_a^{-1} (\mathbf{b}_a - \mathbf{B}_a \hat{\beta})$ 
14:     $s_{t,a} = \mathbf{z}_{t,a}^\top \mathbf{A}_0^{-1} \mathbf{z}_{t,a} - 2 \mathbf{z}_{t,a}^\top \mathbf{A}_0^{-1} \mathbf{B}_a^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}$ 
15:     $\quad + \mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a} + \mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{B}_a \mathbf{A}_0^{-1} \mathbf{B}_a^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}$ 
16:     $p_{t,a} = \mathbf{z}_{t,a}^\top \hat{\beta} + \mathbf{x}_{t,a}^\top \hat{\theta}_a + \alpha \sqrt{s_{t,a}}$ 
17:  end for
18:  Choose  $a_t = \arg \max_{a \in \mathcal{A}_t} p_{t,a}$ 
19:  Observe a real-valued payoff  $r_t$ 
20:   $\mathbf{A}_0^{-1} = \mathbf{A}_0^{-1} - \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top (\mathbf{A}_{a_t} + \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top)^{-1} \mathbf{B}_{a_t} \mathbf{A}_0^{-1}$ 
21:   $\mathbf{b}_0 = \mathbf{b}_0 + \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{b}_{a_t}$ 
22:   $\mathbf{A}_{a_t}^{-1} = \mathbf{A}_{a_t}^{-1} - \frac{\mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t} \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1}}{1 + \mathbf{x}_{t,a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_{t,a_t}}$ 
23:   $\mathbf{B}_{a_t} = \mathbf{B}_{a_t} + \mathbf{x}_{t,a_t} \mathbf{z}_{t,a_t}^\top$ 
24:   $\mathbf{b}_{a_t} = \mathbf{b}_{a_t} + r_t \mathbf{x}_{t,a_t}$ 
25:   $\mathbf{A}_0^{-1} = \mathbf{A}_0^{-1} - \frac{\mathbf{A}_0^{-1} \mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top \mathbf{A}_0^{-1}}{1 + \mathbf{z}_{t,a_t}^\top \mathbf{A}_0^{-1} \mathbf{z}_{t,a_t}}$ 
26:   $\mathbf{A}_0^{-1} = \mathbf{A}_0^{-1} + \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top (\mathbf{A}_{a_t} - \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top)^{-1} \mathbf{B}_{a_t} \mathbf{A}_0^{-1}$ 
27:   $\mathbf{b}_0 = \mathbf{b}_0 + r_t \mathbf{z}_{t,a_t} - \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{b}_{a_t}$ 
28: end for

```

The second update phase in (2.6) involves the sum of the outer product $\mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top$ to $\mathbf{A}_0$ and the subtraction of $\mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}$. The first sum involves a rank-one incremental update, since $\mathbf{z}_{t,a_t}$ is a column vector in $\mathbb{R}^k$. Thus, this step can be efficiently handled using the Sherman-Morrison formula, as depicted in (5.5).

$$(\mathbf{A}_0 + \mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top)^{-1} = \mathbf{A}_0^{-1} - \frac{\mathbf{A}_0^{-1} \mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top \mathbf{A}_0^{-1}}{1 + \mathbf{z}_{t,a_t}^\top \mathbf{A}_0^{-1} \mathbf{z}_{t,a_t}} \quad (5.5)$$

The subtraction, instead, mirrors the mechanics of the first update step. The modified update rule is shown in (5.6).

$$(\mathbf{A}_0 - \mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t})^{-1} = \mathbf{A}_0^{-1} + \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top \cdot (\mathbf{A}_{a_t} - \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \mathbf{B}_{a_t}^\top)^{-1} \mathbf{B}_{a_t} \mathbf{A}_0^{-1} \quad (5.6)$$

Algorithm 2 presents the optimized Hybrid algorithm highlighting the changes to the standard version in [74]. As depicted, this approach only involves the $\mathbf{A}_0^{-1}$

and $\mathbf{A}_a^{-1}$ matrices, updating them using the Sherman-Morrison-Woodbury (SMW) formulas. This enables the direct storage of the inverse matrices and reduces the number of matrix inversions from N inversions of $[d \times d]$ matrices and one inversion of a $k \times k$ matrix per iteration to a constant $3[d \times d]$ inversions per iteration in the update process. This notably decreases computational complexity by removing the need for the large $k \times k$ inversion and eliminating the dependency on N .

Fig. 5.3 depicts the computational complexity trends of the Hybrid algorithm with a constant $N = 8$, while varying f , and d , contrasting the standard and optimized versions. Overall, the optimization leads to an asymptotic computational complexity of $\mathcal{O}(k^2d)$ or $\mathcal{O}(f^2d^3)$ instead of the larger $\mathcal{O}(k^3)$ or $\mathcal{O}(f^3d^3)$. These results also illustrate the relative complexity of the Hybrid algorithm compared to the Disjoint variant, showing how rapidly the computational demands escalate with increasing problem parameters.

Table 5.1 summarizes the impact of the proposed algorithmic optimizations on the computational complexity of the algorithms, both during the action selection and update processes (Stage 1 and Stage 2).

Fig. 5.4 shows the stack memory usage of the hybrid algorithm, measured using a Python memory profiler, with the d and f parameters varying from 4 to 32. The standard algorithm consumes up to 25.26 MB of memory (with 8.74 MB allocated to

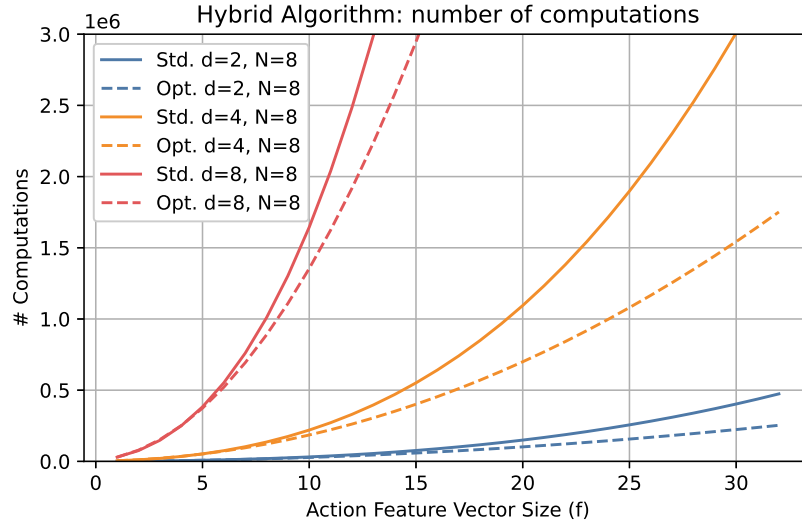


Figure 5.3 Computational complexity trends for the standard Hybrid Algorithm and the proposed optimized version.

Table 5.1 Asymptotic computational complexity in the original Disjoint and Hybrid Algorithms and in the proposed optimized versions

	Disjoint LinUCB		Hybrid LinUCB	
	Stage 1	Stage 2	Stage 1	Stage 2
Standard	$\mathcal{O}(d^3)$	$\mathcal{O}(d^2)$	$\mathcal{O}(k^3)$	$\mathcal{O}(k^2d)$
Optimized	$\mathcal{O}(d^2)$	$\mathcal{O}(d^2)$	$\mathcal{O}(k^2d)$	$\mathcal{O}(k^2d)$

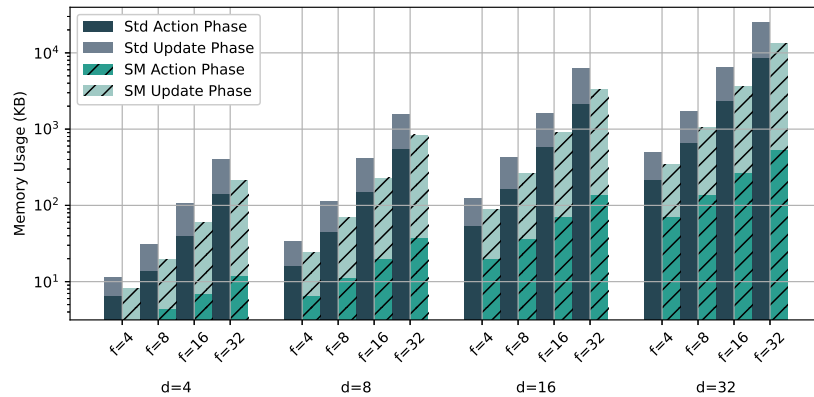


Figure 5.4 Stack Memory Usage for the standard Hybrid Algorithm and the proposed optimized version.

the action phase and 16.52 MB to the update phase), whereas the optimized version requires only 13.35 MB (532 KB for the action phase and 12.81 MB for the update phase). This improvement results in a peak memory reduction of approximately 12 MB, or about 47%.

5.1.3 Stability and Round-off Errors

In the previous section, we demonstrated how the Sherman–Morrison–Woodbury formula can significantly reduce the computational cost of the LinUCB algorithm by avoiding complex and time-consuming matrix inversions. However, while this formula offers an exact solution for updating matrix inverses, its practical application can be affected by round-off errors due to finite-precision arithmetic, potentially leading to instabilities when the matrix becomes ill-conditioned. In this section, we first establish the stability of the proposed approach by demonstrating that the inverse covariance matrices in the LinUCB algorithms remain well-conditioned and stable over time. We then investigate the impact of round-off errors as the number of iterations grows, using synthetic datasets to quantify error accumulation. Finally, we present a novel strategy to mitigate these errors in scenarios where high precision is critical. We begin our analysis with the Disjoint LinUCB algorithm, addressing its simpler structure, and then we extend our findings to the more complex Hybrid one.

Stability of the Covariance Matrices

Round-off or rounding errors arise from the finite-precision arithmetic inherent in digital computations and can accumulate over successive operations, potentially leading to numerical inaccuracies. When recomputing the inverse of a matrix from scratch after each modification, each inversion is independent of previous computations, making this approach generally more resilient to the accumulation of round-off errors. In contrast, in incremental update mechanisms, small errors can accumulate across iterations, which may gradually lead to noticeable inaccuracies. In detail, in the SM formula, round-off errors are prone to amplification and can introduce instabilities in two main scenarios:

1. Denominator Approaching Zero: The SM formula contains a denominator term that, when approaching zero, can render the update unstable. This scenario can lead to large numerical errors or even division by zero, resulting in computational failures.
2. Singular or Ill-Conditioned Inverse Matrices: If the inverse matrix becomes singular or ill-conditioned the calculations may amplify numerical errors. An ill-conditioned matrix is highly sensitive to minor perturbations, which can result in substantial deviations in the output, thereby compromising the algorithm's stability.

To ensure the stability of the SM update in our optimized LinUCB algorithms, in this section we demonstrate that these two conditions are avoided. To do that, is sufficient to show that the involved covariance matrices $\mathbf{A}_{a_t}$ remain positive and definite throughout the iterations [281]. This is due to two key reasons:

1. If $\mathbf{A}_{a_t}$ is positive definite, then the updated matrix $\mathbf{A}_{a_{t+1}} = \mathbf{A}_{a_t} + \mathbf{x}_t \mathbf{x}_t^\top$ and its inverse $\mathbf{A}_{a_{t+1}}^{-1}$ are also positive definite ensuring that the condition number of $\mathbf{A}_{a_t}$ remains bounded, preventing the matrix from becoming ill-conditioned. In fact, if $\mathbf{A}_{a_t}$ is positive definite at iteration t , since $\mathbf{x}_t \mathbf{x}_t^\top$ is positive semi-definite for any non-zero $\mathbf{x}_t$, the sum of a positive definite matrix $\mathbf{A}_{a_t}$ and a positive semi-definite matrix $\mathbf{x}_t \mathbf{x}_t^\top$ remains positive definite.
2. If the matrix is positive definite, the denominator in the Sherman-Morrison formula, $d_t = 1 + \mathbf{x}_t^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_t$, is always strictly greater than one. In fact, since $\mathbf{A}_{a_t}^{-1}$ is positive definite (as proven in the previous step), for any non-zero $\mathbf{x}_t$, we have $\mathbf{x}_t^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_t > 0$. Therefore, the denominator $d_t = 1 + \mathbf{x}_t^\top \mathbf{A}_{a_t}^{-1} \mathbf{x}_t > 1$.

In the optimized Disjoint LinUCB algorithm proposed in Algorithm 1, the inverse covariance matrices $\mathbf{A}_a^{-1}$ are positive definite by construction, as they are initialized as $\mathbf{A}_a = \lambda \mathbf{I}_d$ at time-step $t = 0$, where $\lambda > 0$ is the regularization parameter and $\mathbf{I}_d$ is the $d \times d$ identity matrix. From what we demonstrated, this implies that the denominator in the SM formula remains greater than one and that these matrices remain positive definite and well-conditioned throughout the iterations. As a result, round-off errors are not amplified, and the algorithm is expected to maintain numerical stability over time.

The same analysis extends to the more complex optimized Hybrid LinUCB algorithm presented in Algorithm 2, which incorporates global features shared across all arms. We have already proved that each $\mathbf{A}_{a_t}$ matrix remains positive definite after the update phase; for the Hybrid case, we now demonstrate the same for the global covariance matrix $\mathbf{A}_0$ and its update. At time-step $t = 0$, $\mathbf{A}_0$ is initialized as $\lambda \mathbf{I}_k$, where $\lambda > 0$ and $\mathbf{I}_k$ is the $k \times k$ identity matrix. At each iteration t , $\mathbf{A}_0$ is updated when an arm a_t is selected according to (2.5) and (2.6). In (2.5), $\mathbf{z}_{t,a_t} \mathbf{z}_{t,a_t}^\top$ is positive semi-definite (as it is an outer product of a vector with itself), and thus, adding it to the positive definite $\mathbf{A}_0$ results in a positive definite matrix. The resulting matrix is further updated with the term $\mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}$ in (2.6). To show that this term is positive semi-definite, we need to demonstrate that for any vector $\mathbf{x}$,

$$\mathbf{x}^\top (\mathbf{B}_{a_t}^\top \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}) \mathbf{x} \geq 0.$$

However, since we already demonstrated that the matrix $\mathbf{A}_{a_t}^{-1}$ is positive definite, we know that for any non-zero vector $\mathbf{y}$, $\mathbf{y}^\top \mathbf{A}_{a_t}^{-1} \mathbf{y} > 0$. Setting $\mathbf{y} = \mathbf{B}_{a_t} \mathbf{x}$ gives:

$$\mathbf{x}^\top \mathbf{B}_{a_t} \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}^\top \mathbf{x} = \mathbf{y}^\top \mathbf{A}_{a_t}^{-1} \mathbf{y} \geq 0.$$

This confirms that $\mathbf{B}_{a_t} \mathbf{A}_{a_t}^{-1} \mathbf{B}_{a_t}^\top$ is positive semi-definite, and therefore, $\mathbf{A}_0$ remains positive definite. Consequently, both the global and arm-specific covariance matrices in the Hybrid algorithm remain positive definite over time.

By proving that the matrices of the algorithms are positive definite, we ensure that their inverses are also positive definite and well-conditioned. This positive definiteness of the inverses is crucial for maintaining numerical stability in the SMW updates, as it prevents the amplification of round-off errors and ensures the robustness of the algorithms over time. Thus, the optimized LinUCB algorithm is robust against numerical issues associated with finite-precision arithmetic.

Assessing the Accumulation of Round-Off Errors

In the previous section, we demonstrated that the covariance matrices within the LinUCB algorithms remain well-conditioned throughout the iterations, making them suitable candidates for the Sherman–Morrison update. This condition prevents the amplification of round-off errors over time, allowing the incremental update approach in the optimized LinUCB algorithms to maintain numerical stability effectively.

To further evaluate the practical impact of round-off errors, we conducted an empirical assessment. We ran 100,000 iterations on synthetic datasets for both the Disjoint and Hybrid LinUCB algorithms. At each iteration, we compared the matrices $\mathbf{A}_{a_t}^{-1}$ and $\mathbf{A}_0^{-1}$ obtained through incremental updates with those obtained via the traditional full matrix inversion approach. Figures 5.5 and 5.6 display the error growth, computed as the Frobenius norm of the difference between matrices, and the average reward for both the Disjoint and Hybrid LinUCB algorithms.

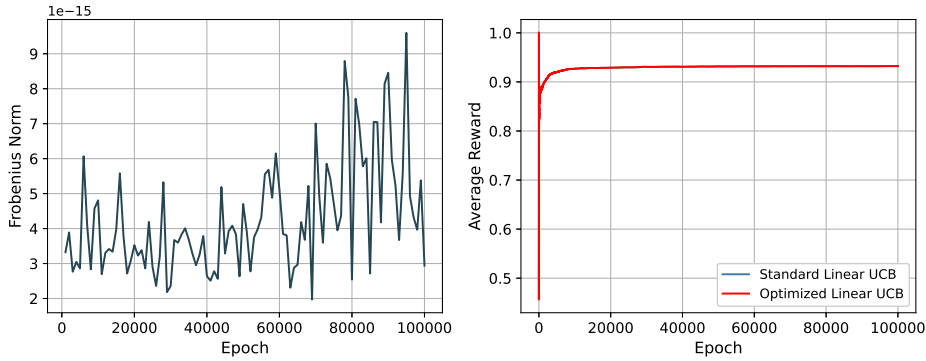


Figure 5.5 Error growth in the Disjoint Linear UCB algorithm, measured by the Frobenius norm of the difference between $\mathbf{A}_{a_t}^{-1}$ (computed incrementally) and the same matrix obtained via full inversion.

For the Disjoint algorithm, the total difference measured using the Frobenius norm, consistently hovers around 1×10^{-15} , indicating that round-off errors accumulate very slowly and remain completely negligible even after 100,000 iterations.

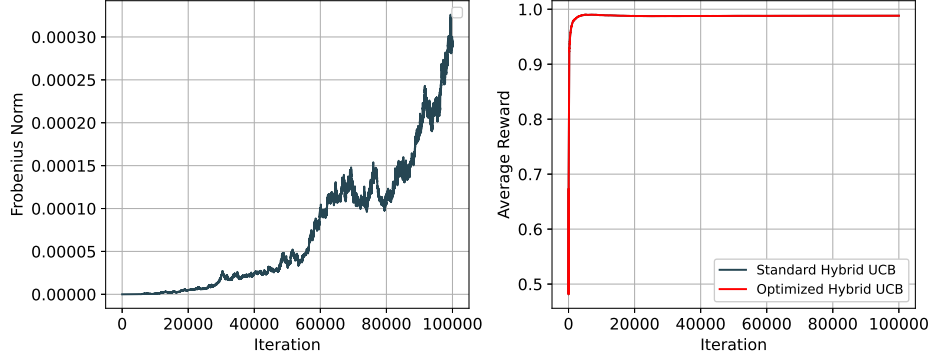


Figure 5.6 Error growth in the Hybrid Linear UCB algorithm, measured by the Frobenius norm of the difference between $\mathbf{A}_0^{-1}$ (computed incrementally) and the same matrix obtained via full inversion.

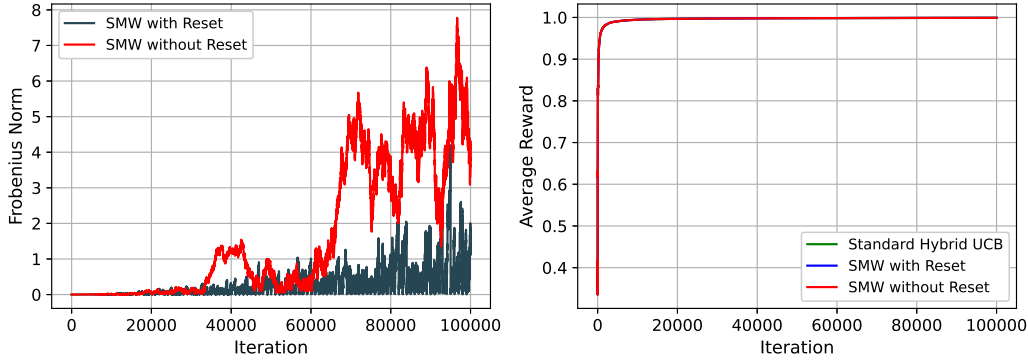


Figure 5.7 Error growth in the Hybrid Linear UCB algorithm with periodic correction of the inverse matrices (every 5000 iterations).

In contrast, in the Hybrid Linear UCB algorithm, we observed a larger accumulation of round-off errors, particularly in the global matrix $\mathbf{A}_0$, where the error grew up to 1×10^{-4} after 100,000 iterations. This larger accumulation is primarily due to the fact that while the arm-specific matrices $\mathbf{A}_a$ are updated only when a particular arm is selected, the global matrix $\mathbf{A}_0$ is updated at every iteration, making it more prone to round-off error accumulation.

An important observation is that in both algorithms, the optimization introduced by the Sherman-Morrison-Woodbury approach had no effect on the decision-making process. Across 100,000 iterations, the actions selected by the optimized methods were identical to those chosen by the traditional matrix inversion, resulting in the same cumulative reward for both approaches. This suggests that the round-off errors introduced by the incremental updates had a negligible impact on the algorithm's performance, even after a substantial number of iterations.

Proposed Solution to Mitigate Round-Off Errors

While our results show that the round-off errors in the optimized SMW approach have negligible impact on performance, in certain applications where the algorithm

may be run for an extremely high number of iterations, it could be important to mitigate the accumulation of these errors. To address this, we propose a method to periodically correct the inverse matrices, thus alleviating the influence of round-off errors over time. This approach consists of periodically replacing or correcting the inverse matrices $\mathbf{A}_a^{-1}$ and $\mathbf{A}_0^{-1}$. When memory constraints are not a major issue, we can store both the direct matrices $\mathbf{A}_a$ and $\mathbf{A}_0$, as well as their inverses. The direct matrices are updated in each iteration using the traditional algorithmic formulation (i.e., without incremental updates), while the inverse matrices are incrementally updated via the SMW approach and used for action selection. Every N iterations, where N is a tunable integer, the inverse matrices $\mathbf{A}_a^{-1}$ and $\mathbf{A}_0^{-1}$ are replaced by the exact inverses computed from their corresponding direct matrices. This periodic correction reduces the impact of round-off errors without requiring matrix inversion at every step, thus maintaining the computational efficiency of the SMW approach. The trade-off here is between the complexity of periodically computing matrix inversions and the degree to which round-off errors are mitigated. The value of N can be adjusted depending on the specific application requirements, balancing between minimizing error growth and minimizing the overhead introduced by performing full matrix inversions.

To demonstrate the effectiveness of this approach, we implemented this mechanism in the Hybrid Linear UCB algorithm and ran it for 100,000 iterations with $N = 5000$. The results, shown in Figure 5.7, indicate that the total round-off error can be reduced by up to two orders of magnitude compared to the original uncorrected approach.

This periodic correction approach provides a solution for applications where the algorithm is expected to run for an extremely high number of iterations, and the accumulation of round-off errors could eventually affect performance. By only computing full matrix inversions every N iterations, this method strikes a balance between computational efficiency and numerical precision. However, it introduces additional memory requirements and complexity. Notably, the overall computational load remains significantly reduced compared to traditional approaches, as matrix inversion is performed only once every N iterations, as opposed to every iteration. The value of N thus provides a flexible parameter to trade off between complexity and round-off error mitigation.

5.2 Vector Computing Acceleration

As discussed in Chapter 4, vector hardware accelerators are specialized computing units designed to efficiently process operations on vectors of numbers, exploiting data-level parallelism [282]. While vector processors have been traditionally employed in supercomputers, there is an evident trend in adopting vector computing support in embedded processing systems [273, 274]. Given the strong dependence of LinUCB algorithms on matrix operations, discussed in Chapter 2, this Section explores if and how vector computing architectural support may be harnessed to improve the algorithm execution time by implementing fundamental matrix operations in terms of elementary operations on vectors.

5.2.1 Reference Embedded Vector Computing Platform

For the purpose of our study, without loss of generality, we adopted the VCU of the Klessydra embedded processor family [114, 273, 283] to explore the impact of vector computing acceleration on the execution of the proposed algorithm. The VCU is controlled by a custom RISC-V compliant instruction set extension and can be configured at synthesis-time in terms of the SIMD, which defines the number of vector elements processed in parallel. The vector size is dynamically specified at runtime using a dedicated Current Status Register. The unit leverages a hardware loop mechanism to efficiently repeat the required instruction until all vector elements are processed—based on the configured SIMD and vector size—without requiring additional instruction fetches. The vector operations used for matrix-related computations in the algorithms of the proposed study are reported in the following section.

As introduced in the Part I Overview, the Klessydra-T13 embedded processor lacks both dynamic memory allocation and floating-point support. Throughout this work, all LinUCB implementations rely on a custom C++ library based on static allocation and fixed-point data representation. To quantify the impact of vector computing support on the final performance, the library supports two execution modes: a software-only mode, compiled on the standard RISC-V instruction set and executed by the scalar processing core, and a vectorized mode exploiting the VCU through the specialized vector intrinsic functions supported by the compiler toolchain.

5.2.2 Mapping LinUCB algorithms on the vector accelerator

The application of the VCU specialized functions to Algorithms 1 and 2 is detailed below:

- matrix addition: this algorithm step was encoded using the *kaddv (rd), (rs1), (rs2)* instruction, which allows the addition of two vectors located in the SPMs in a SIMD fashion. Executing this operation for each row, reduces the matrix addition time by a factor equal to the number of parallel functional units in the VCU.
- matrix multiplication: this algorithm step was encoded using the *kvmulpsrf (rd), (rs1), (rs2)* instruction that performs a fixed-point multiplication between a vector from the scratchpad memory and a scalar from the register file. We used an iterative approach in which an entire row of the first matrix is multiplied by a scalar element from a second row. This produces intermediate products, which are accumulated across vector lanes to form the corresponding row elements of the resultant matrix. By repeating this process for each row, the ability to perform multiple vector-scalar multiplications in a single cycle is fully leveraged, significantly enhancing the operation speed. The vector-scalar product is executed in hardware in double integer precision, and the resulting product is subsequently right-shifted by the appropriate scaling factor to maintain the output data width.

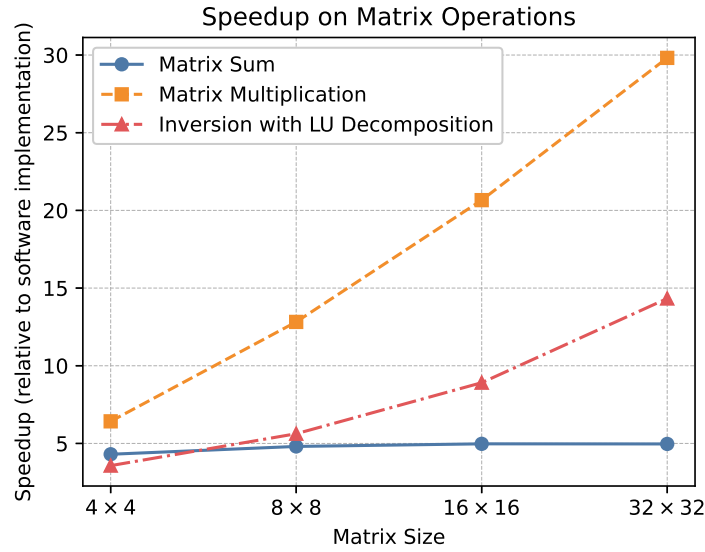


Figure 5.8 Speedup attained by the Klessydra-T13 vector computing acceleration on the fundamental matrix computation kernels used in LinUCB algorithms

- matrix inversion: this algorithm step was implemented by applying the LU factorization technique, which was encoded exploiting the *kdiv (rd)*, (*rs1*), (*rs2*) instruction for executing element-wise vector divisions [272] leveraging the variable latency dividers integrated in the VCU [250].

Figure 5.8 presents the performance gains achieved for the above discussed matrix operations, using a SIMD equal to 4. The speedup metric was chosen as the ratio between the clock cycles required by the standard operation (i.e. compiled for the RISC-V instruction set and executed on the Klessydra-T13 processor without vector computing support) and the cycles required by the accelerated operation (i.e. compiled on the extended instruction set exploiting the vector computing support of the Klessydra-T13 core). The speedup result is analyzed across different matrix sizes. The matrix addition exhibits maximum benefit from parallel processing, demonstrating a consistent speedup across all matrix sizes, equivalent to a factor of $4\times$. Matrix multiplication displays a different speedup pattern as the matrix size grows. The adopted parallel approach reaches a speed up equal to $6.42\times$ in the case of 4×4 matrices, which scales up to $29.82\times$ for 32×32 . The speedup factor greater than the SIMD amount is explained by the heavy memory access overhead suffered in the standard scalar execution. The inversion operation represents the most challenging operation to be parallelized owing to its inherently sequential nature. Nevertheless, using LU matrix factorization techniques and optimized dividers in the VCU achieves a notable acceleration of up to $15\times$ for 32×32 matrices. Notably, this speedup exceeds that reported for the MATINV benchmark in Figure 4.2, where only the division step of the matrix inversion was accelerated by the VCU; here, all steps of the factorization—additions, multiplications, and divisions—execute on the vector unit, yielding a larger overall gain.

5.3 Results on full algorithm execution

This section analyzes the cumulative impact of the proposed dual optimization strategy on the LinUCB algorithms implemented and executed on embedded processor platforms. The purpose of the reported performance data is to contextualize the results obtainable with algorithmic enhancement and explicit vectorization in the low-end embedded system domain. For this reason, we chose the Cortex M4 STM32 Nucleo board, the Cortex A72 Raspberry PI 4 Model B board, and the Klessydra T13 PULPino open-source soft-processor equipped with basic vector acceleration. The contextualization of the results does not aim to demonstrate absolute superior performance to specific commercial products, yet to give evidence of the effectiveness of the explored techniques in limited hardware resource scenarios.

For the Disjoint algorithm, we varied the size of the context feature vector (d) from 4 to 32, while keeping N equal to 8. For the hybrid algorithm, we varied the size of the action feature vector (f) from 4 to 32, keeping d and N equal to 8. The following subsections describe the adopted methodology in detail and discuss the obtained results in terms of execution time and energy consumption.

5.3.1 Methodology

The standard software C++ implementation and the optimized version of both the Disjoint and Hybrid LinUCB algorithms were compiled for the PULPino platform, the STM32 platform, and the Raspberry PI 4 platform. The compilation for PULPino Klessydra T13 was done for no vector computing support and for a $\text{SIMD} = 4$ vector acceleration support. This SIMD width was selected as it offers a balance between performance and area, making it well-suited for embedded applications [114]. The Klessydra PULPino platform was implemented on the Xilinx Vivado tool suite targeting the Kintex-7 KC705 FPGA device (xc7k325tffg900-2), with a target clock frequency of 100 MHz. The synthesis reported no timing violations.

Execution time estimations were done by direct execution on the STM32 and Raspberry boards, while they were done by cycle-accurate RTL simulation on QuestaSim for the Klessydra T13 implementations, with 100 MHz simulated clock frequency. The speedup factor was calculated as the ratio between the execution times of the traditional algorithm and the optimized algorithm.

For energy consumption estimation, we measured the power absorbed by the STM32 and Raspberry Pi boards during algorithm execution using a power supply with integrated power measurement capabilities. The instrument, connected to an AC source and supplying the target boards with DC power, continuously monitored the absorbed current. After synchronizing with a digital trigger signal from the target boards, the average power consumption was recorded during algorithm execution. For the Klessydra T13 implementation, we extracted the post-implementation netlist and conducted gate-level simulations of the algorithms. The simulations provided the switching activity *.saif* files, which were processed by the Vivado power estimator to assess the dynamic power ($P_{dynamic}$) of the core. For all of the Klessydra, Cortex A72 and Cortex M4 processors, the energy consumption was considered for the core and the memories only, excluding all the peripherals. For all the processors, we calculated the total energy consumption as the average power consumption

Table 5.2 Performance results for the Disjoint Algorithm in the Std. and Opt. versions on the examined execution platforms

	d	Cortex M4 STM32 Nucleo		Cortex A72 Raspberry PI 4		Klessydra T13 PULPino			
		Std.	Opt.	Std.	Opt.	Std.	Opt.	Std. with VCU	Opt. with VCU
Cycle count	4	433024	135586	198000	48000	96117	19078	27905	13371
	8	2523904	467402	1117500	141000	484456	52767	131972	28854
	16	19694862	1677126	7629000	487500	2734595	181357	498280	86287
	32	148464048	6499592	55540500	1857000	17562142	682584	2017706	304447
Execution time [μ s]	4	5412	1695	132	32	961	190	279	134
	8	31548	5843	745	94	4844	527	1320	289
	16	246185	20964	5086	325	27345	1813	4983	863
	32	1855800	81244	37027	1238	175621	6825	20177	3044
Energy cons. [μ J]	4	838	262	435	105	185	40	54	27
	8	4890	905	2458	310	939	113	259	59
	16	38158	3249	16783	1072	5387	397	986	179
	32	287649	12592	122189	4085	35651	1549	4095	645

Table 5.3 Performance results for the Hybrid Algorithm in the Std. and Opt. versions on the examined execution platforms

	f	Cortex M4 STM32 Nucleo		Cortex A72 RASPBerry PI 4		Klessydra T13 PULPino			
		Std.	Opt.	Std.	Opt.	Std.	Opt.	Std. with VCU	Opt. with VCU
Cycle count	4	5751676	4355380	2325000	1785000	1029600	909378	413274	327463
	8	28419702	14498906	11142000	5679000	4594114	2919300	1307512	974813
	16	176841698	52908188	67779000	20509500	26216770	10454626	4741130	3338271
	32	N.A.	N.A.	456472500	78625500	176168851	39638697	19156387	12768900
Execution time [μ s]	4	71895	54442	1550	1190	10296	9093	4133	3275
	8	355246	181236	7428	3786	45941	29193	13075	9748
	16	2210521	661352	45186	13673	262167	104546	47411	33383
	32	N.A.	N.A.	304315	52417	1761688	396487	191564	127689
Energy cons. [μ J]	4	11143	8438	5115	3927	1945	1746	847	677
	8	55063	28091	24512	12493	8912	5634	2706	2037
	16	342630	102509	149113	45120	53482	20700	10193	7344
	32	N.A.	N.A.	1004239	172976	389333	80466	43676	30006

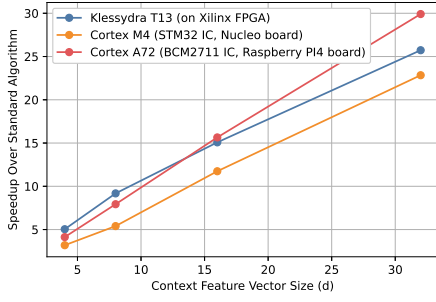
multiplied by the execution time. The energy reduction factor was then calculated as the ratio between the dynamic energy consumption of the original algorithm and the optimized algorithm ($\frac{\text{original_energy}}{\text{optimized_energy}}$).

Tables 5.2 and 5.3 summarize all the obtained results for the Disjoint and Hybrid algorithms, on the chosen embedded execution platforms, while Figures 5.9, 5.10, 5.11, and 5.12 depict them graphically. The analysis of the specific impact of each optimization step is reported in the following.

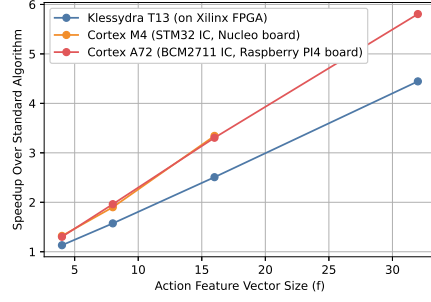
5.3.2 Algorithmic optimization impact on execution time

Efficient execution time is essential for deploying AI algorithms in applications with stringent real-time requirements. Figure 5.9a illustrates the impact of the Sherman-Morrison algorithmic optimization on the Disjoint LinUCB algorithm with N equal to 8. Varying the context feature vector size (d) from 4 to 32 the proposed approach achieves a speed up that ranges from $5.04\times$ to $25.73\times$ on Klessydra T13, from $3.19\times$ to $22.84\times$ on Cortex M4 and from $4.13\times$ to $29.91\times$ on Cortex A72. The observed linear increase in speedup with d aligns with the theoretical reduction in complexity from $\mathcal{O}(d^3)$ to $\mathcal{O}(d^2)$ described in Table 1 and Fig. 5.1. These results highlight how the removed matrix inversions constitute the performance bottleneck of the algorithm and strongly limit its scalability. Notably, increasing the number of actions, N , would further enhance the speedup provided by algorithmic optimization.

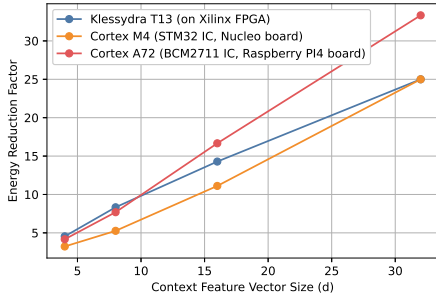
The same analysis on the execution time is depicted in Figure 5.9b for the



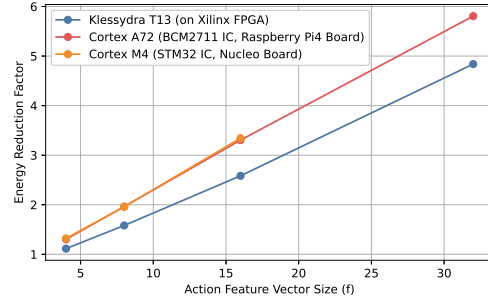
(a) Speedup for the Disjoint algorithm



(b) Speedup for the Hybrid algorithm

Figure 5.9 Algorithmic optimization impact on execution time.

(a) Energy reduction factor for the Disjoint algorithm



(b) Energy reduction factor for the Hybrid algorithm

Figure 5.10 Algorithmic optimization impact on energy consumption.

Hybrid algorithm, varying the action feature vector size (f) from 4 to 32, while keeping d and N equal to 8. The algorithmic optimization based on the Sherman-Morrison-Woodbury formulas results in speedups ranging from $1.13\times$ to $4.45\times$ on the Klessydra-T13 core, and from $1.30\times$ to $5.80\times$ on the Cortex A72. For the Cortex-M4 core, the speedup increases from $1.32\times$ at $f = 4$ to $3.34\times$ at $f = 16$. Notably, data for $f = 32$ are absent due to the memory demands of this configuration, which exceed the capabilities of the target Nucleo Board. The results demonstrate a linear increase in speedup with f , consistent with the theoretical reduction in complexity from $\mathcal{O}(f^3d^3)$ to $\mathcal{O}(f^2d^3)$, as outlined in Table 1 and Fig. 5.3. As detailed in Figure 5.3, increasing the value of d would increase the speedup for the optimized Hybrid algorithm, whereas increasing N has a minimal impact on performance due to the dominance of arithmetic operations on the complex $k \times k$ shared matrix.

5.3.3 Algorithmic optimization impact on energy consumption

Energy consumption is another critical factor when deploying AI algorithms in low-power environments. This section examines how the proposed algorithmic optimizations enhance the energy efficiency of the LinUCB algorithms.

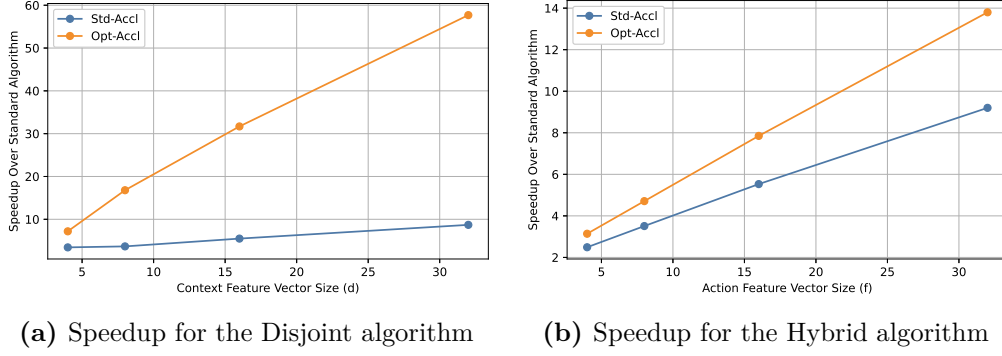


Figure 5.11 Impact of vector computing acceleration on execution time. Std-accl refers to the standard algorithm enhanced only by vector acceleration, while Opt-accl refers to the optimized algorithm enhanced with vector acceleration.

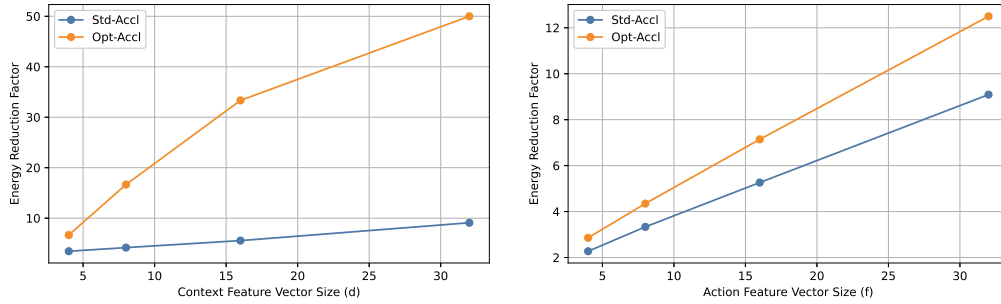
Figure 5.10a depicts the results obtained for the Disjoint algorithm varying the context feature vector size (d) while keeping N equal to 8. As already observed for the execution time, as d increases, so does the advantage brought by algorithmic optimizations. The energy reduction factor grows asymptotically as the ratio $\frac{\mathcal{O}(d^3)}{\mathcal{O}(d^2)} \propto d$, aligning with the theoretical analysis of the computational complexity presented in Table 1 and Fig. 5.1. On the Klessydra-T13 core, the original algorithm consumes from $4.54\times$ to $25\times$ the energy consumed by the optimized version, when d is equal to 4 and 32, respectively. A similar pattern in energy efficiency can be observed for the Cortex-M4 and Cortex A72 platforms, with energy reduction factors ranging from $3\times$ to $24.91\times$ and from $4.16\times$ to $33.33\times$, respectively.

For the Hybrid algorithm, the energy reduction obtained by the algorithmic optimization is shown in Figure 5.10b varying the action feature vector size (f) from 4 to 32, while keeping d and N equal to 8. The trend shows an asymptotic increase as the ratio $\frac{\mathcal{O}(f^3d^3)}{\mathcal{O}(f^2d^3)} \propto f$, as outlined in Table 1 and Fig. 5.3. On Klessydra, the original algorithm consumes from $1.11\times$ to $4.83\times$ the energy consumed by the optimized version. As already discussed for the execution time, the data for $f = 32$ with Cortex-M4 are not present due to the high memory demands of this configuration, but the trend for this platform closely follows the one obtained for the Cortex A72, in which the energy reduction factor ranges from $1.28\times$ to $5.76\times$.

5.3.4 Vector computing acceleration impact on execution time

The impact of vector computing support on the LinUCB algorithms was evaluated by executing both the standard and optimized versions on the Klessydra-T13 core, utilizing the VCU support for accelerating the basic matrix operations, as detailed in Section 5.2. The execution times have been measured and compared with the ones obtained by software executions and the results in terms of speedup are reported in Figures 5.11a and 5.11b for the Disjoint and Hybrid algorithms, respectively.

In the Disjoint version, vector computing acceleration achieves speedups ranging from $3.44\times$ at $d = 4$ to $8.70\times$ at $d = 32$ for the standard version. When combined with algorithmic optimizations, these speedups escalated dramatically ranging from



(a) Energy reduction factor for the Disjoint algorithm (b) Energy reduction factor for the Hybrid algorithm

Figure 5.12 Impact of vector computing acceleration on energy consumption. Std-accl refers to the standard algorithm enhanced only by vector acceleration, while Opt-accl refers to the optimized algorithm enhanced with vector acceleration.

7.18 $\times$ to 57.68 $\times$.

For the Hybrid algorithm, the vector computing acceleration applied to the standard version of the algorithm results in speedups ranging from 2.49 $\times$ when $f = 4$ to 9.19 $\times$ at $f = 32$. When vector acceleration and algorithmic optimization are synergically applied, the speedup ranges from 3.15 $\times$ to 13.82 $\times$, resulting in essential improvement for deploying complex algorithms like the Hybrid one on embedded systems.

Notably, the observed trends for the accelerated standard versions confirm the expected linear speedup provided by vector support when varying the matrix size observed in Fig. 5.8. Since both the algorithmic optimization and the vector acceleration independently contribute linear scaling properties, their combination results in an overall effect on speedup that remains linear but with a greater magnitude.

5.3.5 Vector computing acceleration impact on energy consumption

Finally, the impact of vector computing support on the energy consumption of the target algorithms is depicted in Figures 5.12a and 5.12b. For the standard Disjoint algorithm, the energy reduction factor grows from 3.40 $\times$ to 9.09 $\times$ as d is increased from 4 to 32. The optimized version further increases these values from 6.65 $\times$ to 50 $\times$. In the Hybrid algorithm, the energy reduction factor reaches 9 $\times$ for the standard version and 12.5 $\times$ for the optimized one.

The observed reductions align with theoretical expectations of a linear relationship with d and f due to the linear scaling properties of vector acceleration when varying matrix sizes, as demonstrated in Fig. 5.8.

5.3.6 Discussion

Overall, the obtained results demonstrate the effectiveness of the dual-pronged optimization strategy, which combines algorithmic improvements with vector processing

support to reduce execution time and energy consumption, enabling faster and more power efficient application of LinUCB algorithms in embedded learning systems. For the Disjoint algorithm, the algorithmic refinements based on the Sherman-Morrison formula replace heavy matrix inversions with lightweight incremental updates, resulting in significant execution time improvements across all platforms, with a maximum speedup on Klessydra equal to $25.73\times$. At the same time, although less effective in this case, vector computing support achieves a maximum speedup of $8.70\times$. Combined, the total speedup reaches $58\times$. A different trend can be observed for the Hybrid algorithm, which involves larger shared matrices and is inherently more complex. In this case, the Sherman-Morrison-Woodbury-based algorithmic optimization provides a maximum speedup of $4.45\times$, and the vector computing acceleration achieves $9.19\times$. Together, they yield a total speedup of $13.82\times$. Overall, increasing the size of context and action feature vectors enhances the advantage of vector parallel processing. The presented dual-pronged strategy effectively improves execution time and energy consumption, proving beneficial for deploying AI models in edge computing environments with low-power and real-time requirements.

5.4 Summary of Findings

This chapter introduced a dual-optimization strategy that combines algorithmic and hardware support techniques to address the computational bottlenecks of LinUCB Contextual Bandits on embedded platforms.

On the algorithmic side, we replaced explicit matrix inversion with incremental Sherman–Morrison–Woodbury updates, drastically reducing memory requirements and computational complexity. We extended this technique from the Disjoint to the more complex Hybrid formulation and demonstrated numerical stability through controlled error analysis.

On the hardware side, we mapped the reformulated algorithms onto the Klessydra-T13 Vector Coprocessor equipped with the VLNPD-based vector divider introduced in the previous chapters. By leveraging custom vector instructions for the matrix–vector operations that now dominate execution, we achieved substantial further acceleration.

Experimental validation across three platforms—the Cortex-M4 STM32, the Cortex-A72 Raspberry Pi 4, and the RISC-V Klessydra-T13—demonstrated speedups of up to $58\times$ and energy reductions of up to $50\times$ compared to standard baselines. These results confirm that the combination of algorithmic and architectural co-design can make online contextual bandit learning viable on resource-constrained embedded devices.

Together with the VLNPD (Chapter 3) and its vector integration (Chapter 4), the optimizations presented in this chapter demonstrate that significant gains can be extracted from the linear-algebraic formulation of contextual bandits through cross-layer co-design making linear bandits viable at the edge. However, the fundamental scaling limitations of dense linear algebra remain: covariance matrices still grow quadratically with the context dimension, imposing both a quadratic memory footprint and a computational cost that scales at least quadratically per decision step, making the approach inherently difficult to scale to high-dimensional problems on memory-constrained platforms. Part II explores a radically different path, replac-

ing the linear-algebraic machinery entirely with Hyperdimensional Computing, a neuro-inspired paradigm that operates on lightweight, high-dimensional holographic representations and trades dense matrix algebra for efficient element-wise vector operations.

Part 2
From Linear Algebra to
Neuro-inspired Computing

Overview

Part I of this dissertation investigated how far *arithmetic*, *algorithmic*, and *microarchitectural* co-design can push the efficiency of linear bandits on resource-constrained platforms. We started from the observation that action selection and model updates are dominated by repeated linear algebra operations, and identified matrix inversion (and, in practice, integer division within inversion kernels) as a critical bottleneck for embedded online decision-making. We addressed these bottlenecks at multiple levels: the VLNPD reduced the cost of integer division; its integration into a vector accelerator mitigated the divider bottleneck under SIMD execution; and the Sherman–Morrison–Woodbury reformulation eliminated explicit matrix inversion. This co-design approach achieved up to $58\times$ speedup and $50\times$ energy reduction over standard baselines.

Yet even with these algorithmic and arithmetic optimizations, the inherent limitations of the linear algorithm remain. These algorithms ultimately operate within the constraints of dense linear algebra: real-valued vectors, covariance matrices, and arithmetic whose memory and computation scale quadratically with problem size. To fundamentally decouple learning performance from these costs, a paradigm shift is required.

Part II marks a deliberate change of perspective. Instead of further optimizing *how fast* we execute the same workload, we ask whether we can change the *nature of the computation itself*.

Inspired by the brain’s ability to support robust cognition through massively parallel, noise-tolerant, and low-precision operations, this part leverages hyperdimensional computing (Chapter 2) to replace standard linear models with neuro-inspired representations based on high-dimensional, distributed vectors. By encoding information in the statistical patterns of thousand-dimensional spaces, HDC enables learning rules that are inherently parallel, robust to noise, and computationally lightweight, with complexity that scales *linearly* rather than quadratically with problem size.

The chapters in this part progressively develop and refine this paradigm:

- **Chapter 6** asks whether the precision matrix in linear bandits can be bypassed entirely by encoding contexts in a high-dimensional space. A systematic ablation across low-dimensional, randomly projected, and biologically inspired sparse k WTA-encoded representations shows that high-dimensional nonlinear encoding is the critical ingredient, providing the theoretical motivation for the following chapters.
- **Chapter 7** establishes the theoretical framework by introducing HD-CB, the first model to solve Contextual Bandit problems directly in high-dimensional space using component-wise operations.
- **Chapter 8** specializes the framework for extreme efficiency with HD-CB_{BIN}, a binary variant that minimizes memory footprint for deployment on low-end microcontrollers.
- **Chapter 9** extends the paradigm to hybrid settings, demonstrating how shared feature representations can be learned in hyperspace to model complex context–action interactions.

- **Chapter 10** presents efficient implementations of HD-CB that improve execution speed and reduce memory footprint.
- **Chapter 11** advances the learning foundations of HD-CB, transforming it from a brain-inspired heuristic into a principled, optimization-grounded framework. Probabilistic update rules compatible with bounded integer arithmetic replace unbounded accumulation, enabling deployment with as few as 3 bits per component. A derivation of the implicit loss function minimized by the baseline update rule reveals an ensemble structure and diagnoses its structural limitations. A loss-based formulation then replaces heuristic updates with explicit optimization objectives; its combination with probabilistic updates for low-precision deployment traces a Pareto frontier that no other variant family exceeds. Finally, a formal mapping of the entire HD-CB pipeline onto Random Vector Functional Link networks grounds the framework in established neural network theory.

Chapter 6

From Covariance Matrices to Associative Memories

This Chapter presents original work not yet published

If you can't solve a problem, then
there is an easier problem you can
solve: find it.

George Pólya

Part II Overview argued that overcoming the quadratic complexity and memory of the covariance matrices in linear contextual bandits requires a fundamental change in approaching the problem. This raises an important question:

❓ Research Question

What role makes the collection of covariance matrices indispensable in the first place, and can they be bypassed entirely by encoding contexts in a high-dimensional space?

This chapter addresses this research question through a sequence of controlled ablation experiments that gradually modify the standard linear bandit estimator. We begin by isolating the role of the precision matrix in the linear bandit estimator, showing that it acts as a *decorrelation operator*. We then test whether the decorrelation step could be made redundant by encoding contexts into a high-dimensional space. Three encoding regimes are investigated: (i) the original, low-dimensional features; (ii) a linear random projection into $\mathbb{R}^D$ via a dense Gaussian matrix; and (iii) a biologically-inspired nonlinear encoding [284, 285], sparse random projection followed by k -Winners-Take-All (k WTA) operation.

The experiments yield three main findings. First, *dimensionality expansion alone is not enough*: a linear random projection preserves the correlational structure of the input and, therefore, does not alleviate the need in the precision matrix. Second, *nonlinearity is the critical ingredient of encoding*. Third, for the k WTA representations, we show that the non-inverted covariance matrix reduces to a scaled copy of the reward vector plus cross-talk noise, acting as an auto-associative memory that retrieves information already present in b_a ; the matrix can, therefore, be omitted

entirely, leaving a single high-dimensional vector per action, queried by the inner product with the k WTA representation of the current context.

These findings establish the theoretical foundation and empirical evidence for the transition to Hyperdimensional Computing that follows in the subsequent chapters.

6.1 The Role of Precision Matrix in Linear Bandits

6.1.1 Linear Bandits without Precision Matrix

As established in Chapter 2, for each action $a \in \{1, \dots, N\}$, the linear contextual bandit framework maintains two sufficient statistics: a covariance matrix $A_a \in \mathbb{R}^{d \times d}$ and a reward accumulator vector $b_a \in \mathbb{R}^d$:

$$A_a = \sum_{t: a_t=a} x_t x_t^\top, \quad b_a = \sum_{t: a_t=a} r_t x_t, \quad (6.1)$$

where $x_t \in \mathbb{R}^d$ is the context observed at round t and $r_t \in [0, 1]$ is the corresponding reward. The vector b_a is the reward-weighted superposition of the contexts previously observed for action a . The matrix A_a , in contrast, records how frequently context dimensions co-occur, capturing the second-order structure of the context distribution conditioned on action a .

During action selection, the model computes the parameter estimate:

$$\hat{\theta}_a = A_a^{-1} b_a. \quad (6.2)$$

The predicted reward is then computed as $x_t^\top \hat{\theta}_a$ for the current context x_t .

In this process, the precision matrix A_a^{-1} plays the essential role of *decorrelating* the accumulated context–reward signal in b_a . By acting as the inverse Gram matrix, multiplication by A_a^{-1} compensates for feature overlap, isolating the independent contribution of each direction. From a statistical perspective, $A_a^{-1} b_a$ converts correlated statistics into decorrelated regression coefficients.

This decorrelation step is critical for accurate reward estimation, but—as discussed in Part I—it comes at a substantial computational and memory costs. However, if the precision matrix calculation is omitted and replaced by the forward product $A_a b_a$, the effect is reversed: correlated directions are no longer suppressed but amplified. As a result, the estimate becomes biased toward the most frequently co-occurring feature patterns, regardless of whether those patterns are genuinely predictive of high rewards.

To make this failure mode explicit, we compare the standard linear bandit with the precision matrix ($\hat{\theta}_a = A_a^{-1} b_a$) against a no-inversion variant ($\hat{\theta}_a = A_a b_a$) on synthetic contextual-bandit environments. Table 6.1 reports the average results across a range of problem sizes ($N \in \{5, 10, 15, 20\}$ actions and $d \in \{10, 15, 20\}$ context dimensions). Replacing $A_a^{-1} b_a$ with $A_a b_a$ consistently degrades performance, reducing the average policy value by approximately 9% relative to the standard linear bandit estimator.

These observations prove that *the matrix inverse step is essential because context vectors are correlated*. Hence, the problem can be reformulated: rather than seeking faster algorithms for matrix inverse, we investigate whether the context vectors

Table 6.1 Impact of omitting the matrix inverse step on policy value. The forward product $A_a b_a$ consistently underperforms the ridge regression solution $A_a^{-1} b_a$ by approximately -9% . Results are averaged over 8 configurations ($N \in \{5, 10, 15, 20\}$, $d \in \{10, 15, 20\}$), 10 datasets $\times$ 5 simulation runs per configuration. All policies use $\epsilon = 0.05$.

Model	Avg. policy value	Gap vs. $A_a^{-1} b_a$
LinEPS ($A_a^{-1} b_a$)	0.691	—
LinEPS ($A_a b_a$)	0.597	$\approx -9\%$

Table 6.2 Policy value comparison: linear random projection ($D = 4096$) via a dense Gaussian matrix without the precision matrix versus the linear baselines. The projected policy performs identically to the unprojected baseline without the the precision matrix, and both trail the full-inversion reference by approximately -9% . Increasing D to 8192 results in no change. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.

Model	Avg. policy value	Gap vs. $A_a^{-1} b_a$
LinEPS ($A_a^{-1} b_a$)	0.691	—
LinEPS ($A_a b_a$)	0.597	$\approx -9\%$
Dense RP ($A_a b_a$, $D = 4096$)	0.598	$\approx -9\%$

themselves can be made less correlated, so that the matrix inverse step becomes redundant.

6.1.2 Linear Random Projection

A natural first step in this direction is to transform contexts into a high-dimensional space. The underlying hypothesis is that: if d -dimensional context vectors x_t are projected into a much larger space of dimension $D \gg d$, the resulting representations may become sufficiently separable that the need for explicit decorrelation is reduced.

To test this idea, we introduce a linear random projection via a dense Gaussian matrix $\Phi \in \mathbb{R}^{D \times d}$ with entries drawn i.i.d. from $\mathcal{N}(0, 1/D)$, and define the projected context vector as:

$$\mathcal{X}_t = \Phi x_t \in \mathbb{R}^D. \quad (6.3)$$

The model then operates entirely in the projected space:

$$A_a = A_a + \mathcal{X}_t \mathcal{X}_t^\top, \quad b_a = b_a + r_t \mathcal{X}_t, \quad (6.4)$$

and, in the variant without the precision matrix, the parameter vector is estimated as $\hat{\theta}_a = A_a b_a$.

Table 6.2 reports the results for linear random projection at $D = 4096$, compared with the two baselines from Table 6.1. The result is negative: the linear random projection provides no measurable benefit. The policy with high-dimensional vectors without the precision matrix performs essentially identically to the baseline with low-dimensional context vectors without the precision matrix, and both remain approximately 9% below the standard linear bandit estimator. Increasing the dimensionality D does not lead to any change.

The failure of the linear random projection is anticipated from the Johnson–Lindenstrauss (JL) lemma [245], which states that a linear random projection approximately preserves pair-wise distances: for a random matrix $\Phi \in \mathbb{R}^{D \times d}$ with i.i.d. Gaussian components, with high probability:

$$(1 - \varepsilon) \|x_1 - x_2\|^2 \leq \|\Phi x_1 - \Phi x_2\|^2 \leq (1 + \varepsilon) \|x_1 - x_2\|^2 \quad (6.5)$$

for all pairs x_1, x_2 in a finite dataset, provided $D = \Omega(\varepsilon^{-2} \log n)$. In other words, a linear random projection preserves the geometry: it expands dimensionality, but it does not fundamentally alter the linear correlational structure that makes inversion necessary. Thus, as anticipated, linear transformation of the context vectors cannot by itself remove the need for the precision matrix.

6.2 Variant with FlyHash Algorithm

6.2.1 Nonlinear Encoding via FlyHash Algorithm

Motivated by the negative result of the previous section, we now explore whether an encoding that introduces *nonlinearity* can change the correlational structure and convert correlated context vectors into decorrelated representations.

The encoding investigated here is inspired by the olfactory system of the fruit fly *Drosophila melanogaster*, which Dasgupta et al. [285] formalized as a computational algorithm called **FlyHash** and similar ideas were explored by Kanerva [202] for high-capacity associative memories and by Rachkovskij et al. [284, 286, 287] for forming similarity-preserving representations.

In the fly’s mushroom body, approximately 2,000 Kenyon Cells each receive sparse, random connections from only ~ 6 out of 50 Projection Neurons; a global inhibitory neuron then silences all but the most active $\sim 5\%$ of Kenyon Cells, implementing a *k-Winners-Take-All* competition [288, 289]. In essence, the computation is a two-step nonlinear transformation: a *sparse random projection* that expands dimensionality while keeping each output dependent on a small random subset of inputs, followed by *competitive inhibition* that converts dense activations into a sparse binary vector.

We implement **FlyHash** as a two-stage transformation that maps $x_t \in \mathbb{R}^d$ into a sparse binary vector $\mathcal{X}_t \in \{0, 1\}^D$. First, a *sparse random projection*: each of D output neurons sums exactly s randomly chosen input dimensions,

$$y_j = \sum_{i \in \mathcal{C}(j)} \tilde{x}_i, \quad |\mathcal{C}(j)| = s, \quad (6.6)$$

where $\mathcal{C}(j) \subset \{1, \dots, d\}$ is a fixed random connectivity set. Second, the *kWTA* nonlinearity silences all but the top k components:

$$\mathcal{X}_j = \begin{cases} 1 & \text{if } j \in \text{top-}k(y), \\ 0 & \text{otherwise.} \end{cases} \quad (6.7)$$

This nonlinearity converts graded activations into a sparse binary vector with exactly k active components out of D .

We evaluate the **FlyHash** encoding using the same estimator without the precision matrix as before, $\hat{\theta}_a = A_a b_a$. Table 6.3 reports the results. The **FlyHash**-encoded

Table 6.3 Policy value comparison: FlyHash encoding with unscaled rewards versus linear random projection, both without the precision matrix. FlyHash achieves +5% average improvement over linear projection, demonstrating that the nonlinear encoding alone provides a measurable benefit. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.

Model	Avg. policy value	Gap vs. $A_a^{-1}b_a$	$\Delta_{\text{Fly vs RP}}$
LinEPS ($A_a^{-1}b_a$)	0.691	—	—
Linear rand. proj. ($A_a b_a$)	0.598	$\approx -9\%$	—
FlyHash ($A_a b_a$, unscaled r)	0.659	$\approx -4\%$	+5%

policy achieves an average improvement of +5% over the linear random projection variant, reducing the gap to the standard linear bandit estimator from -9% to approximately -4% .

6.2.2 The A Matrix as an Auto-associative Memory

The results on FlyHash representations show that the nonlinear encoding produces a measurable improvement, but since the covariance matrix A is not inverted—and, thus, no longer performing decorrelation—what role does it play in the variant with FlyHash representations?

With the variant without the precision matrix, the score for action a given a query pattern $\mathcal{X}_q$ is:

$$\text{score}_a = \mathcal{X}_q^\top (A_a b_a). \quad (6.8)$$

When action a is selected, both statistics are updated *simultaneously*:

$$A_a = A_a + \mathcal{X}_t \mathcal{X}_t^\top, \quad b_a = b_a + r_t \mathcal{X}_t. \quad (6.9)$$

Therefore, A_a and b_a are accumulated from the same sequence of past observations for action a . If n_a samples have been observed for that action, we can write:

$$A_a = \sum_{i=1}^{n_a} \mathcal{X}_i \mathcal{X}_i^\top, \quad b_a = \sum_{i=1}^{n_a} r_i \mathcal{X}_i, \quad (6.10)$$

where each $\mathcal{X}_i \in \{0, 1\}^D$ is a sparse binary vector with exactly k active components.

The matrix A_a is, thus, a superposition of outer products, i.e., the classical *auto-associative memory* studied by Kohonen [290] and Anderson [291], queried with b_a :

$$A_a b_a = \left(\sum_{i=1}^{n_a} \mathcal{X}_i \mathcal{X}_i^\top \right) \left(\sum_{j=1}^{n_a} r_j \mathcal{X}_j \right) = \sum_{i=1}^{n_a} \sum_{j=1}^{n_a} r_j \mathcal{X}_i (\mathcal{X}_i^\top \mathcal{X}_j). \quad (6.11)$$

The two indices i and j do not represent different histories; they only enumerate pair-wise interactions within the same set of stored samples.

This expression decomposes into self-terms and cross-terms:

$$A_a b_a = \sum_{i=1}^{n_a} r_i \mathcal{X}_i (\mathcal{X}_i^\top \mathcal{X}_i) + \sum_{i \neq j} r_j \mathcal{X}_i (\mathcal{X}_i^\top \mathcal{X}_j). \quad (6.12)$$

Since each FlyHash code is k -sparse binary, it has constant norm $\mathcal{X}_i^\top \mathcal{X}_i = \|\mathcal{X}_i\|^2 = k$:

$$A_a b_a = k \sum_{i=1}^{n_a} r_i \mathcal{X}_i + \underbrace{\sum_{i \neq j} r_j \mathcal{X}_i (\mathcal{X}_i^\top \mathcal{X}_j)}_{\eta_a} = k b_a + \eta_a. \quad (6.13)$$

The first term is simply a constant rescaling of b_a , while η_a represents the cross-talk noise caused by nonzero overlaps between different k WTA representations stored in the memory.

This reveals that, in the variant without the precision matrix, A_a is no longer acting as a covariance-correction operator. Instead, it behaves as an auto-associative memory applied to b_a . But b_a is itself the reward-weighted superposition of exactly the same previously encountered patterns. Therefore, multiplying by A_a does not introduce new information: it mainly reproduces what is already stored in b_a , up to a constant factor, while adding additional cross-talk noise.

This suggests a radical simplification: *drop A_a entirely* and use b_a directly as the action representation. The score for action a given query pattern $\mathcal{X}_q$ becomes:

$$\text{score}_a = \mathcal{X}_q^\top b_a = \sum_{t: a_t=a} r_t (\mathcal{X}_q^\top \mathcal{X}_t). \quad (6.14)$$

This is a pure *similarity-based* decision rule: the score is the sum of rewards from past contexts, each weighted by its similarity to the current query. In this form, the policy relies directly on the reward-weighted superposition stored in b_a , without the additional memory-retrieval step induced by A_a .

Dropping A_a removes the covariance matrix entirely: there is no $D \times D$ structure to store and no matrix-vector multiplication to compute at inference time. Computationally, this reduces the memory footprint from $\mathcal{O}(ND^2)$ (one $D \times D$ matrix per action) to $\mathcal{O}(ND)$ (one D -dimensional vector per action), and the per-round update from a rank-one matrix update to a simple vector addition.

From a data-structure perspective, the resulting vector b_a can be viewed as a variant of CountSketch algorithm [292]: when the k WTA selection is organized into a block sparse structure with one winner per block [293–295], each block acts as an independent hash row with exactly one active component, mirroring the CountSketch layout directly; the principal difference reduces to the scaling of the hashes, i.e., of the sparse block codes that the FlyHash encoding produces.

Table 6.4 compares the standard linear bandit estimator with two variants with the FlyHash-encoding: the full product $A_a b_a$ and the b_a -only estimator. Omitting A_a results in no measurable difference in policy value compared to the $A_a b_a$ estimator.

6.2.3 Reward Centering

It should be noted that, in the standard linear bandit estimator, the reward accumulator is updated as:

$$b_a \leftarrow b_a + r_t x_t, \quad (6.15)$$

a binary reward of $r_t = 0$ produces no update to b_a . In the standard linear bandit estimator, this is still manageable because the covariance matrix A_a is updated

Table 6.4 Effect of omitting the covariance matrix A_a in the variant with FlyHash. The estimator with only b_a vector matches the $A_a b_a$ variant, confirming that A_a does not contribute useful information. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.

Model	Avg. policy value	Gap vs. $A_a^{-1}b_a$
LinEPS ($A_a^{-1}b_a$)	0.691	—
FlyHash ($A_a b_a$)	0.659	$\approx -4\%$
FlyHash (b_a)	0.657	$\approx -4\%$

Table 6.5 Effect of reward centering ($r' = r - 0.5$) on the b_a -only FlyHash estimator. Centering further closes the gap to the standard linear bandit estimator, from $\approx -4\%$ to $\approx -2\%$. Results are averaged over 8 configurations, 10 datasets $\times$ 5 simulation runs.

Model	Avg. policy value	Gap vs. $A_a^{-1}b_a$	Δ vs. Dense RP
LinEPS ($A_a^{-1}b_a$)	0.691	—	—
Dense RP ($A_a b_a$)	0.598	$\approx -9\%$	—
FlyHash (b_a only, unscaled r)	0.657	$\approx -4\%$	+5%
FlyHash (b_a only, centered r')	0.674	$\approx -2\%$	+7%

regardless of reward and, therefore, continues to track the occurrence of context vectors with low rewards. In the b_a -only FlyHash variant, however, the model must receive an explicit negative signal to learn from mistakes, especially in environments with binary rewards.

A simple correction is to center the binary reward around its midpoint:

$$r'_t = r_t - 0.5. \quad (6.16)$$

Under this centering correction, a positive reward results in a positive update, while a zero reward produces an *active negative signal* that pushes b_a away from the observed context vector, enabling the model to learn symmetrically from both positive and negative outcomes.

Table 6.5 reports the results when rewards are centered. The FlyHash variant improves from approximately -4% to approximately -2% relative to the standard linear bandit estimator, corresponding to a $+7\%$ improvement over the linear random projection variant.

6.3 From Ablation to Hyperdimensional Computing

The ablation trajectory presented in this chapter can be summarized as a sequence of four progressive simplifications, each justified by the corresponding experiment:

1. **Remove the precision matrix:** Replacing $A_a^{-1}b_a$ with $A_a b_a$ costs $\approx 9\%$ in policy value, confirming that decorrelation is essential when operating on original or randomly projected context vectors.
2. **Introduce nonlinear encoding:** The FlyHash encoding—sparse random projection followed by k WTA nonlinearity—produces sparse binary representations that recover 5% of the lost performance without the precision matrix.

3. **Omit the covariance matrix:** In the variant with the FlyHash encoding, A_a degenerates into an auto-associative memory that adds only cross-talk noise; omitting it has no negative impact and simplifies the model to a single high-dimensional vector per action.
4. **Center the reward signal:** Correcting $r' = r - 0.5$ provides an explicit negative signal, recovering an additional 2% and leaving the model only $\approx 2\%$ below the standard linear bandit estimator while eliminating all matrix operations.

The resulting model at the end of these ablations is strikingly simple:

- Each action a is represented by a single D -dimensional vector b_a .
- At each round, the context x_t is encoded through a fixed nonlinear map to produce $\mathcal{X}_t \in \{0, 1\}^D$.
- Action selection is done by inner product: $a^* = \arg \max_a \mathcal{X}_t^\top b_a$.
- After observing reward r_t , the selected action's representation is updated incrementally: $b_{a^*} \leftarrow b_{a^*} + r'_t \mathcal{X}_t$.

This operational description is, in every detail, an instance of a *Hyperdimensional Computing*: an approach that encodes inputs into high-dimensional, quasi-orthogonal representations, stores class prototypes as superpositions of encoded training examples, and predicts based on similarity to these prototypes.

Equipped with this understanding, the next chapter introduces the HD-CB (*Hyperdimensional Computing for Contextual Bandits*), which generalizes and extends the model derived here to a broader family of models grounded in the theory of Hyperdimensional Computing. The ablation results of this chapter guarantee that the transition is not an ad hoc design choice but a principled consequence of eliminating the computational bottleneck at the core of the standard linear bandit estimator.

Chapter 7

HD-CB: The First Exploration of Hyperdimensional Computing for Contextual Bandits Problems

Chapter based on Paper 4 [225]: Marco Angioli, Antonello Rosato, Marcello Barbirotta, Rocco Martino, Francesco Menichelli, and Mauro Olivieri. Hd-cb: The first exploration of hyperdimensional computing for contextual bandits problems. IEEE Open Journal of the Computer Society, pages 1–12, 2025. doi: 10.1109/OJCS.2025.3642108

Those who are inspired by a model
other than Nature labor in vain.

Leonardo da Vinci

The previous chapter established, through controlled ablation, that nonlinear encoding into a high-dimensional space is the critical ingredient that allows the precision matrix to be dropped from the linear bandit estimator. Under this encoding, a single high-dimensional vector per action, queried by inner product, suffices for competitive decision-making. Hyperdimensional computing offers exactly this computational model: structured nonlinear encoding, distributed high-dimensional representations, and simple element-wise operations that are inherently parallel, noise-tolerant, and compatible with low-precision arithmetic. This naturally raises the next question:

❓ Research Question

Can hyperdimensional computing serve as a complete computational substrate for contextual bandit learning, inheriting its simple, distributed, and low-precision properties while retaining competitive decision-making performance?

As introduced in Chapter 2, HDC represents information through high-dimensional hypervectors manipulated using simple primitives—bundling, binding, and permutation—that are naturally parallel, tolerant to noise and hardware faults, and well suited to low-precision implementations [216, 217, 296, 297]. These properties have enabled competitive performance across classification [218], clustering [221], and regression [219] tasks, with growing interest in reinforcement learning settings [224, 298].

Yet no prior work has applied HDC to the contextual bandit problem.

This chapter, mainly based on Paper 5 [225], bridges the CB and HDC research lines by casting the entire contextual bandit pipeline into high-dimensional space. The resulting framework, termed *Hyperdimensional Contextual Bandits* (HD-CB), encodes contexts as hypervectors, represents each action by a dedicated prototype, and performs action selection via similarity computation. Learning proceeds by adjusting action prototypes toward or away from observed contexts based on received rewards, replacing the costly ridge-regression machinery of linear bandits with simple, parallel vector arithmetic.

This reformulation yields several advantages over classical linear bandits. HD-CB achieves linear scaling in both computational complexity and memory, faster convergence, and significant advantages in execution time when accelerated in hardware, emerging as a strong candidate for real-time, resource-constrained environments.

The main contributions of the study are the following:

- The work represents the first exploration of HDC to model and automate CB problems and ranks among the earliest explorations of HDC-based RL.
- It details the approach in four variants tailored to different exploration strategies and resource constraints.
- It analyzes the performance in terms of average reward, execution time, convergence speed, and memory footprint while varying the problem parameters and hypervector size.
- It compares all of the proposed schemes with traditional CB algorithms from the literature using synthetic datasets and real benchmarks.

Importantly, while this chapter employs real-valued HVs following the Multiply-Add-Permute-C implementation [5] from Gayler [228], the proposed methodology can be extended to other representations, such as complex or binary elements, offering flexibility to balance performance and computational complexity. Subsequent chapters will explore these alternatives.

7.1 Proposed Approach

7.1.1 HD-CB Basics

Figure 7.1 provides an overview of the working principle of the proposed HD-CB algorithm. Each action $a \in \{1, \dots, N\}$ is modeled by a dedicated representative vector $\mathcal{A}_a$ in the high-dimensional space. This vector is initialized to zero and is iteratively updated over time to capture the quality of the action within the current state of the environment.

At each iteration t , the model maps the observed context $x_{t,a}$ into a high-dimensional HV, $\mathcal{X}_{t,a}$. In this work, this encoding step is performed using the *record-based* technique introduced in Chapter 2, which quantizes each feature into discrete levels, maps each level to a hypervector, and binds the result with a position-specific basis hypervector before bundling all key-value pairs into a single representation. This technique is a nonlinear projection: a property identified in

Chapter 6 as necessary for bypassing the precision matrix. The HD-CB framework is encoding-agnostic, and alternative techniques may be equally integrated, provided they preserve this nonlinearity.

The estimated payoff $e_{t,a}$ of each action a is then computed as the similarity between $\mathcal{A}_a$ and $\mathcal{X}_{t,a}$, as in (7.1).

$$e_{t,a} = \delta(\mathcal{A}_a, \mathcal{X}_{t,a}) \quad (7.1)$$

These estimations serve as the basis for selecting the action a_t according to the adopted exploration strategy, whose integration will be discussed in the following subsections. After selecting an action, the model receives a real-valued payoff r_{t,a_t} , which is projected into the hypervector $\mathcal{R}_{t,a_t}$ and used to update the HD-CB model as in (7.2).

$$\mathcal{A}_{a_t} = \mathcal{A}_{a_t} \oplus (\mathcal{X}_{t,a_t} \otimes \mathcal{R}_{t,a_t}) \quad (7.2)$$

Specifically, $\mathcal{R}_{t,a_t}$ is generated using a thermometer code [237] where the number of active elements increases with the magnitude of the scalar value. This encoding ensures that $\mathcal{A}_{a_t}$ is adjusted closer to or further from the encoded context based on the received reward, learning at each iteration and capturing the stochastic nature of the environment. Additionally, it eliminates the need for the explicit reward centering that Chapter 6 required to generate negative feedback from binary rewards. In Figure 7.1, the collection of $\mathcal{X}_{t,a}$ and $e_{t,a}$ for all actions are denoted as $\mathcal{X}_t$ and $\mathbf{e}_t$.

An important qualitative property of the HD-CB update rule is its inherent tendency to promote exploration, which emerges naturally from the interplay between thermometer-encoded rewards and cosine similarity.

7.1.2 HD-CB_{EPS}

Algorithm 3 illustrates the HD-CB_{EPS} model, which incorporates the approach outlined in the previous section and employs the ϵ -greedy exploration strategy. At each iteration, with a probability of $1 - \epsilon$, the algorithm selects the action with the highest estimated reward $e_{t,a}$, while with a probability of ϵ , it explores by selecting a random action. This simple approach requires tuning only one hyperparameter (ϵ) and is well-suited for resource-constrained edge applications thanks to its low computational complexity. However, as discussed in Chapter 2, the ϵ -greedy strategy

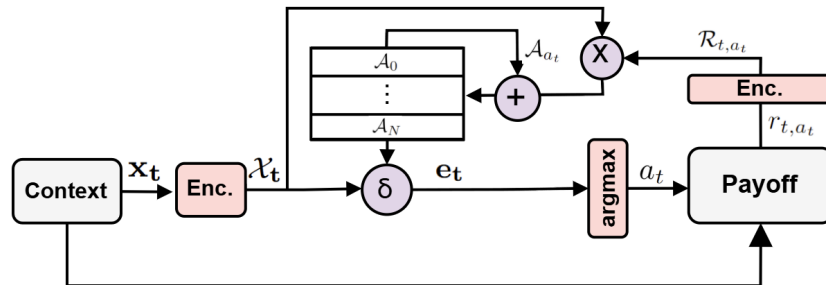


Figure 7.1 Illustration of the working principle of HD-CB. The collection of encoded hypervectors $\mathcal{X}_{t,a}$ for all actions a at time step t is denoted as $\mathcal{X}_t$, while the set of estimated payoffs $e_{t,a}$ is represented as $\mathbf{e}_t$.

Algorithm 3 HD-CB_{EPS} model

```

1: Input:  $\epsilon \in [0, 1]$ 
2: for  $t = 1, 2, \dots, T$  do
3:   for  $a = 1, 2, \dots, N$  do
4:     Observe context  $\mathbf{x}_{t,a} \in \mathbb{R}^d$ 
5:     Encode context into hypervector  $\mathcal{X}_{t,a}$ 
6:     if  $t = 1$  then ▷ Model initialization
7:        $\mathcal{A}_a = \mathbf{0}_D$ 
8:     end if
9:      $e_{t,a} = \delta(\mathcal{A}_a, \mathcal{X}_{t,a})$ 
10:   end for
11:   Draw  $n \sim \text{Uniform}(0, 1)$ 
12:   if  $n < \epsilon$  then ▷ Exploration
13:     Choose  $a_t$  uniformly at random from the action space
14:   else ▷ Exploitation
15:      $a_t = \arg \max_a e_{t,a}$ 
16:   end if
17:   Observe real-valued payoff  $r_{t,a_t}$ 
18:   Encode reward into hypervector  $\mathcal{R}_{t,a_t}$ 
19:    $\mathcal{A}_{a_t} = \mathcal{A}_{a_t} \oplus (\mathcal{X}_{t,a_t} \otimes \mathcal{R}_{t,a_t})$ 
20: end for

```

does not account for the confidence of the model in the expected payoffs of actions, potentially leading the algorithm to over-exploit actions with high uncertainty or under-explore potentially better alternatives. For example, if an action has a high estimated reward but has been sampled only a few times, the algorithm may over-exploit it—mistakenly treating it as the optimal choice. In contrast, an action with a slightly lower but more reliably estimated reward could be unfairly neglected, leading to suboptimal long-term performance.

7.1.3 HD-CB_{UNC1}: Integrating Contextual Confidence

To address the limitations of the ϵ -greedy strategy, we propose the HD-CB_{UNC1} model, depicted in Figure 7.2 and detailed in Algorithm 4. The core idea of this approach is to integrate a confidence measure into the expected payoff of each action. This is achieved by introducing an additional high-dimensional vector, $\mathcal{B}_a$, for each action a , which tracks how frequently the action has been chosen in similar contexts, independently of the received reward. To do this, after an action a_t is selected, the corresponding $\mathcal{B}_{a_t}$ vector is updated towards the current context $\mathcal{X}_{t,a_t}$ using the exponential moving average rule in (7.3).

$$\mathcal{B}_{a_t} = (1 - \beta)\mathcal{B}_{a_t} + \beta\mathcal{X}_{t,a_t} \quad (7.3)$$

Here, the *smoothing factor* β is a hyperparameter controlling the weight of each new observation. A smaller β leads to slower adaptation, allowing $\mathcal{B}_{a_t}$ to evolve gradually while preserving long-term memory. This incremental adjustment inherently increases

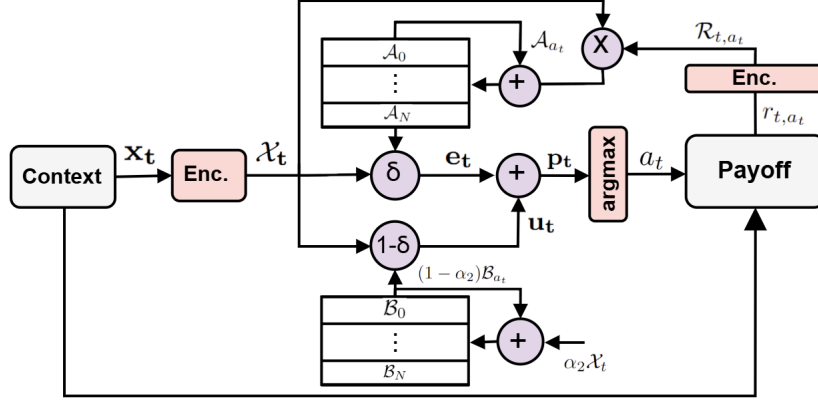


Figure 7.2 Schematic of HD-CB_{UNC1}. Each action a is associated with a confidence vector $\mathcal{B}_a$. The vectors $\mathcal{X}_t$, $\mathbf{e}_t$, $\mathbf{u}_t$, and $\mathbf{p}_t$ denote the context, estimated payoffs, uncertainties, and potentials for all actions at time t .

the similarity between $\mathcal{B}_{a_t}$ and the contexts where the action is frequently selected, directly translating into a confidence measure.

During action selection, HD-CB_{UNC1} evaluates each action based on two metrics: the estimated payoff ($e_{t,a}$) computed as in (7.1), and the uncertainty ($u_{t,a}$), derived from the confidence ($c_{t,a}$). The confidence is explicitly defined as the similarity between $\mathcal{B}_a$ and $\mathcal{X}_{t,a}$, and quantifies how familiar the model is with the current context.

$$u_{t,a} = 1 - c_{t,a} = 1 - \delta(\mathcal{B}_a, \mathcal{X}_{t,a}) \quad (7.4)$$

The sum of the estimated payoff and uncertainty determines the potential expectation ($p_{t,a}$) in (7.5). In this equation, the *exploration factor* α regulates the balance between exploration and exploitation.

$$p_{t,a} = e_{t,a} + \alpha \cdot u_{t,a} \quad (7.5)$$

Once $p_{t,a}$ is calculated for all actions, the one with the highest potential value, $a_t = \arg \max_a(p_{t,a})$, is selected. This formulation ensures that the HD-CB_{UNC1} effectively combines the expected payoff and the confidence level.

7.1.4 HD-CB_{UNC2}: Controlling Updates with Confidence

This subsection introduces HD-CB_{UNC2}, a variant that refines the confidence-based update mechanism of HD-CB_{UNC1} by replacing the fixed-rate exponential moving average with a confidence-driven *thinned update* mechanism. In this approach, the confidence value c_{t,a_t} of the selected action determines the proportion of dimensions in $\mathcal{B}_{a_t}$ that are updated at each time step. Specifically, a binary mask is generated, where each element is set to 1 with probability c_{t,a_t} and 0 otherwise. This mask is then applied to $\mathcal{X}_{t,a}$ as shown in (7.6), selectively updating $\mathcal{B}_{a_t}$.

$$\text{mask}_i = \begin{cases} 1, & \text{with prob. } c_{t,a_t}, \\ 0, & \text{otherwise,} \end{cases} \quad \forall i \in \{1, \dots, \text{HVsize}\}, \quad (7.6)$$

$$\mathcal{B}_{a_t} = \mathcal{B}_{a_t} \oplus (\mathcal{X}_{t,a} \odot \text{mask})$$

Algorithm 4 HD-CB_{unc1} model

```

1: Input:  $\alpha \in \mathbb{R}_+, \beta \in \mathbb{R}_+$ 
2: for  $t = 1, 2, \dots, T$  do
3:   for  $a = 1, 2, \dots, N$  do
4:     Observe context  $\mathbf{x}_{t,a} \in \mathbb{R}^d$ 
5:     Encode context into hypervector  $\mathcal{X}_{t,a}$ 
6:     if  $t = 1$  then ▷ Model initialization
7:        $\mathcal{A}_a = \mathbf{0}_D$ 
8:        $\mathcal{B}_a = \mathbf{0}_D$ 
9:     end if
10:     $e_{t,a} = \delta(\mathcal{A}_a, \mathcal{X}_{t,a})$ 
11:     $c_{t,a} = \delta(\mathcal{B}_a, \mathcal{X}_{t,a})$ 
12:     $u_{t,a} = 1 - c_{t,a}$ 
13:     $p_{t,a} = e_{t,a} + \alpha u_{t,a}$ 
14:  end for
15:   $a_t = \arg \max_a p_{t,a}$ 
16:  Observe real-valued payoff  $r_{t,a_t}$ 
17:  Encode reward into hypervector  $\mathcal{R}_{t,a_t}$ 
18:   $\mathcal{A}_{a_t} = \mathcal{A}_{a_t} \oplus (\mathcal{X}_{t,a_t} \otimes \mathcal{R}_{t,a_t})$ 
19:   $\mathcal{B}_{a_t} = (1 - \beta) \mathcal{B}_{a_t} + \beta \mathcal{X}_{t,a_t}$ 
20: end for

```

This method adaptively scales the impact of each update: when confidence is low, only a small fraction of dimensions is modified, resulting in a subtle increase in similarity with the current context, whereas when confidence is high, a larger portion of components is updated.

This approach offers two key advantages. First, it eliminates the need for the β hyperparameter, simplifying the model and removing the requirement for costly tuning procedures. Second, it replaces the scalar multiplication in (7.3) with bundling, confining all operations strictly to the standard HDC arithmetic. This restriction is especially beneficial in resource-constrained environments, as it keeps the HVs in the same representation space (e.g., integers) and facilitates hardware deployment.

7.1.5 HD-CB_{UNC3}: Compact Representation via Permutation

The HD-CB_{UNC1} and HD-CB_{UNC2} models require two distinct HVs, $\mathcal{A}_a$ and $\mathcal{B}_a$, for each action a to effectively track the estimated reward and uncertainty over time. However, as the number of actions and the HV size increase, this can lead to high memory overhead, potentially making these models impractical for resource-constrained embedded systems.

To address this challenge, the HD-CB_{UNC3} variant optimizes memory usage by leveraging the mathematical properties of the permutation operator. In this model, for a given action a , a unique permutation ρ^a is applied to the encoded context vector $\mathcal{X}_{t,a}$, generating an hypervector $\mathcal{S}_{t,a}$ that represents the context-action pair, as in (7.7).

Algorithm 5 HD-CB_{UNC2}

```

1: Input:  $\alpha \in \mathbb{R}_+$ 
2: for  $t = 1, 2, \dots, T$  do
3:   for  $a = 1, 2, \dots, K$  do
4:     Observe the context  $x_{t,a} \in \mathbb{R}^d$ 
5:     Encode the context into the  $\mathcal{X}_{t,a}$  HV
6:     if  $t = 1$  then
7:        $\mathcal{A}_a = \mathbf{0}_D$ 
8:        $\mathcal{B}_a = \mathbf{0}_D$ 
9:     end if
10:     $e_{t,a} = \delta(\mathcal{A}_a, \mathcal{X}_{t,a})$ 
11:     $c_{t,a} = \delta(\mathcal{B}_a, \mathcal{X}_{t,a})$ 
12:     $u_{t,a} = 1 - c_{t,a}$ 
13:     $p_{t,a} = e_{t,a} + \alpha \cdot u_{t,a}$ 
14:  end for
15:  Select action  $a_t = \arg \max_a (p_{t,a})$ 
16:  Observe a real-valued payoff  $r_{t,a_t}$ 
17:  Encode the reward as high-dimensional vector  $\mathcal{R}_{t,a_t}$ 
18:  Update model:
19:  -  $\mathcal{A}_{a_t} = \mathcal{A}_{a_t} \oplus (\mathcal{X}_{t,a_t} \otimes \mathcal{R}_{t,a_t})$ 
20:  - Thinning update: select  $c_{t,a_t} D$  random indices to form a mask.  $\mathcal{B}_{a_t}[mask] =$ 
     $(\mathcal{B}_{a_t} \oplus \mathcal{X}_{t,a_t})[mask]$ 
21: end for

```

$$\mathcal{S}_{t,a} = \rho^a(\mathcal{X}_{t,a}) \quad (7.7)$$

Permutation operation in HDC yields HVs that are nearly orthogonal to their originals [217]. This mathematical property ensures that each context-action pair is mapped to a unique, high-dimensional representation, allowing them to be superimposed into shared memory structures without destructive interference. As a result, instead of maintaining separate vectors for every action, HD-CB_{UNC3} can encode and retrieve relevant information using just two shared hypervectors, $\mathcal{M}_\mathcal{A}$ and $\mathcal{M}_\mathcal{B}$, initialized to zero. $\mathcal{M}_\mathcal{A}$ acts as a global repository for the outcomes of the selected context-action pairs, while $\mathcal{M}_\mathcal{B}$ encodes the frequency of encountered context-action pairs, regardless of the rewards.

At each iteration, the potential value of each action $p_{t,a}$ is computed as a combination of the estimated reward $e_{t,a}$ and the uncertainty $u_{t,a}$, calculated as in (7.8).

$$\begin{aligned} e_{t,a} &= \delta(\mathcal{M}_\mathcal{A}, \mathcal{S}_{t,a}) \\ u_{t,a} &= 1 - \delta(\mathcal{M}_\mathcal{B}, \mathcal{S}_{t,a}) \end{aligned} \quad (7.8)$$

After receiving the reward, the model updates as in (7.9).

$$\begin{aligned} \mathcal{M}_\mathcal{A} &= \mathcal{M}_\mathcal{A} \oplus (\mathcal{S}_{t,a_t} \otimes \mathcal{R}_{t,a_t}) \\ \mathcal{M}_\mathcal{B} &= (1 - \beta)\mathcal{M}_\mathcal{B} + \beta\mathcal{S}_{t,a_t} \end{aligned} \quad (7.9)$$

Notably, the thinned update from HD-CB_{UNC2} can also be applied here, eliminating scalar multiplications and further simplifying the update process for embedded applications.

7.2 Results

To comprehensively evaluate the performance and efficiency of the proposed HD-CB models, we conducted a series of experiments across diverse datasets and metrics. First, we assessed the online learning capabilities of the models on synthetic datasets by analyzing average reward, cumulative regret, and convergence time. Then, we benchmarked HD-CB against state-of-the-art approaches using the Open Bandit Pipeline (OBP) framework [76, 299] for off-policy evaluation, and the MovieLens-100k dataset in a real-world recommendation setting. Next, we investigated the impact of HV size on average reward and execution time, highlighting trade-offs between accuracy and processing cost. We further analyzed the scalability of HD-CB and LinUCB with respect to context size, evaluating both computational complexity and memory overhead.

7.2.1 Online Learning Capabilities

To assess the online learning capabilities of HD-CB, we designed synthetic datasets emulating typical CB problems. In these scenarios, the models iteratively observe new contexts, select actions, and receive rewards, mimicking real-world decision-making processes. These controlled environments offer a robust and replicable framework for evaluating the models, enabling a detailed analysis of their adaptability and learning performance under consistent and well-defined conditions. Specifically, each synthetic dataset consisted of continuous rewards in $[0,1]$, 20 possible actions, and continuous context vectors with $d = 10$ randomly sampled features. Context vectors were independently sampled at every iteration. For each action-context pair, the reward was determined by a distinct linear function parameterized by an action-specific weight vector, randomly initialized at the beginning of each dataset. To emulate realistic stochastic variability, continuous rewards were perturbed with Gaussian noise ($\mu = 0, \sigma = 0.1$). The encoding of the contexts into high-dimensional space was performed using the record-based encoding technique, employing $L = 10$ quantization levels and HV size $D = 1000$, with BVs generated using Python’s pseudo-random number generators.

Performance was evaluated over 100 independently generated synthetic datasets using two key metrics: *average reward* and *cumulative regret*. The average reward measures how effectively the model selects actions that maximize total payoff, while cumulative regret quantifies the difference between the rewards of optimal actions and those chosen by the model. A lower cumulative regret reflects the capability of the model to adapt quickly and align its decision-making with the optimal strategy over time. Experiments were run over 500 iterations. Figure 7.3 summarizes the results obtained varying the hyperparameters ϵ and α . All the subplots consistently demonstrate the capability of HD-CB models to adaptively learn and converge towards optimal solutions, as reflected by the average reward approaching 1.

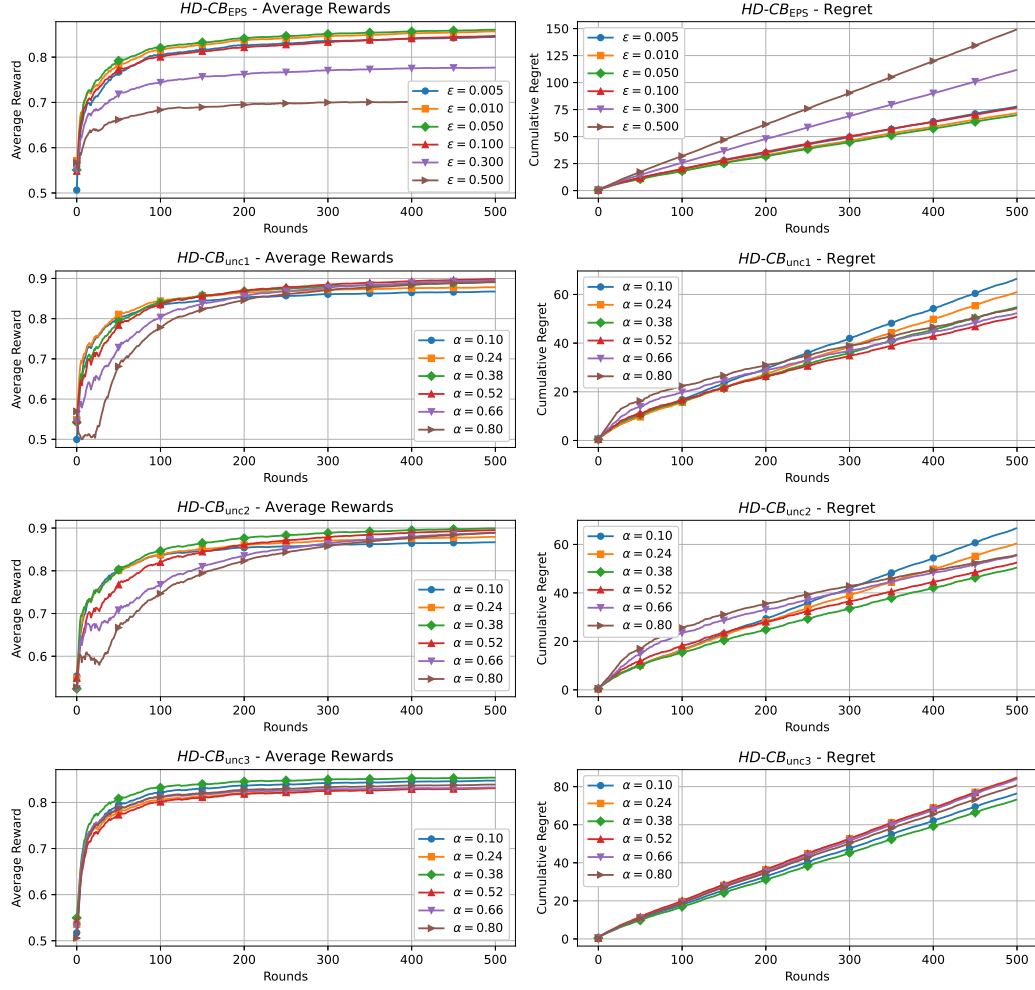


Figure 7.3 Average reward and cumulative regret trends for HD-CB models as a function of the exploration parameters (ϵ for HD-CB_{EPS} and α for the uncertainty-driven models), over 500 iterations, averaged across 100 synthetic datasets. The experiments were conducted with a hypervector size of 1000, 20 actions, continuous rewards in the range $[0,1]$, and a context feature vector size of 10.

Table 7.1 Off-policy evaluation for HD-CB models and traditional CB algorithms on synthetic datasets with binary rewards, with varying number of actions N and context feature vector sizes d . Average rewards $\pm$ std. deviation were calculated for each configuration across 50 datasets using OBP.

N	d	LinEPS	LinUCB	HD-CB _{EPS}	HD-CB _{UNC1}	HD-CB _{UNC2}	HD-CB _{UNC3}
10	5	0.628 $\pm$ 0.067	0.674 $\pm$ 0.061	0.670 $\pm$ 0.059	0.691 $\pm$ 0.062	0.691 $\pm$ 0.065	0.679 $\pm$ 0.062
10	10	0.689 $\pm$ 0.087	0.704 $\pm$ 0.055	0.690 $\pm$ 0.067	0.707 $\pm$ 0.068	0.711 $\pm$ 0.072	0.690 $\pm$ 0.068
15	5	0.734 $\pm$ 0.065	0.753 $\pm$ 0.050	0.758 $\pm$ 0.041	0.787 $\pm$ 0.046	0.794 $\pm$ 0.044	0.772 $\pm$ 0.046
15	10	0.691 $\pm$ 0.090	0.695 $\pm$ 0.074	0.696 $\pm$ 0.075	0.718 $\pm$ 0.078	0.724 $\pm$ 0.078	0.696 $\pm$ 0.078
20	5	0.705 $\pm$ 0.067	0.715 $\pm$ 0.059	0.722 $\pm$ 0.062	0.760 $\pm$ 0.062	0.761 $\pm$ 0.063	0.737 $\pm$ 0.060
20	10	0.631 $\pm$ 0.098	0.648 $\pm$ 0.088	0.647 $\pm$ 0.092	0.668 $\pm$ 0.093	0.674 $\pm$ 0.091	0.645 $\pm$ 0.090

Similar to the traditional CB algorithms, the performance of all HD-CB models is influenced by the exploration parameters (i.e., ϵ in HD-CB_{EPS} and α in the other variants). Small parameter values (e.g., $\epsilon = 0.01$ or $\alpha = 0.1$) result in insufficient exploration, leading to suboptimal performance characterized by slow reward growth and minimal reduction in cumulative regret. Conversely, high values lead to over-exploration, where the models continue to explore unnecessarily even after identifying the optimal action.

When comparing the models in terms of average rewards, HD-CB_{UNC2} demonstrates the highest performance, achieving an average reward of approximately 0.908 and a cumulative regret of 50.36 with $\alpha = 0.38$. The HD-CB_{UNC1} and HD-CB_{UNC3} models follow with maximum average rewards of 0.899 and 0.863 and cumulative regrets of 50.70 and 69.13 when using α equal to 0.52 and 0.38, respectively. Finally, HD-CB_{EPS} achieves a maximum reward of 0.85 and a regret of 69.76 when $\epsilon = 0.05$. This ranking aligns with the computational complexity of the models: HD-CB_{EPS} and HD-CB_{UNC3} are the simplest and least computationally demanding, whereas HD-CB_{UNC1} and HD-CB_{UNC2} are more complex but deliver higher performance.

7.2.2 Off-Policy Evaluation

To benchmark the performance of the proposed models against traditional linear algorithms, we utilized the OBP [76]: an open-source framework designed to generate synthetic datasets and provide standardized off-policy evaluations of bandit algorithms. Off-policy evaluation (OPE) plays a critical role in reinforcement learning and contextual bandit settings, offering a safe, efficient, and reproducible way to estimate the performance of a new policy. In an online setting, in fact, only the reward for the selected action is observed at each time step [300], making offline evaluation on logged data essential for obtaining unbiased performance estimations of target policies. OPE is particularly important in high-stakes domains, such as healthcare, where deploying an untested model could lead to costly or risky outcomes. Using the OBP framework, we conducted tests on 50 different synthetic datasets with binary rewards, varying the number of actions (N) and the context feature vector size (d). Table 7.1 summarizes the average rewards achieved by all HD-CB models compared to LinEPS and LinUCB algorithms. For all the algorithms, we performed a grid search over the ϵ , α and β (only for HD-CB_{UNC1} and HD-CB_{UNC3}) parameters in the range $[0.1, 0.2, \dots, 1]$. Additionally, we searched for the optimal number of quantization levels from the set $[10, 15, 20]$ for the HD-CB models.

When employing an ϵ -greedy strategy, HD-CB $_{\epsilon}$ consistently outperforms LinEPS, with an average improvement of 1.75% across all dataset configurations. This result is particularly significant as it directly compares the simplest versions of both frameworks, utilizing the same exploration strategy. The superior performance of HD-CB underscores the potential of HDC over traditional linear algorithms, even when paired with simple exploration strategies.

The uncertainty-driven HD-CB models consistently match or outperform LinUCB across all tested configurations. HD-CB_{UNC2} achieves the highest average reward, outperforming LinUCB by 4% on average. HD-CB_{UNC3}, which uses a more compact representation via permutation, also performs competitively while requiring significantly less memory, making it ideal for memory-constrained environments.

Overall, these findings validate the effectiveness of the proposed HD-CB framework for modelling and solving CB problems, demonstrating comparable or superior performance to traditional algorithms. By mapping the problem into a high-dimensional space, HD-CB effectively separates context-action pairs, enhancing learning efficiency.

7.2.3 Evaluation on the MovieLens-100k Dataset

To evaluate the performance of HD-CB models on real-world data, we conducted tests using the MovieLens-100k dataset, a standard benchmark for recommendation systems containing 100,000 ratings (ranging from 1 to 5) from 943 users on 1,682 movies. This dataset is often used to evaluate recommender systems in a batch setting, where models have full access to the user-item matrix and can achieve RMSE values between 0.90 and 0.98 [301, 302]. In contrast, this section frames the task as a contextual bandit problem. For each encountered user (and their associated contextual information), the policy must recommend a movie (treated as an arm), and only observes the binary reward based on the user's actual rating. Specifically, ratings of 4 or higher are considered positive and assigned a reward of 1, while all others receive a reward of 0. Before evaluation, we preprocessed the dataset by grouping users into age buckets and applying one-hot encoding to the age group, gender, and occupation, forming the context vector for each decision.

Figure 7.4 shows the average rewards achieved by the traditional LinUCB algorithm and our best-performing HD-CB_{UNC2} model, averaged over 50 runs, with the number of available actions (i.e., movies) ranging from 100 to 1000. A grid search was used to determine the optimal values for the α and β hyperparameters. The results indicate that HD-CB consistently outperforms LinUCB, particularly as the number of actions increases. For $N = 100$, both models achieve similar average rewards, with HD-CB_{UNC2} slightly outperforming LinUCB (0.824 vs. 0.821). However, as N increases, the performance gap becomes more pronounced. For $N = 1000$, HD-CB_{UNC2} achieves an average reward of 0.720, significantly higher than LinUCB's 0.320, demonstrating robustness and better scalability in large action spaces.

7.2.4 Convergence Time Analysis

In online decision-making, convergence time is a key metric. Unlike offline learning, where models are trained prior to deployment, online algorithms must continuously adapt in real time based on observed rewards. Rapid convergence enables better decisions early in the learning process, essential in applications such as healthcare and autonomous systems.

HDC has demonstrated faster convergence and lower data requirements in various learning tasks [239, 296], owing to its ability to enhance separability in high-dimensional space. To evaluate this property in the HD-CB framework, we compared its convergence speed to that of traditional LinUCB models on synthetic datasets, averaging results over 100 runs while varying the exploration parameter α .

Figure 7.5 illustrates the results in terms of cumulative reward as a percentage of the final reward over time. Across all tested α values, HD-CB models consistently

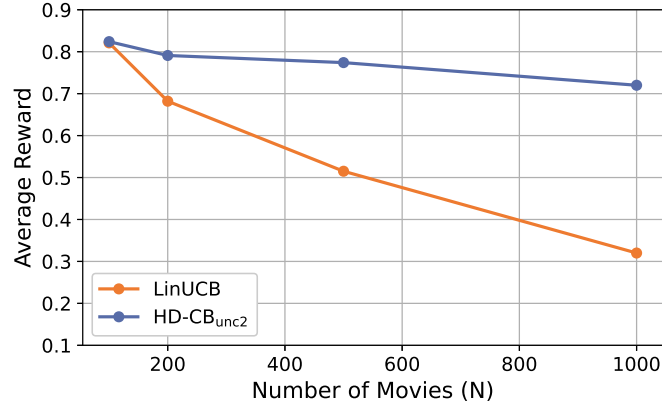


Figure 7.4 Results obtained on the MovieLens-100k dataset by LinUCB and HD-CB models, varying the number of movies.

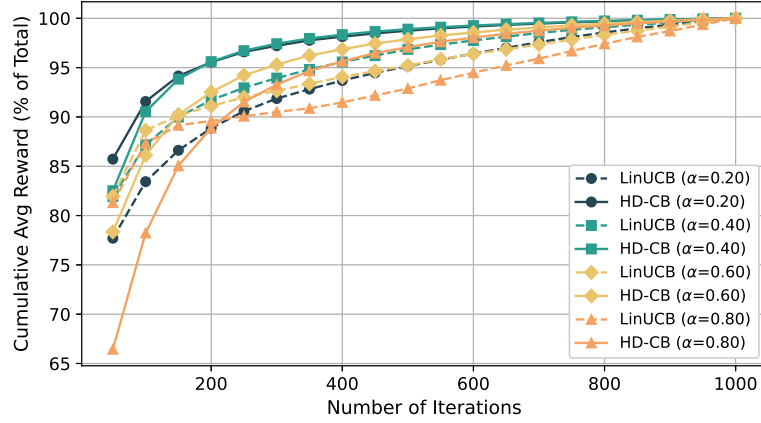


Figure 7.5 Convergence time as a function of α , comparing the LinUCB algorithms and the HD-CB_{UNC1} model.

converged faster than LinUCB, confirming their suitability for scenarios requiring rapid adaptation.

7.2.5 Computational Complexity Analysis

The computational complexity of CB algorithms can be critical in resource-constrained applications requiring rapid decision-making. Linear bandits rely on the inversion of $N \times d \times d$ covariance matrices at each step, resulting in an asymptotic complexity of $\mathcal{O}(d^3)$, and a significant computational burden that can create performance bottlenecks. This complexity can be reduced to $\mathcal{O}(d^2)$ by employing the incremental update mechanism proposed in Chapter 5, which leverages the Sherman–Morrison theorem to efficiently update the inverse matrices without recomputing them from scratch.

The HD-CB framework eliminates the need for matrix inversion entirely. By

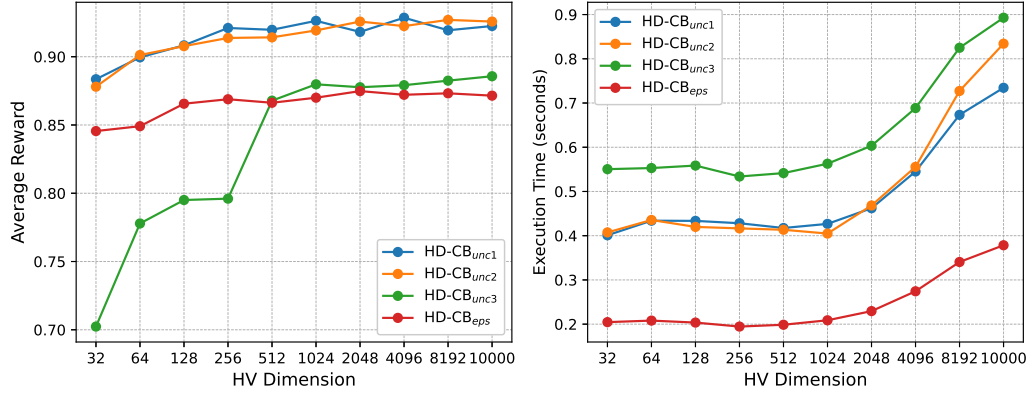


Figure 7.6 Impact of HV size on average reward and execution time for the proposed HD-CB models. The figure highlights the trade-offs between accuracy and computational efficiency as HV size varies from 32 to 10,000.

representing each action with a dedicated hypervector and using simple arithmetic operations in the high-dimensional space, HD-CB reduces the complexity with respect to the context size to $\mathcal{O}(d)$. Additionally, HD-CB maintains linear complexity in the number of actions ($\mathcal{O}(N)$) and introduces a dependence on HV size ($\mathcal{O}(D)$), making its computational cost primarily determined by D . The linear dependence of HD-CB on d provides a substantial advantage. Traditional CB algorithms often require dimensionality reduction techniques to avoid the computational burden of high-dimensional contexts, especially in domains such as personalized recommendations, where contexts may represent detailed user profiles. HD-CB addresses this issue, eliminating the need for additional pre-processing methods.

Table 7.2 summarizes the computational complexity of the algorithms, highlighting their respective dependencies on context size, number of actions, and HV size.

Table 7.2 Computational complexity of contextual bandit algorithms

Algorithm	Context Size (d)	N. Actions (N)	HV Size (D)
LinUCB [74]	$\mathcal{O}(d^3)$	$\mathcal{O}(N)$	—
LinUCB-SM [82]	$\mathcal{O}(d^2)$	$\mathcal{O}(N)$	—
HD-CB	$\mathcal{O}(d)$	$\mathcal{O}(N)$	$\mathcal{O}(D)$

7.2.6 Scalability Analysis

As shown in the previous subsection, the computational cost of HD-CB scales with the HV size D . However, in HDC, larger HV sizes enhance the representational capacity of the high-dimensional space, allowing the model to encode and discriminate between distinct objects more effectively [5]. For this reason, striking a balance between accuracy and efficiency is essential in HDC-based systems. To analyze this trade-off, we evaluated HD-CB models on 100 synthetic datasets with 10 features and 10 actions. The HV size was varied from 32 to 10,000, and $(\epsilon, \alpha, \beta)$ were selected

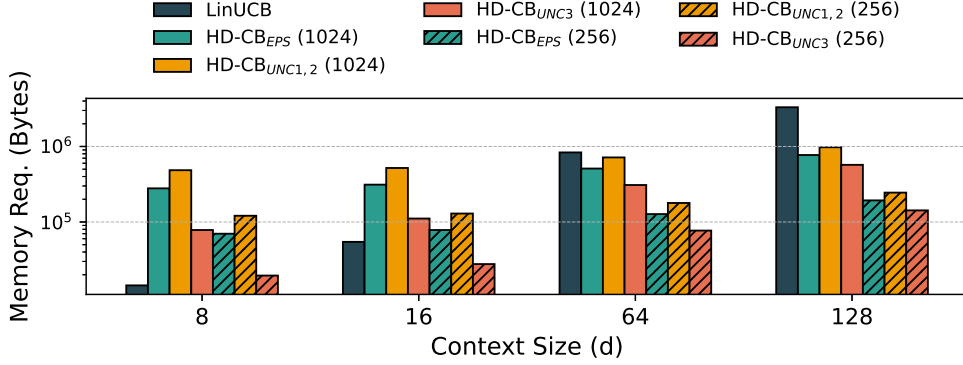


Figure 7.7 Memory requirements comparison for LinUCB and HD-CB, across varying context feature vector sizes (d) and two HV sizes (D).

via grid search, consistent with prior experiments. Figure 7.6 shows the resulting trends in average reward and execution time.

From a performance perspective, the HD-CB_{EPS}, HD-CB_{UNC1}, and HD-CB_{UNC2} models demonstrate strong robustness to reduced HV size, with performance losses below 2% when decreasing from 10,000 to 256 dimensions, and under 9% even at 32 dimensions. This resilience stems from their use of distinct representative HVs per action, which maintains discrimination capabilities even at low dimensionalities. In contrast, HD-CB_{UNC3} is more sensitive to HV size, showing a 10% drop at 256 dimensions and 20% at 32. This is expected since the model uses only two HVs shared across all actions and relies on permutation. These shared HVs are updated incrementally for all context-action pairs, quickly saturating the representational capacity at lower dimensionalities. However, its memory usage is reduced by a factor of $2N$ compared to other models, making it a strong candidate in memory-constrained environments and allowing for the deployment of higher HV sizes to recover performance.

In terms of execution time, HD-CB_{EPS} is the fastest due to its simplicity. HD-CB_{UNC1} and HD-CB_{UNC2} are roughly twice as slow, reflecting the overhead of maintaining two HVs per action, with HD-CB_{UNC2} slightly slower due to its thinning update. HD-CB_{UNC3} is the slowest, as it requires N additional permutations per iteration to encode actions using a reduced number of shared HVs.

7.2.7 Memory Footprint Analysis

Although HDC is often considered memory-consuming with respect to conventional computing due to the use of high-dimensional vectors, HD-CB demonstrates that this is not always the case. By modeling actions using HVs rather than matrices, HD-CB exhibits linear scaling of memory usage with respect to the context size d , in contrast to the quadratic scaling typical of traditional methods. Specifically, LinUCB must store N covariance matrices of size $(d \times d)$ and N regression vectors of size

($d \times 1$), leading to quadratic memory space growth. HD-CB, in comparison, requires ($d + L$) HVs for record-based encoding, along with a limited set of representative HVs: $2N$ for HD-CB_{UNC1} and HD-CB_{UNC2}, N for HD-CB_{EPS}, and just two for HD-CB_{UNC3}.

Figure 7.7 gives evidence to these differences, comparing the memory footprints of LinUCB with the four HD-CB variants across context sizes $d = 8$ to 128, for HV dimensions $D = 256$ and $D = 1024$ and $N = 50$ possible actions. While LinUCB is more memory-efficient for small d , HD-CB becomes increasingly advantageous as d grows. Among the HD-CB models, as expected HD-CB_{UNC3} emerges as the most compact variant. At $d = 128$, the HD-CB_{UNC2} version achieves memory reductions of up to 92.6% ($D = 256$) and 70.5% ($D = 1024$). Importantly, alternative encoding strategies such as n-gram-based encoding [303] can further reduce HD-CB’s memory footprint by eliminating the need for d BVs, albeit at the cost of slightly increased execution time.

These results confirm HD-CB is a scalable and efficient alternative to LinUCB in high-dimensional context settings.

7.2.8 Implicit Exploration Bias

Beyond the quantitative advantages documented in the preceding sections, the HD-CB update mechanism exhibits a qualitative property that contributes to its faster convergence and competitive reward performance: an *inherent tendency to promote exploration*, which arises naturally from the interplay between thermometer-encoded rewards and cosine similarity-based action selection. This section formalizes this observation through a step-by-step comparison with the linear bandit setting.

At the start of learning, both a linear bandit and an HD-CB assign zero estimated quality to every action. In the linear case, $\hat{\theta}_a = \mathbf{0}$ implies $\hat{r}_{t,a} = \mathbf{x}_t^\top \hat{\theta}_a = 0$ for all actions a . In the hyperdimensional case, $\mathcal{A}_a = \mathbf{0}$ implies $e_{t,a} = \delta(\mathcal{A}_a, \mathcal{X}_{t,a}) = 0$ for all actions. Both models, therefore, begin with a uniform landscape of expected rewards.

Suppose that at the first round, both models select a suboptimal action a_t and receive a low reward, say $r_{t,a_t} = 0.1$. In the linear bandit, the ridge-regression update incorporates this observation into the parameter estimate $\hat{\theta}_{a_t}$. Because the reward is positive, the updated estimate yields a strictly positive predicted payoff $\hat{r}_{t,a_t} = \mathbf{x}_t^\top \hat{\theta}_{a_t} > 0$, which is now higher than the zero estimate of every unexplored action. Consequently, in the absence of an explicit exploration mechanism, the policy will continue to select a_t in subsequent rounds, becoming trapped in a locally suboptimal choice until the exploration strategy forces it to try an alternative.

The dynamics in HD-CB are fundamentally different, due to the structure of the reward encoding and the properties of the similarity metric. When the reward $r_{t,a_t} = 0.1$ is encoded via thermometer code [237], the resulting reward hypervector $\mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ is predominantly composed of -1 entries: only a small fraction of its components are activated (set to $+1$) for such a low reward value, while the remaining majority retains the value -1 . When this bipolar reward hypervector is bound with $\mathcal{X}_{t,a_t}$, the update term $\mathcal{R}_{t,a_t} \odot \mathcal{X}_{t,a_t}$ will be predominantly *anti-aligned* with the context and the cosine similarity $\delta(\mathcal{A}_{a_t}, \mathcal{X}_{t,a_t})$ becomes negative (i.e., $e_{t,a_t} < 0$).

At the next iteration, the estimated payoff of the previously selected suboptimal action is now strictly lower than the zero estimate of all unexplored actions. The model will therefore naturally move on to a different action without requiring any explicit exploration mechanism to override its greedy choice. In effect, HD-CB encodes the information that “this action was tried and yielded a poor outcome” directly in the geometry of the hyperdimensional space, making the action less attractive in a self-correcting manner. Conversely, when an action receives a high reward (e.g., $r_{t,a_t} = 0.9$), the thermometer-encoded vector $\mathcal{R}_{t,a_t}$ is predominantly composed of +1 entries, and the update pulls the representative *toward* the context, yielding a positive cosine similarity. This creates a natural asymmetry: good actions are reinforced, while poor actions are actively repelled, both under the same update rule.

It is important to stress that this implicit exploration bias does not replace the explicit exploration strategies detailed in Sections 7.1.2–7.1.5. Rather, it provides an additional layer of robustness against early commitment to suboptimal actions, complementing the explicit mechanisms. This dual exploration effect—inherent repulsion combined with deliberate strategic exploration—partly explains the faster convergence observed in Section 7.2.4 and the competitive reward performance across the evaluation benchmarks presented above.

The critical threshold in this mechanism is the value $r = 0.5$, which corresponds to the neutral point of the thermometer code: when exactly half of the components of $\mathcal{R}_{t,a_t}$ are set to +1, the update neither attracts nor repels the representative. Rewards below this neutral point produce repulsive updates (implicit exploration), while rewards above it produce attractive updates (implicit exploitation). Naturally, one could argue that this symmetric partition at 0.5 is a heuristic choice, and that the optimal neutral point may depend on the specific application. In environments where rewards are concentrated in the upper portion of the $[0, 1]$ range, a threshold of 0.5 may be overly aggressive in penalizing actions, classifying outcomes that are in fact close to the environmental average as “poor“. Conversely, in environments with predominantly low rewards, the same threshold may be too lenient, failing to sufficiently repel genuinely suboptimal actions. The neutral point of the thermometer code is, in effect, an *implicit hyperparameter* of the HD-CB framework whose optimal value is not determined by the algorithm itself but can be tuned for the target application.

This limitation reflects a deeper issue in the heuristic nature of the reward-driven update rule adopted in this chapter and will be addressed and resolved in Section 11.2.

7.3 Summary of Findings

This Chapter introduced the HD-CB framework, the first exploration of Hyperdimensional Computing in the literature for modeling and solving sequential decision-making problems in the Contextual Bandits domain. HD-CB leverages the mathematical properties of high-dimensional spaces to encode the state of the environment and represent each action using dedicated hypervectors. At each iteration, these hypervectors are used to identify the action with the highest expected reward and

are updated based on the received payoff.

We detailed HD-CB in four distinct variants, each addressing specific challenges and requirements in CB problems. HD-CB_{EPS} employs an ϵ -greedy exploration strategy, providing a simple and computationally lightweight solution well-suited for resource-constrained systems and less complex environments. HD-CB_{UNC1} enhances decision accuracy by incorporating confidence-driven exploration but introduces additional complexity and requires tuning an extra hyperparameter, β . HD-CB_{UNC2} simplifies this approach by dynamically controlling updates based on model confidence, eliminating the need for β . Finally, HD-CB_{UNC3} optimizes memory efficiency by leveraging the permutation operation.

Experimental validation across synthetic and real-world datasets demonstrated that HD-CB achieves competitive performance, faster convergence (partly attributable to an implicit self-correcting exploration bias), and lower computational complexity compared to traditional algorithms. By replacing the costly ridge regression procedures of traditional CB algorithms with lightweight, highly parallel vector operations, HD-CB significantly reduces computational overhead, resulting well-suited for resource-constrained systems and an ideal candidate for hardware acceleration.

Two limitations of the approach proposed in this chapter should be noted. First, the update rule is accumulation-based: the action representative grows unboundedly over time as new reward-context products are bundled into it, which can lead to precision issues and complicates deployment on low-precision hardware. Second, the update rule is heuristic in nature: unlike linear bandits, where the ridge-regression update minimizes a well-defined objective and provides theoretical regret guarantees, the HD-CB update is shaped by HDC intuitions rather than derived from an explicit optimization criterion. As a consequence, the learning dynamics depend on design choices, such as the neutral threshold of the thermometer encoding, which lack formal justification and must be tuned empirically. Chapter 11 will address both limitations—first by deriving the implicit loss function that the baseline update rule already minimizes, clarifying what the heuristic actually optimizes, and then by introducing an explicit loss-based formulation that replaces heuristic updates with an explicit optimization objective.

Chapter 8

HD-CB_{BIN}: A Binary Neuro-Inspired Framework for Embedded Contextual Bandits

Chapter based on Paper 5 [304]: Marco Angioli, Antonello Rosato, Marcello Barbirotta, Rocco Martino, Andrea Marcelli, Antonio Mastrandrea, and Mauro Olivieri. Hd-cb bin: A lightweight approach for contextual bandit learning in real-time applications. In 2025 International Joint Conference on Neural Networks (IJCNN), pages 1–8. IEEE, 2025

"Simplicity is the ultimate sophistication."

Anonymous

The previous chapter introduced HD-CB, a neuro-inspired reformulation of contextual bandits based on hyperdimensional computing. By replacing computationally intensive linear algebra with lightweight vector operations, HD-CB demonstrated how principles inspired by the brain—robustness, parallelism, and low-precision arithmetic—can yield learning models that are both fast and hardware-friendly. However, that initial formulation employed *real-valued* hypervectors, which provide a convenient baseline for algorithmic exploration but remain expensive for highly resource-constrained embedded deployments. Multiplication and accumulation of floating-point or high-precision integer elements still demand significant memory bandwidth and arithmetic resources

These considerations motivate the research question guiding this chapter:

❓ Research Question

Can HD-CB be realized with *binary* hypervectors, preserving its online learning capabilities while reducing memory footprint and computational cost for extreme edge deployment?

This chapter addresses this question by presenting HD-CB_{BIN}, a streamlined variant of HD-CB designed specifically for resource-constrained embedded systems. By introducing significant modifications to the original algorithm—including hybrid binarization strategies and optimized thinning updates—HD-CB_{BIN} successfully

transitions to binary hypervectors, addressing the challenge of preserving precision during iterative updates, and achieves performance comparable to its real-valued counterpart. This transition drastically reduces memory overhead, execution time, and computational complexity, while enabling the deployment of small, efficient hardware accelerators. These properties make the proposed approach perfect for low-power embedded applications where execution time is critical.

To demonstrate the advantages of HD-CB_{BIN}, we implement it across a range of platforms: a general-purpose CPU, the Raspberry Pi 4, and the Klessydra-T03 RISC-V core. Extensive evaluation against LinUCB confirms HD-CB_{BIN}'s superior computational efficiency, scalability, and reduced execution time across increasing context dimensions. Furthermore, to showcase the benefits of HD-CB_{BIN}'s inherent parallelism, we accelerate both algorithms using dedicated hardware accelerators: the HDCU [209] (that will be introduced in Chapter 12) for HD-CB_{BIN} and the VCU [114] for LinUCB. These experiments demonstrate the clear advantages of HD-CB_{BIN} in achieving high performance and energy efficiency, making it an ideal choice for time-critical, embedded applications.

The main contributions of this chapter are as follows:

- Developing HD-CB_{BIN}, A binary hypervector-based implementation of the HD-CB framework that reduces computational complexity and memory requirements while maintaining comparable performance.
- Comparing HD-CB_{BIN} with the traditional LinUCB algorithm in terms of computational complexity, memory usage, and execution time.
- Demonstrating the benefits of hardware acceleration for HD-CB, emphasizing how HDC-based solutions can exploit parallelism significantly more efficiently than traditional algorithms, achieving substantial speedups and establishing their suitability for time-critical applications.

The chapter is organized as follows. Section 8.1 introduces the HD-CB_{BIN} framework, detailing the binary encoding, hybrid binarization strategy, and optimized update rules. Sections 8.2 and 8.4 present an extensive evaluation of online learning performance, execution time, and memory requirements. The impact of hardware acceleration is also analyzed. Finally, Section 8.5 summarizes the findings.

8.1 Proposed Approach

Among the four HD-CB variants presented in [225], this work selects HD-CB_{UNC2} as the baseline due to its superior performance. For simplicity, this variant is referred to as HD-CB throughout the remainder of this Chapter.

Pseudocode 5 summarizes this algorithm. We remind that in this framework, each action a is modelled by two high-dimensional vectors: $\mathcal{A}_a$, encoding the expected quality of the action, and $\mathcal{B}_a$, representing the frequency of contexts in which the action has been selected, thereby embedding a measure of confidence in the expected payoff. These representative HVs are initialized to zero and updated iteratively during online learning.

At each time step t , the model observes a context $x_{t,a}$ and encodes it as a high-dimensional vector $\mathcal{X}_{t,a}$ using the record-based encoding technique described in Chapter 2. For each action a , the estimated payoff ($e_{t,a}$) is calculated as the similarity between $\mathcal{A}_a$ and $\mathcal{X}_{t,a}$, while the uncertainty ($u_{t,a}$) is defined as the complement of confidence ($c_{t,a}$), which is the similarity between $\mathcal{B}_a$ and $\mathcal{X}_{t,a}$. The potential expectation ($p_{t,a}$) combines these metrics with an exploration factor α , balancing exploration and exploitation. The action a_t is then selected as $\arg \max_a (p_{t,a})$, and the environment provides a real-valued reward r_{t,a_t} . The received reward is encoded into the high-dimensional space, $\mathcal{R}_{t,a_t}$, and used to update the $\mathcal{A}_{a_t}$ HV, moving it closer to or further from the encoded context, depending on the received payoff.

The confidence vector $\mathcal{B}_{a_t}$, on the other hand, is updated using a thinning update mechanism, in which the confidence value c_{t,a_t} directly controls how many elements of $\mathcal{B}_{a_t}$ are bundled with $\mathcal{X}_{t,a}$. This approach balances integrating new information with preserving prior knowledge, according to the confidence level of the model.

HD-CB_{BIN} is the first lightweight and efficient redesign of the HD-CB framework to address the unique challenges posed by resource-constrained systems operating under strict real-time requirements. Binary HVs, as detailed in Chapter 2, offer several advantages in such environments, including reduced memory usage, simpler arithmetic operations, and compatibility with hardware acceleration. However, the shift from real-valued to binary HVs in an online-learning context introduces challenges, particularly in preserving precision during model updates. This section presents the algorithmic modifications proposed to overcome these challenges.

In the original HD-CB framework, action selection depends on mapping the context vector $x_{t,a}$ into a high-dimensional HV $\mathcal{X}_{t,a}$ and computing its similarity with the representative HVs $\mathcal{A}_{t,a}$ and $\mathcal{B}_{t,a}$. In HD-CB_{BIN}, this process is re-engineered to operate in the binary domain, ensuring all three hypervectors— $\mathcal{X}_{t,a}$, $\mathcal{A}_{t,a}$, and $\mathcal{B}_{t,a}$ —are binary.

To encode $x_{t,a}$ as a binary HV without significant precision loss, we employ record-based encoding but defer binarization until all d key-value pairs have been bundled. Binarization is then applied using a threshold of $d/2$, ensuring that the semantic properties of the context are preserved. This approach requires intermediate precision of $\log_2(d)$ bits per HV element, striking a balance between precision and efficiency.

For the representative HVs $\mathcal{A}_{t,a}$ and $\mathcal{B}_{t,a}$, on the other hand, the transition to binary format is more challenging. Direct binarization after each update would severely degrade precision and compromise the model’s online-learning capability. To mitigate this, we propose storing these HVs in a Q -bit integer format and performing binarization only during the action selection phase. Each action a is associated with a counter ($count_a$), which tracks the number of times the action has been selected and serves as a threshold for binarization. During action selection, the binarized HVs are used to efficiently compute similarity metrics, such as the Hamming distance, and are discarded immediately afterward to preserve precision in the stored representations. To prevent overflow in the Q -bit integer format, the representative HVs are periodically replaced with their binarized versions, and the corresponding counters are reset every 2^Q iterations. This hybrid strategy, combining precision-preserving updates with the computational efficiency of binary operations, ensures that HD-CB_{BIN} remains highly effective for resource-constrained

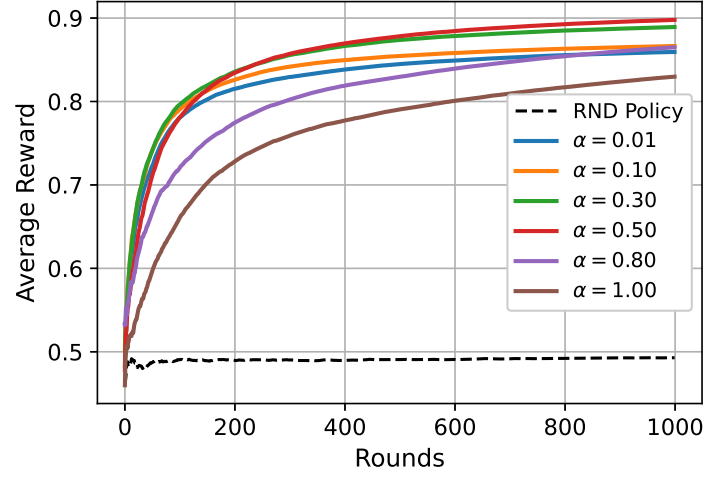


Figure 8.1 Average reward for HD-CB_{BIN} as a function of the exploration parameters α , over 1000 iterations, averaged across 100 synthetic datasets. The experiments were conducted with a HV size of 1000, $N = 20$, continuous rewards in the range $[0,1]$, and $d = 10$.

and real-time applications.

Since the Hamming distance increases with the dissimilarity of HVs, the similarity results in HD-CB_{BIN} must be interpreted accordingly. Specifically, $e_{t,a}$ is defined as the complement of $\delta_{HAM}(\mathcal{X}_{t,a}, \mathcal{A}_{t,a})$, while the uncertainty $u_{t,a}$ is directly given by $\delta_{HAM}(\mathcal{X}_{t,a}, \mathcal{B}_{t,a})$. The potential expectation $p_{t,a}$ is then calculated as the sum of $e_{t,a}$ and $u_{t,a}$, with the confidence value $c_{t,a}$ derived as the complement of $u_{t,a}$. These values guide the update phase of the algorithm. Importantly, the normalized Hamming distance is not used in these calculations to avoid costly division operations, as the relative ordering of $p_{t,a}$ values is sufficient for action selection.

To further optimize HD-CB_{BIN}, we propose reducing the computational overhead of the thinning update mechanism by discretizing the confidence value $c_{t,a}$ into M levels. Each level is associated with a precomputed mask HV, which is generated during initialization with sparsity proportional to the corresponding confidence level. For example, a mask associated with $c_{t,a} = 0.1$ will have approximately 10% of its elements set to 1, while the remaining elements are set to 0. During the update phase, the mask corresponding to the current confidence level is retrieved and permuted by a random number of positions to introduce stochasticity into the updates. The permuted mask is then applied to the binary $\mathcal{X}_{t,a}$ using an element-wise AND operation. The resulting HV is subsequently bundled with $\mathcal{B}_{t,a}$. This method reduces computational complexity by leveraging precomputed masks and ensures that all operations conform to standard HDC arithmetic. Consequently, it enables efficient implementations in both software frameworks [305] and hardware accelerators [209].

8.2 Online Learning and Parameter Analysis

To evaluate the online learning capabilities of the proposed HD-CB_{BIN} model, we conducted extensive simulations on synthetic datasets emulating CB problems. In these experiments, the models iteratively observe new contexts, select actions, and receive rewards, mimicking real-world sequential decision-making scenarios. Synthetic datasets provide a robust and replicable framework for assessing model performance under controlled conditions, enabling a detailed analysis of adaptability, learning efficiency, and convergence behavior.

8.2.1 Learning Capabilities

The learning performance of HD-CB_{BIN} was assessed using the average reward as key metric. Average reward directly reflects how effectively the model selects actions to maximize cumulative payoff, indicating exploration efficiency and alignment with the optimal strategy over time.

The evaluation involved testing HD-CB_{BIN} on 100 synthetic datasets, using a HV size of 1000, while varying the context feature vector size and numbers of actions. For clarity, Figure 8.1 specifically illustrates the performance of HD-CB_{BIN} configured with 20 actions, continuous rewards in the range $[0, 1]$, a context feature vector size of 10, and varying the exploration parameter α . The results, evaluated over 1000 iterations, demonstrate that HD-CB_{BIN} consistently learns and converges toward optimal solutions, with the average reward approaching 1.

Similar to traditional CB algorithms, the performance of HD-CB_{BIN} is influenced by its exploration parameter α . When α is too small (e.g., $\alpha = 0.01$), the model explores insufficiently, resulting in a suboptimal performance characterized by slow reward growth and high cumulative regret. Conversely, excessively large values of α lead to over-exploration, where the model continues exploring unnecessarily after identifying the best actions, reducing its cumulative reward.

8.2.2 Impact of the Q Parameter

The parameter Q , representing the bit-width used to store each element in the representative hypervectors $\mathcal{A}_{t,a}$ and $\mathcal{B}_{t,a}$, plays a critical role in balancing model precision and computational efficiency. To analyze its impact, we conducted experiments varying Q across 1 (binary), 2, 4, 8, and 16.

Figure 8.2 illustrates the average reward trends achieved by HD-CB_{BIN} across 100 synthetic datasets for different values of Q . The results reveal the following insights:

- For $Q = 1$, where the HVs are binarized at every iteration, the significant loss of precision severely hinders the model's ability to learn effectively in real-time scenarios.
- Increasing Q progressively enhances performance, with substantial improvements observed up to $Q = 8$. Beyond this, no noticeable difference is observed between $Q = 8$ and $Q = 16$, indicating that 8-bit integers provide sufficient precision for representing $\mathcal{A}_{t,a}$ and $\mathcal{B}_{t,a}$ without compromising performance.

- Interestingly, even with $Q = 4$, the average reward decreases by only 4% compared to $Q = 8$. This suggests that $Q = 4$ can be a practical choice for applications where minimizing resource usage is critical.

Based on these findings, $Q = 8$ is selected for subsequent experiments, as it strikes an optimal balance between precision, computational efficiency, and compatibility with the standard 8-bit format prevalent in embedded systems.

8.2.3 Comparative Analysis

We compared the performance of HD-CB_{BIN} against both the real-valued HD-CB implementation and the traditional LinUCB algorithm using synthetic datasets, with average reward serving as the primary performance metric. We varied the context feature vector size (d) and the number of actions (N) in the ranges $[10, 15, 20]$ and $[5, 10, 20]$, respectively.

The results in Table 8.1 show that HD-CB_{BIN} achieves performance similar to the real-valued HD-CB, with average rewards closely aligned across all configurations. Furthermore, both HD-CB variants consistently outperform or match LinUCB, highlighting that HD-CB_{BIN} delivers competitive results using only binary HVs.

8.3 Memory Requirements Analysis

Memory overhead is a critical factor in embedded systems, where resources are limited and efficient utilization is essential. HD-CB_{BIN} requires $2N$ HV with Q -bit precision, $(d + L)$ binary HVs for record-based encoding (where L is the number of quantization levels), and M binary HVs for the pre-computed masks. In contrast,

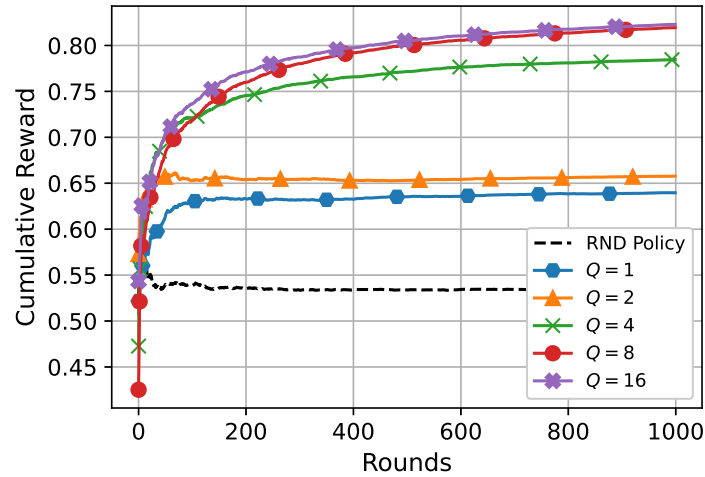
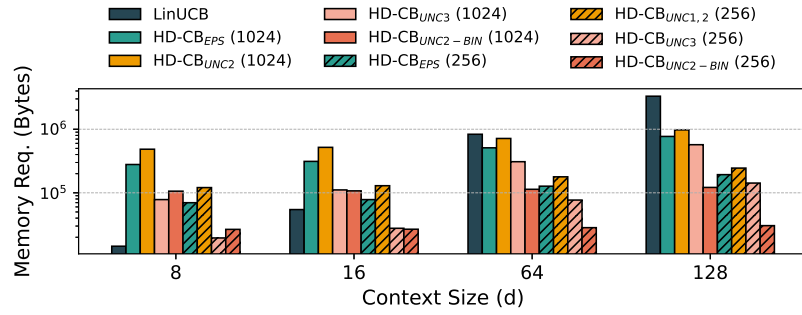


Figure 8.2 Impact of the Q -parameter on the learning performance of HD-CB_{BIN}. The plot shows the average reward trends across 100 synthetic datasets for different Q values, where Q represents the bit-width used to store each element in the representative HVs $\mathcal{A}_{t,a}$ and $\mathcal{B}_{t,a}$.

Table 8.1 Performance Comparison of LinUCB, Real-Valued HD-CB, and HD-CB_{BIN} Across Context Sizes (d) and Number of Actions (N).

d	N	LinUCB	HD-CB	HD-CB _{BIN}
10	5	0.800	0.787	0.785
10	10	0.784	0.770	0.774
10	20	0.854	0.949	0.947
15	5	0.970	0.988	0.990
15	10	0.862	0.874	0.861
15	20	0.878	0.890	0.880
20	5	0.849	0.791	0.778
20	10	0.902	0.920	0.912
20	20	0.789	0.937	0.940

**Figure 8.3** Memory requirements comparison for LinUCB, HD-CB, and HD-CB_{BIN} across varying context feature vector sizes (d) and HV sizes (D).

the LinUCB algorithm relies on $N \times d \times d$ matrices and $N \times d \times 1$ vectors, with elements represented in floating-point or fixed-point formats.

Fig. 8.3 extends Fig. 7.7 in the previous chapter by including HD-CB_{BIN} to highlight the significant memory savings achievable simply by adopting a more compact data type. The results demonstrate the consistent memory savings of HD-CB_{BIN} over the real-valued counterpart and its linear dependence on the context feature vector size (d), compared to the quadratic one of LinUCB. This distinction is critical, as it confirms that as d increases, HD-CB_{BIN} introduces significantly less memory overhead than LinUCB, establishing HD-CB_{BIN} as a strong candidate for efficient deployment in memory-constrained systems.

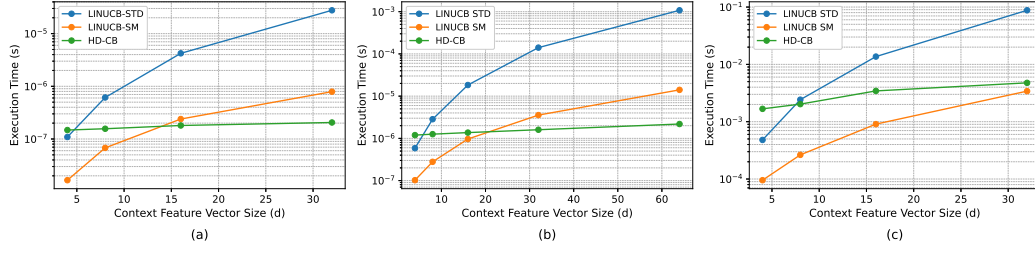


Figure 8.4 Execution time comparison for LinUCB, Real-Valued HD-CB, and HD-CB_{BIN} across varying context feature vector sizes (d). Measurements are conducted on a) a general-purpose CPU (@ 2.30 GHz), b) Raspberry Pi 4 (@ 1.8 GHz), and c) Klessydra-T03 RISC-V core (@ 200 MHz), illustrating the linear execution time trend of HD-CB_{BIN} compared to the cubic and quadratic trends of LinUCB and LinUCB-SM, respectively.

8.4 Execution Time Analysis

8.4.1 Software Execution

To evaluate the efficiency of HD-CB_{BIN} and its feasibility for time-critical applications, we measured its execution time across different hardware platforms and compared it against the LinUCB algorithm, both in its traditional formulation and with the Sherman-Morrison optimization introduced in Chapter 5. The experiments were conducted on three platforms: a general-purpose CPU (11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz), a Raspberry Pi 4, and the embedded Klessydra-T03 RISC-V core. All algorithms were implemented in C++ without pre-defined libraries or dynamic memory allocation to ensure compatibility with embedded systems. Execution times were recorded by direct execution on the general-purpose CPU and Raspberry board, while they were done by cycle-accurate RTL simulation on QuestaSim for the Klessydra T03 implementations, with a 200 MHz simulated clock frequency. The simulation was conducted for varying context feature vector sizes (d) and averaged over multiple runs.

Figure 8.4 shows the results obtained across the three platforms. Independent of the absolute execution times, which naturally vary across platforms due to differences in hardware capabilities (general-purpose CPU being the fastest, followed by the Raspberry Pi and the Klessydra core), the key insight lies in the growth trends of execution time with respect to the context feature vector size (d). The experimental results strongly validate the theoretical expectations described in Chapter 2. The LinUCB algorithm, in its standard (STD) implementation, exhibits a cubic growth in execution time, aligning with its $\mathcal{O}(d^3)$ complexity. The SM variant mitigates this overhead by employing incremental updates, reducing the complexity to $\mathcal{O}(d^2)$. In contrast, HD-CB_{BIN} demonstrates a linear growth in execution time with respect to d , matching its theoretical complexity of $\mathcal{O}(d)$. As d increases, HD-CB_{BIN} demonstrates superior scalability compared to LinUCB, becoming faster than both the STD and SM variants of LinUCB. This efficiency does not come at the expense of performance, as previously demonstrated in Section 8.2, where HD-CB_{BIN} was shown to provide competitive learning capabilities.

As expected, by replacing the complex and time-consuming matrix operations

Table 8.2 Accelerator Configuration and Resource Utilization

Accelerator	LUTs	FFs	DSPs	$f_{\max}$ (MHz)
VCU (SIMD = 4)	11366	3301	23	140
HDCU (SIMD = 128, B = 8 bits)	11496	3017	8	218

required by LinUCB with simple and fast arithmetic operations between vectors in the high-dimensional space, HD-CB_{BIN} achieves substantial computational advantages, making it an ideal candidate for real-world applications where computational resources are constrained, and real-time responsiveness is critical.

8.4.2 Hardware Acceleration

Hardware acceleration can play a critical role in maximizing execution speed in time-constrained applications by offloading computationally intensive operations to specialized hardware units [306, 307]. To evaluate the benefits of hardware acceleration on CB algorithms, we adopt the Klessydra VCU [114] for the LinUCB to handle its matrix operations and a hyperdimensional computing accelerator [209] (that will be introduced in Chapter 12) for HD-CB_{BIN}. Both architectures can be selectively integrated into the Klessydra-T03 RISC-V core at synthesis time, feature dedicated local memories, and are controlled through custom RISC-V instruction set extensions fully integrated into the GCC compiler and exposed to the programmer as simple intrinsics. Notably, neither of these architectures is application-specific. The VCU is designed to efficiently process operations on vectors of numbers, exploiting data-level parallelism, and is chosen due to the strong dependence of LinUCB algorithms on matrix operations, as discussed in Chapter 2. The HDCU, on the other hand, is a general-purpose and flexible hardware accelerator designed to optimize the execution time of the basic arithmetic operations between binary HVs, perfectly fitting HD-CB_{BIN}. Both accelerators are reconfigurable at synthesis time in terms of the SIMD (Single Instruction Multiple Data) width, which defines the number of integer vector elements or binary HV elements processed in parallel by the VCU and HDCU, respectively. The vector sizes and HV sizes are set at run-time through dedicated Current Status Registers. Both accelerators leverage a hardware loop mechanism to efficiently repeat instructions until all vector elements are processed—based on the configured SIMD width and vector size—without requiring additional instruction fetches. The HDCU includes further configurability in terms of the bundling capacity (B), allowing the bit precision used during this operation to be set at synthesis time. To ensure a fair comparison, the accelerators were configured to utilize similar hardware resources, as summarized in Table 8.2.

To quantify the impact of hardware acceleration on the execution time, the C++ implementation of the algorithms was modified to operate in two distinct modes: a software-only mode executed on the scalar processing core and a vectorized mode utilizing the VCU and HDCU. In the vectorized mode, arithmetic matrix operations for LinUCB and HDC operations for HD-CB were parallelized using the specialized vector intrinsic functions supported by the compiler toolchain.

Figure 8.5 presents the execution time results for the accelerated and non-accelerated versions of LinUCB (standard and SM variants) and HD-CB across

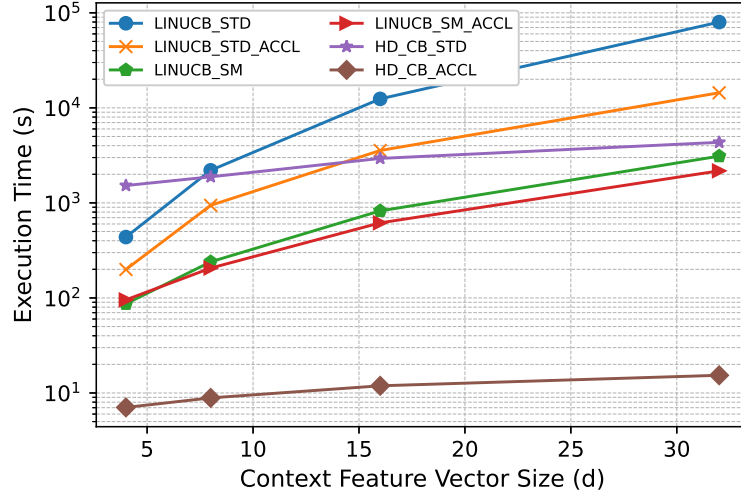


Figure 8.5 Execution time comparison for LinUCB (standard and SM variants) and HD-CB_{BIN}, with and without hardware acceleration. HD-CB_{BIN} achieves up to 217 $\times$ speed-up using the HDCU over the software implementation and outperforms LinUCB-SM-ACCL by 13.56 $\times$ to 141.7 $\times$ as d increases, showcasing superior scalability and efficiency.

various context feature vector sizes. The results demonstrate significant speed-ups enabled by hardware acceleration. The VCU achieved speed-ups of up to 5.54 $\times$ for LinUCB-STD and 1.5 $\times$ for LinUCB-SM, reducing the impact of matrix operations on execution time. At the same time, the HDCU shows a remarkable speed-up of up to 217 $\times$ for HD-CB_{BIN}, showcasing the significant benefits of leveraging the parallel nature of HDC. This version achieves consistent and substantial speed-ups compared to LinUCB-SM with hardware acceleration (LinUCB-SM-ACCL). The speed-up ranges from 13.56 $\times$ when $d = 4$ to 141.7 $\times$ when $d = 32$, further emphasizing the scalability and efficiency of the HD-CB framework in hardware-accelerated environments.

8.5 Summary of Findings

This chapter introduced HD-CB_{BIN}, a lightweight and highly efficient implementation of the HD-CB framework designed for automating contextual bandits problems on resource-constrained systems operating under stringent real-time requirements. By introducing modifications to the original algorithm, including hybrid binarization strategies and optimized thinning updates, HD-CB_{BIN} significantly reduces computational complexity, execution time, and memory overhead using binary HVs.

We benchmarked HD-CB_{BIN} on synthetic datasets, comparing its performance to the traditional state-of-the-art LinUCB algorithm and the real-valued HD-CB framework. The results demonstrated that HD-CB_{BIN} achieves competitive rewards and low cumulative regret, highlighting its robustness in sequential decision-making tasks. To evaluate computational efficiency, we implemented HD-CB_{BIN} and Lin-

UCB on various platforms, including a general-purpose CPU, a Raspberry Pi 4, and the embedded Klessydra-T03 core. The experiments demonstrated that HD-CB_{BIN} achieves linear execution time with respect to the context feature vector size, providing significant scalability advantages over LinUCB, which exhibits cubic and quadratic complexity in its standard and Sherman-Morrison variants, respectively. Finally, hardware acceleration was leveraged to optimize the execution speed further. Using the HDCU we demonstrated the transformative potential of hardware support for HD-CB. The presented algorithm achieved up to a $141\times$ speedup over LinUCB, highlighting the advantages of HDC's inherently parallel operations.

The combination of algorithmic efficiency, the impact of hardware acceleration, and scalability underscores HD-CB_{BIN} as a strong candidate for CB problems in time-critical applications that require rapid, autonomous decision-making.

However, two aspects of the evaluation should be noted. First, the analysis in this chapter focused exclusively on the HD-CB_{UNC2} variant, which was selected as the baseline due to its superior performance and higher hardware demands compared to the other variants introduced in Chapter 7. Second, the evaluation relied on online learning within controlled synthetic environments. While this is a natural setting for cross-platform benchmarking, it would benefit from a comprehensive assessment of how periodic binarization affects decision quality in more complex problem configurations.

These design choices leave open a critical question: whether the periodic binarization mechanism introduces a systematic accuracy penalty that becomes evident in more challenging evaluation settings. In particular, the simpler ε -greedy variant, which lacks the confidence-driven exploration of HD-CB_{UNC2}, may prove to be more sensitive to precision loss during binarization resets. Chapter 11 investigates this directly: Section 11.1 evaluates HD-CB_{BIN} alongside a novel proposed probabilistic variant, employing ε -greedy exploration and off-policy evaluation on more complex standardized environments.

Chapter 9

Hybrid HD-CB: A High-Dimensional Approach to Contextual Bandit Problems in Hybrid Settings

This Chapter presents original work not yet published

“Alone we can do so little; together we can do so much.”

Helen Keller

The previous chapters demonstrated how HDC can simplify and accelerate contextual bandit learning by replacing matrix-based operations with lightweight, parallel-friendly vector arithmetic. HD-CB recasts the problem into the high-dimensional space, achieving robust performance and linear complexity, and HD-CB_{BIN} pushes this further with binary representations for extreme edge deployment.

However, both formulations treated actions independently, mirroring the Disjoint approach (Chapter 2) without leveraging structure or correlations across actions. In many real-world applications, such as recommender systems, robotic control and adaptive interfaces, actions share features or latent relationships, and exploiting these can accelerate convergence and improve cumulative reward. While the linear hybrid algorithm [74] addresses this by introducing shared parameters learned across all actions, its computational cost creates a bottleneck for real-time edge computing, effectively restricting the majority of modern libraries and systems to the less efficient disjoint formulation [76].

These considerations motivate the research question guiding this chapter:

❓ Research Question

Can HD-CB be extended to exploit shared structure across actions while preserving the linear complexity and hardware efficiency of the original framework?

This chapter introduces Hybrid HD-CB, a novel extension that encodes both contextual information and action features into high-dimensional space. Shared

hypervectors, updated across all actions at each learning step, enable the model to generalize from previously explored context–action pairs and estimate rewards for tuples not yet observed — all through lightweight HDC arithmetic that preserves the low overhead and parallelism essential for embedded deployment. Evaluation across synthetic and real-world dataset shows that Hybrid HD-CB consistently outperforms Hybrid LinUCB in learning quality. Moreover, we propose a binarized variant of the algorithm inspired by Chapter 8 and demonstrate that it can achieve up to $48.9\times$ faster execution under hardware acceleration while reducing memory consumption by more than $50\times$.

The main contributions of this chapter are:

- Hybrid HD-CB: a novel extension of the HD-CB framework to hybrid contextual bandit settings. By encoding action features into high-dimensional space and introducing shared hypervectors, the model captures inter-action correlations while maintaining linear computational complexity.
- Hybrid HD-CB_{BIN}: a binary implementation of the hybrid model tailored for resource-constrained platforms, leveraging quantized updates and binary hypervectors to minimize memory footprint and arithmetic cost.
- Extensive comparative evaluation against Hybrid LinUCB (both standard and Sherman–Morrison–Woodbury variants) on synthetic and real-world benchmarks, demonstrating superior scalability, faster convergence, and significantly lower memory requirements.
- Hardware acceleration: evaluating Hybrid HD-CB_{BIN} on the HDCU and comparing it against Hybrid LinUCB accelerated on the VCU, quantifying the gains enabled by the intrinsic parallelism of HDC operations.

The chapter is organized as follows. Section 9.1 introduces the Hybrid HD-CB framework, detailing the encoding of action features and the shared hypervector update mechanism. Section 9.2 evaluates online learning performance on synthetic datasets with both binary and continuous rewards. Section 9.3 presents evaluation on the MovieLens-100k real-world benchmark. Section 9.4 introduces the binary variant and evaluates execution time on embedded platforms. Section 9.5 analyzes the impact of hardware acceleration using the HDCU and VCU. Section 9.6 compares memory requirements. Finally, Section 9.7 summarizes the findings.

9.1 Proposed Approach

Hybrid HD-CB incorporates both action-specific and shared parameters to exploit relationships among actions. The key idea is to encode the interaction between context and action features directly in high-dimensional space, so that actions with similar feature representations naturally produce similar estimates, enabling the model to generalize from limited observations without maintaining covariance matrices.

9.1.1 Intuition

In the disjoint formulation of HD-CB (Chapter 7), each action is modelled independently: its representative HV accumulates information only from rounds in which that specific action is selected. This mirrors the Disjoint LinUCB approach [74] and works well when actions are unrelated. However, when actions share an underlying feature representation—for instance, movies described by genre, director, and cast—information gathered about one action should inform estimates for similar ones.

Hybrid HD-CB addresses this by introducing a state-action encoding. Prior to online learning, during the initialization phase, for each action a , the action feature vector $\mathbf{y}_a \in \mathbb{R}^f$ is encoded into the high-dimensional vector $\mathcal{Y}_a$ using the record-based encoding method described in Chapter 2. Then, at each time step, the observed context hypervector $\mathcal{X}_{t,a}$ is bound with every a action-feature hypervector $\mathcal{Y}_a$ to produce a composite representation $\mathcal{S}_{t,a} = \mathcal{X}_{t,a} \otimes \mathcal{Y}_a$ representing the state-action pair. The structured similarity property of the binding operator ensures two forms of generalization:

- If $\mathcal{Y}_a$ is similar to $\mathcal{Y}_{a'}$, then $\mathcal{S}_{t,a}$ will be similar to $\mathcal{S}_{t,a'}$, allowing the shared representative hypervectors to transfer knowledge between them.
- Similarly, if $\mathcal{X}_{t,a}$ is similar to $\mathcal{X}_{t',a}$, then $\mathcal{S}_{t,a}$ will be similar to $\mathcal{S}_{t',a}$, enabling generalization across similar contexts over time.

These properties allow Hybrid HD-CB to learn faster than the disjoint formulation when action features carry meaningful structure, while degenerating gracefully to the disjoint case when actions are orthogonal in feature space.

9.1.2 Algorithm description

The proposed approach, outlined in Pseudocode 8, builds upon HD-CB by introducing two additional representative hypervectors: $\mathcal{M}_A$ and $\mathcal{M}_B$. These are shared across all actions, initialized to zero, and updated iteratively during learning. By incorporating these shared components, Hybrid HD-CB adds new terms to both the estimated reward and its associated uncertainty, enabling cross-action generalization.

At each round t , the observed context $x_{t,a} \in \mathbb{R}^d$ is encoded into the hypervector $\mathcal{X}_{t,a}$. For each action a , the algorithm computes two groups of estimates:

- Action-specific terms: The estimated reward $e_{t,a}$ and uncertainty $u_{t,a}$ are computed exactly as in HD-CB, by measuring the cosine similarity between $\mathcal{X}_{t,a}$ and the action-specific representative hypervectors $\mathcal{A}_a$ and $\mathcal{B}_a$, respectively.
- Shared terms: The state-action hypervector $\mathcal{S}_{t,a} = \mathcal{X}_{t,a} \otimes \mathcal{Y}_a$ is constructed by binding the context with the action descriptor. Its similarity to $\mathcal{M}_A$ yields the shared reward estimate $se_{t,a}$, and its similarity to $\mathcal{M}_B$ yields the shared confidence $sc_{t,a}$, from which the shared uncertainty $su_{t,a} = 1 - sc_{t,a}$ is derived.

The combined potential payoff is then:

$$p_{t,a} = (e_{t,a} + se_{t,a}) + \alpha(u_{t,a} + su_{t,a}), \quad (9.1)$$

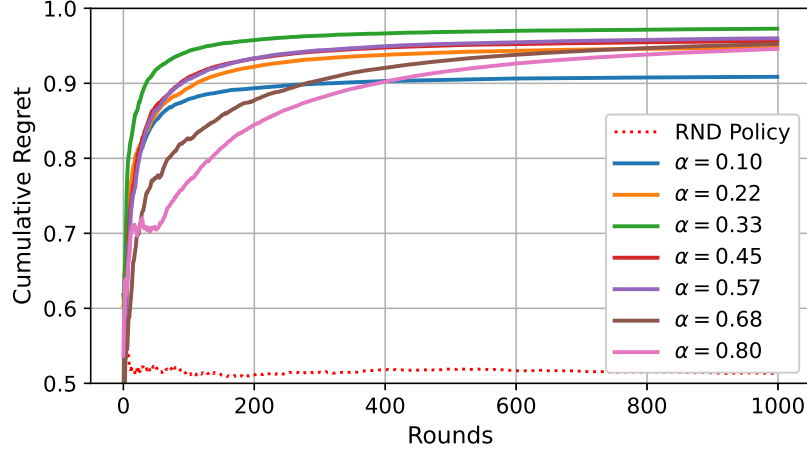


Figure 9.1 Average reward for HD-CB-Hyb as a function of the exploration parameters α , over 1000 iterations, averaged across 100 synthetic datasets. The experiments were conducted with a HV size of 1000, $N = 20$, continuous rewards in the range $[0,1]$, $d = 15$ and $f = 10$

where $\alpha > 0$ balances exploration and exploitation. The action with the highest potential payoff is selected $a_t = \arg \max_a p_{t,a}$.

After observing the reward r_{t,a_t} , the model updates both components. The action-specific HVs, $\mathcal{A}_a$ and $\mathcal{B}_a$, are updated following the same rules as in HD-CB. The shared hypervectors, $\mathcal{M}_A$ and $\mathcal{M}_B$, are updated as in (9.2) and (9.3).

$$\mathcal{M}_A \leftarrow \mathcal{M}_A \oplus (\mathcal{S}_{t,a_t} \otimes \mathcal{R}_{t,a_t}), \quad (9.2)$$

$$\mathcal{M}_B \leftarrow \mathcal{M}_B \oplus \text{thinning}(\mathcal{S}_{t,a_t}, sc_{t,a}). \quad (9.3)$$

Specifically, $\mathcal{R}_{t,a_t}$ is bound with the context vector $\mathcal{S}_{t,a_t}$ and used to update $\mathcal{M}_A$ through a bundling operation. $\mathcal{M}_B$, on the other hand, is updated using the thinning operation explained in Chapter 7 controlled by the shared-confidence value $sc_{t,a}$, to balance incorporating new information while retaining historical knowledge.

9.2 Online Learning Performance

To evaluate the online learning capability of Hybrid HD-CB, we conducted extensive simulations in a synthetic hybrid contextual bandit environment. Each dataset defines an N -armed problem where, at each round t , the learner observes a context vector $\mathbf{x}_{t,a} \in \mathbb{R}^d$ and selects one of N actions, each associated with a fixed action-feature vector $\mathbf{y}_a \in \mathbb{R}^f$. Rewards are generated using a hybrid model combining a context-dependent term, an action-dependent term, and an interaction term coupling context and action features. We evaluate both *binary* rewards, sampled from a Bernoulli distribution with mean given by a logistic link applied to the latent score, and *continuous* rewards in $[0, 1]$, obtained by mapping the latent score to the unit interval.

Figure 9.1 illustrates the learning behavior of Hybrid HD-CB across 100 synthetic datasets configured with $D = 1000$, $N = 20$, continuous rewards, $d = 15$ and $f = 10$.

Algorithm 6 Hybrid HD-CB

```

1: Input: exploration factors  $\alpha_1 \in \mathbb{R}_+$ ,  $\alpha_2 \in \mathbb{R}_+$ 
2: for  $t = 1, 2, \dots, T$  do
3:   for  $a = 1, 2, \dots, K$  do
4:     Encode context  $x_{t,a}$  in  $\mathcal{X}_{t,a}$ 
5:     if  $t = 1$  then
6:        $\mathcal{A}_a \leftarrow \mathbf{0}_D$ 
7:        $\mathcal{B}_a \leftarrow \mathbf{0}_D$ 
8:        $\mathcal{M}_A \leftarrow \mathbf{0}_D$ 
9:        $\mathcal{M}_B \leftarrow \mathbf{0}_D$ 
10:      Encode action-specific feature vector into  $\mathcal{Y}_a$ 
11:    end if
12:    Action-specific estimations
13:     $e_{t,a} \leftarrow \delta_{\cos}(\mathcal{A}_a, \mathcal{X}_{t,a})$ 
14:     $c_{t,a} \leftarrow \delta_{\cos}(\mathcal{B}_a, \mathcal{X}_{t,a})$ 
15:     $u_{t,a} \leftarrow 1 - c_{t,a}$ 
16:    Shared estimations
17:     $\mathcal{S}_{t,a} \leftarrow \mathcal{X}_{t,a} \otimes \mathcal{Y}_a$ 
18:     $se_{t,a} \leftarrow \delta_{\cos}(\mathcal{M}_A, \mathcal{S}_{t,a})$ 
19:     $sc_{t,a} \leftarrow \delta_{\cos}(\mathcal{M}_B, \mathcal{S}_{t,a})$ 
20:     $su_{t,a} \leftarrow 1 - sc_{t,a}$ 
21:    Combined potential payoff
22:     $p_{t,a} \leftarrow (e_{t,a} + se_{t,a}) + \alpha_1 (u_{t,a} + su_{t,a})$ 
23:  end for
24:   $a_t \leftarrow \arg \max_a p_{t,a}$ 
25:  Observe reward  $r_{t,a_t}$ 
26:  Encode reward into hypervector  $\mathcal{R}_{t,a_t}$ 
27:  Action-specific update
28:   $\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} \oplus (\mathcal{X}_{t,a_t} \otimes \mathcal{R}_{t,a_t})$ 
29:   $\mathcal{B}_{a_t} \leftarrow \mathcal{B}_{a_t} \oplus \text{thinning}(\mathcal{X}_{t,a_t}, c_{t,a_t})$ 
30:  Shared update
31:   $\mathcal{M}_A \leftarrow \mathcal{M}_A \oplus (\mathcal{S}_{t,a_t} \otimes \mathcal{R}_{t,a_t})$ 
32:   $\mathcal{M}_B \leftarrow \mathcal{M}_B \oplus \text{thinning}(\mathcal{S}_{t,a_t}, sc_{t,a_t})$ 
33: end for

```

The exploration parameter α was varied between 0.1 and 0.8, and the model was evaluated over 1000 iterations. The results confirm that Hybrid HD-CB consistently learns and converges toward optimal solutions across the tested range. The average reward approaches the maximum as iterations progress, demonstrating effective sequential decision-making. The exploration parameter plays a critical role: values that are too small (e.g., $\alpha = 0.1$) limit the model's ability to identify optimal actions, while excessively large values cause over-exploration and slower convergence. Intermediate values of α yield the best accuracy–convergence trade-off.

Tables 9.1 and 9.2 summarize the per-round execution time and the average regret for the hybrid setting under binary and continuous rewards, respectively (for each method, α is selected as the best-performing value over the explored range).

Table 9.1 Hybrid bandit: per-round execution time (ms) and average regret (best α per model). Binary Reward.

Dims		Hybrid LinUCB		Hybrid HD-CB	
d	f	Time (ms)	AvgRegret	Time (ms)	AvgRegret
8	8	5.006	0.0893	1.404	0.0411
8	16	12.058	0.0840	1.444	0.0501
8	32	21.150	0.0704	1.452	0.0434
16	8	13.854	0.0797	1.504	0.0465
16	16	25.382	0.0752	1.558	0.0286
16	32	50.062	0.0669	1.540	0.0322
32	8	35.897	0.0587	1.640	0.0310
32	16	87.625	0.0589	1.674	0.0386
32	32	141.534	0.0563	1.701	0.0355

Table 9.2 Hybrid bandit: per-round execution time (ms) and average regret (best α per model). Continuous Reward.

Dims		Hybrid LinUCB		Hybrid HD-CB	
d	f	Time (ms)	AvgRegret	Time (ms)	AvgRegret
8	8	4.921	0.0460	1.471	0.0491
8	16	11.486	0.0418	1.487	0.0399
8	32	20.774	0.0420	1.538	0.0361
16	8	13.833	0.0438	1.589	0.0274
16	16	25.491	0.0461	1.596	0.0270
16	32	50.473	0.0365	1.607	0.0224
32	8	35.970	0.0492	1.754	0.0157
32	16	89.283	0.0405	1.753	0.0142
32	32	150.700	0.0325	1.656	0.0136

Two clear trends emerge. First, Hybrid LinUCB exhibits a steep increase in per-round latency as the context and action-feature dimensions grow (from ≈ 5 ms up to ≈ 151 ms), reflecting the cubic cost of matrix-based updates. In contrast, Hybrid HD-CB remains nearly constant at ≈ 1.4 – 1.7 ms across all configurations, yielding speedups ranging from $3.3\times$ to $91\times$. Second, Hybrid HD-CB consistently achieves lower regret in the binary setting for all tested (d, f) pairs (typically reducing regret by roughly $1.5\times$ – $2.6\times$ compared to Hybrid LinUCB). In the continuous setting, Hybrid LinUCB is marginally better only in the smallest reported configuration $(d, f) = (8, 8)$, whereas Hybrid HD-CB becomes clearly superior as dimensionality increases, reaching up to $\approx 3.1\times$ lower regret. Overall, these results indicate that Hybrid HD-CB provides a markedly better accuracy–efficiency trade-off, particularly in higher-dimensional hybrid problems where conventional linear methods incur both higher computational overhead and higher regret.

Figure 9.2 provides a complementary view of the computational advantage by reporting the speedup, defined as the ratio between the per-round execution time of Hybrid LinUCB and that of Hybrid HD-CB, while jointly increasing the

context and action-feature dimensions d and f in $\{4, 8, 16, 32, 64\}$. The speedup grows rapidly with dimensionality, ranging from $1.23\times$ for $(d, f) = (4, 4)$ up to $1871\times$ for $(d, f) = (64, 64)$, highlighting an increasingly pronounced benefit in high-dimensional hybrid settings. This behavior reflects the different computational profiles of the two methods. Hybrid LinUCB relies on matrix-based updates with cost $\mathcal{O}(k^3) = \mathcal{O}(f^3d^3)$, where $k = df$ is the interaction feature dimension, as explained in Chapter 2. Even under the Sherman–Morrison–Woodbury optimization, the dominant cost remains $\mathcal{O}(f^2d^3)$. Hybrid HD-CB, by contrast, is dominated by elementwise operations that scale linearly with d and f for fixed hypervector dimension. As a result, the computational advantage becomes increasingly pronounced in high-dimensional hybrid problems, precisely the regime where the hybrid formulation is most valuable.

9.3 Evaluation on the MovieLens-100k Dataset

To assess whether the advantages observed on synthetic data transfer to a real-world setting, we evaluated Hybrid HD-CB on the MovieLens-100k dataset, following the same contextual bandit protocol described in Chapter 7. Each decision round corresponds to a user interaction: the user profile constitutes the context, the movie characteristics are the action features, the policy recommends one movie among a candidate set of N arms, and the reward is observed only for the chosen movie. Ratings were binarized (1 for ratings ≥ 4 , 0 otherwise), and the user-side context was obtained by discretizing age into buckets and applying one-hot encoding to age group, gender, and occupation.

Table 9.3 reports the average regret of Hybrid LinUCB and Hybrid HD-CB as the number of candidate movies N increases. For small candidate sets ($N = 10$ and $N = 50$), Hybrid HD-CB achieves substantially lower regret, while as N grows, the gap narrows. Overall, these results confirm that the proposed high-dimensional hybrid formulation remains competitive on real recommendation data and can

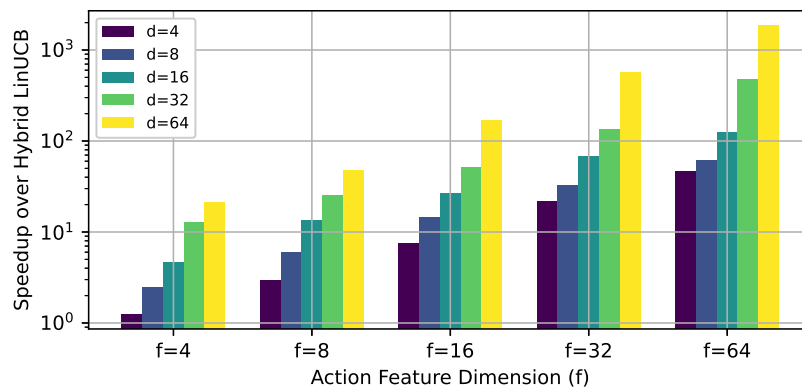


Figure 9.2 Speedup achieved by Hybrid HD-CB compared to Hybrid LinUCB on synthetic datasets with varying context (d) and action (f) feature vector sizes. The speedup, calculated as the ratio of Hybrid LinUCB’s execution time to Hybrid HD-CB’s execution time, dramatically increases as f and d grow.

provide clear benefits.

Table 9.3 MovieLens: Average model-based regret for Hybrid LinUCB and Hybrid HD-CB as the number of movies N increases.

N (movies)	Hybrid LinUCB	Hybrid HD-CB
10	0.1565	0.0608
50	0.1790	0.0624
100	0.1633	0.1624
200	0.1641	0.1486

9.4 Hybrid HD-CB on Embedded Systems

To evaluate the efficiency of the proposed Hybrid HD-CB algorithm on embedded systems and hardware accelerators, we implemented Hybrid HD-CB_{BIN}, a binary version of the model in C++, inspired by the approach introduced in Chapter 8 [304].

Hybrid HD-CB_{BIN} applies binarization after encoding the context $\mathcal{X}$ and the actions $\mathcal{Y}$, using $d/2$ and $f/2$ as thresholds. All representative HVs maintained by the learner—both action-specific $(\mathcal{A}_a, \mathcal{B}_a)$ and shared $(\mathcal{M}_\mathcal{A}, \mathcal{M}_\mathcal{B})$ —are stored using 8-bit precision. This choice is motivated by the findings in Chapter 8, which showed that directly binarizing representative HVs after every update introduces significant quantization noise and hinder online learning, whereas 8-bit integers can preserve the learning behavior. To support this mechanism, compact counters track the number of updates applied to each representative HV. During action selection, the representatives are temporarily binarized using half of the corresponding counter values as thresholds and are then used to compute similarities using Hamming distance. To prevent saturation of the 8-bit representatives, they are periodically refreshed by replacing them with their binarized versions every $2^8 = 256$ iterations, balancing precision retention with the lightweight requirements of embedded systems.

We evaluated the execution time of Hybrid HD-CB_{BIN} on a general-purpose CPU (11th Gen Intel® Core™ i7-11800H @ 2.30 GHz) and on the embedded Klessydra-T03 RISC-V core, comparing against Hybrid LinUCB in both its standard formulation and its Sherman–Morrison–Woodbury optimized variant described in Chapter 5. All implementations were written in C++ without external numerical libraries and without dynamic memory allocation to ensure compatibility with embedded targets. Execution time on the Intel CPU was measured directly, while cycle-accurate RTL simulations were performed on QuestaSim for the Klessydra-T03 using a 100 MHz simulated clock frequency.

Figure 9.3 reports execution time while varying the action feature dimension f from 4 to 32 (with $d = 8$ and $N = 8$ fixed). Consistent with the previous scaling analysis, Hybrid HD-CB_{BIN} maintains near-constant latency across the tested range, while both Hybrid LinUCB variants exhibit steep growth driven by their cubic and quadratic dependence on f .

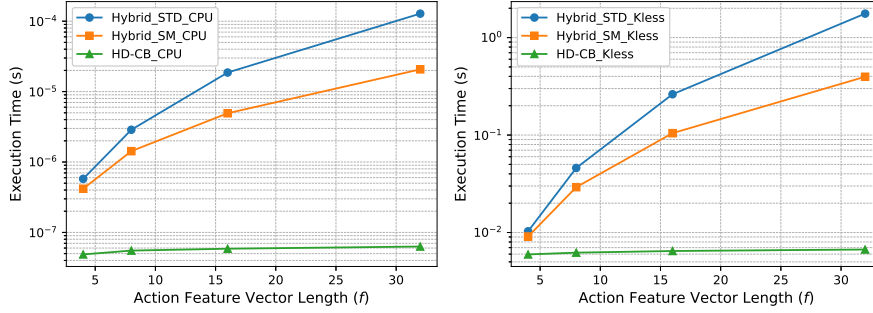


Figure 9.3 Execution time results for Hybrid HD-CB and Hybrid LinUCB (standard and SMW variants) evaluated on a general-purpose CPU and the embedded Klessydra core. The experiments were conducted by varying the action feature vector size (f) while keeping the context feature vector size ($d = 8$) and the number of actions ($N = 8$) fixed.

9.5 Hardware Acceleration

To quantify the additional gains achievable through dedicated hardware support, we leveraged the VCU for Hybrid LinUCB and the HDCU (that will be introduced in Chapter 12) for Hybrid HD-CB. As already noted for HD-CB, both architectures are general-purpose and not application-specific: the VCU is optimized for vector and matrix operations exploiting data-level parallelism, while the HDCU targets binary hypervector operations. For a fair comparison, the two accelerators were configured to occupy similar hardware resources ($\sim 11,400$ LUTs, $\sim 3,100$ FFs).

Both algorithms were implemented in C++ under two execution modes: software-only (scalar core) and vectorized (leveraging the respective accelerator through dedicated intrinsic functions).

Figure 9.4 presents the execution time (in clock cycles) for the accelerated and non-accelerated versions of Hybrid LinUCB (standard and SMW variants) and Hybrid HD-CB across varying action feature dimensions f . The VCU provides measurable speedups for Hybrid LinUCB, reaching up to $3.4\times$ for the standard variant and $2.9\times$ for the SMW variant, thereby reducing the overhead of matrix-based operations. In contrast, the HDCU delivers substantially larger gains for Hybrid HD-CB, achieving up to $275\times$ acceleration relative to its software-only implementation. This substantial improvement showcases the ability of the HDCU to exploit the parallelism inherent in HDC operations.

When comparing the hardware-accelerated implementations, Hybrid HD-CB with HDCU consistently outperforms Hybrid LinUCB with VCU. In particular, the speedup of Hybrid HD-CB over Hybrid-SMW-ACCL ranges from $5.1\times$ at $f = 4$ to $48.9\times$ at $f = 32$, highlighting superior scalability as the action feature dimension increases. Overall, the combination of an intrinsically scalable algorithmic structure and an accelerator tailored to binary HV operations positions Hybrid HD-CB as a highly efficient solution for contextual bandits in embedded and time-critical applications.

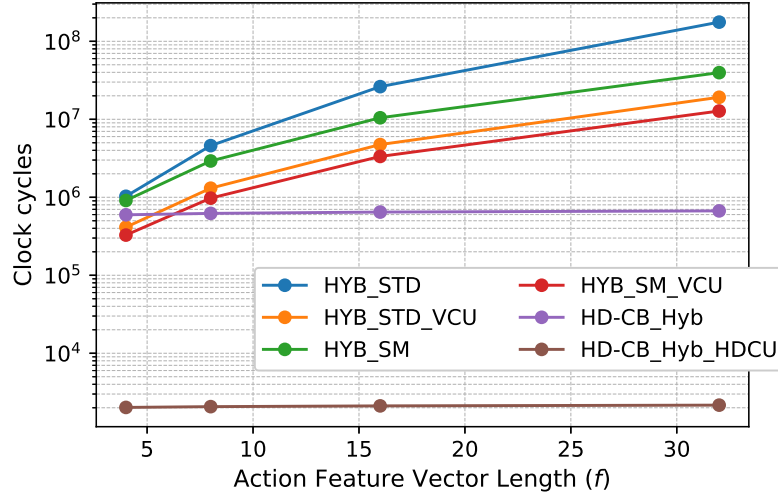


Figure 9.4 Execution time results (in clock cycles) for Hybrid HD-CB and Hybrid LinUCB (standard and SM variants) with hardware acceleration using the HDCU and VCU, respectively. The experiments varied the action feature vector size (f) while keeping $d = 8$ and $N = 8$ fixed.

9.6 Memory Requirements

A final dimension of comparison concerns memory overhead, which is particularly constrained on embedded platforms. Hybrid HD-CB_{BIN} requires $2 + 2N$ HV with Q -bit precision for the shared and dedicated representative HVs, $(d + L)$ binary HVs for record-based encoding (where L is the number of quantization levels), and M binary HVs for the pre-computed thinning masks. In contrast, the LinUCB algorithm relies on a $d \times d$ matrices, a $d \times 1$ vectors, and a $d \times k$ vector for each action, and one $k \times k$ shared matrix and $k \times 1$ shared vector, each with elements represented in 32-bit floating-point or fixed-point formats.

Figure 8.5 compares memory requirements across d and f ranging from 8 to 128, with $D = 1024$, $N = 8$, $L = 15$, $M = 10$, and $Q = 8$. Hybrid HD-CB_{BIN} exhibits linear growth with d and f , while Hybrid LinUCB grows quadratically due to its dependence on $d \times d$ and $k \times k$ matrices. The gap widens rapidly: at $d = f = 128$, Hybrid HD-CB_{BIN} achieves over 50 $\times$ memory reduction: a decisive advantage for platforms where RAM is measured in kilobytes.

9.7 Summary of Findings

This chapter introduced *Hybrid HD-CB*, a high-dimensional framework for contextual bandit problems in *hybrid* settings. The core idea—encoding context–action interactions in high-dimensional space through binding—enables cross-action generalization without covariance matrices that make linear hybrid models prohibitively expensive.

The evaluation progressed along four axes:

- Learning quality: Hybrid HD-CB demonstrated lower regret than Hybrid

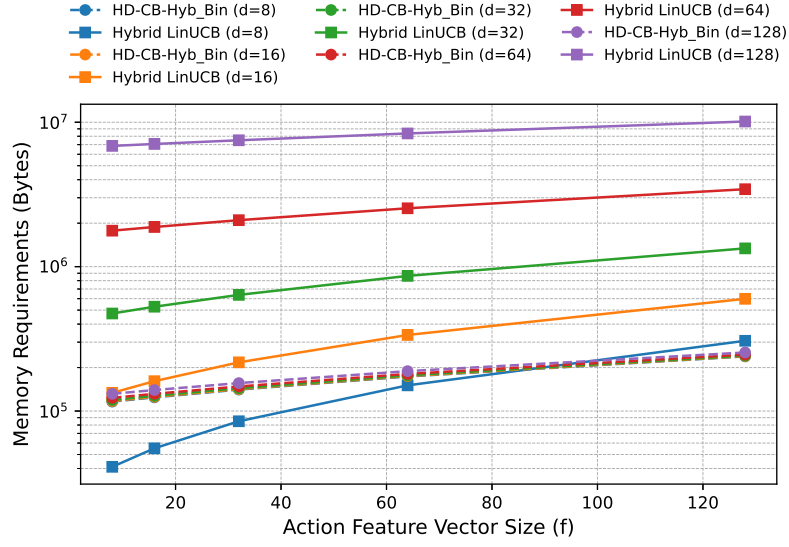


Figure 9.5 Comparison of memory requirements for HD-CB Hyb_{BIN} and Hybrid LinUCB across varying context feature vector sizes ($d = 8$ to 128) and action feature vector sizes ($f = 8$ to 128).

LinUCB on synthetic benchmarks with both binary and continuous rewards, and confirmed these advantages on the MovieLens-100k dataset.

- **Computational efficiency:** The linear scaling of Hybrid HD-CB contrasts sharply with the $\mathcal{O}(d^3 f^3)$ cost of standard Hybrid LinUCB (or $\mathcal{O}(d^3 f^2)$ under SMW updates). On synthetic benchmarks, this translated into speedups of up to $1871\times$ for large configurations. On the embedded Klessydra-T03, the binary variant confirmed these scalability trends on actual hardware.
- **Hardware acceleration:** The HDCU amplified the advantages by a factor of $275\times$ relative to software-only execution, while the VCU provided moderate gains (up to $3.4\times$) for Hybrid LinUCB. In the accelerated head-to-head comparison, Hybrid HD-CB outperformed by up to $48.9\times$.
- **Memory footprint:** The binary formulation reduced memory requirements by over $50\times$ relative to Hybrid LinUCB, scaling linearly rather than quadratically with feature dimensionality.

Together, these results position Hybrid HD-CB as a practical and scalable solution for real-time decision-making in hybrid bandit problems, particularly on embedded platforms where computational, memory, and energy budgets are severely constrained. The framework’s advantages grow with problem dimensionality, precisely the regime where the hybrid formulation is most needed.

Chapter 10

Efficient Implementations of HD-CB

This Chapter presents original work not yet published

"Mathematics is the art of giving the same name to different things."

Henri Poincaré

The preceding chapters established HD-CB as a viable and efficient substrate for contextual bandit learning in the high-dimensional space. This chapter presents engineering refinements that address two orthogonal efficiency concerns.

First, Section 10.1 derives mathematically equivalent matrix-form reformulations of HD-CB and its variants, reorganizing the operations into dense linear-algebra kernels that expose data parallelism without changing the learning dynamics. Second, Section 10.2 presents memory-optimized variants that exploit the properties of the permutation operation to compress the prototype hypervectors from one or more per-action to a single global memory vector, making HD-CB deployment feasible on memory-constrained platforms. Section 10.3 summarizes the findings and discusses their implications for embedded deployment.

10.1 Matrix-Vector Reformulation of HD-CB

The previous chapters formulated the learning of HD-CB models using HDC arithmetic (binding, bundling, permutation and similarity), typically expressed through per-action procedures with explicit loops. While this abstraction is elegant and faithful to the HDC intuition, it can be difficult to follow for readers unfamiliar with HDC notation, and it is not ideal for execution on Basic Linear Algebra Subprograms (BLAS)-capable backend, which achieves peak performance when computations are expressed as dense linear-algebra kernels. This section asks:

❓ Research Question

Can HD-CB-style bandit learners be reformulated in matrix form without changing their behavior?

Algorithm 7 HD-CB_{EPS} (matrix form)

```

1: Input:  $\epsilon \in [0, 1]$ 
2: Initialization:  $\mathbf{A} \leftarrow \mathbf{0}_{N \times D}$ 
3: for  $t = 1, \dots, T$  do
4:   Encode context(s)  $\rightarrow \mathbf{X}_t \in \{-1, +1\}^{N \times D}$ 

5:   Action selection
6:      $\mathbf{s}_t \leftarrow \text{diag}(\mathbf{A}\mathbf{X}_t^\top) \in \mathbb{R}^N$ 
7:      $\nu_A \leftarrow \sqrt{\text{diag}(\mathbf{A}\mathbf{A}^\top)}$ 
8:      $\mathbf{e}_t \leftarrow \frac{1}{\sqrt{D}} \frac{\mathbf{s}_t}{\nu_A}$ 
9:     Draw  $u \sim \text{Uniform}(0, 1)$ 
10:    if  $u < \epsilon$  then
11:       $a_t \leftarrow \text{RandomAction}()$ 
12:    else
13:       $a_t \leftarrow \arg \max_{a \in \{1, \dots, N\}} e_{t,a}$ 
14:    end if

15:   Update
16:     Observe payoff  $r_{t,a_t} \in \mathbb{R}$  and encode  $\mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ 
17:      $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + (\mathcal{R}_{t,a_t} \odot \mathcal{X}_{t,a_t})$ 
18: end for

```

and derives *exactly equivalent* matrix-form formulations for the main HD-CB variants. We restrict ourselves to the MAP family (as in Chapter 7) and stress that no operations are changed: the reformulations reorganize computation to expose data parallelism, yielding identical decisions and learning dynamics.

10.1.1 General principle

We illustrate the construction on HD-CB_{EPS}, the simplest variant, and then extend to the remaining models. As defined in Chapter 7, the baseline agent maintains a separate prototype hypervector $\mathcal{A}_a$ for each available action $a \in \{1, \dots, N\}$ and computes the action scores by iterating over all actions.

To reformulate the agent in matrix-form, we stack the N action prototypes as rows of $\mathbf{A} \in \mathbb{R}^{N \times D}$ and the per-action encoded contexts as rows of $\mathbf{X}_t$ (when the context is shared across actions, all rows of $\mathbf{X}_t$ coincide). In this form, the dot-product score for each action can be obtained as the diagonal of $\mathbf{A}\mathbf{X}_t^\top$.

$$s_{t,a} = \langle \mathbf{A}_a, \mathcal{X}_{t,a} \rangle, \quad \mathbf{s}_t = \text{diag}(\mathbf{A}\mathbf{X}_t^\top) \in \mathbb{R}^N. \quad (10.1)$$

When the context is shared, this reduces to the matrix-vector product $\mathbf{s}_t = \mathbf{A}\mathcal{X}_t$.

If we assume that the encoded context is bipolar (i.e. a sign function is applied after the record-based encoding, so that $\mathcal{X}_{t,a} \in \{-1, +1\}^D$) its norm is constant: $\|\mathcal{X}_{t,a}\|_2 = \sqrt{D}$ and cosine similarity requires only the per-action model norms:

$$\nu_A = \sqrt{\text{diag}(\mathbf{A}\mathbf{A}^\top)} \in \mathbb{R}^N, \quad \mathbf{e}_t = \frac{1}{\sqrt{D}} \frac{\mathbf{s}_t}{\nu_A}. \quad (10.2)$$

Action selection proceeds via ϵ -greedy on $\mathbf{e}_t$; the update remains a single row addition:

$$\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + (\mathcal{R}_{t,a_t} \odot \mathcal{X}_{t,a_t}). \quad (10.3)$$

The complete matrix-form algorithm is listed in Algorithm 7.

10.1.2 Extension to uncertainty-aware and hybrid variants

The same principle extends to the remaining variants with minimal modification.

Uncertainty-aware HD-CB. This variant maintains a second matrix $\mathbf{B} \in \mathbb{R}^{N \times D}$ for confidence estimation. The estimate and confidence vectors are computed in parallel as:

$$\mathbf{e}_t = \frac{1}{\sqrt{D}} \frac{\text{diag}(\mathbf{A}\mathbf{X}_t^\top)}{\sqrt{\text{diag}(\mathbf{A}\mathbf{A}^\top)}}, \quad \mathbf{c}_t = \frac{1}{\sqrt{D}} \frac{\text{diag}(\mathbf{B}\mathbf{X}_t^\top)}{\sqrt{\text{diag}(\mathbf{B}\mathbf{B}^\top)}}. \quad (10.4)$$

The action potential $\mathbf{p}_t = \mathbf{e}_t + \alpha(\mathbf{1} - \mathbf{c}_t)$ and the row-wise updates to $\mathbf{A}_{a_t}$ and $\mathbf{B}_{a_t}$ remain identical to the original formulation:

$$\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + (\mathcal{R}_{t,a_t} \odot \mathcal{X}_{t,a_t}), \quad \mathbf{B}_{a_t} \leftarrow \mathbf{B}_{a_t} + (\mathbf{m}_t \odot \mathcal{X}_{t,a_t}), \quad (10.5)$$

where $\mathbf{m}_t \in \{0, 1\}^D$ is the confidence-dependent mask.

Hybrid HD-CB. For the hybrid model, let $\mathbf{Y} \in \{-1, +1\}^{N \times D}$ stack the encoded action-feature HVs and define the bound context-action matrix $\mathbf{S}_t = \mathbf{Y} \odot \mathbf{X}_t \in \{-1, +1\}^{N \times D}$. Because both factors are bipolar, every row of $\mathbf{S}_t$ has norm $\sqrt{D}$, and cosine normalization for the shared components only requires the norm of the shared memories.

$$\mathbf{se}_t = \frac{1}{\sqrt{D}} \frac{\mathbf{S}_t \mathcal{M}_A}{\|\mathcal{M}_A\|_2}, \quad \mathbf{sc}_t = \mathbf{1} - \frac{1}{\sqrt{D}} \frac{\mathbf{S}_t \mathcal{M}_B}{\|\mathcal{M}_B\|_2}. \quad (10.6)$$

The final potential $\mathbf{p}_t$ combines action-specific and shared terms exactly as in the original algorithm; updates consist of row updates to $\mathbf{A}_{a_t}$, $\mathbf{B}_{a_t}$ and shared-memory updates driven by $\mathbf{S}_{t,a_t}$.

10.1.3 Computational analysis and performance demonstration

The reformulations above are mathematically equivalent to the loop-based algorithms: they produce identical action choices, regret, and learning dynamics. Their purpose is purely computational: by expressing similarity computations as dense matrix operations, the implementation can leverage optimized BLAS kernels (NumPy/MKL on CPUs, cuBLAS on GPUs).

To quantify this effect, we implemented the most demanding variant, Hybrid HD-CB, in two NumPy implementations: the original iterative form with explicit per-action loops (Loop) and the matrix-vector formulation derived above (Matrix). Both are compared against a standard Hybrid LinUCB baseline. Table 10.1 reports the mean per-round execution time across three hardware targets.

Three observations emerge. First, on the CPU the matrix formulation yields a consistent $2.3\times$ – $3.6\times$ speedup over the loop-based implementation. Second, the

advantage of Hybrid HD-CB over Hybrid LinUCB remains overwhelming, as discussed in Chapter 9. Third, on GPUs (RTX 3050 and T4) the matrix form remains systematically faster than the loop version, though gains are more modest ($1.1\times$ – $1.4\times$) because per-round workloads are small enough that kernel-launch overhead becomes non-negligible.

Overall, these results confirm that the matrix-form view is clearer from a modelling perspective and accelerates HD-CB on general-purpose hardware without requiring HDC-specific support.

Table 10.1 Per-round execution time (ms) across devices (means only). HD-CB is implemented in the original version (Loop) and Matrix.

Config		CPU i7-11800H (ms)			GPU RTX 3050 (ms)			GPU T4 (ms)		
d	f	LINUCB	Loop	Matrix	LINUCB	Loop	Matrix	LINUCB	Loop	Matrix
4	4	1369.449	1177.798	493.665	5.281	5.005	4.376	3.746	1.356	1.200
4	8	2212.563	1193.821	482.612	10.242	5.326	4.798	3.067	1.321	1.199
4	16	4172.073	1146.704	441.126	10.384	5.219	4.654	3.182	1.367	1.191
4	32	10 033.161	1192.312	466.750	6.571	5.275	5.192	3.733	1.751	1.214
8	4	2927.776	1146.312	496.890	11.671	6.145	5.212	3.047	1.399	1.187
8	8	5074.613	1219.161	488.074	9.832	5.252	4.515	3.179	1.378	1.172
8	16	28 139.676	1539.790	579.105	9.575	5.370	4.695	4.282	1.766	1.670
8	32	205 794.737	2834.616	849.386	7.721	6.451	5.317	3.523	1.347	1.185
16	4	32 293.778	3225.185	1026.497	10.512	5.061	4.377	3.176	1.356	1.173
16	8	99 630.848	3207.573	1021.705	6.323	4.977	4.403	3.324	1.384	1.459
16	16	249 301.224	3159.496	1015.472	6.746	5.267	5.483	4.329	1.365	1.189
16	32	604 975.346	3721.883	1021.867	6.416	6.102	4.528	3.990	1.374	1.161
32	4	170 328.144	3282.692	1151.511	8.289	5.256	4.410	3.604	1.369	1.217
32	8	309 019.130	3499.950	1179.127	6.207	7.057	5.354	4.846	1.352	1.183
32	16	686 497.052	3359.164	1167.034	9.409	7.096	5.242	4.359	1.387	1.177
32	32	1 080 394.730	3379.553	1167.567	13.963	6.382	4.942	7.125	1.818	1.565

10.2 Memory-Optimized HD-CB via Permutations

This section addresses a critical constraint for edge deployment: the *memory footprint*.

The HD-CB algorithms in their original formulation rely on dedicated high-dimensional representatives for each action, allocating a non-negligible amount of memory. The baseline HD-CB_{EPS} maintains one representative HV per action, resulting in N hypervectors of dimension D . The uncertainty-aware variants require two representatives per action (estimate and confidence), i.e., $2N$ HVs, while the hybrid extension additionally introduces shared components, yielding a total of $(2N + 2)$ HVs. Consequently, although HD-CB avoids the matrix-centric state of classical linear bandits (whose memory grows quadratically with the problem dimensionality), its memory footprint still scales linearly with the number of actions N and the HV size D .

A fundamental property of high-dimensional spaces, however, is their large *representational capacity*: multiple items can be stored in superposition within a single hypervector without catastrophic information loss. This section exploits that capacity to answer the following question:

❓ Research Question

Can the memory footprint of HD-CB be reduced from per-action storage to a single global memory vector by exploiting the superposition capacity of high-dimensional spaces?

To answer this question, we pursue two progressively more aggressive compression strategies, both grounded in standard HDC properties: (i) *permutation-based channel multiplexing*, which eliminates the dedicated confidence memory in the HD-CB_{UNC} variants by encoding confidence in a permuted channel of the action prototype; and (ii) *label-based superposition*, which collapses all per-action prototypes into one global hypervector by tagging each action’s contribution with a unique random key. After presenting each mechanism, we analyze the resulting trade-offs among accuracy, memory, and throughput.

10.2.1 Compressing confidence into the estimate memory

The uncertainty-aware variant HD-CB_{UNC2} traditionally maintains two per-action prototypes: $\mathcal{A}_a$ for the estimate and $\mathcal{B}_a$ for the confidence. The key observation in this section is that permutations generate quasi-orthogonal hypervectors and can therefore be used to multiplex two independent information channels within a single hypervector.

Concretely, instead of allocating separate vectors, we superimpose the confidence update into a *permuted channel* of the estimate memory. The update that would normally target $\mathcal{B}_{a_t}$ is instead permuted and accumulated directly into $\mathcal{A}_{a_t}$:

$$\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} + (\mathcal{R}_{t,a_t} \odot \mathcal{X}_{t,a_t}) + \rho(\mathbf{m}_t \odot \mathcal{X}_{t,a_t}), \quad (10.7)$$

where $\mathbf{m}_t \in \{0,1\}^D$ is the binary mask whose density depends on the estimated confidence. During action selection, the confidence of action a is recovered by correlating $\mathcal{A}_a$ with the *permuted* context HV:

$$c_{t,a} = \delta(\mathcal{A}_a, \rho(\mathcal{X}_{t,a})). \quad (10.8)$$

Because the permuted and unpermuted channels are approximately decorrelated, the estimate and confidence contributions remain separable. The resulting formulation halves the memory requirement from $2N$ to N representative hypervectors.

Empirically, this compression has negligible impact on learning quality. Figure 10.1 reports results on 100 synthetic datasets, showing that the compact variant can learn and improve over time, converging towards ideal unitary reward values.

10.2.2 From per-action memories to a single global memory

The confidence multiplexing introduced above reduces the number of HVs per action (from two to one) but still requires one vector per action. We now pursue the more ambitious goal of eliminating per-action storage altogether, collapsing all action-specific prototypes into a *single* global memory vector $\mathbf{M} \in \mathbb{R}^D$.

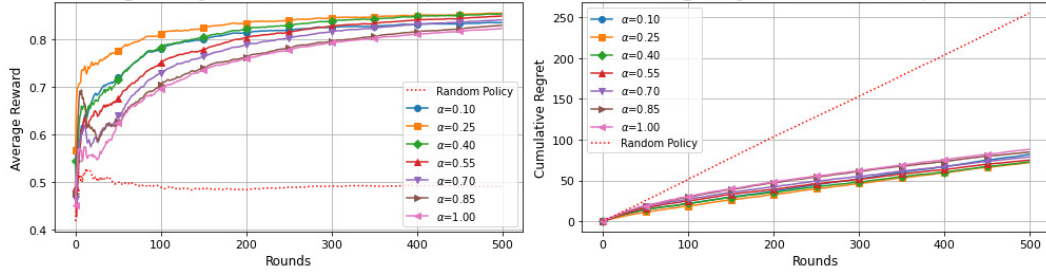


Figure 10.1 Learning behavior of HD-CB_{UNC2} with confidence multiplexed into the estimate memory via permutation.

Core mechanism: label-based superposition. To separate the contributions of different actions within a shared memory, we assign each action a a unique random label *hypervector* $\mathbf{L}_a \in \{-1, +1\}^D$, which acts as a nearly orthogonal key.

At round t , an action-specific key is formed by binding the encoded context with the label:

$$\mathbf{q}_{t,a} = \mathcal{X}_{t,a} \odot \mathbf{L}_a \in \{-1, +1\}^D. \quad (10.9)$$

The reward-modulated update for the selected action is then "tagged" and accumulated in the global memory by superposition:

$$\mathbf{M} \leftarrow \mathbf{M} + (\mathcal{R}_{t,a_t} \odot \mathbf{q}_{t,a_t}). \quad (10.10)$$

A useful consequence of this construction is that *cosine normalization is no longer required for action selection*. Since both $\mathcal{X}_{t,a}$ and $\mathbf{L}_a$ are bipolar, every key $\mathbf{q}_{t,a}$ has constant norm $\sqrt{D}$, and the memory norm $\|\mathbf{M}\|_2$ is common to all actions at a given time step. The cosine similarity, therefore, differs from the dot product by the same positive constant for all actions, leaving the $\arg \max$ unchanged. This removes one division per action from the inference path; a notable simplification for embedded implementations.

$$s_t(a) = \langle \mathbf{M}, \mathbf{q}_{t,a} \rangle, \quad a_t = \arg \max_a s_t(a). \quad (10.11)$$

This yields a more compact algorithmic formulation, and it enables a straightforward matrix implementation. Stacking labels into $\mathbf{L} \in \{-1, +1\}^{N \times D}$ and defining the key matrix $\mathbf{Q}_t = \mathbf{X}_t \odot \mathbf{L}$, Eq. (10.11) becomes:

$$\mathbf{s}_t = \mathbf{Q}_t \mathbf{M} = (\mathbf{X}_t \odot \mathbf{L}) \mathbf{M} \in \mathbb{R}^N, \quad (10.12)$$

exposing a single dense matrix–vector kernel. Algorithm 8 lists the complete single-memory HD-CB_{EPS} in vectorized form. Notably, action labels are bipolar and can be stored as bit-packed binary HVs, or generated on-the-fly from a single seed via permutation ($\mathbf{L}_a = \rho^a(\mathbf{L}_0)$), further reducing persistent memory to a single vector.

Figure 10.2 confirms that the single-memory variant learns and converges toward ideal reward values, validating the approach on 100 synthetic datasets.

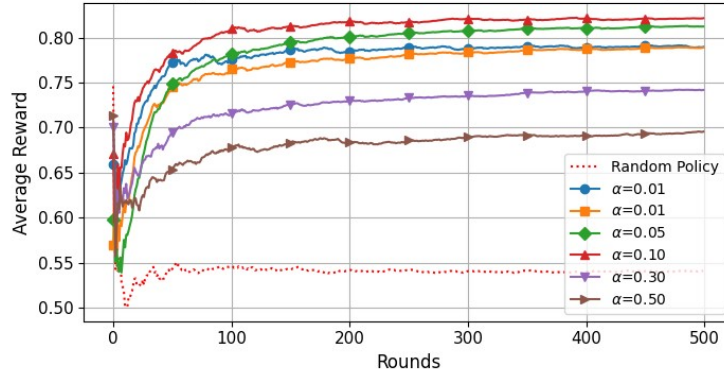
Extension to uncertainty-aware and hybrid models. Both compression mechanisms compose naturally. For the single-memory uncertainty-aware variant, we

Algorithm 8 Single-memory HD-CB_{EPS} (vectorized)

```

1: Input:  $\epsilon \in [0, 1]$ 
2: Initialization: draw action-label matrix  $\mathbf{L} \in \{-1, +1\}^{N \times D}$ ; set global memory  $\mathbf{M} \leftarrow \mathbf{0}_D$ 
3: for  $t = 1, \dots, T$  do
4:   Encode context(s)  $\rightarrow \mathbf{X}_t \in \{-1, +1\}^{N \times D}$  (rows are  $\mathcal{X}_{t,a}$ ; often identical)
5:    $\mathbf{Q}_t \leftarrow \mathbf{X}_t \odot \mathbf{L}$  (row-wise keys  $\mathbf{q}_{t,a} = \mathcal{X}_{t,a} \odot \mathbf{L}_a$ )
6:    $\mathbf{s}_t \leftarrow \mathbf{Q}_t \mathbf{M}$  (scores via dot products)
7:   Draw  $u \sim \text{Uniform}(0, 1)$ 
8:   if  $u < \epsilon$  then
9:      $a_t \leftarrow \text{RandomAction}()$ 
10:  else
11:     $a_t \leftarrow \arg \max_{a \in \{1, \dots, N\}} s_t(a)$ 
12:  end if
13:  Observe payoff  $r_{t,a_t}$  and encode reward HV  $\mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ 
14:   $\mathbf{M} \leftarrow \mathbf{M} + (\mathcal{R}_{t,a_t} \odot \mathbf{q}_{t,a_t})$ 
15: end for

```

**Figure 10.2** Learning behavior of HD-CB_{EPS} with one single global memory .

combine label-based superposition with permutation-based confidence multiplexing: the global memory is updated as

$$\mathbf{M} \leftarrow \mathbf{M} + (\mathcal{R}_{t,a_t} \odot \mathbf{q}_{t,a_t}) + \rho(\mathbf{m}_t \odot \mathbf{q}_{t,a_t}), \quad (10.13)$$

and action selection uses $\mathbf{s}_t = \mathbf{Q}_t \mathbf{M} - \alpha \rho(\mathbf{Q}_t) \mathbf{M}$, where the subtraction of the permuted-channel score implements the confidence-style exploration bonus. Cosine similarity is computed only for the selected action to derive the absolute confidence value required for mask generation:

$$c_{t,a_t} = \frac{1}{2} \left(1 + \frac{\langle \mathbf{M}, \rho(\mathbf{q}_{t,a_t}) \rangle}{\|\mathbf{M}\|_2 \sqrt{D}} \right). \quad (10.14)$$

The complete procedure is given in Algorithm 9.

Single-memory hybrid variant. For the hybrid setting, we combine action labels with action-feature hypervectors to construct composite keys. Let $\mathbf{Y} \in \{-1, +1\}^{N \times D}$

Algorithm 9 Single-memory HD-CB_{UNC2} (vectorized, multiplexed confidence; query permutation)

```

1: Input:  $\alpha > 0$ 
2: Initialization: draw  $\mathbf{L} \in \{-1, +1\}^{N \times D}$ ; set global memory  $\mathbf{M} \leftarrow \mathbf{0}_D$ 
3: for  $t = 1, \dots, T$  do
4:   Encode context(s)  $\rightarrow \mathbf{X}_t \in \{-1, +1\}^{N \times D}$ 
5:    $\mathbf{Q}_t \leftarrow \mathbf{X}_t \odot \mathbf{L}$ 
6:    $\mathbf{s}_t \leftarrow \mathbf{Q}_t \mathbf{M} - \alpha \rho(\mathbf{Q}_t) \mathbf{M}$ 
7:    $a_t \leftarrow \arg \max_a s_t(a)$ 
8:    $c_{t,a_t} \leftarrow \frac{1}{2} \left( 1 + \frac{\langle \mathbf{M}, \rho(\mathbf{q}_{t,a_t}) \rangle}{\|\mathbf{M}\|_2 \sqrt{D}} \right)$ 
9:   Sample mask  $\mathbf{m}_t \in \{0, 1\}^D$  with  $\lceil c_{t,a_t} D \rceil$  active indices
10:  Observe payoff  $r_{t,a_t}$  and encode  $\mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ 
11:   $\mathbf{M} \leftarrow \mathbf{M} + (\mathcal{R}_{t,a_t} \odot \mathbf{q}_{t,a_t}) + \rho(\mathbf{m}_t \odot \mathbf{q}_{t,a_t})$ 
12: end for

```

stack the action-feature HVs $\mathcal{Y}_a$ (one per action), obtained by encoding action features $y_a \in \mathbb{R}^f$. We build a composite key matrix $\mathbf{C} \in \{-1, +1\}^{N \times D}$ by bundling label and action-feature vectors row-wise:

For the hybrid setting, we replace action labels with *composite keys* that bundle label and action-feature information: $\mathbf{C}_a = \text{sgn}(\mathbf{L}_a + \mathbf{Y}_a)$, where $\mathbf{Y}_a$ is the encoded action-feature HV. Because $\mathbf{C}_a$ is obtained through superposition and thus retains partial correlation with both $\mathbf{L}_a$ and $\mathbf{Y}_a$, querying $\mathbf{M}$ with either component individually recovers the corresponding channel’s contribution, enabling separate confidence estimates for shared and dedicated components. The complete single-memory hybrid algorithm is given in Algorithm 10.

10.2.3 Accuracy–memory–throughput trade-offs

Storing multiple action-specific associations in superposition inevitably introduces cross-talk, potentially degrading performance relative to fully dedicated formulations. Table 10.2 quantifies this trade-off on the most demanding setting, the hybrid model, comparing the single-memory variant (**vec**) against the original Hybrid HD-CB and Hybrid LinUCB. Despite the aggressive compression, **vec** performs better than Hybrid LinUCB in average regret across all tested (d, f) configurations, while simultaneously reducing per-round execution time and collapsing the memory footprint from $(2N + 2)$ HVs to one.

The moderate regret increase relative to the original Hybrid HD-CB is therefore the expected cost of packing multiple action channels into a single superposed memory. Importantly, this gap scales with the representational capacity of the hyperspace. As the hypervector dimension D increases, near-orthogonality between superposed components improves and cross-talk diminishes. Figure 10.3 confirms this trend: the regret gap between the original and compact formulations decreases monotonically with D , indicating that the compression penalty can be mitigated by increasing the HV dimensionality.

From a deployment perspective, the central outcome is the change in memory

Algorithm 10 Single-memory Hyb-HD-CB_{UNC2} (vectorized; query permutation multiplexing)

```

1: Input:  $\alpha > 0$ 
2: Initialization:
3:   draw labels  $\mathbf{L} \in \{-1, +1\}^{N \times D}$ ; set global memory  $\mathbf{M} \leftarrow \mathbf{0}_D$ 
4:   encode action features  $y_a \in \mathbb{R}^f \rightarrow \mathbf{Y}_a$  for all  $a$ ; stack  $\mathbf{Y} \in \{-1, +1\}^{N \times D}$ 
5:   build composite keys  $\mathbf{C} \in \{-1, +1\}^{N \times D}$  with rows  $\mathbf{C}_a \leftarrow \text{sgn}(\mathbf{L}_a + \mathbf{Y}_a)$ 
6: for  $t = 1, \dots, T$  do
7:   Encode context(s)  $\rightarrow \mathbf{X}_t \in \{-1, +1\}^{N \times D}$ 
8:    $\mathbf{Q}_t \leftarrow \mathbf{X}_t \odot \mathbf{C}$ 
9:    $\mathbf{s}_t \leftarrow \mathbf{Q}_t \mathbf{M} - \alpha \rho(\mathbf{Q}_t) \mathbf{M}$   $\triangleright$  subtraction: confidence  $\rightarrow$  exploration bonus
10:   $a_t \leftarrow \arg \max_a s_t(a)$ 
11:  Compute confidences for selected action:
12:     $c_{t,a_t}^{(\text{sh})} \leftarrow \frac{1}{2} \left( 1 + \frac{\langle \mathbf{M}, \rho(\mathcal{X}_{t,a_t} \odot \mathbf{Y}_{a_t}) \rangle}{\|\mathbf{M}\|_2 \sqrt{D}} \right)$   $\triangleright$  shared channel
13:     $c_{t,a_t}^{(\text{ded})} \leftarrow \frac{1}{2} \left( 1 + \frac{\langle \mathbf{M}, \rho(\mathcal{X}_{t,a_t} \odot \mathbf{L}_{a_t}) \rangle}{\|\mathbf{M}\|_2 \sqrt{D}} \right)$   $\triangleright$  dedicated channel
14:
15:  Form masks  $\mathbf{m}_s, \mathbf{m}_d \in \{0, 1\}^D$  with  $\lceil c_{t,a_t}^{(\text{sh})} D \rceil$  and  $\lceil c_{t,a_t}^{(\text{ded})} D \rceil$  active indices
16:  Observe payoff  $r_{t,a_t}$  and encode  $\mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ 
17:   $\mathbf{M} \leftarrow \mathbf{M} + (\mathcal{R}_{t,a_t} \odot (\mathcal{X}_{t,a_t} \odot \mathbf{C}_{a_t})) + \rho(\mathbf{m}_s \odot (\mathcal{X}_{t,a_t} \odot \mathbf{Y}_{a_t})) + \rho(\mathbf{m}_d \odot (\mathcal{X}_{t,a_t} \odot \mathbf{L}_{a_t}))$ 
18: end for

```

scaling. Figure 10.4 reports memory footprint as a function of the number of actions N . The baseline variants exhibit steep linear growth (N , $2N$, or $2N + 2$ HVs). The single-memory formulations shift the N prototypes to lightweight action identifiers: when stored explicitly as bit-packed bipolar HVs, the slope is substantially reduced; when generated on-the-fly from a single seed via permutation ($\mathbf{L}_a = \rho^a(\mathbf{L}_0)$), memory becomes effectively constant with N .

Finally, Table 10.3 complements the analysis with a throughput perspective, reporting per-round execution time across devices for the loop-based, matrix-form, and compact vectorized (**vec**) implementations. Across all platforms, the compact formulation not only improves memory scalability but also yields consistently lower runtimes.

10.3 Summary of Findings

This chapter presented two engineering refinements designed to enhance the execution efficiency of HD-CB algorithms on general-purpose hardware and reduce their memory footprint on embedded platforms.

- **Matrix-form reformulations.** We derived mathematically equivalent matrix-form expressions for HD-CB and its variants that expose data parallelism without changing the learning dynamics. On CPUs, the matrix formulation yields $2.3\times$ – $3.6\times$ speedups over loop-based implementations by enabling BLAS-level optimization. On GPUs, the matrix form provides consistent $1.1\times$ –

Table 10.2 Hybrid bandit: average per-round execution time (ms) and average regret (best α per model). Binary Reward.

Dims		Hybrid LinUCB		Hybrid HD-CB		Hybrid HD-CB-vec	
d	m	Time (ms)	AvgRegret	Time (ms)	AvgRegret	Time (ms)	AvgRegret
8	8	5.006	0.0893	1.404	0.0411	0.571	0.0551
8	16	12.058	0.0840	1.444	0.0501	0.572	0.0640
8	32	21.150	0.0704	1.452	0.0434	0.575	0.0586
16	8	13.854	0.0797	1.504	0.0465	0.649	0.0485
16	16	25.382	0.0752	1.558	0.0286	0.647	0.0584
16	32	50.062	0.0669	1.540	0.0322	0.656	0.0453
32	8	35.897	0.0587	1.640	0.0310	0.769	0.0495
32	16	87.625	0.0589	1.674	0.0386	0.765	0.0458
32	32	141.534	0.0563	1.701	0.0355	0.771	0.0440

Table 10.3 Average per-round execution time (ms) across devices.

Config		CPU i7-11800H (ms)				GPU RTX 3050 (ms)				GPU T4 (ms)			
d	f	LINUCB	Loop	Matrix	Vec	LINUCB	Loop	Matrix	Vec	LINUCB	Loop	Matrix	Vec
4	4	1369.449	1177.798	493.665	381.060	5.281	5.005	4.376	3.483	3.746	1.356	1.200	0.956
4	8	2212.563	1193.821	482.612	372.596	10.242	5.326	4.798	3.553	3.067	1.321	1.199	0.928
4	16	4172.073	1146.704	441.126	363.298	10.384	5.219	4.654	3.719	3.182	1.367	1.191	0.916
4	32	10033.161	1192.312	466.750	359.620	6.571	5.275	5.192	3.310	3.733	1.751	1.214	1.159
8	4	2927.776	1146.312	496.890	396.475	11.671	6.145	5.212	4.001	3.047	1.399	1.187	0.949
8	8	5074.613	1219.161	488.074	385.238	9.832	5.252	4.515	4.097	3.179	1.378	1.172	0.917
8	16	28139.676	1539.790	579.105	475.264	9.575	5.370	4.695	3.734	4.282	1.766	1.670	1.171
8	32	205794.737	2834.616	849.386	725.192	7.721	6.451	5.317	3.364	3.523	1.347	1.185	0.955
16	4	32293.778	3225.185	1026.497	922.889	10.512	5.061	4.377	3.560	3.176	1.356	1.173	0.914
16	8	99630.848	3207.573	1021.705	924.849	6.323	4.977	4.403	3.474	3.324	1.384	1.459	0.914
16	16	249301.224	3159.496	1015.472	914.883	6.746	5.267	5.483	3.418	4.329	1.365	1.189	0.901
16	32	604975.346	3721.883	1021.867	913.637	6.416	6.102	4.528	2.460	3.990	1.374	1.161	0.962
32	4	170328.144	3282.692	1151.511	1078.560	8.289	5.256	4.410	3.102	3.604	1.369	1.217	0.920
32	8	309019.130	3499.950	1179.127	1105.928	6.207	7.057	5.354	2.847	4.846	1.352	1.183	0.933
32	16	686497.052	3359.164	1167.034	1095.252	9.409	7.096	5.242	2.842	4.359	1.387	1.177	0.955
32	32	1080394.730	3379.553	1167.567	1093.764	13.963	6.382	4.942	2.574	7.125	1.818	1.565	0.911

1.4 $\times$ improvements. These reformulations bridge the gap between the HDC abstraction and efficient execution on any vectorization-capable backend.

- **Memory-optimized variants.** By exploiting permutation-based channel multiplexing and label-based superposition, we demonstrated that the memory footprint of HD-CB can be reduced from $\mathcal{O}(ND)$ (one or more hypervectors per action) to $\mathcal{O}(D)$ (a single global memory vector). The compression introduces cross-talk between superposed action channels, but this penalty diminishes as the hypervector dimension D increases, and the compact variants remain superior to Hybrid LinUCB in both regret and execution time. For embedded platforms where storing N separate hypervectors would exceed on-chip memory, this reformulation makes HD-CB deployment feasible.

Both techniques are general-purpose optimizations applicable, in principle, to any HD-CB variant. The advanced learning formulations introduced in the following chapter were developed and evaluated independently of these implementation refinements; integrating them is left as a promising direction for future work.

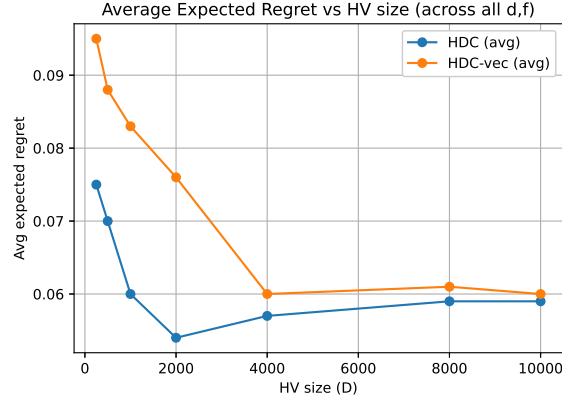


Figure 10.3 Effect of hypervector dimensionality on the regret penalty induced by the compact single-memory formulation. The plot reports the average expected regret of the original hybrid HD-CB model and its compressed variant as a function of the HV size D , averaged over 100 independent synthetic datasets. As D increases, near-orthogonality improves and superposition cross-talk diminishes, causing the regret of the compact model to progressively approach that of the original formulation.

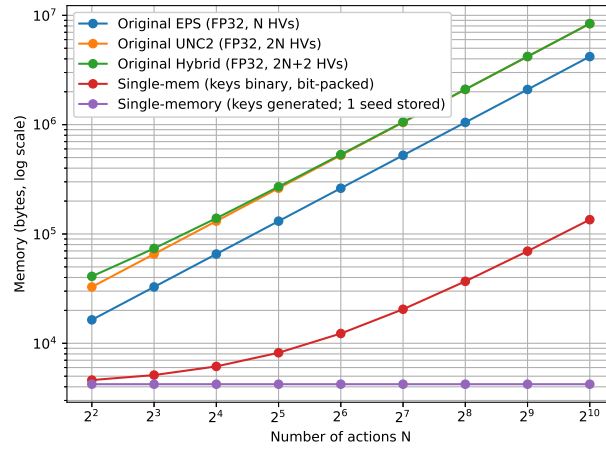


Figure 10.4 Memory footprint versus number of actions N (log scale) for baseline HD-CB variants and proposed single-memory formulations. Baselines store representative HVs per action. The single-memory formulation stores one global model HV and uses bipolar action keys: if keys are stored explicitly they can be bit-packed; alternatively, keys can be generated on-the-fly as permutations of a single seed.

Chapter 11

From Heuristic Updates to Optimization-Based Learning

This Chapter presents original work not yet published

"Change is the end result of all true learning."

Leo Buscaglia

The previous chapters of this dissertation have established HDC/VSA as an effective and efficient substrate for modelling and solving contextual bandit problems. Chapter 7 introduced HD-CB as a foundational framework; Chapter 8 addressed extreme resource constraints through binarization; Chapter 9 extended the approach to capture inter-action correlations; and Chapter 10 introduced efficient implementations. Together, these contributions demonstrated that HDC can deliver robust online learning while avoiding the unfavorable cubic scaling of classical linear bandits.

However, the HD-CB framework, as presented so far, relies on heuristic update rules whose empirical effectiveness has been demonstrated but not formally justified.

This chapter asks a deeper question:

❓ Research Question

Why does HD-CB work, and what is its relationship to established learning models?

We answer this question from three converging angles that progressively transform HD-CB from a brain-inspired heuristic into a principled learning framework with rigorous theoretical grounding.

First, we address the practical limitation of unbounded accumulation in standard HD-CB updates. Section 11.1 introduces Probabilistic HD-CB, which replaces deterministic updates with bounded stochastic ones governed by time-decaying probabilities and saturating integer counters. This enables deployment with as few as 3 bits per component while avoiding the information loss of periodic binarization. Second, we tackle the theoretical gap between HD-CB's heuristic updates and the optimization-grounded formulations of classical bandits. Section 11.2 first derives the implicit loss function that the baseline update rule minimizes, revealing an

ensemble structure where each hypervector component acts as an independent weak learner. Identifying the limitations of this implicit objective, we then formulate Loss-Based HD-CB, which replaces heuristic updates with explicit minimization of a well-defined loss function. The loss-based and probabilistic mechanisms are combined into Loss-Probabilistic HD-CB, which integrates error-driven learning within fixed-precision integer arithmetic. Third, Section 11.3 formalizes the mapping between the HD-CB architecture and Random Vector Functional Link networks, connecting HD-CB to the broader theory of neural networks and opening a bridge toward spiking neural networks and neuromorphic hardware.

11.1 Probabilistic HD-CB via Integer Self-Organizing Maps-Inspired Bounded Stochastic Updates

While the optimizations presented thus far have focused on reorganizing computations and reducing memory overhead, as highlighted in Chapter 7, a significant barrier to deploying HD-CB on resource-constrained platforms is its *accumulation-based* update rule: vectors representing actions are incremented or decremented at every action selection, and the growth of their components is not limited over time. Preventing overflow, therefore, requires either integer or floating-point types with high-precision, both of which are expensive in terms of computational and memory costs.

To address this problem and reduce these costs, in Chapter 8 [304] we introduced HD-CB_{BIN}, a low-precision variant that stores representatives in a Q -bit integer format and periodically applies *hard binarization*. During action selection, representations are binarized to enable Hamming-distance similarity, and every Q updates the accumulated memories are projected back to a binary domain to keep the state bounded.

However, while effective in reducing computational and memory costs, these mechanisms rely on *hard binarization*. The periodic reset indiscriminately discards the information accumulated in components between resets: a component that has reached a value of, e.g., 100 becomes indistinguishable from one with 2, erasing the ranking information that the model has built over many interactions. As noted at the end of Chapter 8, this raises the question of whether periodic binarization introduces a systematic accuracy penalty when simpler exploration strategies—such as ϵ -greedy, which lacks the confidence-driven correction of HD-CB_{UNC2}—are employed, or when the model is evaluated in more complex and diverse environments. The experimental results presented later in this section confirm that this is indeed the case: under off-policy evaluation on standardized benchmarks with ϵ -greedy exploration, HD-CB_{BIN} incurs a non-negligible performance degradation compared to its real-valued counterpart, with the penalty becoming more pronounced at lower bit-widths. This raises an important design choice between HD-CB, with its large memory footprint, and HD-CB_{BIN}, which degrades performance after periodic resets. Motivated by this limitation, this chapter pursue a different strategy, asking:

? Research Question

Rather than periodically compressing an otherwise unbounded update, can we *change the update rule itself* so that the learning dynamics remain bounded by design to a predefined dynamic range?

We answer by presenting *probabilistic HD-CB* (HD-CB_{PROB}), a bounded-by-design variant based on a probabilistic update rule inspired by the previous work on low-precision neural networks [308, 309]. The proposed approach updates only a randomly selected subset of vector components at each action selection, enabling the use of low-precision, saturated integer counters (limited to a range $[-\kappa, \kappa]$) by controlling the learning rate with time-decaying update probabilities. This approach allows the agent to retain historical information with the high-fidelity in low-precision vector components. As a result, the overflow is avoided, update costs are decreasing with time, and learning no longer depends on periodic binarization steps that may severely affect usefulness of representations.

11.1.1 Learning low-precision integer representations

The probabilistic update rule leveraged in this Section is inspired by the previous studies of self-organizing maps that proposed a reformulation of the algorithm, designed to enable low-cost learning with digital hardware [308, 309]. While the original update rule was presented in another context, the formulation can be used for updating signed integer representations via simple increment/decrement steps. Two concepts are particularly relevant to the CB setting: (i) a bounded integer state enforced by saturation and (ii) a probabilistic per-component update controlled by an update probability that decreases with time [308, 309].

Unlike the standard accumulation-based update, which allows vector components to grow indefinitely, the probabilistic update rule constrains each component to a fixed integer range $[-\kappa, +\kappa]$. This is achieved by applying a clipping function $f_\kappa(\cdot)$ after every update:

$$f_\kappa(v) = \max(-\kappa, \min(v, +\kappa)), \quad (11.1)$$

where v denotes the value being clipped, and κ is a configurable saturation threshold. It controls the trade-off between representational capacity and dynamic range: small κ yields very low-precision representations (e.g., ternary for $\kappa = 1$), while larger κ allows the agent to accumulate more information. If v is restricted to integers, $f_\kappa(v)$ can be stored within $\lceil \log_2(2\kappa + 1) \rceil$ bitwidth, preventing the unlimited growth typical of standard accumulation-based updates.

To enable learning within such low-precision representations, the deterministic accumulation is replaced with *probabilistic* updates. Let $w_{t,i}$ denote the i -th component of the weight vector at time step t , and let $\mathcal{T}_{t,i} \in \{-1, +1\}$ denote the corresponding component of the current update signal. Instead of using $\mathcal{T}_{t,i}$ at every update, components are updated with probability α_t , which decreases over time [308, 309]. Specifically, for each component i , a random value $m_{t,i} \sim \mathcal{U}(0, 1)$ is

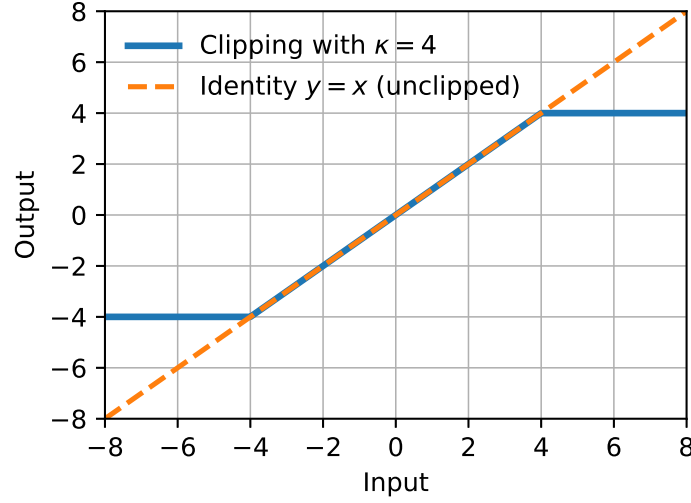


Figure 11.1 Clipping function used to bound integer-valued memories. The solid curve shows $f_{\delta}(x) = \min(\kappa, \max(-\kappa, x))$ for $\kappa = 4$, while the dashed line is the identity $y = x$ (unclipped). Clipping prevents unbounded growth by saturating values outside the interval $[-\kappa, \kappa]$.

sampled; the update is performed only if $m_{t,i} \leq \alpha_t$:

$$w_{t,i} = \begin{cases} f_{\kappa}(w_{t-1,i} + \mathcal{T}_{t,i}), & \text{if } m_{t,i} \leq \alpha_t, \\ w_{t-1,i}, & \text{otherwise.} \end{cases} \quad (11.2)$$

In contrast to many other learning rules that scale the update value over time, the update rule in Eq. (11.2) effectively scales the *frequency* of updates through α_t . By adopting this principle, we can update the action vectors in HD-CB using only integer addition and saturation, achieving a bounded integer representation and reduced write-access frequency without the information loss of binarization.

11.1.2 Probabilistic HD-CB

This section introduces HD-CB_{PROB}, a variant of HD-CB designed to operate under strict memory constraints without the information loss associated with periodic resets as in HD-CB_{BIN}. The core intuition is to replace the unlimited, magnitude-based update of the standard HD-CB with the probabilistic update rule on components with low-precision. By treating the learning rate not as a scaling factor for the update value, but as a *probability of update*, we can design an agent using low-precision saturating integers only.

11.1.3 Algorithm overview

HD-CB_{PROB} (Algorithm 11) maintains a set of action hypervectors $\mathcal{A}_a \in \{-\kappa, \dots, +\kappa\}^D$, one for each action $a \in \{1, \dots, N\}$. Unlike standard HD-CB, where components are high-precision integers or floats, here every component $\mathcal{A}_{a,i}$ is strictly bounded by the saturation threshold κ .

HD-CB_{PROB} proceeds as follows:

1. *Context encoding:* Context $\mathbf{x}_t$ is encoded into a hypervector $\mathcal{X}_t \in \{-1, +1\}^D$ using a record-based transformation [225] (hereafter, we use the $\mathbf{x}_t$ notation even if contexts are action-dependent). The fact that it is bipolar is important, as updates are done simply in ± 1 steps, remaining compatible with low-precision saturating integers.
2. *Action selection:* The agent computes the similarity between the context $\mathcal{X}_t$ and all action hypervectors $\mathcal{A}_a$, selecting action a_t via an ε -greedy strategy. Notably, in this variant, similarity can be efficiently computed using the inner product, as the norm of the saturating hypervectors remains implicitly bounded over time.
3. *Probabilistic update:* Upon receiving reward r_{t,a_t} , the agent encodes it into a hypervector $\mathcal{R}_{t,a_t}$ (using the thermometer encoding) and binds it with $\mathcal{X}_t$ to form an update hypervector. Instead of adding it to $\mathcal{A}_a$, the probabilistic update rule is applied, leading to a randomly selected subset of components being modified, and the result is clipped to the range $[-\kappa, \kappa]$.

11.1.4 Probabilistic update rule

The key feature of HD-CB_{PROB} is its update rule. Following the probabilistic update rule in Eq. (11.2), each component is updated with probability α_t and then clipped to the range $[-\kappa, \kappa]$. The parameter $\alpha_t \in [0, 1]$ reflects a time-decaying learning schedule and is computed as $\alpha_t = \alpha_0 (1 - (t - 1)/T)$ for $t = 1, \dots, T$, where $\alpha_t = 0$ for $t > T$.

At each time step, for each component $i \in \{1, \dots, D\}$, an i.i.d. random variable $m_{t,i} \sim \mathcal{U}(0, 1)$ is sampled and a binary update mask $\mathcal{M}_{t,i} \in \{0, 1\}$ is computed as $\mathcal{M}_{t,i} = \mathbb{I}(m_{t,i} \leq \alpha_t)$, where $\mathbb{I}(\cdot)$ is the indicator function. The update directions are determined by the result of the binding of the reward and context hypervectors: $\mathcal{T}_t = \mathcal{R}_{t,a_t} \odot \mathcal{X}_t \in \{-1, +1\}^D$. Finally, the chosen action hypervector is updated as:

$$\mathcal{A}_{a_t} = f_{\kappa}(\mathcal{A}_{a_t} + \mathcal{M}_t \odot \mathcal{T}_t), \quad (11.3)$$

Intuitively, this update acts as a *saturating random walk* driven by the bipolar update signal: active components (where $\mathcal{M}_{t,i} = 1$) take a single ± 1 step towards the saturation boundaries $\pm \kappa$, ensuring overflow is always prevented while preserving the direction of the accumulated hypervectors.

HD-CB_{PROB} addresses the limitations of periodic binarization while preserving the properties that motivated HD-CB_{BIN}. *Low-precision components*, $\mathcal{A} \in [-\kappa, \kappa]^{D \times N}$, are guaranteed by design via saturation. *Update cost is reduced* since updates occur only when $\mathcal{M}_{t,i} = 1$ and the number of write operations to memory is controlled by α_t . In later stages when α_t is getting close to zero, the updates become highly sparse, significantly reducing computational costs. *Periodic resets are not needed* as the agent can accumulate information in $\mathcal{A} \in [-\kappa, \kappa]^{D \times N}$ without periodically binarizing to $\{-1, +1\}$.

Algorithm 11 Probabilistic HD-CB with ε -greedy strategy

```

1: Input:  $N, T, D, \varepsilon, \kappa, \alpha_0$ ,
2: Initialization: Set  $\mathcal{A}_a$  to  $\mathbf{0}^D$  for all  $a \in \{1, \dots, N\}$ 
3: for  $t = 1, \dots, T$  do
4:   // 1. Context encoding
5:   Receive context  $\mathbf{x}_t$ 
6:   Encode  $\mathbf{x}_t \rightarrow \mathcal{X}_t \in \{-1, +1\}^D$ 
7:   // 2. Action selection ( $\varepsilon$ -greedy strategy)
8:   Draw  $q \sim \mathcal{U}(0, 1)$ 
9:   if  $q < \varepsilon$  then
10:    Select random action  $a_t \sim \mathcal{U}\{1, \dots, N\}$ 
11:   else
12:    Select  $a_t = \arg \max_a (\mathcal{X}_t^\top \mathcal{A}_a)$ 
13:   end if
14:   // 3. Observation
15:   Execute action  $a_t$ , observe reward  $r_{t,a_t}$ 
16:   // 4. Probabilistic update rule
17:   Compute decay  $\alpha_t \leftarrow \alpha_0 \max(0, 1 - \frac{t-1}{T})$ 
18:   Encode  $\mathbf{r}_{t,a_t} \rightarrow \mathcal{R}_{t,a_t} \in \{-1, +1\}^D$ 
19:   Compute update hypervector  $\mathcal{T}_t \leftarrow \mathcal{R}_{t,a_t} \odot \mathcal{X}_t$ 
20:   for  $i = 1, \dots, D$  do
21:     Draw  $\mathcal{M}_{t,i} \sim \mathcal{U}(0, 1)$ 
22:     if  $\mathcal{M}_{t,i} < \alpha_t$  then
23:       Updated value  $v \leftarrow \mathcal{A}_{a_t,i} + \mathcal{T}_{t,i}$ 
24:       Saturated value  $\mathcal{A}_{a_t,i} \leftarrow \max(-\kappa, \min(v, +\kappa))$ 
25:     end if
26:   end for
27: end for

```

11.1.5 Experimental Evaluation

To validate the proposed approach, we conducted an off-policy evaluation using the OBP [299] and performed a comparative analysis of the memory footprint of several approaches. These experiments quantify the trade-off between the low-precision action hypervectors formed with the probabilistic update rule and the quality of the decision-making. We compare four approaches introduced earlier. To be consistent, all of them use ε -greedy exploration strategy:

1. LinEPS: Standard linear CB algorithm based on covariance matrices and the ridge regression [74].
2. HD-CB_{REAL}: Real-valued HD-CB variant [225].
3. HD-CB_{BIN}: Binarized variant, relying on periodic binarization to prevent overflow [304].
4. HD-CB_{PROB}: Approach with probabilistic updates, Eq. (11.3).

Table 11.1 Off-policy evaluation results on the OBP library. The table compares the real-valued variant against two variants with low-precision representations. Best values among these variants are underlined; global bests are in bold.

N	d	LinEPS	HD-CB _{REAL}	HD-CB _{BIN}			HD-CB _{PROB} (Ours)		
				2 bits	3 bits	4 bits	2 bits	3 bits	4 bits
10	5	0.614	0.656	0.579	0.595	0.605	0.631	<u>0.648</u>	0.642
10	10	0.683	0.684	0.581	0.598	0.614	0.631	<u>0.652</u>	0.635
10	15	0.663	0.659	0.569	0.569	0.588	0.603	<u>0.631</u>	0.618
15	5	0.711	0.735	0.645	0.663	0.682	0.707	<u>0.718</u>	0.705
15	10	0.703	0.708	0.598	0.614	0.636	0.642	<u>0.671</u>	0.643
15	15	0.678	0.664	0.565	0.573	0.583	0.601	<u>0.630</u>	0.608
20	5	0.716	0.733	0.643	0.658	0.677	0.700	<u>0.728</u>	0.718
20	10	0.708	0.724	0.614	0.634	0.656	0.661	<u>0.685</u>	0.657
20	15	0.690	0.681	0.584	0.591	0.601	0.623	<u>0.656</u>	0.621

Off-policy evaluation Table 11.1 reports results obtained on the OBP library in the case of binary rewards for varying number of actions $N \in \{10, 15, 20\}$ and context dimensionality $d \in \{5, 10, 15\}$. The dimensionality of hypervectors is fixed at $D = 1024$. Cumulative rewards are averaged over 50 datasets per configuration. For HD-CB_{PROB}, the initial update probability is set to $\alpha_0 = 0.4$. We evaluate low-precision variants at 2, 3, and 4 bits per component. For HD-CB_{PROB}, these correspond to saturation thresholds of $\kappa = 1$ (ternary), $\kappa = 3$, and $\kappa = 7$ respectively. For HD-CB_{BIN}, instead, these correspond to $Q = 2, 3$ and 4, that is $2^Q = 4, 8, 16$, respectively. The exploration parameter ε is tuned via grid search for each configuration, and the best cumulative rewards on the OBP library are reported.

Three important observations could be made. First, HD-CB_{PROB} outperforms HD-CB_{BIN} for every tested configuration (N, d) and every bitwidth, it yields higher cumulative rewards than HD-CB_{BIN}. The advantage is particularly pronounced at low precision: averaged across all configurations, 3-bit HD-CB_{PROB} ($\kappa = 3$) improves over 3-bit HD-CB_{BIN} by approximately 5.8%. This confirms the hypothesis that adding information via probabilistic updates preserves more ranking information than periodic binarization steps. Second, 3-bit HD-CB_{PROB} consistently approaches the performance of HD-CB_{REAL} baseline, with an average gap of only 2.5%. This indicates that a very low precision is sufficient to capture the relative quality of arms, provided the update rule allows for fine-grained frequency-based adjustments.

Third, it is notable that increasing precision from 3 to 4 bits yields diminishing returns (or slightly negative) returns for HD-CB_{PROB}. This suggests that, in this setting, increasing the dynamic range provides limited additional benefit once the agent can already encode a modest amount of information per component. Additionally, the regularization effect of the tighter saturation bound $\kappa = 3$ may actually be beneficial in these synthetic scenarios.

Memory footprint analysis Figure 11.2 compares the memory footprint of the considered agents as a function of the context feature dimension d (reported in KiB on a logarithmic scale), for $d \in \{8, 16, 32, 64, 128\}$ and for 2-, 3-, and 4-bit instantiations

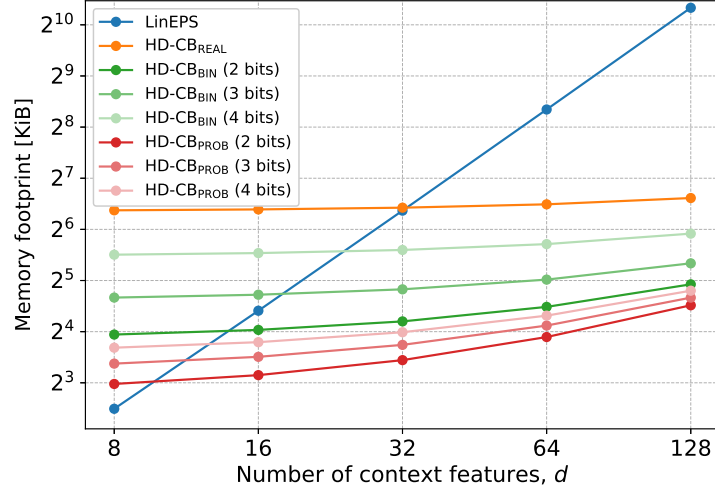


Figure 11.2 Memory footprint (KiB, log scale) vs. context dimensionality d . HD-CB_{PROB} (red) consistently requires less memory than HD-CB_{BIN} (green) at the same bitwidth as it does not require auxiliary counters or separate copies for inference.

of HD-CB_{BIN} and HD-CB_{PROB}. While LinEPS exhibits quadratic scaling with d , the HD-CB variants remain constant with respect to context dimensionality, scaling only with hyperdimension D . Crucially, at any fixed bitwidth, HD-CB_{PROB} requires *strictly less memory* than HD-CB_{BIN}. This efficiency stems from the architectural simplicity of the probabilistic approach. HD-CB_{BIN} must maintain additional machinery per action: (1) a Q -bit integer hypervector to track history, (2) a binarized copy for efficient inference and (3) per-action update counters to trigger re-binarization. In contrast, HD-CB_{PROB} stores action hypervectors directly in their final bounded format ($\lceil \log_2(2\kappa + 1) \rceil$ bits) and no additional copies or counters are required.

Overall, these results confirm that the proposed approach provides a better trade-off: compared to HD-CB_{BIN}, it systematically yields a higher average reward at equal bitwidth, while simultaneously reducing memory footprint and avoiding the performance degradation induced by periodic binarization.

These results position probabilistic HD-CB as a compelling solution for edge applications, offering a way to deploy adaptive, autonomous agents on devices with limited power and memory budgets.

11.2 Loss-Based HD-CB: From Heuristic Updates to Explicit Optimization

The HD-CB framework introduced in Chapter 7, and progressively refined through binary (Chapter 8), hybrid (Chapter 9), memory-optimized (Section 10.2), and probabilistic (Section 11.1) variants, has demonstrated that contextual bandit problems can be effectively solved in a high-dimensional space. Across synthetic benchmarks and real-world scenarios, HD-CB agents have consistently achieved strong performance while offering faster convergence, better scalability, and more hardware-friendly

implementations than classical linear bandits.

Nevertheless, some fundamental theoretical distinction separates all HD-CB formulations presented so far from classical linear bandit algorithms:

- **Absence of an explicit optimization objective.** Linear bandits are grounded in well-defined optimization problems, minimizing a regularized least-squares error, which provide convergence guarantees and regret bounds. All HD-CB variants discussed so far rely on geometric heuristics: the update either adds or subtracts the context hypervector from the action representative, weighted by the observed reward. While these rules are intuitive from an HDC/VSA perspective and empirically effective, they do not derive from any explicit loss function. As a consequence, there is no formal guarantee that the updates decrease a well-defined error over time.
- **Magnitude-insensitive error correction.** In the heuristic update, the magnitude of the update signal is determined solely by the reward encoding, not by the prediction error. If the model already predicts the observed reward accurately, the heuristic still applies a full-magnitude update, which can overwrite useful information. Conversely, if the prediction is far from the observed reward, the same fixed-magnitude update may be insufficient to correct the error. This insensitivity to the prediction residual means the heuristic lacks the self-correcting property that gradient-based methods possess by construction.
- **Norm growth and similarity saturation.** Under the accumulation-based update, the norm of the action representatives grows monotonically. As highlighted in Section 11.1, this unbounded growth requires either integer or floating-point types with high precision and makes the model increasingly insensitive to new evidence, since the relative contribution of a single update diminishes as the norm grows.

To address these limitations, this section asks two complementary questions. We begin by asking:

? Research Question

What implicit loss function does the baseline HD-CB update rule minimize, and what are its structural limitations?

To answer this question, we mirror a classical result in machine learning theory: the connection between the perceptron update rule and hinge loss minimization via stochastic gradient descent, as presented in the "Patterns, Predictions, and Actions" textbook [310] from Hardt and Recht. We apply this reasoning to the baseline HD-CB update, first for a simplified scalar-reward case and then for the full thermometer-encoded rule. This analysis reveals that the baseline HD-CB implicitly minimizes a per-component linear loss, with each action hypervector component acting as an independent weak learner operating at a distinct reward threshold. It also identifies the specific structural limitations of the implicit objective, which directly motivate the second question:

❓ Research Question

Research question. Can these limitations be addressed through an explicit loss formulation that bridges the gap between the heuristic update rules proposed in the preceding chapters and optimization-grounded learning?

To answer this question, we introduce a loss-based formulation of HD-CB inspired by the work in [311]. We define a per-round squared-error loss that measures the discrepancy between the reward predicted by the hyperspace model and the reward observed from the environment. We then derive the corresponding gradient with respect to the selected action hypervector and obtain an update rule that is explicitly grounded in first-order optimization. This transformation moves HD-CB from a heuristic method to a true optimization-based learning algorithm, providing a formal foundation for its empirical success and opening the door to the theoretical connections explored later in this thesis.

The section is structured as follows. We first derive the implicit loss function of the baseline update rule through a progressive analysis spanning the scalar-reward and thermometer-encoded cases, and discuss the resulting ensemble interpretation (Section 11.2.1). We then define the per-round explicit loss and derive the gradient for both cosine and sigmoid-based estimators (Sections 11.2.2–11.2.4), introducing Loss-HD-CB. We present an extended experimental evaluation including loss dynamics over time and a comparison between the cosine and sigmoid loss formulations (Section 11.2.5). Next, we introduce Loss-Probabilistic HD-CB, a variant that combines loss-based update with bounded stochastic integer updates (Section 11.2.6). Finally, we present a comprehensive comparison across all HD-CB variants as a function of the representation precision (Section 11.2.7).

11.2.1 Implicit loss of the baseline update rule

This section develops a theoretical analysis to identify the *implicit loss function* that the baseline HD-CB update rule minimizes. The methodology mirrors the well-known connection between the perceptron update rule and hinge loss minimization [310]. We proceed in three steps of increasing complexity:

- (i) **Perceptron loss:** we first review how the implicit loss can be derived from the perceptron update rule, establishing the methodology.
- (ii) **Simple scalar case:** we apply the same reasoning to a simplified HD-CB update $\mathcal{A}_a \leftarrow \mathcal{A}_a + r_t \mathcal{X}_t$, where r_t is the raw scalar reward. Although this variant does not learn effectively in practice, it provides an instructive intermediate step that isolates the core mechanism before the additional structure introduced by the thermometer encoding.
- (iii) **Thermometer-encoded case:** $\mathcal{A}_a \leftarrow \mathcal{A}_a + \mathcal{R}_t \odot \mathcal{X}_t$, where $\mathcal{R}_t$ is the thermometer-encoded reward hypervector. This is the baseline HD-CB update rule.

Notation. Throughout this subsection, we denote by $\mathcal{A}_a \in \mathbb{R}^D$ the prototype hypervector for action a , by $\mathcal{X}_t \in \{-1, +1\}^D$ the encoded context at round t , and

by $\mathcal{R}_t \in \{-1, +1\}^D$ the thermometer-encoded reward. We write $\mathcal{A}_{a,j}$, $\mathcal{X}_{t,j}$, $\mathcal{R}_{t,j}$ for the j -th component of the respective hypervectors.

Recap: the perceptron and its implicit loss

The classical perceptron operates in $\mathbb{R}^d$ with binary labels $y \in \{-1, +1\}$. Given a weight vector $\mathbf{w}_t$, an input feature vector $\mathbf{x}_t$, and the corresponding label y_t , the perceptron update rule is:

$$\begin{cases} \mathbf{w}_{t+1} = \mathbf{w}_t + y_t \mathbf{x}_t & \text{when } y_t \langle \mathbf{w}_t, \mathbf{x}_t \rangle < 1, \\ \mathbf{w}_{t+1} = \mathbf{w}_t & \text{otherwise.} \end{cases} \quad (11.4)$$

The “Patterns, Predictions, and Actions” book [310] proves that this update corresponds to stochastic gradient descent on the *hinge loss*:

$$\mathcal{L}_{\text{hinge}}(\mathbf{w}, \mathbf{x}, y) = \max\{1 - y \langle \mathbf{w}, \mathbf{x} \rangle, 0\}. \quad (11.5)$$

In fact, the gradient is:

$$\nabla_{\mathbf{w}} \mathcal{L}_{\text{hinge}} = \begin{cases} -y \mathbf{x} & \text{if } y \langle \mathbf{w}, \mathbf{x} \rangle < 1, \\ \mathbf{0} & \text{if } y \langle \mathbf{w}, \mathbf{x} \rangle > 1. \end{cases} \quad (11.6)$$

The simple HD-CB case: scalar reward update

Let’s start from a simplified HD-CB case, in which the scalar reward r_t is directly multiplied with the encoded context $\mathcal{X}_t$. In practice, in this variant, the model is unable to learn, because the similarity grows too fast independent of reward is received. However, this is an instructive and simple step for starting the analysis.

In this simplified HD-CB variant, the algorithm:

- models each action a with a dedicated prototype HV $\mathcal{A}_a \in \mathbb{R}^D$,
- encodes the observed context in the high-dimensional space $\mathcal{X}_t \in \{-1, +1\}^D$,
- performs prediction based on the cosine similarity:

$$\hat{a}_t = \arg \max_a \frac{\mathcal{A}_a \cdot \mathcal{X}_t}{\|\mathcal{A}_a\| \|\mathcal{X}_t\|}, \quad (11.7)$$

- observes a reward r_t for the chosen action,
- updates the model according to:

$$\mathcal{A}_a \leftarrow \mathcal{A}_a + r_t \mathcal{X}_t. \quad (11.8)$$

If we interpret the update as a step of stochastic gradient descent with unit learning rate ($\eta = 1$):

$$\mathcal{A}_a \leftarrow \mathcal{A}_a - \nabla_{\mathcal{A}_a} \mathcal{L}, \quad (11.9)$$

then we require:

$$\nabla_{\mathcal{A}_a} \mathcal{L} = -r_t \mathcal{X}_t. \quad (11.10)$$

To find the loss function, we integrate this gradient *component after component*. For each dimension $j = 1, \dots, D$:

$$\frac{\partial \mathcal{L}}{\partial \mathcal{A}_{a,j}} = -r_t \mathcal{X}_{t,j} \quad \implies \quad \mathcal{L} = -r_t \mathcal{X}_{t,j} \mathcal{A}_{a,j} + C_j. \quad (11.11)$$

Setting all integration constants $C_j = 0$ and summing over all D components, we obtain:

$$\mathcal{L}_{\text{scalar}}(\mathcal{A}_a, \mathcal{X}_t, r_t) = \sum_{j=1}^D (-r_t \mathcal{X}_{t,j} \mathcal{A}_{a,j}) = -r_t \sum_{j=1}^D \mathcal{X}_{t,j} \mathcal{A}_{a,j} = -r_t \langle \mathcal{A}_a, \mathcal{X}_t \rangle. \quad (11.12)$$

In summary:

$$\boxed{\mathcal{L}_{\text{scalar}}(\mathcal{A}_a, \mathcal{X}_t, r_t) = -r_t \langle \mathcal{A}_a, \mathcal{X}_t \rangle} \quad (11.13)$$

This is a *linear loss* in the inner product $\langle \mathcal{A}_a, \mathcal{X}_t \rangle$, scaled by the reward r_t . Minimizing this loss is equivalent to:

- **Maximizing** $\langle \mathcal{A}_a, \mathcal{X}_t \rangle$ when $r_t > 0$ (push the action prototype towards rewarding contexts).
- **Minimizing** $\langle \mathcal{A}_a, \mathcal{X}_t \rangle$ when $r_t < 0$ (push it away from punishing contexts).

If the reward was $r_t \in \{-1, +1\}$ the update $\mathcal{A}_a \leftarrow \mathcal{A}_a + r_t \mathcal{X}_t$ would be *structurally identical* to the perceptron update $\mathbf{w} \leftarrow \mathbf{w} + y \mathbf{x}$, with the difference that the perceptron only updates when $y \langle \mathbf{w}, \mathbf{x} \rangle < 1$, while the HD-CB baseline updates unconditionally.

A critical observation is that when $r_t \in [0, 1]$ (as is typical in bandits), the update always adds a non-negative multiple of $\mathcal{X}_t$. These updates just differ in magnitude, so the prototype always moves *toward* the context, regardless of whether the reward was good or bad. This could be the reason why this variant does not work: without sign changes in the update, the algorithm cannot push prototypes *away* from contexts that led to poor rewards. This observation is empirically confirmed by the fact that the earliest working variant of the algorithm used $(0.5 - r_t) \cdot \mathcal{X}_t$ as the update, effectively centering the reward around its mean and restoring the ability to produce both positive and negative updates.

The thermometer-encoded case

In the baseline variant of HD-CB, the obtained reward is encoded in the high-dimensional space using a thermometer code. It maps a scalar reward r_t (quantized into L levels) into a D -dimensional bipolar hypervector $\mathcal{R}_t \in \{-1, +1\}^D$. In particular, for a reward at level $\ell \in \{0, 1, \dots, L\}$, the encoding produces:

$$\mathcal{R}_{t,j} = \begin{cases} +1 & \text{if } j \leq \ell \cdot (D/L), \\ -1 & \text{otherwise.} \end{cases} \quad (11.14)$$

For higher rewards, more components are “activated” (i.e., set to +1).

The baseline HD-CB update rule uses binding between the encoded reward and the encoded context hypervectors:

$$\mathcal{A}_a \leftarrow \mathcal{A}_a + \mathcal{R}_t \odot \mathcal{X}_t, \quad (11.15)$$

where $\odot$ is the Hadamard (component-wise) product.

Component-wise decomposition and per-component loss. A key property of the Hadamard product is that it makes the action hypervector update *separable across dimensions*: each component j of the prototype hypervector is updated independently of all other components. Explicitly, for $j = 1, \dots, D$:

$$\mathcal{A}_{a,j} \leftarrow \mathcal{A}_{a,j} + \mathcal{R}_{t,j} \cdot \mathcal{X}_{t,j}. \quad (11.16)$$

This is precisely the *perceptron update rule* where $\mathcal{A}_{a,j}$ plays the role of the scalar weight, $\mathcal{X}_{t,j} \in \{-1, +1\}$ plays the role of the scalar input, and $\mathcal{R}_{t,j} \in \{-1, +1\}$ plays the role of the label.

By repeating the same integration step as in Section 11.2.1, we interpret each component update as a stochastic gradient step with unit learning rate:

$$\mathcal{A}_{a,j} \leftarrow \mathcal{A}_{a,j} - \frac{\partial \mathcal{L}_j}{\partial \mathcal{A}_{a,j}}, \quad (11.17)$$

which requires:

$$\frac{\partial \mathcal{L}_j}{\partial \mathcal{A}_{a,j}} = -\mathcal{R}_{t,j} \cdot \mathcal{X}_{t,j}. \quad (11.18)$$

Integrating with respect to $\mathcal{A}_{a,j}$:

$$\boxed{\mathcal{L}_j(\mathcal{A}_{a,j}, \mathcal{X}_{t,j}, \mathcal{R}_{t,j}) = -\mathcal{R}_{t,j} \cdot \mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j}} \quad (11.19)$$

Total loss as sum of per-component losses. Since the Hadamard product makes the update independent across dimensions (each component j is updated using only $\mathcal{R}_{t,j}$ and $\mathcal{X}_{t,j}$, with no cross-component coupling) the total loss can be written as the sum of D scalar losses:

$$\boxed{\begin{aligned} \mathcal{L}_{\text{therm}}(\mathcal{A}_a, \mathcal{X}_t, \mathcal{R}_t) &= -\sum_{j=1}^D \mathcal{R}_{t,j} \cdot \mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j} = -\langle \mathcal{R}_t \odot \mathcal{X}_t, \mathcal{A}_a \rangle \\ &\stackrel{L=D}{=} -\sum_{j=1}^D 2([j \leq r_t D] - 0.5) \mathcal{A}_{a,j} \mathcal{X}_{t,j} \end{aligned}} \quad (11.20)$$

In fact, if we take the gradient of $\mathcal{L}_{\text{therm}}$ with respect to $\mathcal{A}_{a,j}$, only the j -th term in the sum survives (all other terms are independent of $\mathcal{A}_{a,j}$), and we recover $\partial \mathcal{L}_{\text{therm}} / \partial \mathcal{A}_{a,j} = -\mathcal{R}_{t,j} \cdot \mathcal{X}_{t,j}$, which is exactly the negative of the update applied to component j in Eq. (11.16). Therefore, the sum correctly reconstructs the complete vector update rule $-\mathcal{R}_t \odot \mathcal{X}_t$.

This is a *linear loss on the bound (reward-context) hypervector*. Comparing with the simple case: the role previously played by the scalar r_t is now played by the encoded hypervector $\mathcal{R}_t$, and the effective input to the loss is $\mathcal{R}_t \odot \mathcal{X}_t$ instead of $r_t \mathcal{X}_t$.

Position-dependent probabilistic structure. The very interesting aspect is that, because of the thermometer encoding structure, the “label” $\mathcal{R}_{t,j}$ for each component j is *not independently and identically distributed* across positions. Instead, each position has a distinct marginal probability of being $+1$ or -1 that depends on the reward distribution and the component index. In particular:

- **Low-index components** (j small): $\mathcal{R}_{t,j} = +1$ for nearly all reward levels $\ell \geq 1$. Thus $\Pr(\mathcal{R}_{t,j} = +1)$ is **high**.
- **High-index components** (j large): $\mathcal{R}_{t,j} = +1$ only for the highest reward levels. Thus $\Pr(\mathcal{R}_{t,j} = +1)$ is **low**, and $\Pr(\mathcal{R}_{t,j} = -1)$ is high.

Formally, the probability of having $+1$ in position j when receiving a reward level ℓ drawn from some distribution $p(\ell)$ can be expressed as:

$$\Pr(\mathcal{R}_{t,j} = +1) = \Pr\left(\ell \geq \frac{j \cdot L}{D}\right). \quad (11.21)$$

Implicit ensemble interpretation. This position-dependent structure suggests an intriguing interpretation: the thermometer-encoded HD-CB can be viewed as an *implicit ensemble* of D weak learners (one per component), each with a different operating threshold on the reward scale. Specifically:

- Component j effectively asks: “Is the reward above the threshold $\tau_j = j \cdot L/D$?”
- The answer is $\mathcal{R}_{t,j} \in \{-1, +1\}$.
- Each component independently learns a classifier for this binarized question.
- The cosine similarity at prediction time aggregates all D “votes” into a single decision.

Intuitively, the sign of $\mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j}$ reflects whether, historically, contexts with this projection pattern tended to produce rewards above threshold τ_j when action a was chosen. This is reminiscent of AdaBoost or stacked binary classifiers. The thermometer encoding naturally creates this decomposition.

Role of different thermometer positions. This analysis reveals that the different components of the hypervector play distinct roles:

- **Early positions** (high $\Pr(\mathcal{R}_{t,j} = +1)$): these components are almost always updated in the same direction regardless of reward. They primarily encode the *frequency* of choosing action a in context $\mathcal{X}_t$, contributing a “popularity” signal rather than reward discrimination.
- **Late positions** (high $\Pr(\mathcal{R}_{t,j} = -1)$): these components are $+1$ only for the highest rewards. They act as *selective reward detectors*, contributing only when the action is truly excellent for the context.
- **Middle positions**: these have the most balanced label distribution and carry the most *information* in an information-theoretic sense (maximum entropy of $\mathcal{R}_{t,j}$).

Expected loss. Over time, the expected implicit loss for component j can be obtained by integrating over the reward distribution. Denoting the corresponding level distribution as $p(\ell)$:

$$\mathbb{E}_r[\mathcal{L}_j] = -\mathbb{E}_r[\mathcal{R}_{t,j}] \cdot \mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j} = -\beta_j \cdot \mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j}. \quad (11.22)$$

where β_j is a position-dependent bias:

$$\beta_j = \Pr(\mathcal{R}_{t,j} = +1) - \Pr(\mathcal{R}_{t,j} = -1) = 2 \Pr(\mathcal{R}_{t,j} = +1) - 1, \quad (11.23)$$

that varies monotonically: $\beta_1 \geq \beta_2 \geq \dots \geq \beta_D$, with early positions having strong positive bias and late positions having strong negative bias.

The expected total loss is therefore:

$$\mathbb{E}_r[\mathcal{L}_{\text{therm}}] = -\sum_{j=1}^D \beta_j \cdot \mathcal{A}_{a,j} \cdot \mathcal{X}_{t,j} = -\langle \beta \odot \mathcal{X}_t, \mathcal{A}_a \rangle, \quad (11.24)$$

where $\beta = (\beta_1, \dots, \beta_D)^T$ is the vector of position-dependent biases.

The expected loss is a *weighted inner product*, where the weights β_j modulate the importance of each component. This is structurally similar to the simple scalar case $\mathcal{L} = -r_t \langle \mathcal{A}_a, \mathcal{X}_t \rangle$, but with the scalar r_t replaced by a component-specific weighting.

11.2.2 Per-round loss definition

The preceding analysis showed that the baseline update rule already implicitly performs a form of stochastic gradient descent, but on a linear, unregularized loss that updates unconditionally. We now replace this implicit objective with an explicit, well-defined loss function that addresses each of these shortcomings.

A key structural property of contextual bandit problems that distinguishes them from full reinforcement learning is that the environment provides *immediate, per-round feedback* for the selected action. At every round t , after the agent selects action a_t given context x_t , it observes the reward $r_{t,a_t} \in [0, 1]$ *before* the next context is presented. No delayed consequences, no state transitions, and no credit assignment across time steps are involved: each round is a self-contained example consisting of an input (the encoded context x_t), a model prediction ($\hat{r}_{t,a_t}$), and a ground-truth label (r_{t,a_t}).

This means that the reward signal in contextual bandits can be treated as a *regression target* for the chosen action. As a result, the classical tools of supervised learning can be directly applied to the learning problem, provided that the model produces a differentiable prediction.

In the HD-CB framework, the model does produce such a prediction. Recall that in the baseline HD-CB formulation (Chapter 7) the estimated quality of action a in context x_t is computed as the cosine similarity between the encoded context $\mathcal{X}_t$ and the action representative $\mathcal{A}_a$:

$$\cos(\mathcal{X}_t, \mathcal{A}_a) = \frac{\mathcal{X}_t \cdot \mathcal{A}_a}{\|\mathcal{X}_t\| \|\mathcal{A}_a\|}, \quad \text{defined for } \mathcal{X}_t \neq \mathbf{0}, \mathcal{A}_a \neq \mathbf{0}. \quad (11.25)$$

Since cosine similarity lies in $[-1, 1]$ while rewards lie in $[0, 1]$, we can interpret similarity as an estimated reward by rescaling it:

$$\hat{r}_{t,a} = \frac{\cos(\mathcal{X}_t, \mathcal{A}_a) + 1}{2} = \frac{1}{2} \left(\frac{\mathcal{X}_t \cdot \mathcal{A}_a}{\|\mathcal{X}_t\| \|\mathcal{A}_a\|} + 1 \right). \quad (11.26)$$

This mapping preserves the original HD-CB semantics (actions whose representatives are aligned with the current context receive higher predicted reward) while ensuring that $\hat{r}_{t,a} \in [0, 1]$, making it comparable to the observed reward.

With both a prediction and a target available at every round, we can now define a standard regression loss. Specifically, we define a *per-round squared-error loss* for the chosen action:

$$\mathcal{L}_t = \frac{1}{2} (\hat{r}_{t,a_t} - r_{t,a_t})^2, \quad (11.27)$$

As for the standard bandit setting, the loss in Eq. (11.27) is defined *only for the selected action* a_t and the rewards for the remaining $N - 1$ unchosen actions are *not observed*.

The loss $\mathcal{L}_t$ depends on $\mathcal{A}_{a_t}$ through the differentiable function $\hat{r}_{t,a_t}$ (Eq. (11.26)). This makes it possible to compute the gradient $\partial \mathcal{L}_t / \partial \mathcal{A}_{a_t}$ and apply stochastic gradient descent to update the representative.

11.2.3 Gradient of the loss with respect to the representative

To achieve a loss-based HD-CB variant, we seek to update the representative of the chosen action, $\mathcal{A}_{a_t}$, so as to minimize $\mathcal{L}_t$ over time. We apply stochastic gradient descent, which requires computing the gradient of the loss with respect to the model parameters $\mathcal{A}_{a_t}$. Since $\mathcal{L}_t$ depends on $\mathcal{A}_{a_t}$ only through $\hat{r}_{t,a_t}$, we apply the chain rule:

$$\frac{\partial \mathcal{L}_t}{\partial \mathcal{A}_{a_t}} = \frac{\partial \mathcal{L}_t}{\partial \hat{r}_{t,a_t}} \cdot \frac{\partial \hat{r}_{t,a_t}}{\partial \mathcal{A}_{a_t}}. \quad (11.28)$$

- The first term is the derivative of the squared error, which represents the prediction error, and is immediate:

$$\frac{\partial \mathcal{L}_t}{\partial \hat{r}_{t,a_t}} = \hat{r}_{t,a_t} - r_{t,a_t}.$$

- The second term involves differentiating the normalized dot product.

$$\frac{\partial \hat{r}_{t,a_t}}{\partial \mathcal{A}_{a_t}} = \frac{1}{2} \frac{\partial}{\partial \mathcal{A}_{a_t}} \left(\frac{\mathcal{X}_t \cdot \mathcal{A}_{a_t}}{\|\mathcal{X}_t\| \|\mathcal{A}_{a_t}\|} \right).$$

Since $\|\mathcal{X}_t\|$ is constant with respect to $\mathcal{A}_{a_t}$, the remaining derivative is the gradient of cosine similarity with respect to its second argument. For simplicity, let's define $u = \mathcal{X}_t$ and $v = \mathcal{A}_{a_t}$.

$$c(u, v) = \frac{u \cdot v}{\|u\| \|v\|}.$$

Since $\|u\|$ is constant w.r.t. v ,

$$\frac{\partial}{\partial v} (c(u, v)) = \frac{\partial}{\partial v} \left(\frac{u \cdot v}{\|u\| \|v\|} \right) = \frac{1}{\|u\|} \frac{\partial}{\partial v} \left(\frac{u \cdot v}{\|v\|} \right)$$

From the quotient rule we remember that if $f = g/h$ then $f' = \frac{hg' - gh'}{h^2}$. Therefore:

$$= \frac{1}{\|u\|} \frac{\partial}{\partial v} \left(\frac{u \cdot v}{\|v\|} \right) = \frac{1}{\|u\|} \frac{\|v\| \nabla(u \cdot v) - (u \cdot v) \nabla(\|v\|)}{\|v\|^2}$$

Where:

$$\begin{aligned} - \nabla(u \cdot v) &= u \\ - \nabla(\|v\|) &= \nabla(\sqrt{v \cdot v}) = 1/2 (v \cdot v)^{-1/2} \nabla(v \cdot v) = v / \sqrt{(v \cdot v)} = v / \|v\| \end{aligned}$$

Therefore,

$$\frac{1}{\|u\|} \frac{\partial}{\partial v} \left(\frac{u \cdot v}{\|v\|} \right) = \frac{1}{\|u\|} \frac{\|v\| u - (u \cdot v) \frac{v}{\|v\|}}{\|v\|^2} = \frac{1}{\|v\| \|u\|} \left(u - \frac{(u \cdot v)}{\|v\|^2} v \right).$$

Substituting back $u = \mathcal{X}_t$, $v = \mathcal{A}_{a_t}$:

$$\begin{aligned} \frac{\partial \hat{r}_{t,a_t}}{\partial \mathcal{A}_{a_t}} &= \frac{1}{2} \cdot \frac{1}{\|\mathcal{X}_t\| \|\mathcal{A}_{a_t}\|} \left(\mathcal{X}_t - \frac{\mathcal{X}_t \mathcal{A}_{a_t}}{\|\mathcal{A}_{a_t}\|^2} \mathcal{A}_{a_t} \right). \\ \frac{\partial \hat{r}_{t,a_t}}{\partial \mathcal{A}_{a_t}} &= \frac{1}{2} \cdot \frac{1}{\|\mathcal{X}_t\| \|\mathcal{A}_{a_t}\|} \left(\mathcal{X}_t - \frac{\mathcal{X}_t \mathcal{A}_{a_t}}{\|\mathcal{A}_{a_t}\|^2} \mathcal{A}_{a_t} \right). \end{aligned}$$

Combining both terms yields the gradient of the per-round loss with respect to the chosen representative:

$$\boxed{\frac{\partial \mathcal{L}_t}{\partial \mathcal{A}_{a_t}} = (\hat{r}_{t,a_t} - r_{t,a_t}) \cdot \frac{1}{2} \cdot \frac{1}{\|\mathcal{X}_t\| \|\mathcal{A}_{a_t}\|} \left(\mathcal{X}_t - \frac{\mathcal{X}_t \mathcal{A}_{a_t}}{\|\mathcal{A}_{a_t}\|^2} \mathcal{A}_{a_t} \right)} \quad (11.29)$$

Equation (11.29) provides a principled direction in which to update $\mathcal{A}_{a_t}$: when the predicted reward $\hat{r}_{t,a_t}$ exceeds the observed reward, the gradient pushes the representative away from the current context direction (reducing similarity); when the predicted reward is too low, it pushes the representative towards the context direction (increasing similarity). This mechanism is qualitatively similar to the baseline HD-CB, but unlike the heuristic update rules used in that formulation, this behavior follows directly from minimizing the explicit loss in Eq. (11.27).

The full gradient-descent step for the selected action can be written as:

$$\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} - \eta_t \left((\hat{r}_{t,a_t} - r_{t,a_t}) \cdot \frac{1}{2} \cdot \frac{1}{\|\mathcal{X}_t\| \|\mathcal{A}_{a_t}\|} \left(\mathcal{X}_t - \frac{\mathcal{X}_t \mathcal{A}_{a_t}}{\|\mathcal{A}_{a_t}\|^2} \mathcal{A}_{a_t} \right) \right), \quad (11.30)$$

where $\eta_t > 0$ is a learning rate (possibly time-dependent).

11.2.4 Simplifying via Sigmoid and Regularization

While the gradient derivation in Eq. (11.29) is theoretically exact for optimizing cosine similarity, it introduces significant computational complexity. Specifically, computing the term $\frac{1}{\|\mathcal{A}_{a_t}\|^2} \mathcal{A}_{a_t}$ requires divisions and norm calculations at every update step. In resource-constrained environments, implementing these division-heavy updates can be costly and time-consuming, as observed in Chapter 2. To

avoid this computational burden, we observe that the behavior of the exact cosine gradient can be effectively approximated by replacing the cosine similarity estimator with a sigmoid function on the dot product combined with L_2 regularization.

The exact cosine gradient (Eq. (11.29)) consists of two forces:

1. A force aligning $\mathcal{A}_{a_t}$ with the context $\mathcal{X}_t$ (the $\mathcal{X}_t$ term).
2. A "rejection" force that subtracts the component of $\mathcal{A}_{a_t}$ parallel to $\mathcal{X}_t$ (the $-\frac{\mathcal{X}_t \cdot \mathcal{A}_{a_t}}{\|\mathcal{X}_t\|^2} \mathcal{A}_{a_t}$ term).

The second term effectively prevents the norm $\|\mathcal{A}_{a_t}\|$ from growing indefinitely, ensuring the vector only rotates. The core idea of this subsection is that we can achieve a similar effect by allowing the dot product to drive the alignment while using an explicit regularization term to constrain the norm.

Derivation Let us redefine the estimated reward $\hat{r}_{t,a}$ using the logistic sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ applied to the raw (unnormalized) dot product:

$$\hat{r}_{t,a} = \sigma(\mathcal{X}_t \cdot \mathcal{A}_a). \quad (11.31)$$

To prevent the magnitude of $\mathcal{A}_a$ from growing unbounded, we add an L_2 regularization term to the loss:

$$\mathcal{L}_t^{\text{sig}} = \frac{1}{2}(\hat{r}_{t,a_t} - r_{t,a_t})^2 + \frac{\lambda}{2}\|\mathcal{A}_{a_t}\|^2, \quad (11.32)$$

with

$$\hat{r}_{t,a_t} = \sigma(z_t), \quad z_t = \mathcal{X}_t \cdot \mathcal{A}_{a_t}. \quad (11.33)$$

and λ is the regularization parameter. To derive the update rule, we compute the gradient of $\mathcal{L}_t^{\text{sig}}$ with respect to $\mathcal{A}_{a_t}$. By the chain rule:

$$\frac{\partial \mathcal{L}_t^{\text{sig}}}{\partial \mathcal{A}_{a_t}} = \frac{\partial \mathcal{L}_t^{\text{sig}}}{\partial \hat{r}_{t,a_t}} \cdot \frac{\partial \hat{r}_{t,a_t}}{\partial z_t} \cdot \frac{\partial z_t}{\partial \mathcal{A}_{a_t}} + \lambda \mathcal{A}_{a_t}. \quad (11.34)$$

Each term is straightforward:

- The derivative of the squared error:

$$\frac{\partial \mathcal{L}_t^{\text{sig}}}{\partial \hat{r}_{t,a_t}} = \hat{r}_{t,a_t} - r_{t,a_t}.$$

- The derivative of the sigmoid:

$$\frac{\partial \hat{r}_{t,a_t}}{\partial z_t} = \sigma(z_t)(1 - \sigma(z_t)) = \hat{r}_{t,a_t}(1 - \hat{r}_{t,a_t}).$$

- The derivative of the dot product:

$$\frac{\partial z_t}{\partial \mathcal{A}_{a_t}} = \mathcal{X}_t.$$

Substituting into Eq. (11.34), we obtain:

$$\boxed{\frac{\partial \mathcal{L}_t^{\text{sig}}}{\partial \mathcal{A}_{a_t}} = (\hat{r}_{t,a_t} - r_{t,a_t}) \hat{r}_{t,a_t} (1 - \hat{r}_{t,a_t}) \mathcal{X}_t + \lambda \mathcal{A}_{a_t}} \quad (11.35)$$

Resulting update rule The full gradient descent step can be written as:

$$\boxed{\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} - \eta_t \left[(\hat{r}_{t,a_t} - r_{t,a_t}) \hat{r}_{t,a_t} (1 - \hat{r}_{t,a_t}) \mathcal{X}_t + \lambda \mathcal{A}_{a_t} \right]} \quad (11.36)$$

For convenience, and compactness we can define the scalar

$$g_t = (\hat{r}_{t,a_t} - r_{t,a_t}) \hat{r}_{t,a_t} (1 - \hat{r}_{t,a_t}), \quad \hat{r}_{t,a_t} = \sigma(\mathcal{A}_{a_t}^\top \mathcal{X}_t). \quad (11.37)$$

and write the the sigmoid-based loss update in Eq. (11.36) in compact form:

$$\boxed{\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} - \eta_t \left(\underbrace{g_t \mathcal{X}_t}_{\text{update direction}} + \underbrace{\lambda \mathcal{A}_{a_t}}_{\text{norm control}} \right)} \quad (11.38)$$

This update preserves the same qualitative behavior as the cosine-based formulation: if $\hat{r}_{t,a_t} < r_{t,a_t}$, the representative $\mathcal{A}_{a_t}$ is pulled towards $\mathcal{X}_t$; if $\hat{r}_{t,a_t} > r_{t,a_t}$, it is pushed away. However, unlike the cosine gradient in Eq. (11.29), the sigmoid-based update avoids explicit normalisation and division operations.

The sigmoid-based formulation can be interpreted as a computationally efficient approximation of cosine-based learning. The sigmoid derivative $\hat{r}(1 - \hat{r})$ naturally dampens updates when the dot product becomes large in magnitude, while the L_2 regularization term explicitly counteracts norm growth. Together, these two effects emulate the role of the rejection term in the cosine gradient, making the sigmoid-based loss particularly attractive for embedded and low-power implementations.

Algorithm 12 depicts the complete Loss-HD-CB model with sigmoid-based updates.

11.2.5 Experimental Evaluation

To validate the loss-based formulation, we conduct three complementary analyses. We first examine the loss dynamics over time, providing direct evidence that the update rule is minimizing the squared-error objective (Section 11.2.5). We then present per-configuration off-policy evaluation results comparing loss-based updates (in both cosine and sigmoid formulations) with heuristic updates (Section 11.2.5).

Loss dynamics over time

Before examining the average reward, it is important to verify that the per-round prediction error decreases systematically over time rather than merely fluctuating. To this end, we examine the per-round prediction loss $\mathcal{L}_t = \frac{1}{2}(\hat{r}_{t,a_t} - r_{t,a_t})^2$ over 1000 rounds on a synthetic CB dataset ($N = 10$, $d = 5$, $D = 1000$).

Figure 11.3 reports the per-round loss for the Loss-HD-CB_{EPS} model with $\epsilon = 0.05$. The raw loss (grey) decreases rapidly during the initial learning phase within the first 200 rounds, and then stabilizes at a low residual level. The smoothed curve (moving average with window 50) clearly reveals the convergence trajectory: the loss decreases monotonically during the learning phase and stabilizes around a small residual value.

This convergence profile provides direct empirical evidence that the loss-based update rule is indeed minimizing the squared-error objective. The monotonic decrease

Algorithm 12 LOSS-HDC_{SIG} (Epsilon-Greedy, Sigmoid + Regularization)

```

1: Input:  $\epsilon \in [0, 1]$ , learning rate  $\eta > 0$ , regularization  $\lambda \geq 0$ 
2: Initialize: action counts  $n_a \leftarrow 0$  for all  $a \in \{1, \dots, N\}$ 
3: for  $t = 1, 2, \dots, T$  do
4:   Observe context  $x_t \in \mathbb{R}^d$ 
5:   Encode context into hypervector  $\mathcal{X}_t \leftarrow \text{Encode}(x_t) \in \mathbb{R}^D$ 
6:   for  $a = 1, 2, \dots, N$  do
7:     Compute score  $z_{t,a} \leftarrow \mathcal{A}_a^\top \mathcal{X}_t$ 
8:     Predicted reward  $\hat{r}_{t,a} \leftarrow \sigma(z_{t,a})$ 
9:     Sample  $u \sim \text{Uniform}(0, 1)$ 
10:    if  $u < \epsilon$  then ▷ exploration
11:      Select  $a_t$  uniformly at random from  $\{1, \dots, N\}$ 
12:    else ▷ exploitation
13:      Select  $a_t \leftarrow \arg \max_a \hat{r}_{t,a}$  ▷ break ties uniformly
14:    endif
15:  endif
16:  Execute  $a_t$  and observe reward  $r_{t,a_t} \in [0, 1]$ 
17:  Compute  $g_t \leftarrow (\hat{r}_{t,a_t} - r_{t,a_t}) \hat{r}_{t,a_t} (1 - \hat{r}_{t,a_t})$ 
18:  Update selected representative:
19:     $\mathcal{A}_{a_t} \leftarrow \mathcal{A}_{a_t} - \eta(g_t \mathcal{X}_t + \lambda \mathcal{A}_{a_t})$ 
20: end for

```

of the smoothed loss mirrors the behavior expected from a stochastic gradient descent procedure applied to a well-defined cost function. Moreover, the rapid initial decrease followed by stabilization suggests that the model quickly identifies the dominant reward structure and then enters a fine-tuning phase where only residual corrections are needed.

This behavior stands in sharp contrast to the heuristic update, where no analogous convergence analysis is possible because no explicit loss is being minimized. While the heuristic HD-CB also learns effective representations empirically, the absence of a loss function makes it impossible to monitor or guarantee that the learning dynamics are improving the prediction quality over time. The loss-based formulation closes this gap: for the first time in the HD-CB family, the learning process can be tracked through a well-defined, monotonically decreasing cost, bringing it in line with the optimization-based perspective that underpins classical linear bandit algorithms.

Off-policy evaluation

Having established that the loss-based update converges, we now evaluate its impact on decision quality through a systematic off-policy evaluation.

Table 11.2 reports the average reward obtained on synthetic contextual bandit datasets using the OBP framework, while varying the number of actions N and the context dimension d . Four models are compared: the linear ϵ -greedy baseline (LinEPS), the standard real-valued HD-CB with heuristic updates (HD-CB_{EPS}), and two loss-based variants using the cosine gradient (Loss-Cos) and the sigmoid approximation (Loss-Sig), respectively.

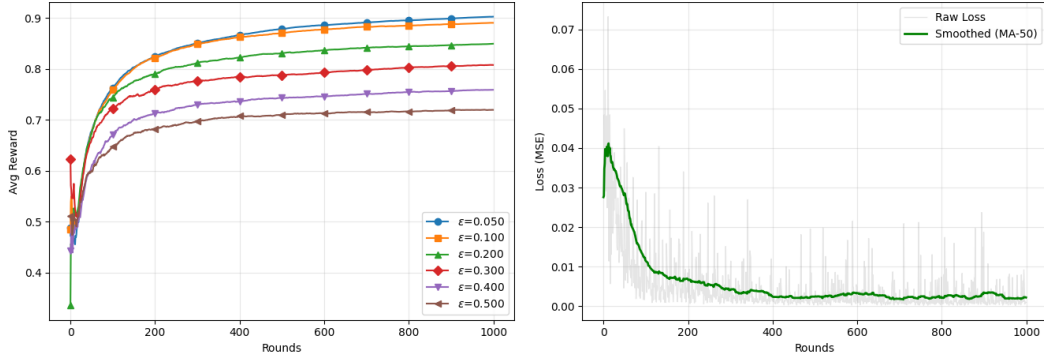


Figure 11.3 Per-round prediction loss of Loss-HD-CB_{EPS} on a synthetic contextual bandit dataset ($N = 10$, $d = 5$, $D = 1000$). The raw loss (grey) and its moving average (green, window 50) show clear convergence from approximately 0.04 to a residual level below 0.005, confirming that the update rule effectively minimizes the squared-error objective.

Several important observations emerge from this comparison. First, Loss-Cos and Loss-Sig achieve the highest average reward in every tested (N, d) configuration, without exception. The improvements over the heuristic HD-CB_{EPS} range from approximately 1.3% (at $N = 20$, $d = 5$, where HD-CB_{EPS} already performs well at 0.733) to a striking 4.6% (at $N = 10$, $d = 15$, where the loss variant reaches 0.705 compared to 0.659), confirming the structural benefit arising from the error-corrective nature of gradient-based updates.

Second, a particularly revealing pattern emerges at larger context dimensions. At $d = 15$, the heuristic HD-CB_{EPS} *underperforms* the LinEPS baseline in the majority of configurations: at $(10, 15)$, HD-CB achieves 0.659 versus LinEPS at 0.663; at $(15, 15)$, the gap widens to 0.664 versus 0.678; at $(20, 15)$, HD-CB obtains 0.681 versus 0.690. This suggests that in more complex environments, the heuristic accumulation becomes noisier, as the fixed-magnitude update does not account for the model’s current prediction accuracy and progressively corrupts the representations. The loss-based variants completely reverse this trend. At the same $(N, 15)$ configurations, Loss-Cos achieves 0.705, 0.703, and 0.713, respectively, consistently outperforming *both* the heuristic HD-CB and the linear baseline. The absolute improvement of Loss-based models over HD-CB_{EPS} tends to increase with the context dimension d . At $d = 5$, the gains are moderate (1.3–1.7%), reflecting that low-dimensional problems are already well-handled by the heuristic accumulation. At $d = 10$, the gains rise to 2.2–3.9%. At $d = 15$, where the heuristic update struggles most, the gains reach 3.2–4.6%.

Crucially, the enhanced accuracy does not come at the expense of the lightweight per-round structure of HD-CB. In fact, the sigmoid update is *computationally lighter* than the heuristic rule. In the standard HD-CB, the observed reward must first be encoded into a full D -dimensional bipolar hypervector $\mathcal{R}_{t,a_t}$ via thermometer code, then bound element-wise with the context $\mathcal{X}_t$, and finally bundled into $\mathcal{A}_{a_t}$. The sigmoid update (Eq. (11.38)) bypasses the reward encoding entirely: the scalar gradient factor g_t (Eq. (11.37)) is computed from the observed reward and the predicted reward using scalar operations, and the representative is updated via a

Table 11.2 Off-policy evaluation results comparing LinEPS, HD-CB_{REAL}, and the estimated Loss variants. The best performing method in each row is highlighted in bold.

N	d	LinEPS	HD-CB _{EPS}	Loss Variants	
				Loss-Cos	Loss-Sig
10	5	0.614	0.656	0.670	0.668
10	10	0.683	0.684	0.723	0.719
10	15	0.663	0.659	0.705	0.701
15	5	0.711	0.735	0.752	0.751
15	10	0.703	0.708	0.730	0.729
15	15	0.678	0.664	0.703	0.703
20	5	0.716	0.733	0.746	0.742
20	10	0.708	0.724	0.746	0.742
20	15	0.690	0.681	0.713	0.714

single scalar-vector multiplication $g_t \mathcal{X}_t$ followed by a vector addition.

As a result, Loss-HD-CB_{EPS} is not only more accurate but also *more efficient per round* than the heuristic baseline, retaining full hardware friendliness while providing substantially improved and more stable learning.

The results in Table 11.2 also allow a direct comparison between the two loss formulations. Loss-Cos achieves marginally higher accuracy than Loss-Sig in most configurations, with differences never exceeding 0.4%. Given this near-equivalence, and considering that the sigmoid update avoids the division-heavy norm calculations required by the exact cosine gradient, operations that can be an order of magnitude slower on embedded platforms as demonstrated in Chapter 2, the sigmoid-based loss represents the *recommended default* for hardware-oriented deployments.

11.2.6 Loss-Probabilistic HD-CB

The previous sections presented two complementary extensions of HD-CB that address the distinct limitations of the heuristic update: the loss-based formulation (Sections 11.2.2–11.2.4) replaced these updates with gradient-derived corrections, while the probabilistic formulation (Section 11.1) replaced unbounded accumulation with bounded stochastic integer updates. A natural question arises: can these two ideas be composed to yield a variant that simultaneously benefits from principled error-driven learning *and* low-precision bounded representations?

This section answers by presenting *Loss-Probabilistic HD-CB* (HD-CB_{LOSS-PROB}), a variant that uses the magnitude of the prediction error as the update probability, combining the error sensitivity of the loss-based approach with the memory efficiency of probabilistic updates.

In the standard probabilistic HD-CB, the update probability α_t depends solely on time: it decreases according to a predetermined schedule regardless of the model’s current prediction quality. This means the schedule is *agnostic to the actual prediction error*: even if the model makes a grossly incorrect prediction at a late time step, the update probability may be too low to correct it. Conversely, early in

training, when some predictions are already close to the observed reward, the high α_t still triggers unnecessary updates that inject noise into the representations. The loss-based formulation suggests a more principled mechanism: the prediction error itself should modulate the update intensity.

The core idea is to modify the update rule as follows. Upon observing reward r_{t,a_t} , the model computes the prediction error $G_t = r_{t,a_t} - \hat{r}_{t,a_t}$. The update probability is then set as:

$$P_t = \alpha_t \cdot |G_t|, \quad (11.39)$$

where $\alpha_t = \alpha_0 \cdot (1 - (t-1)/T_{\max})$ is the time-decaying base probability. For each component $i \in \{1, \dots, D\}$, a random value $m_{t,i} \sim U(0, 1)$ is sampled, and the component is updated only if $m_{t,i} \leq P_t$:

$$\mathcal{A}_{a_t,i} = \begin{cases} f_\kappa(\mathcal{A}_{a_t,i} + \text{sign}(G_t) \cdot \mathcal{X}_{t,i}), & \text{if } m_{t,i} \leq P_t, \\ \mathcal{A}_{a_t,i}, & \text{otherwise,} \end{cases} \quad (11.40)$$

where $f_\kappa(\cdot) = \max(-\kappa, \min(\cdot, +\kappa))$ is the saturation function bounding each component to $[-\kappa, +\kappa]$.

In this way, the update probability P_t now depends on $|G_t|$, the absolute prediction error. This introduces an *error-driven modulation*: when the model predicts well ($|G_t| \approx 0$), the effective update probability drops toward zero regardless of the time schedule, conserving the current representation. When the prediction is poor ($|G_t| \approx 1$), the probability approaches α_t , triggering a correction. The update direction is determined by $\text{sign}(G_t) \cdot \mathcal{X}_{t,i}$: when the observed reward exceeds the prediction ($G_t > 0$), the representative is pulled toward the context; when the prediction overshoots ($G_t < 0$), it is pushed away.

However, a non-trivial design issue arises in action selection. In the full-precision Loss-HD-CB (Section 11.2.4), the sigmoid estimator $\hat{r}_{t,a} = \sigma(\mathcal{A}_a^\top \mathcal{X}_t)$ is kept in the active region by the L_2 regularization term $\lambda \|\mathcal{A}_a\|^2$, which penalizes norm growth and maintains the dot products within a range where the sigmoid gradient $\hat{r}_{t,a_t}(1 - \hat{r}_{t,a_t})$ remains appreciable. In the probabilistic setting, the action representatives are bounded by design to $\mathcal{A}_a \in [-\kappa, +\kappa]^D$ through saturation, but the dot product $\mathcal{A}_a^\top \mathcal{X}_t$ can reach magnitudes up to $D\kappa$, which for typical values ($D = 1000$, $\kappa = 3$) exceeds 3000. At such magnitudes, the sigmoid is deeply saturated, causing the gradient factor $\hat{r}(1 - \hat{r})$ to *vanish* and halting learning even when the prediction error is large.

Instead of introducing a temperature-scaled sigmoid, which would add a hyperparameter dependent on both D and κ and still require evaluating an exponential on hardware—we adopt a simpler approach: a *normalized clipped ReLU*. We define a normalization constant $\nu = D\kappa/2$, corresponding to the expected magnitude of the dot product when the representative components are uniformly distributed in $[-\kappa, +\kappa]$, and compute the predicted reward as:

$$\hat{r}_{t,a} = \min \left(1, \max \left(0, \frac{\mathcal{A}_a^\top \mathcal{X}_t}{\nu} \right) \right). \quad (11.41)$$

Within the active region $0 < \hat{r}_{t,a} < 1$, the estimator is *linear* in the dot product, so the prediction error $G_t = r_{t,a_t} - \hat{r}_{t,a_t}$ is always well-defined and never vanishes due to gradient saturation.

This estimator is an ideal candidate for hardware. The constant ν is computed once at initialization. For power-of-two choices of D and κ , the division reduces to an arithmetic shift. The remaining operations are standard integer primitives, making the entire prediction pipeline compatible with the low-precision arithmetic of the saturating-integer action representatives.

The Loss-Probabilistic update can be understood as a *stochastic, quantized approximation* of the sigmoid-based gradient update (Eq. (11.38)). In the sigmoid-based loss, the gradient scalar modulates the continuous-valued update *magnitude*; in HD-CB_{LOSS-PROB}, the error modulates the discrete update *frequency* instead, while the per-component update magnitude remains fixed at ± 1 . This frequency-based modulation is the mechanism that enables the update rule to operate entirely within low-precision saturating integer arithmetic while retaining the error-sensitivity of gradient-based learning.

11.2.7 Comparison: Performance versus Representation Precision

To provide a complete and unified perspective, we close this section presenting a comprehensive of all HD-CB models. In particular, since HD-CB_{LOSS-PROB} was introduced algorithmically in Section 11.2.6 without a dedicated evaluation, its experimental validation is presented here alongside all other variants.

All models were evaluated using the OBP framework [76]. Each reported average reward is obtained by averaging across all (N, d) configurations (with $N \in \{10, 15, 20\}$ and $d \in \{5, 10, 15\}$) and 50 independent seeds per configuration, with $D = 1000$. Table 11.3 reports the per-configuration results, while Figure 11.4 plots the configuration-averaged reward against the number of bits per element, providing a panoramic view of the reward-precision trade-off across the entire HD-CB family.

Table 11.3 reveals that HD-CB_{LOSS-PROB} consistently outperforms the standard probabilistic variant HD-CB_{PROB} at every tested precision level and in every (N, d) configuration. The improvement is systematic and substantial. At 3-bit precision ($\kappa = 3$), Loss-Prob achieves the largest and most consistent gains, outperforming standard Prob by 1.4-2.9% across all configurations. In several cases, the 3-bit Loss-Prob variant matches or exceeds even the performance of the real-valued (32 bits) heuristic HD-CB_{EPS}: at (15, 5), Loss-Prob at 3 bits reaches 0.745 compared to HD-CB_{EPS} at 0.735; at (20, 5), it achieves 0.737 versus 0.733. This is a remarkable result: a 3-bit integer model surpassing its 32-bit heuristic counterpart, enabled purely by the error-driven update mechanism.

At 4-bit precision ($\kappa = 7$), Loss-Prob provides dramatic improvements in the higher-dimensional configurations where the standard Prob variant struggles most. At (10, 10), Loss-Prob reaches 0.692 versus Prob at 0.646 (+4.61%); at (20, 10), it achieves 0.713 versus 0.653 (+6.0%); at (20, 15), 0.672 versus 0.612 (+6.0%). An interesting pattern emerges: while standard Prob at $\kappa = 7$ often *underperforms* standard Prob at $\kappa = 3$ —suggesting that the additional dynamic range is wasted or even harmful—Loss-Prob at $\kappa = 7$ consistently outperforms Loss-Prob at $\kappa = 3$ at higher dimensions. The error-driven modulation effectively unlocks the representational capacity of the wider integer range, which the heuristic schedule fails to exploit.

Figure 11.4 synthesises the comparison across all twelve variants. Overall, the

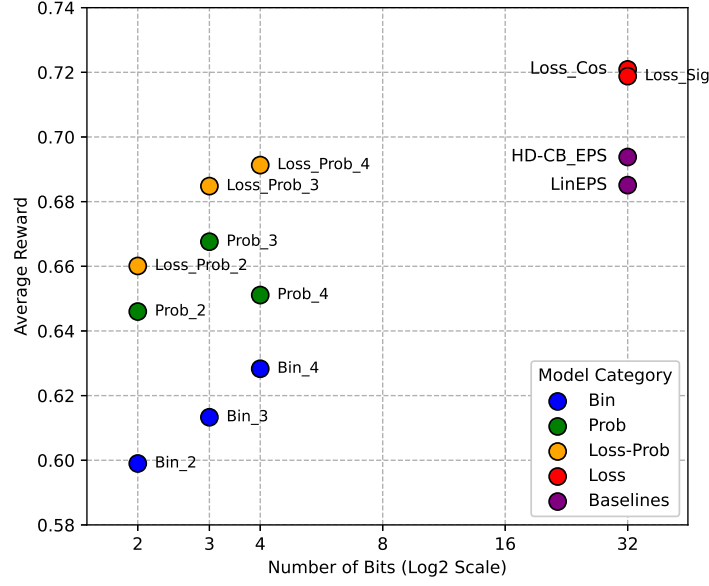


Figure 11.4 Comprehensive comparison of all HD-CB variants: average reward (OPE, averaged across all (N, d) configurations) versus bits per element ($\log_2$ scale).

real-valued Loss-HD-CB_{EPS} (0.7151) outperforms the heuristic HD-CB_{EPS} (0.6911) by 2.4%. At low precision, HD-CB_{LOSS-PROB} outperforms HD-CB_{PROB} at every value of κ : by 1.4% at $\kappa = 1$, by 1.7% at $\kappa = 3$, and by 4.0% at $\kappa = 7$.

HD-CB_{LOSS-PROB} with $\kappa = 7$ (4 bits) achieves a configuration-averaged reward of 0.6876, exceeding the LinEPS baseline (0.6858) despite using $8\times$ less memory per element. Even at $\kappa = 3$ (3 bits), it reaches 0.6811, within 0.5%pp of the 32-bit linear baseline: at (15, 5), Loss-Prob at 3 bits (0.745) exceeds both LinEPS (0.711) and HD-CB_{EPS} (0.735).

Connecting the Loss-HD-CB variants across precision levels traces a clear Pareto frontier: Loss-Prob $\kappa = 1$ (2 bits, 0.6563) $\rightarrow$ Loss-Prob $\kappa = 3$ (3 bits, 0.6811) $\rightarrow$ Loss-Prob $\kappa = 7$ (4 bits, 0.6876) $\rightarrow$ Loss-HD-CB real (32 bits, 0.7151). No other variant family lies above this frontier, establishing the loss-based family as the uniformly best choice for any given bit budget.

Notably, the reward curve flattens between 4 bits (0.6876) and 32 bits (0.7151): an $8\times$ increase in precision yields only 2.8% of improvement. This suggests that 3–4 bi representations offer an excellent accuracy-efficiency sweet spot for edge deployment.

Figure 11.4 serves as a practical design chart. Given a target memory budget, the designer can read off the maximum κ supported and estimate the expected reward. For example, with $D = 1000$ and $N = 20$ actions, a 4-bit representation requires $20 \times 1000 \times 4/8 = 10$ KB for action representatives; reducing to 3 bits brings this to ≈ 7.5 KB with only 0.65% reward loss.

Table 11.3 Off-policy evaluation results comparing LinEPS, HD-CB_{EPS}, and all proposed variants. Columns report average reward for each (N, d) configuration. The best performing method in each row is highlighted in bold.

N	d	LinEPS	HD-CB _{EPS}	Loss Vars		HD-CB _{BIN} (bits)			HD-CB _{PROB} (bits)			HD-CB _{Loss-Prob} (bits)		
				Cos	Sig	2	3	4	2	3	4	2	3	4
10	5	0.614	0.656	0.670	0.668	0.578	0.594	0.604	0.630	0.647	0.641	0.641	0.661	0.655
10	10	0.683	0.684	0.723	0.719	0.592	0.610	0.625	0.643	0.664	0.646	0.652	0.681	0.692
10	15	0.663	0.659	0.705	0.701	0.570	0.579	0.589	0.604	0.632	0.619	0.616	0.646	0.658
15	5	0.711	0.735	0.752	0.751	0.656	0.674	0.693	0.718	0.729	0.715	0.737	0.745	0.740
15	10	0.703	0.708	0.730	0.729	0.594	0.610	0.631	0.638	0.662	0.639	0.652	0.681	0.692
15	15	0.678	0.664	0.703	0.703	0.571	0.579	0.588	0.607	0.635	0.614	0.624	0.654	0.669
20	5	0.716	0.733	0.746	0.742	0.645	0.661	0.680	0.703	0.731	0.721	0.715	0.737	0.731
20	10	0.708	0.724	0.746	0.742	0.610	0.630	0.652	0.657	0.671	0.653	0.673	0.700	0.713
20	15	0.690	0.681	0.713	0.714	0.575	0.583	0.593	0.614	0.637	0.612	0.631	0.658	0.672

11.3 A Neural Network Interpretation of HD-CB

This section closes the second part of this dissertation by showing that the entire HD-CB pipeline—from context encoding to action selection and update—can be formally interpreted as a shallow, single-hidden-layer feedforward neural network whose architecture closely parallels that of *Random Vector Functional Link* (RVFL) networks [312, 313] and *Extreme Learning Machines* (ELMs) [314, 315].

This interpretation connects HD-CB with the broader landscape of neural network theory established over three decades of research.

11.3.1 Background: RVFL networks and extreme learning machines

RVFL networks, introduced by Igel and Pao [312, 313], and Extreme Learning Machines (ELMs), popularized by Huang et al. [314, 315], belong to a family of single-hidden-layer feedforward neural networks characterized by a distinctive training paradigm: the weights connecting the input layer to the hidden layer are randomly initialized and never updated during training, while only the output-layer weights are learned. This technique yields several compelling properties, including extremely fast training [312, 316].

Formally, an RVFL/ELM with n input features, D hidden neurons, and C output neurons computes the mapping $f: \mathbb{R}^n \rightarrow \mathbb{R}^C$ defined as:

$$f(\mathbf{x}) = \mathbf{W}_{\text{out}} \sigma(\mathbf{W}_{\text{in}} \mathbf{x} + \mathbf{b}), \quad (11.42)$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input, $\mathbf{W}_{\text{in}} \in \mathbb{R}^{D \times n}$ is the random, frozen input-to-hidden weight matrix, $\mathbf{b} \in \mathbb{R}^D$ is a random, frozen bias vector, $\sigma(\cdot)$ is an element-wise nonlinear activation, and $\mathbf{W}_{\text{out}} \in \mathbb{R}^{C \times D}$ is the *only trainable* weight matrix.

The essential architectural principle is a strict separation between a *frozen nonlinear feature expansion* $\mathbf{x} \mapsto \sigma(\mathbf{W}_{\text{in}} \mathbf{x} + \mathbf{b}) \in \mathbb{R}^D$ and a *trainable linear readout* $\mathbf{W}_{\text{out}}$ that operates in the expanded space.

In their most general form, RVFL networks may also include direct input-to-output (skip) connections in addition to the random hidden layer. In this thesis we consider the widely adopted randomized single-layer feedforward formulation without skip connections and show that HD-CB exhibits precisely this separation.

11.3.2 The HD-CB Encoding Layer as a Frozen Random Nonlinear Projection

The HD-CB algorithms can always be decomposed into three main stages. At each time step t , first the observed raw context vector $\mathbf{x}_t \in \mathbb{R}^d$ is incoded into the high-dimensional space $\mathcal{X}_t \in \{-1, +1\}^D$, where $D \gg d$. This encoding step, as highlighted in Chapter 7 can be performed using any encoding technique. Then, action selection is performed by computing the similarity between each action prototype and the encoded context. Finally, after obtaining the reward r_{t,a_t} , the selected action is updated to better reflect the quality of the selected option.

Notably, by collecting the per-action prototypes $\{\mathcal{A}_a\}_{a=1}^N$ as rows of the matrix $\mathbf{A} \in \mathbb{R}^{N \times D}$ introduced in Section 10.1, we can notice that every learning rule proposed in this thesis modifies only $\mathbf{A}$, while the encoding mapping remains fixed. This suggests that the encoding mapping can be seen as a fixed *nonlinear random projection* $\mathbb{R}^d \rightarrow \{-1, +1\}^D$ whose parameters are fixed at initialization and never updated: the defining property of the RVFL hidden layer.

The equivalence is immediate for the projection-based encoding described in Chapter 2:

$$\mathcal{X}_t = \text{sign}(\mathbf{\Phi} \mathbf{x}_t), \quad \mathbf{\Phi} \in \mathbb{R}^{D \times d}, \quad \Phi_{ij} \sim \mathcal{N}(0, 1). \quad (11.43)$$

This is exactly the hidden layer of a sign-activated RVFL network with $\mathbf{W}_{\text{in}} = \mathbf{\Phi}$, $\mathbf{b} = \mathbf{0}$, and $\sigma(\cdot) = \text{sign}(\cdot)$.

The same holds for the record-based encoding employed throughout this thesis, which replaces the unstructured matrix multiplication with a compositional construction exploiting the algebraic properties of HD/VSA. As detailed in Chapter 2, in this technique each continuous feature $x_{t,j}$ is first quantized into one of L levels, yielding a level hypervector $LV_j \in \{-1, +1\}^D$ retrieved from a pre-generated codebook in which adjacent levels produce similar hypervectors while distant levels are quasi-orthogonal. Each encoded level is then bound with a feature-specific *base hypervector* $BV_j \in \{-1, +1\}^D$, drawn i.i.d. at random during initialization and held fixed thereafter, via element-wise multiplication $p_j = BV_j \odot LV_j$. Finally, the d bound key-value pairs are superposed and binarized:

$$\mathcal{X}_t = \text{sign}\left(\sum_{j=1}^d BV_j \odot LV_j\right) \in \{-1, +1\}^D. \quad (11.44)$$

Although different from a single matrix-vector product, this pipeline produces the same functional outcome: a fixed, nonlinear, high-dimensional mapping whose parameters are never updated during training. The base hypervectors $\{BV_j\}_{j=1}^d$ are randomly generated and subsequently frozen, serving the exact same role as the random input weights $\mathbf{W}_{\text{in}}$ in an RVFL network. The level hypervectors encode a constant nonlinear mapping of each scalar feature; the binding operation creates cross-products between these encodings and the random basis vectors; and $\text{sign}(\cdot)$ acts as the nonlinear activation function $\sigma(\cdot)$, analogous to RVFL [314]. This architectural correspondence was first formalized by Kleyko et al. [317] that demonstrated the compositional encoding, despite being composed by two frozen sub-layers (a density-based representation layer $\mathbf{F} \in \{-1, +1\}^{D \times d}$ where each feature is independently mapped to its level hypervector via feature-dependent links, and a hidden layer

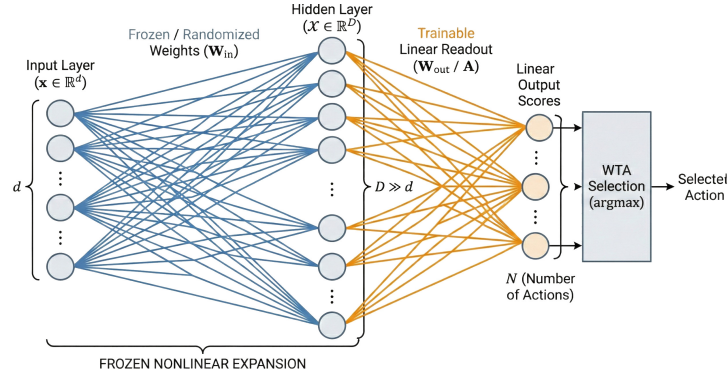


Figure 11.5 Graphical correspondence between RVFL and HD-CB

obtained by binding $\mathbf{F}$ with a random weight matrix and then bundling with sign binarization), fulfills the role of a single RVFL hidden layer, since all parameters before the output remain fixed at initialization.

Regardless of the specific encoding variant adopted—and, as discussed in Chapter 7, HD-CB is encoding-agnostic—the critical invariant is that the entire encoding stage is set once and never updated. No gradients flow through these components. This is the defining characteristic of the RVFL paradigm. This decomposition also formalizes the central finding of the ablation study in Chapter 6: the frozen nonlinear projection layer performs the heavy lifting by producing quasi-orthogonal representations that render the covariance matrix redundant, while the trainable output weights—the action prototypes—need only accumulate reward-weighted projections to achieve competitive decision quality.

11.3.3 The Action Prototypes as Trainable Output Weights

Once the context has been projected into the high-dimensional space, the second stage of HD-CB performs action selection and learning. As formalized in the matrix notation of Section 10.1, each action $a \in \{1, \dots, N\}$ is associated with a prototype $\mathcal{A}_a \in \mathbb{R}^D$, and the action scores are computed via

$$\mathbf{s}_t = \mathbf{A} \mathcal{X}_t \in \mathbb{R}^N, \quad (11.45)$$

where $\mathbf{A} = [\mathcal{A}_1, \dots, \mathcal{A}_N]^\top \in \mathbb{R}^{N \times D}$ is the prototype matrix. This is a linear readout—the standard output layer of an RVFL network, where the hidden-layer activations $\mathcal{X}_t$ are linearly combined by the weight matrix $\mathbf{A}$ to produce the N -dimensional output. In the cosine-normalised form, the readout additionally includes a per-row normalisation by $\|\mathcal{A}_a\|$, which does not alter the linear nature of the computation but ensures score comparability across actions.

The final action selection, $a_t = \arg \max_a (e_{t,a})$, can be viewed as a Winner-Take-All (WTA) output nonlinearity. Figure 11.5 formalizes the complete component-by-component correspondence.

Training Procedures

The architectural equivalence established above extends naturally to the training procedure. In the RVFL framework, only the output weights are learned; similarly, in HD-CB the encoding layer is fixed, and all learning occurs in the prototype matrix $\mathbf{A} \in \mathbb{R}^{N \times D}$.

The base HD-CB update, $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + (\mathcal{R}_t \odot \mathcal{X}_t)$, can be read as a reward-modulated Hebbian rule on the output weights. The loss-based formulation makes this connection even more explicit. The gradient flows exclusively through $\mathbf{A}_{a_t}$; the encoding layer remains frozen, preserving the RVFL architecture by construction. Loss-Based HD-CB is the standard online RVFL training procedure, adapted to the partial feedback of the bandit setting.

Connections and Perspectives Opened by the RVFL Interpretation

The structural equivalence between HD-CB and RVFL networks does not, by itself, guarantee that classical RVFL results transfer without modification to the bandit setting—differences in feedback (partial vs. full), loss landscape, and data distribution may require dedicated analysis. Nevertheless, the mapping opens a number of concrete connections that motivate future theoretical and empirical investigation. We outline the most promising below, framing each as a direction to be explored rather than a claim already established.

- **Universal approximation.** Single-hidden-layer networks with random hidden parameters are known to be universal approximators under mild conditions on the activation function and weight distribution [312, 318]. In particular, Igel and Pao [312] showed that the mean-square approximation error of an RVFL decays as $\mathcal{O}(C/\sqrt{D})$ with the number of hidden neurons D . Since HD-CB shares the same architecture, it is natural to conjecture that increasing the hypervector dimensionality D should progressively improve the model’s capacity to approximate complex reward functions $r(\mathbf{x}, a)$. Verifying whether the standard universality proofs extend to bipolar activations and bandit-style partial updates remains an open question.
- **Implicit kernel approximation.** Randomized hidden-layer networks admit an equivalent interpretation as kernel machines operating through random feature maps [246]. Under this view, the HD-CB encoding defines a nonlinear map $\phi : \mathbb{R}^d \rightarrow \{-1, +1\}^D$, and the linear readout computes action scores as inner products $s_a(\mathbf{x}) = \langle \mathbf{A}_a, \phi(\mathbf{x}) \rangle$. As $D \rightarrow \infty$, these inner products converge in expectation to a kernel function whose form depends on the encoding scheme [244, 246]. This suggests that HD-CB may be understood as performing linear learning in an implicitly defined reproducing kernel Hilbert space, without ever computing or storing a kernel matrix—a property particularly attractive for online, resource-constrained operation. Characterizing the specific kernel induced by the record-based HDC encoding, and understanding how it interacts with bandit exploration, is a promising avenue for future work.
- **Depth and expressiveness.** A natural concern is whether HD-CB, as a “shallow” network, suffers from the expressiveness limitations that motivate

costly deep neural networks in complex, non-linear, and high-dimensional environments. Whether the current single-layer encoding suffices for such settings, whether stacking multiple encoding layers could improve performance, and whether partially or fully training the encoder—rather than keeping it frozen as in the standard RVFL paradigm—could enhance representational capacity without sacrificing hardware-friendliness, are open questions that deserve systematic empirical investigation.

- **Connections to spiking neural networks and neuromorphic hardware.** The RVFL interpretation also suggests a conceptual bridge to spiking neural networks and neuromorphic deployment. Both HDC-style encodings and SNNs operate with binary or low-precision states and rely on inexpensive primitives that map naturally onto event-driven neuromorphic processors such as Intel Loihi [319, 320]. Under this perspective, the HD-CB encoder could be realized as a fixed spiking feature extractor, with the action prototypes corresponding to trainable synaptic weights and action selection implemented via a WTA circuit. Online learning would proceed through local, reward-modulated synaptic updates that adjust only the readout weights while keeping the encoder frozen—directly mirroring the RVFL training paradigm. Exploring this mapping toward ultra-low-power, real-time HD-CB agents on neuromorphic substrates is a natural direction for future work.

11.4 Summary of Findings

This chapter presented a series of extensions and optimizations that transform HD-CB from a brain-inspired heuristic into a principled learning architecture with clear connections to the well-established theory of randomized neural networks.

- **Probabilistic HD-CB.** To address the unbounded growth of standard accumulation-based updates, we introduced HD-CB_{PROB}, which replaces deterministic updates with probabilistic ones governed by a time-decaying probability α_t . Combined with saturating integer counters bounded to $[-\kappa, +\kappa]$, this approach enables deployment with as few as 3 bits per component while outperforming HD-CB_{BIN} by approximately 5.8% in average reward. Unlike periodic binarization, which indiscriminately discards accumulated information, the probabilistic rule preserves ranking information through frequency-based modulation.
- **Implicit loss of the baseline update rule.** By interpreting the baseline thermometer-encoded update as a step of stochastic gradient descent—mirroring the classical perceptron–hinge-loss connection—we derived the implicit loss function that the heuristic rule minimizes. The analysis revealed that each hypervector component acts as an independent weak learner operating at a distinct reward threshold. While this result proves that the heuristic is not entirely unprincipled, it also identifies its structural limitations, which directly motivated the explicit loss formulation.

- **Loss-based HD-CB.** We addressed the theoretical gap between HD-CB’s heuristic updates and the optimization-grounded formulations of classical bandits. By defining a per-round squared-error loss and deriving the gradient with respect to action representatives, we obtained an update rule that is explicitly grounded in first-order optimization. The sigmoid-based simplification avoids the division-heavy normalization of the exact cosine gradient while preserving the same qualitative behavior. Off-policy evaluation confirmed that Loss-HD-CB_{EPS} outperforms the standard heuristic variant in every tested configuration, with improvements of up to 3 percentage points in accuracy.
- **Loss-Probabilistic HD-CB.** Combining the loss-based and probabilistic formulations, we introduced HD-CB_{LOSS-PROB}, which uses the prediction error magnitude as the update probability, combining principled error-driven updates with the low-precision bounded representations of HD-CB_{PROB}.
- **Cross-family comparison and Pareto analysis.** The comprehensive comparison across all twelve variants (Table 11.3 and Figure 11.4) reveals a clear hierarchy. At full precision, the loss-based variants consistently outperform their heuristic counterparts; at low precision, HD-CB_{LOSS-PROB} dominates HD-CB_{PROB} at every tested bit width. Connecting the loss-based variants across precision levels traces a Pareto frontier that no other variant family exceeds, establishing the loss-based family as the uniformly best choice for any given bit budget. Remarkably, the 4-bit Loss-Probabilistic configuration surpasses the 32-bit LinEPS baseline while using $8\times$ less memory per element, and the reward curve flattens markedly beyond 3–4 bits, confirming that ultra-low-precision representations offer an excellent accuracy-efficiency sweet spot for edge deployment.
- **Neural network interpretation.** We demonstrated that the entire HD-CB pipeline can be formally mapped onto a Random Vector Functional Link network: the encoding stage constitutes a frozen nonlinear random projection (the RVFL hidden layer), while the action prototype matrix $\mathbf{A}$ acts as the trainable output weights. This identification connects HD-CB to randomized neural network research and opens a natural bridge toward spiking neural networks and deployment on neuromorphic hardware.

Throughout this chapter the experimental evaluation has deliberately employed the simplest ε -greedy exploration strategy. This choice was intentional: by fixing the exploration mechanism to its most elementary form, the impact of each algorithmic contribution could be isolated and measured against a common baseline without confounding effects introduced by more sophisticated exploration policies. In particular, the ε -greedy setting was instrumental in revealing the performance degradation caused by hard binarization in HD-CB_{BIN}, a phenomenon that was masked by a more aggressive confidence-based rule in Chapter 8. Investigating more complex exploration strategies within the HD-CB framework represents a natural direction for future research.

This chapter completes the algorithmic development of the HD-CB family. Part III will now turn to the hardware accelerators that enable these algorithms to achieve their full potential on embedded platforms.

Part 3

Hardware Acceleration and Co-Design

Overview

Parts I and II of this dissertation progressively redefined how online decision-making algorithms can be executed on resource-constrained platforms. Part I attacked the dominant bottlenecks of linear contextual bandits across the computing stack: the VLNPD and its integration into a vector accelerator mitigated the division bottleneck, while the Sherman-Morrison-Woodbury reformulation significantly reduces computational cost replacing explicit matrix inversions with incremental updates. This co-design approach achieved up to $58\times$ speedup and $50\times$ energy reduction over standard baselines. However, the fundamental quadratic scaling of covariance-matrices remained.

Part II addressed this limitation at a more fundamental level, reconsidering not merely *how fast* we execute linear algebra but the *nature of the computation itself*. By adopting hyperdimensional computing as a neuro-inspired substrate, we introduced the HD-CB family and its variants—binary, hybrid, probabilistic, loss-based, and loss-probabilistic—demonstrating that contextual bandit learning can be performed with simple, distributed, and low-precision operations while achieving superior performance.

Yet the full potential of neuro-inspired algorithms can only be realized when the underlying hardware is tailored to its unique operations. General-purpose hardware (e.g. CPUs and GPUs), even when optimized as in Part I, cannot fully exploit the massive parallelism and low-precision arithmetic inherent in HDC. The properties of the paradigm—low-precision components, extremely parallel operations, simple arithmetic—make it a compelling candidate for dedicated hardware acceleration [321, 322]. Prior works have explored FPGA [205, 323] and ASIC [206, 229] implementations, but these architectures typically commit to fixed design choices—specific tasks, datasets, hypervector dimensions, and encoding schemes—requiring the accelerator to be redesigned from scratch whenever a new algorithm or application is targeted. This drastically limits the practical adoption of these accelerators by the research community and makes investigating the impact of dedicated hardware on new emerging algorithms, like HD-CB, strongly impractical. Literature currently lacks general-purpose hardware solutions that can be adapted to efficiently support the execution of a wide range of HDC tasks.

Part III addresses this gap by developing a *family of open-source, general-purpose hardware accelerators* for HDC whose flexibility can support the full range of HDC learning tasks—classification, clustering, regression, and online decision-making—across diverse application constraints. By making all hardware designs, ISA extensions, and toolchains publicly available as open-source artifacts, we aim to lower the barrier for researchers and practitioners who wish to deploy HDC on

embedded platforms without requiring specialized hardware expertise.

The contributions of this part are organized as follows.

- **Chapter 12** introduces the *Hyperdimensional Coprocessor Unit* (HDCU), a configurable RISC-V extension for accelerating binary HDC primitives. Integrated into the Klessydra core family, the HDCU is parametric at synthesis time in data parallelism, supported operations, and memory size, while allowing the hypervector dimension to be set at runtime. A custom instruction set extension, fully integrated into the GCC toolchain, exposes the accelerator to software via simple intrinsics. The HDCU achieves speedups of up to $1968\times$ on individual primitives and up to $275\times$ on complete HD-CB inference compared to software execution, while requiring as few as 1030 LUTs in its minimal configuration.
- **Chapter 13** extends architectural support to *Modular Composite Representations* (MCR), challenging the widespread assumption that binary hypervectors are the most efficient choice for embedded platforms. We demonstrate that modular arithmetic achieves a unique balance of information capacity, classification accuracy, and hardware efficiency: the MCR-HDCU yields $2.7\times$ faster execution and $3.9\times$ lower energy than binary models at equivalent accuracy.
- **Chapter 14** presents a discretized, hardware-friendly accelerator for the FHRR, enabling complex-valued HDC operations with integer and fixed-point arithmetic. The design includes, to the best of author’s knowledge, the first hardware implementation of a Resonator Network.
- **Chapter 15** explores the computational boundaries of the FHRR paradigm by applying Resonator Networks to NP-complete problems (3-SAT). This proof-of-concept validates the broader applicability of the proposed FHRR accelerator while revealing fundamental capacity limits that inform future algorithm and hardware design.
- **Chapter 16** introduces *AeneasHDC*, an open-source automated end-to-end framework that streamlines the path from algorithmic specification to synthesizable FPGA accelerators. AeneasHDC enables systematic design-space exploration, reducing accelerator development time from weeks to hours and making hardware acceleration accessible to researchers without hardware design expertise.

Together, these contributions establish a complete, open-source hardware ecosystem for hyperdimensional computing that complements and enables the algorithmic advances developed in Part II. Importantly, the contributions of this part are not limited to the contextual bandit algorithms in Part II. The general-purpose nature of the proposed HDC accelerators means they can support any HDC learning task on the same hardware. By releasing all designs as open-source artifacts, we aim to promote adoption across the broader HDC community, enabling researchers and practitioners to deploy and evaluate hardware-accelerated HDC models without requiring specialized hardware design expertise.

Chapter 12

Configurable Hardware Acceleration for Hyperdimensional Computing Extension on RISC-V

Chapter based on Paper 6 and 7 [209]: Rocco Martino, Marco Angioli, Antonello Rosato, Marcello Barbirotta, Abdallah Cheikh, and Mauro Olivieri. *Configurable hardware acceleration for hyperdimensional computing extension on risc-v*. IEEE Transactions on Computers, 2025

"Civilization advances by extending the number of important operations which we can perform without thinking about them."

Alfred North Whitehead

Part III overview observed that existing HDC accelerators commit to fixed design choices, requiring a complete redesign whenever the target algorithm changes. This chapter begins to address that challenge starting from the simplest and most hardware-friendly representation: the binary spatter code (BSC)[164]. In particular, we ask:

❓ Research Question

Can we design a *configurable* and *general-purpose* HDC accelerator that efficiently supports diverse learning tasks, while enabling systematic trade-offs between performance, energy, and area through both synthesis-time and runtime configurability?

We answer affirmatively by introducing the *Hyperdimensional Coprocessor Unit (HDCU)*, an innovative highly flexible hardware accelerator that optimizes the core arithmetic operations on binary HVs of any size, while meeting the specific constraints of different target applications through hardware configurability and software programmability. The HDCU acts like a coprocessor tightly coupled with a RISC-V processor core, and features optimized functional units for each primitive

HDC operation along with dedicated local memories. Crucially, the architecture is configurable at synthesis time in data parallelism, supported operations, and memory size, while allowing the hypervector dimension to be set at runtime. A custom RISC-V Instruction Set Extension, fully integrated into the GCC compiler toolchain, exposes the accelerator to software via simple intrinsics.

The HDCU was previously employed in Chapters 8 and 9 to accelerate HD-CB_{BIN} and Hybrid HD-CB_{BIN}, achieving speedups of up to 275× over software execution. This chapter provides the architectural details and systematic evaluation of the accelerator.

The main contributions of this work are:

- Introducing a flexible hardware accelerator microarchitecture for HDC learning tasks, integrated into a RISC-V core and configurable at synthesis time;
- Allowing an efficient trade-off between time and resource usage to meet the constraints of diverse applications.
- Proposing a RISC-V ISE that maps the primitive operations of HDC, enabling the acceleration of generic learning tasks within a unified architecture;
- Characterizing the hardware requirements of the accelerator by synthesis on FPGA with varying data parallelism;
- Evaluating the speed-up and energy reduction provided by the proposed design, both on the basic operations and on complete learning tasks;
- Demonstrating the effectiveness of a general-purpose solution with respect to application-specific architectures;

The chapter is organized as follows. Section 12.1 surveys related HDC accelerators. Section 12.2 details the HDCU architecture and ISE. Section 12.3 presents configuration parameters and the programming model. Section 12.4 reports FPGA implementation results. Section 12.5 evaluates performance and energy efficiency. Section 12.6 contrasts with state-of-the-art designs. Finally, Section 12.7 summarizes the findings.

12.1 Related Works

Since HDC is well-suited to hardware acceleration thanks to its massive data-level parallelism, several FPGA and ASIC architectures have been proposed to speed up its core operations [205, 323, 324]. However, most of these accelerators are highly specialized—tuned for particular datasets, HV sizes, and hardware constraints—making them difficult to repurpose for new HDC tasks.

Initial steps toward more flexible HDC hardware come from architectures that allow configurability across datasets or encoding schemes, either at synthesis or via lightweight runtime control. In [206], the authors present Tiny-HD, a compact, energy-efficient ASIC for edge classification. Tiny-HD is highly optimized for energy and area and can be configured at synthesis for HV size and class count. However, it only supports inference (no training, or retraining), cannot perform permutation (so

it cannot encode sequences), and its datapath is fully deterministic and hard-wired with no user control. Similarly, the study in [325] proposes a fully learnable HDC framework and an ultratiny FPGA accelerator for vision applications, reformulating random-projection encoding as a binary neural layer and providing runtime configurability of the hypervector dimension and channel width. This enables adaptation across datasets within the image-classification domain and achieves strong accuracy–efficiency trade-offs, yet the pipeline remains fixed and cannot be extended to other HDC tasks. LAHDC [326] replaces similarity search with a logic-aggregation query and provides an automated search that, at synthesis time, selects the hypervector dimension, the query width, and the gate set to generate an ultratiny accelerator. This design achieves strong area and energy savings, supporting various classification datasets. However, the learned gate connections are compiled into hardware, which requires per-task synthesis, and its fixed learning pipeline prevents support for other HDC learning tasks. The work in [229] offers a higher degree of customization, with an accelerator based on standard cell memory, always on, ultra-low power, in 22 nm for sensor node classification, which exploits in-memory computing. The accelerator is always-on and wakes up the host only when a target class exceeds a configurable threshold. This design supports configurability in the HV size, number of classes, and parallelism at synthesis time, while a low-level microcode-based ISA allows the operation to be selected in the encoding tasks (permutation or binding), which are performed in a combinatorial datapath. While this approach offers significantly more flexibility compared to the other works in the literature, its pipeline strictly involves the encoding block and the associative memory, preventing support for retraining, clean-up, or other learning tasks (clustering, regression, and reinforcement learning). Its ASIC-specific optimizations and minimal host interface also hinder FPGA deployment and algorithm prototyping. Finally, the study in [327] demonstrates that the encoding techniques employed by both [206] and [229] perform poorly on several benchmarks.

Another step toward flexibility comes from automatic-generation frameworks. F5-HD [328] introduces a framework that allows users to specify dataset properties, target FPGA, and power/accuracy goals and then instantiates a matching accelerator using predefined processing elements. HD2FPGA [329] similarly generates a host-attached accelerator, adding support for clustering, variable HV sizes, and auto-tuning of parallelism according to user-specific requirements. However, both these frameworks still rely on fixed, hard-wired encoding and learning pipelines, making them unsuitable for temporal sequences or diverse input types, and cannot adjust parallelism across different learning stages of the learning process. Our prior work, AeneasHDC [330] (presented in detail in Chapter 16), seeks to overcome these limitations by introducing an automatic framework based on HLS for generating highly flexible hardware accelerators that can be configured across model configuration, encoding techniques and hardware parallelism, and can be generated to perform different learning tasks. Its focus, however, is on exploring hardware–accuracy trade-offs, making the accelerator unsuitable for applications in resource-constrained systems or with stringent real-time requirements, where optimizing each individual Functional Unit (FU) is crucial. Moreover, as in previous works, each accelerator is specifically synthesized for a single HDC model and task, and must be re-synthesized to target new problems.

A further step toward flexibility leverages the open ISA of RISC-V to introduce domain-specific instructions for HDC. RISC-HD [331] extends the low-power RISCY core with built-in support for the inference of HDC classification tasks with binary HVs. This core features dedicated hardware units, three 4.5 kB ROMs for HVs, and introduces two custom instructions in the RISC-V ISE: one for encoding, one for inference. While highly efficient and dataset-agnostic, this architecture still exhibits flexibility limitations, as it does not support training and is restricted to processing 1000-dimensional binary HVs with a fixed parallelism of 100 bits per clock cycle. The designed hardware units and the two RISC-V instructions are restricted to record-based encoding [330] and classification tasks, preventing their use in other learning problems or with different input data, such as temporal sequences, which significantly limits the flexibility and applicability of this core.

Overall, the literature on hardware designs for HDC is evolving towards more flexible and adaptable solutions, but the field still lacks a general-purpose flexible architecture that can be adapted to different applications, controlled at a high-level and easily deployed by the community (especially by researchers and practitioners without a hardware background) to prototype new algorithms and assess the impact of hardware acceleration on HDC-based learning tasks.

12.2 Hyperdimensional Coprocessor Unit

The designed HDCU is integrated into the execution stage of the Klessydra-T03 core, as depicted in Figure 12.1, and can run concurrently with the scalar execution by operating in parallel with the Load-Store Unit (LSU) and the Arithmetic Logic Unit (ALU). The accelerator can be synthesized to be replicated for each thread running in the core or shared among them.

12.2.1 Architecture overview

The HDCU operates as a tightly integrated coprocessor in charge of executing HDC operations. A custom RISC-V ISE was defined corresponding to the basic HDC operation set on binary HVs, leveraging the open and modular nature of the RISC-V ISA. Each instruction in the custom ISE is encoded following the standard RISC-V R-type format: the `opcode` identifies the instruction as part of the HDC extension, the `funct3` specifies the datatype (currently fixed to binary, but designed to support additional types in future extensions), and the `funct7` encodes the specific operation. These custom instructions have been fully integrated into the RISC-V GCC toolchain by extending the assembler and disassembler backends to recognize their encodings and enable proper parsing, emission, and disassembly. Each instruction is exposed to the programmer as a C-language intrinsic function, defined as a static inline wrapper with embedded assembly, to ensure accessibility even for non-hardware developers. This approach also enables compiler-level optimizations such as inlining, register allocation, and instruction scheduling, allowing HDC operations to be efficiently interleaved with conventional control or arithmetic code. Table 12.1 summarizes the available HDC instructions with their intrinsic syntax. Unlike task-specific approaches [331], our design exposes the fundamental VSA operators, enabling users to compose any arbitrary HDC algorithm in the literature purely through

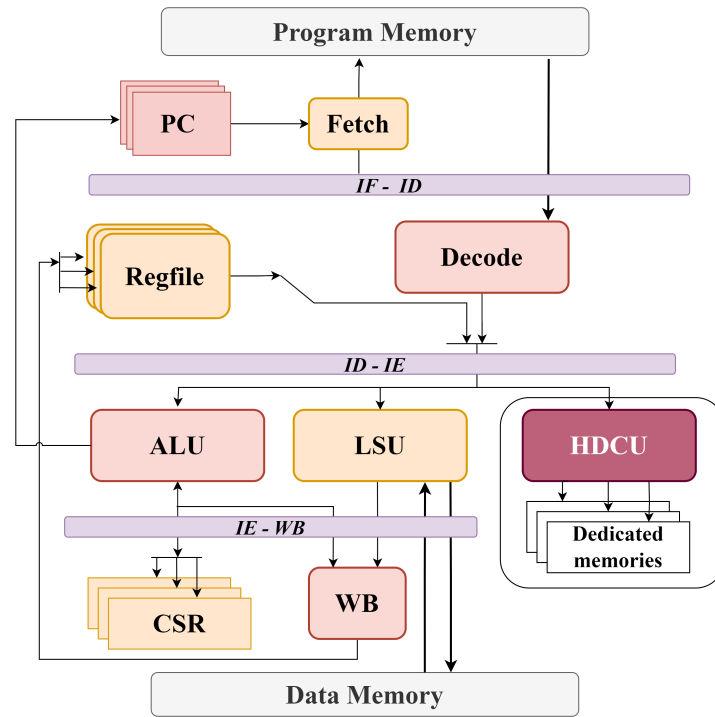


Figure 12.1 Integration of the HDCU in the Klessydra-T03 core.

Table 12.1 ISE for HDC Operations

Instruction	Description
hvmemld(*rd, *rs1, size)	Loads an HV from memory location rs1 into the SPM memory location rd . The size specifies the number of bytes to load.
hvmemstr(*rd, *rs1, size)	Stores an HV from the SPM memory location rs1 into data memory location rd . The size specifies the number of bytes to store.
hvbundle(*rd, *rs1, *rs2)	Bundle the <i>M</i> -bit precision HV in rs1 with the binary HV in rs2 to create a new HV in rd .
hvbind(*rd, *rs1, *rs2)	Bound the HVs in rs1 and rs2 , resulting in a new HV in rd .
hvperm(*rd, *rs1, *rs2)	Permute the HV in rs1 by rs2 positions storing the result in rd .
hvsim(*rd, *rs1, *rs2)	Hamming distance between the HVs in rs1 and rs2 . The similarity is stored in rd .
hvclip(*rd, *rs1, *rs2)	Binarize the <i>M</i> -bit precision HV in rs1 using the threshold in rs2 . Result in rd .
hvsearch(*rd, *rs1, *rs2)	Compare the HV in rs1 with CSR_HVCLASS HVs in rs2 . Stores the closest match in rd .

software, while adopting the same hardware backend. In the decode phase of the Klessydra-T03 pipeline, when the fetched instruction belongs to the HDC ISE and the HDCU is available, the coprocessor is invoked to handle the operation autonomously, without intervention of the host core.

The instructions executed by the HDCU operate on HVs contained in dedicated local Scratchpad Memories (SPMs), which are designed to provide low-latency, high-bandwidth access for parallel operations. The size of each SPM can be configured at synthesis time. When the learning task is known in advance, the SPMs can be dimensioned to match precisely the required number of HVs, ensuring optimal hardware and energy efficiency. On the other hand, for general-purpose deployments where the learning task may vary at runtime, larger SPMs may be synthesized to support broader flexibility without reconfiguration. The required size of SPMs depends on the number of input features, the number of classes, and the dimensionality of the HVs, while it is independent of dataset sizes (which reside entirely in host memory). The accelerator operates on HVs whose addresses in the SPMs are provided by the host processor according to the instruction syntax outlined in Table 12.1. Each SPM includes one read and one write port, enabling results to overwrite operands directly without intermediate buffers.

The host processor interacts with these memories via the Scratchpad Memory Interface (SPMI), using the dedicated *hvmemld* and *hvmemstr* instructions, where the *size* parameter indicates the byte count to be transferred. This interface and tight integration offer unprecedented flexibility compared to prior architectures in the literature. Users can seamlessly interleave standard RISC-V instructions with HDC operations, enabling real-time debugging, incremental development, hybrid CPU-accelerator workflows, and experimentation without hardware redesign. While the SPMI width is fixed at 32 bits (introducing latency of $HV_SIZE/32$ cycles per HV transfer), these interactions typically occur only at the initiation or completion of the task (e.g., loading initial HVs or collecting results) and, therefore, do not interfere or introduce bottlenecks during iterative computations. Once loaded, in fact, the HDCU operates directly on locally stored HVs, with no further dependence on host memory. As a concrete reference, for an HV size of 8192 and a representative mid-range hardware configuration of the HDCU, initial SPM loading requires 0.10 *ms* against 14.4 *ms* for a full inference run on the CARDIO dataset [332] (0.7% overhead), while on the larger UCIHAR [333] dataset it takes 0.75 *ms* compared to 5.0 *s* for a complete training pass (< 0.02% overhead). Notably, when the target task is known in advance, the required HVs can be preloaded into the SPMs at synthesis time, eliminating the need for runtime transfers entirely. By default, two out of four SPMs are automatically initialized at synthesis time: one with random BVs and the other with LVs, which are common components in many HDC pipelines. Notably, hierarchical caching could further reduce load-time latency, but we omitted such mechanisms from this initial design to limit complexity and overhead; future implementations will explore these options.

The tight in-pipeline integration of the accelerator not only enables flexible and high-level software programmability through standard instructions but also provides direct access to CPU registers and immediate operands. This eliminates the need for explicit data movement, synchronization, and additional control logic typically required in DMA-based or loosely coupled designs, reducing latency, hardware overhead, and programming complexity.

Figure 12.2 shows a detailed view of the microarchitecture of the HDCU, which includes dedicated pipelines and highly optimized FUs for each of the basic HDC arithmetic operations. These units operate independently and concurrently with

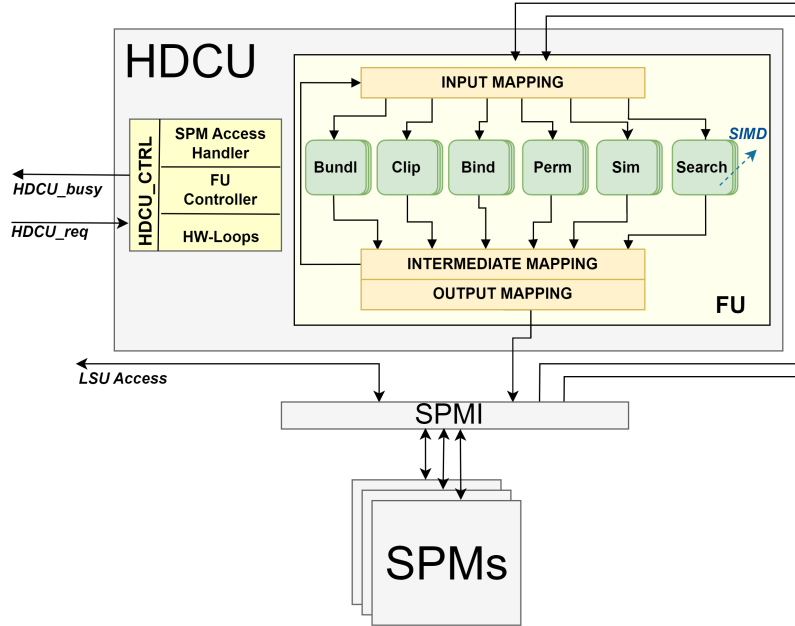


Figure 12.2 High-level schematic of the HDCU microarchitecture showing functional units and scratchpad memories.

the scalar pipeline, enabling efficient parallel execution of HDC instructions. The hardware parallelism inside each functional unit can be configured at synthesis time using a parameter called *SIMD* degree (Single Instruction Multiple Data). For instance, setting *SIMD* equal to 256 replicates the hardware to process 256 binary HV elements in one clock cycle. This flexibility enables trading off execution time with energy consumption and hardware requirements, allowing the accelerator to adapt to the specific needs of various application scenarios. Additionally, each functional unit can be optionally disabled to reduce hardware requirements when necessary, leaving the software execution of the corresponding HDC operation on the core.

Differently from existing solutions in the literature, the *HV size* in HDCU is not fixed but can be set at *runtime* by writing into a dedicated Control and Status Register (CSR), denoted as *HVSIZE* and visible to the software program through the *CSR_HVSIZE* instruction. The value of *HVSIZE* controls the hardware loops in the functional units. The hardware loops keep the required functional unit busy until *HVSIZE* elements are processed, avoiding the need for repetitive software loops in which the same instructions are repeatedly fetched and decoded. If the *HVSIZE* is bigger than the *SIMD*, the functional units process chunks of *SIMD* bits per clock cycle. For example, with *SIMD* = 32 and *HVSIZE* = 1024, the functional unit will be busy for 1024/32 clock cycles. All data movement between the SPMs and the HDCU datapath is internally pipelined and overlapped with the execution of the corresponding operation, introducing no additional latency.

12.2.2 Functional Units

This Section details the design of each functional unit in the HDCU, showing how they have been optimized for performance, resource efficiency, scalability and flexibility.

Binding Unit. Binding is implemented as a bit-wise logic XOR, with the SIMD parameter defining how many XOR gates operate in parallel.

Bundling Unit. In BSC, bundling is performed as an element-wise sum between HVs, resulting in an integer HV that is later binarized through a clipping operator. However, in practical pipelines such as encoding or training, binarization is typically applied only after several bundling steps [218, 330]. For example, when encoding an input sequence with N features, each feature HV is successively bundled, and binarization is deferred until all N HVs have been accumulated, requiring intermediate HVs to maintain integer precision until the final clipping step [205]. To support this feature, in the view of flexibility and scalability, we decouple bundling and binarization into two separate instructions. The number of HVs to be bundled before clipping is defined by the software program, with only an upper limit referred to as the Bundle Capacity (BC), representing the maximum number of HVs that can be accumulated through bundling. The parameter BC is a power of 2 and is set at synthesis time as the number M of bits needed to represent it, i.e. $M = \log_2 BC$. For instance, if inference is performed on a dataset with 5 features, we can set $M = 3$; whereas for optimizing the training process on a large FPGA, we can set $M = 32$.

The bundling hardware unit is composed of an array of M -bit counters. In order to maintain M -bit integer precision in cumulative bundling, the unit assumes the first HV operand, pointed to by `rs1`, has a *widened* format – i.e. each element being M bits wide – and it produces a similarly widened result. The value of each counter is initialized to the corresponding M -bit element of the HV pointed to by `rs1`, and it increments the count value if a '1' occurs in the corresponding position of the second binary operand, `rs2`. The output of each counter represents one element of the output HV of the operation.

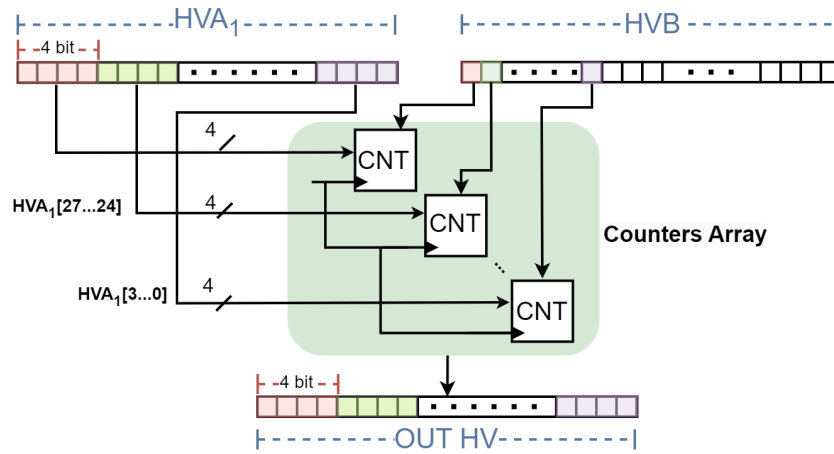


Figure 12.3 Bundling Unit with M set to 4

This approach enables iterative HV bundling with M -bit element precision, allowing the output of a bundling operation to be used as the first operand in subsequent bundling operations. In order for the software program to operate correctly, the first of a cumulative bundling sequence should always have a zeroed HV as its destination.

The number of counters in the functional unit is

$$\text{Counters} = \frac{SIMD}{M} \quad (12.1)$$

while the number of HV elements processed per clock cycle by the bundling unit is expressed by

$$\text{Bundling latency} = \frac{HVSIZE * M}{SIMD} \quad (12.2)$$

Clipping Unit. Binary normalization is performed by the Clipping Unit using a majority rule: each element of the output binary HV is set to '1' if the arithmetic value of the corresponding M -bit element in the widened input HV exceeds a threshold, which is usually half the number of bundled HVs, with ties broken at random. To ensure flexibility and not constrain the programmer to any number of features in the dataset, we decided to make the value threshold explicit in the Clipping operation invoked by the program.

The hardware structure of the clipping unit is similar to that of the bundling unit but replaces the array of counters with an array of comparators. In this case, the unit takes as input one widened HV with M -bit elements and returns a binary HV. The latency for clipping completion is given by

$$\text{Clipping latency} = \frac{HVSIZE * M}{SIMD} \quad (12.3)$$

Permutation Unit. As explained in Chapter 2, the HDC permutation operation consists of a cyclic right shift (rotation), where the least significant bits that fall out of the HV replace the most significant ones.

Like the other FUs, the structure of the Permutation Unit is also HV length agnostic and supports configurable hardware parallelism. To rotate arbitrarily long vectors, as illustrated in Figure 12.4, a temporary register (buffer) stores the least

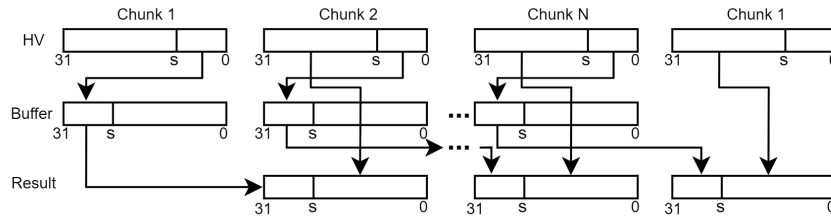


Figure 12.4 Permutation Principle Scheme applied to a 128-bit HV processed using a SIMD of 32

significant bits of the HV chunk processed in the current clock cycle. In the first clock cycle, the most significant chunk of the HV pointed by `rs1` is read from the local memory, and a number of bits equal to the shift amount `s` are shifted out and stored in the buffer. The output of the first cycle is not written back to memory. In the second clock cycle, the buffer content is moved to the most significant `s` bits of the subsequent HV chunk, while the least significant `s` bits of this chunk are saved in the buffer, replacing its previous content. This process iterates until the entire HV has been read from memory and permuted. Once the last chunk has been processed, the first chunk is reloaded, shifted to the right, and the least significant `s` bits of the last processed chunk are inserted as its most significant bits.

Similarity Unit. The similarity measurement between two binary HVs can be computed using the Hamming distance as described in Chapter 2. The similarity hardware unit first performs a bitwise XOR across *SIMD* bits of the two operands, then counts the number of '1's in that SIMD-wide result (using a pop-count function), and finally accumulates these counts over the entire HV.

To implement the pop-count, we evaluated the six architectures reviewed in [334], comparing their resource utilization, critical path delays, scalability and latency. Among them, we selected the basic loop-based design, which provides excellent scalability in terms of hardware requirements and latency. This solution is well-suited for our configurable accelerator, where hardware parallelism is set at synthesis time and functional units operate on binary HVs. Future extensions of the HDCU targeting more complex datatypes (e.g., fixed-point or complex HVs) may benefit from adopting a pipelined tree-adder design, which reduces critical path length and improves throughput while maintaining a high operating frequency. The output of the pop-count is accumulated through a 13-bit adder. The similarity unit computes *SIMD* elements per clock cycle, requiring a total latency of $(SIMD/HVSIZE + 2)$. After processing the entire HV, the final value in memory represents the Hamming distance.

Search Unit. The last operation implemented in the HDCU is the *search*. While this is not a standard HDC operator, it is crucial during the inference phase of many learning tasks to identify the closest prototype among a set of HVs stored in memory. Performing this procedure in software would require calling the designed `hvsim` instruction k times in a k -class problem, which is inefficient and limits the advantages of hardware acceleration. Therefore, having a specific hardware implementation for this task is highly beneficial. While in-memory computing techniques are often used in the literature to implement this operation, such approaches are not feasible for FPGA implementations or soft-processors like Klessydra. Consequently, we decided to reuse the similarity unit and introduce a dedicated hardware instruction.

Unlike all previous operations, the search needs one additional information: the number of prototypes K . This is provided to the HDCU at *runtime* through the dedicated CSR function `CSR_PROT_COUNT()`. The search instruction reads the query HV from `rs1` and the base address of the prototype array from `rs2`. A hardware loop then compares the query against the K prototypes (each of `HVSIZE` bits), keeping the query latched on one input of the similarity unit while the other input iterates

Table 12.2 HDCU Configuration Parameters

Parameter	Configuration Time
SPM Size	Synthesis
SPM Number	Synthesis
Hardware Parallelism (<i>SIMD</i>)	Synthesis
Functional Unit Enable/Disable	Synthesis
Bundle Capacity (<i>M</i>)	Synthesis
HVSIZE	Runtime
PROT_COUNT	Runtime

through the prototypes. Each time a similarity is computed, it is compared with the best similarity stored in a 13-bit register that is eventually updated. When the search completes, the functional unit writes the index of the class that returned the best similarity to the scratchpad memory.

12.3 HDCU configuration and programming

Overall, the presented hardware accelerator exemplifies significant flexibility through configurability and software programmability via the RISC-V ISE, making it versatile for a wide range of learning tasks and of application contexts.

12.3.1 Hardware configuration

Table 12.2 summarizes the parameters that can be tuned at synthesis time and at runtime. At synthesis time, the size and number of SPMs, the hardware parallelism, the Bundle Capacity, and the enable/disable status of each functional unit can be configured. These parameters allow the designer to balance the trade-offs between speed, energy, and resource utilization according to the specific requirements of the target application. At runtime, the HV size and the number of prototypes can be set through dedicated CSRs that control the hardware loops. This capability enhances the flexibility of the HDCU, enabling dynamic adaptation to varying workloads, using the same accelerator.

12.3.2 Software programming

The set of intrinsic functions available to the programmer introduced in Section 12.2, directly mapped on the RISC-V ISE and fully integrated into the GCC compiler toolchain, constitutes a simple yet effective API to implement HDC workloads.

As a reference example, pseudocode 13 demonstrates the record-based encoding [330] implemented through the presented ISE. In this technique, feature indices are assigned to randomly generated base HVs (BVs); while the feature values are discretized and mapped in level HVs (LVs) generated ensuring similar values are transformed into similar HVs [218]. The BVs and LVs are then bound to form a key-value pair; all pairs are subsequently bundled to yield the final HV representing the entire feature vector. On the HDCU, by default, two of the SPM banks (spmA and spmB) are automatically initialized at synthesis time with BVs and LVs, respectively.

As a result, explicit loading is not required unless the user wishes to override these defaults with custom-generated HVs. The size of the HVs is set at runtime through the `CSR_HVSIZE()` function, which controls the hardware loops of the following arithmetic instructions. These initial steps are performed only once before starting the encoding process and should not be repeated for each input feature vector. The record-based encoding then begins. For each feature i , its raw scalar value is quantized on-the-fly in software (via simple linear scaling or threshold comparisons) to produce an integer level index $q[i]$. We then bind the $BV[i]$ to the corresponding $LV[q[i]]$ via the `hvbnd` instruction, pointing to the respective addresses in the SPMs. The output of this process is bundled using the `hvbndle` instruction to create the encoded HV with an element precision of M -bits. After processing all features, the resulting HV is binarized using the `hvclip` instruction. This completes the encoding operation, with the resulting HV stored in the SPM. If the accelerator is no longer required, the output can be saved to the data memory via the `hvmemstr` instruction. Otherwise, the training and inference processes can proceed, continuing to point to the encoded HV in SPM.

This example illustrates that the architecture does not assume any fixed encoding strategy. All encoding logic remains programmable at the software level, enabling users to implement virtually any technique from the literature [218] by simply initializing the SPMs accordingly and composing HDC instructions as needed. Notably, by decoupling binarization from bundling, the HDCU enables intermediate HVs to retain multi-bit precision throughout the encoding and training phases. Users can also offload intermediate multi-bit HVs to the host processor and explore mixed-precision pipelines.

Algorithm 13 Record-based Encoding Algorithm

```

1: // Configure HV size
2: CSR_HVSIZE(HVSIZE)

3: // Bind and Bundle
4: for  $i = 0$  to  $DS\_FEATURE\_SIZE - 1$  do
5:   level_addr = spmA +  $q[i] \times HVSIZE$ 
6:   base_addr = spmB +  $i \times HVSIZE$ 
7:   hvbnd(spmC, level_addr, base_addr)
8:   hvbndle(spmD, spmD, spmC)
9: end for

10: // Clip final result
11: hvclip(spmC, spmD, DS_FEATURE_SIZE)

```

12.3.3 HDCU Software Library

On top of the API constituted by the HDCU intrinsic functions, we developed a higher-level software library in *C++* to simplify accelerator usage, evaluate performance, and streamline the development of HDC applications. The library provides

Table 12.3 Hardware Requirements of the Klessydra-T03 and the HDCU with SIMD = 32; $f = 220$ MHz; SPMs = 2 kB

Device	#LUTs	#FFs	#DSPs	#BRAM
Klessydra-T03	4281	1418	7	0
HDCU	1030	450	0	0
SPMI	384	151	0	2
SPMs	268	0	0	2

custom types for binary and integer HVs, along with pre-defined functions that abstract well-known procedures from the HDC literature. Examples include functions for generating base and level HVs using both linear and thermometer quantization, record-based and n-gram encoding strategies [330], temporal encoding, classification training and inference, retraining, clustering, and regression. These high-level functions internally orchestrate the appropriate sequence of intrinsic operations listed in Table 12.1 and automatically configure the required CSRs, further abstracting hardware control from the user. For instance, the record-based encoding operation described in Pseudocode 13 can be performed with a single call to the `hdc_recordbased_encoding` function, that handles all required memory setup, arithmetic steps, and final binarization using the HDCU instructions. All library functions can be executed either as pure software routines on the scalar RISC-V core or using hardware acceleration on the HDCU, enabling correctness validation and performance benchmarking across hardware and software. The HDCU Software Library is released as open-source on GitHub [335], along with examples and documentation, to support ease of use and community-driven extension.

12.4 Hardware implementation

Being hardware configurability a principal feature of the proposed design, the target implementation is on FPGA devices. To assess the hardware requirements and performance of the proposed acceleration units, we synthesized the Klessydra-T03 core equipped with the novel HDCU on the Xilinx Zynq UltraScale+ ZCU106 (EK-U1-ZCU106-G) device, using the Vivado 2023.2 tool suite.

Table 12.3 provides a detailed breakdown of hardware resource utilization of the baseline configuration of the HDCU integrated with the Klessydra-T03 core. For this configuration, we adopted a SIMD width of 32, a bundling precision of 4 bits, and 3 SPMs of 2 KB each. This setup represents the minimal hardware parallelism supported by the HDCU, and it is suitable for resource-constrained environments, such as IoT applications on the edge. The hardware utilization is detailed as the utilized Look-Up Tables (LUTs), Flip-Flops (FFs), Digital Signal Processors (DSPs), and Block-RAM (BRAM) on the FPGA device. The baseline HDCU occupies 1030 LUTs and 450 FFs and, importantly, it does not affect the critical path of the microarchitecture, allowing the processor to maintain its original operating frequency of 220 MHz.

In order to explore how the hardware resource requirements of the HDCU

Table 12.4 Hardware Requirements of HDCU for Different SIMD Configurations

SIMD	#LUTs	#FFS	#CARRY8	$f_{\max}$ [MHz]
32	1030	450	87	234
64	2040	651	101	221
128	3243	868	131	218
256	4016	1101	172	215
512	13811	3731	464	160
1024	34189	7478	1424	140

Table 12.5 Execution time for each operation (in μs)

Scheme	HV Size	Binding	Bundling	Clipping	Permutation	Similarity	Search
Software	64	0.19	9.75	3.18	0.26	0.78	6.54
Software	512	1.84	73.00	25.34	2.05	6.12	57.41
Software	2048	7.44	298.96	100.08	6.95	25.05	224.83
SIMD 32	64	0.06	0.21	0.13	0.10	0.03	0.08
SIMD 32	512	0.10	0.73	0.42	0.17	0.06	0.42
SIMD 32	2048	0.31	2.58	1.40	0.42	0.17	1.53
SIMD 256	512	0.05	0.21	0.13	0.10	0.03	0.09
SIMD 256	2048	0.11	0.45	0.26	0.14	0.05	0.23
SIMD 1024	2048	0.10	0.34	0.20	0.16	0.24	0.14

scale when varying the SIMD width from 32 to 1024, we synthesized other HDCU configurations, reporting the resulting hardware utilization details in Table 12.4. This analysis is crucial for understanding how the accelerator can be adapted to different application scenarios, trading-off performance for hardware utilization. When increasing the SIMD from 32 to 1024, the number of LUTs increases from 1030 to 34189, evidencing the importance of selecting an appropriate SIMD configuration based on the specific needs of the target application. In the context of resource-constrained applications focused on edge inference computation, a low SIMD such as 32 is likely to be suitable. Conversely, when resource constraints are less of an issue and the goal is also to accelerate the training phase, higher parallelism with SIMD 1024 may be employed to maximize performance.

Although the Klessydra core nominally operates at 220 MHz, in practical SoC deployments (e.g., in PULPino) the system frequency is typically constrained by peripheral and memory interfaces to around 100 MHz. This frequency remains below the post-implementation $f_{\max}$ of our HDCU, even for aggressive configurations (e.g., SIMD = 1024). Additionally, during non-HDC workloads, FPGA partial reconfiguration can be exploited to disable the HDCU when not required and reconfigure the core region for the highest achievable frequency.

Overall, the obtained results demonstrate the flexibility and scalability of the proposed configurable design. By carefully tuning its parameters, users can effectively trade off execution time with area occupancy, pursuing the optimal balance based on the specific application requirements. These characteristics establish the HDCU as a versatile component capable of addressing the diverse needs of various contexts.

Table 12.6 Energy consumption for each operation (in μJ)

Scheme	HV Size	Binding	Bundling	Clipping	Permutation	Similarity	Search
Software	64	0.021	1.025	0.335	0.028	0.083	0.687
Software	512	0.220	8.688	3.016	0.244	0.729	6.832
Software	2048	0.894	35.876	12.010	0.835	3.006	26.980
SIMD 32	64	0.006	0.021	0.013	0.018	0.003	0.008
SIMD 32	512	0.010	0.074	0.043	0.018	0.007	0.043
SIMD 32	2048	0.033	0.276	0.150	0.045	0.019	0.163
SIMD 256	512	0.007	0.026	0.016	0.013	0.004	0.011
SIMD 256	2048	0.035	0.062	0.035	0.019	0.007	0.032
SIMD 1024	2048	0.014	0.046	0.028	0.023	0.033	0.019

12.5 Performance Results

This section evaluates the advantages of the HDCU on both basic HDC operations and real-world computation kernels, demonstrating the accelerator’s performance and flexibility. All benchmarks were implemented using the custom software library introduced in Section 12.3.3, allowing each computation kernel to be executed in both a standard (non-accelerated) configuration, compiled for the RISC-V instruction set and executed on the Klessydra-T03 processor without HDCU support, and an accelerated configuration, compiled with the extended instruction set exploiting the HDCU support.

Execution time estimations were performed using cycle-accurate Register-Transfer-Level simulations in QuestaSim, extracting the number of clock cycles and multiplying them by the post-implementation maximum operating frequency measured for each configuration and reported in Table 12.2. Speedup factors were then computed as the ratio between baseline and accelerated execution times. For energy consumption, we performed post-implementation gate-level simulations to generate switching activity files (`.saif`) for each kernel, which were then analyzed using the Vivado power estimator to derive the dynamic power of both the core and the accelerator. Energy reduction factors were then calculated as the ratio of the baseline and accelerated dynamic energy consumption.

All experiments were conducted across a range of HV sizes and parallelism configurations to explore scalability and design trade-offs. Specifically, we evaluate three representative SIMD configurations (32, 256, and 1024) and three HV sizes (64, 512, and 2048). To improve readability, we omit configurations where $\text{HVSIZE} < \text{SIMD}$, as they incur the same execution time and energy consumption as the $\text{HVSIZE} = \text{SIMD}$ case and do not contribute additional insight.

12.5.1 Impact on the Basic HDC Arithmetic Operation

The execution time and the dynamic energy consumption achieved by the HDCU for each core arithmetic operation in HDC are summarized in Table 12.5 and Table 12.6 respectively. As the HV size increases, the hardware loops inside the HDCU allow efficient processing without re-fetching instructions repeatedly, maximizing the execution speed. On the other hand, increasing SIMD values enables greater parallelism, allowing for the simultaneous processing of more HV elements, but at

the cost of increased hardware complexity, as demonstrated in the previous Section.

For the *Binding* operation the HDCU achieves speedups ranging from $3\times$ at $\text{SIMD} = 32$ and $\text{HVSIZE} = 64$ to $70\times$ at $\text{SIMD} = 1024$ and $\text{HVSIZE} = 2048$. A similar scaling trend is observed for the *Permutation* operator, where despite its more complex data movement and cyclic rotation, the HDCU sustains up to $49\times$ speedup. Substantially higher performance gains are observed in the more complex *Bundling* and *Clipping* operations, which benefit from hardware-level accumulation with configurable precision (M bits), as discussed in Section 12.2. The former achieves speedups from $45\times$ to $868\times$. The latter exhibits speedups in the range of $24\times$ to $478\times$. All these operations, especially the bundling, are critical for encoding, which is often the most computationally intensive stage in HDC training pipelines [336]. Reducing the latency of the *Similarity* operation is equally important, as it directly impacts inference speed. The HDCU accelerates this task with speedups ranging from $20\times$ to $471\times$, scaling efficiently with both SIMD and HV size. Even more notable is the Search operation, which combines multiple similarity evaluations in a single instruction. In classification problems with three prototypes, this operator yields up to $1540\times$ speedup, showcasing the advantage of offloading this task to dedicated hardware.

Energy efficiency follows a similar trend. Across all operations, the software-only implementation on the baseline RISC-V pipeline consumes between $2.56\times$ and $1359\times$ more energy than its HDCU-accelerated counterpart. This substantial reduction highlights the value of specialized hardware, particularly for low-power applications, even when accelerating only individual arithmetic operators.

12.5.2 Performance on real datasets

To demonstrate the flexibility of the proposed accelerator and evaluate its performance in real-world scenarios, we implemented a suite of diverse HDC-based tasks. In each experiment, we measured both execution time and energy consumption across a single iteration of the encoding, training, and inference phases. Table 12.7 and 12.8 summarizes the obtained results. It is important to note that the focus of this analysis is not to assess the accuracy achieved by the accelerator, since each arithmetic operation implemented in the HDCU, along with the encoding, training, and inference stages, has been rigorously tested and validated to ensure full consistency with the software model. Consequently, the accelerator delivers the exact same accuracy as an HDC software model.

For classification tasks, we considered three datasets to explore a range of problem complexities and feature characteristics. The first dataset, *CARDIO* [332], includes 3 classes and 21 features, which are encoded using the record-based technique implemented in Pseudocode 13. On this benchmark, the HDCU significantly accelerates all phases of the pipeline, with execution time reduced by up to $208\times$ in encoding, $203\times$ in training, and $212\times$ in inference. Energy consumption is also substantially reduced, with a maximum reduction factor of $176\times$. Interestingly, due to the relatively low number of features and classes, the highest performance is obtained with the $\text{SIMD} = 256$ configuration. In this case, the increased speedup from $\text{SIMD} = 1024$ does not compensate for the lower clock frequency resulting from higher hardware complexity.

Table 12.7 Execution time in μs per task and configuration. Configurations achieving the highest speedup over software execution are highlighted in bold.

Configuration		CLASSIFICATION									CLUSTERING		RL	
Scheme	HV SIZE	CARDIO			EMG			UCIHAR			IRIS		HD-CB _{BIN}	
		Enc.	Train.	Infer.	Enc.	Train.	Infer.	Enc.	Train.	Infer.	Train.	Infer.	Act. Sel.	Update
Software	64	111.62	113.69	116.30	68.81	73.71	76.67	11464.50	11486.88	11476.85	3346.33	2423.33	604.99	119.76
Software	512	874.03	897.85	918.65	544.90	581.42	633.83	44930.88	89321.29	89271.74	24637.75	18055.37	2286.68	1025.44
Software	2048	3436.15	3579.83	3629.21	2123.90	2268.87	2452.87	354513.86	355175.26	11476.85	97002.72	70782.61	17873.42	3015.77
SIMD 32	64	3.80	3.90	3.90	2.45	2.55	2.53	55.54	56.35	55.66	7.18	3.37	19.08	9.70
SIMD 32	512	11.95	12.39	12.03	6.65	6.96	6.80	259.87	264.56	259.99	22.01	16.84	20.91	17.04
SIMD 32	2048	42.42	44.58	42.50	23.25	24.43	23.62	995.33	1013.07	995.45	72.03	62.21	38.50	42.99
SIMD 256	512	6.66	6.82	6.77	2.50	2.61	2.59	70.59	75.38	70.71	21.13	17.24	20.28	17.44
SIMD 256	2048	17.05	17.62	17.13	4.00	4.17	4.11	200.71	218.87	200.83	68.15	63.66	23.47	43.84
SIMD 1024	2048	22.58	23.23	22.75	3.86	4.03	4.00	180.23	208.10	180.41	103.73	97.76	34.16	67.32

Table 12.8 Energy Consumption in μJ per Task and Configuration. Configurations achieving the highest energy reduction factor over software execution are highlighted in bold.

Configuration		CLASSIFICATION									CLUSTERING		RL	
Scheme	HV SIZE	CARDIO			EMG			UCIHAR			IRIS		HD-CB _{BIN}	
		Enc.	Train.	Infer.	Enc.	Train.	Infer.	Enc.	Train.	Infer.	Train.	Infer.	Act. Sel.	Update
Software	64	11.72	11.82	12.10	7.02	7.59	7.90	1203.77	1194.64	1193.59	351.36	232.64	62.31	12.22
Software	512	104.01	103.25	109.32	59.94	65.12	72.26	5346.77	10271.95	10623.34	2562.33	1877.76	237.81	105.62
Software	2048	412.34	422.42	435.50	252.74	270.00	294.34	42541.66	41910.68	1377.22	11252.32	8069.22	1894.58	316.66
SIMD 32	64	0.38	0.39	0.39	0.25	0.26	0.26	5.61	5.64	5.57	0.72	0.34	1.93	0.97
SIMD 32	512	1.22	1.26	1.21	0.69	0.70	0.69	26.51	26.98	26.26	2.20	1.72	2.15	1.72
SIMD 32	2048	4.58	4.77	4.51	2.46	2.64	2.53	107.50	131.70	105.52	7.42	7.28	4.04	12.34
SIMD 256	512	0.82	0.84	0.83	0.31	0.32	0.32	8.68	9.27	8.63	2.54	2.12	2.53	2.13
SIMD 256	2048	2.22	2.40	2.23	0.51	0.54	0.53	26.09	29.77	26.11	8.45	8.02	3.10	5.66
SIMD 1024	2048	6.55	6.99	6.67	1.12	1.19	1.17	52.27	62.64	52.86	29.67	28.25	10.01	19.32

A different trend is observed in the second dataset, *UCIHAR* [333], which presents a significantly more complex classification task with 561 features and 6 classes. Here, the HDCU reduces encoding, training, and inference times by up to $1967\times$, $1706\times$, and $1968\times$, respectively, using the $\text{SIMD} = 1024$ configuration. This workload benefits from higher hardware parallelism, which greatly improves throughput as the dimensionality of the HVs increases. The energy reduction is also remarkable, reaching $813\times$ with $\text{SIMD} = 1024$. Notably, although $\text{SIMD} = 1024$ achieves the lowest latency, the configuration with $\text{SIMD} = 256$ provides the best energy efficiency, yielding energy savings of up to $1630\times$ —making it a more optimal choice in power-constrained scenarios.

The third dataset, *EMG* [337], is a temporal classification task involving 4 features and 5 classes. Here, we adopted a temporal encoding scheme, extending the record-based technique by incorporating permutation operations to capture temporal dependencies across three time steps. Essentially, each pattern was mapped into the high-dimensional space, permuted based on its position in the time window, and then bundled with the other HVs of the same temporal window. In this case, the HDCU achieves up to $549\times$ speedup in encoding, $563\times$ in training, and $613\times$ in inference, with energy consumption reduced by a factor of up to $554\times$.

To evaluate the HDCU’s applicability beyond classification, we tested it on an unsupervised learning task using the *IRIS* dataset [338], which includes 4 features and 3 classes. The clustering process has been implemented according to [221]. In this scenario, the HDCU achieves execution time reductions of up to $761\times$ in training and $1111\times$ in inference with $\text{SIMD} = 1024$, alongside energy savings up to $1516\times$.

Finally, to demonstrate the HDCU’s ability to support novel learning paradigms

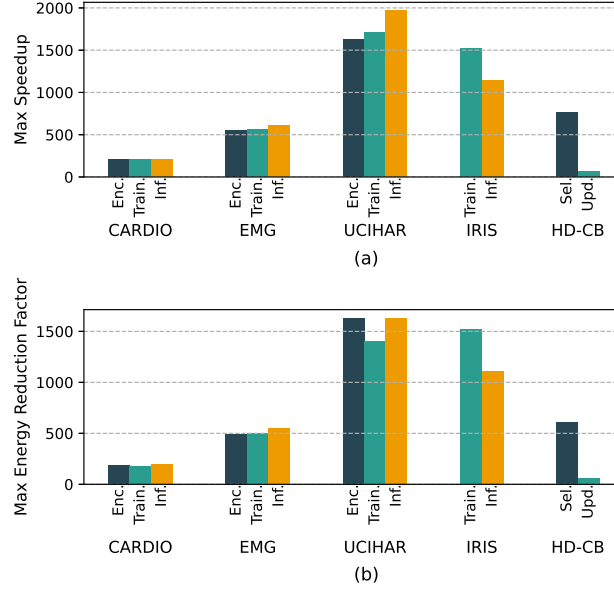


Figure 12.5 Maximum achieved (a) speedup and (b) energy reduction factor for each benchmark and stage

and allow algorithm prototyping, we accelerated our HD-CB_{BIN} (as described in Chapter 8). No prior work has proposed hardware implementations for this algorithm and investigated the impact of acceleration, also due to the required non-standard HDC operators in its learning loop. However, thanks to its flexibility, high-level control and tight integration with the host core, the HDCU can easily optimize this task by interleaving the specialized instruction of the proposed ISA with standard arithmetic and control logic. With a SIMD = 512, the HDCU achieves up to $761\times$ speedup in action selection and $70\times$ in the update phase, while reducing energy consumption by factors of up to $611\times$ and $66\times$, respectively. This analysis also reveals that SPM energy consumption ranges from 19% to 57% of the total energy and is mainly determined by hardware parallelism and HV size, while being only marginally affected by the SPM size.

12.6 Discussion and Comparison

The results presented in this work demonstrate the flexibility and scalability of the proposed configurable HDCU. Leveraging the tight integration with the RISC-V core and the custom instruction set extension, the HDCU enables high-level, software-controlled dataflow management, supporting both standard HDC tasks and the prototyping of novel algorithms, including those requiring non-standard operations in the learning pipeline. Thanks to its configurable hardware parameters, users can adjust the trade-off between execution time and area occupancy at synthesis time, facilitating deployment across a wide range of applications and resource constraints.

While the HDCU has been deliberately designed to prioritize flexibility and programmability over peak performance, introducing the first general-purpose, task-agnostic accelerator for HDC, it still achieves substantial gains—reducing execution

Table 12.9 Comparison of HDC Architectures and Feature Support

Architecture	Device	SIMD	LUTs	FFs	DSPs	Area	f (MHz)	Time (us)	A×T	S. Enc.	T. Enc.	Train	Ret.	Inf	Clust.	Regr.	RL.
[339]	Kintex-7	224	189534	130k	0	47384	-	8	379.07	✓	×	×	×	✓	✓	×	×
[328]	Kintex-7	481	167000	-	244	65450	-	0.2	13.09	✓	×	✓	✓	✓	✓	×	×
[330]	ZCU-102	64 [†]	3522	5562	0	1765	100	113.64	200.62	✓	×	×	×	✓	×	×	×
[331]**	Artix-7	100	6073	2955	5	2932	41	62.10	182.07	✓	×	×	×	✓	✓	×	×
This work**	Artix-7	128 [†]	7524	2712	7	3420	90	67.97	232.45	✓	✓	✓	✓	✓	✓	✓	✓

[†] Settable at synthesis time.

** Hardware requirements of the entire core embedding support for HDC.

time by one to three orders of magnitude compared to software execution and lowering energy consumption by up to 99%. This confirms its effectiveness for both high-performance and energy-efficient applications.

To further emphasize the flexibility and highlight the trade-offs and positioning of the proposed design, Table 12.9 compares the HDCU with selected state-of-the-art HDC accelerators introduced in Section 12.1, all synthesized on FPGAs. The comparison covers hardware resource requirements, average time per operation, and applicability across diverse tasks. Following [331], we also report the AT (Area × Time) metric as an efficiency indicator. For works not reporting these metrics directly, we derived them from available data as accurately as possible. *Quant-HD* and the F5-HD framework deliver the fastest inference times, owing to their high hardware parallelism and aggressive resource utilization. However, these designs are fixed-function accelerators limited to classification pipelines with spatial encoding, and cannot be adapted for tasks such as temporal encoding, clustering, or reinforcement learning. The accelerator generated by *AeneasHDC* (Chapter 16), by contrast, offers the smallest footprint and supports multiple tasks configurable at synthesis time. However, the low parallelism selected for this analysis and the HLS-based implementation result in higher inference latencies compared to other solutions. *RISC-HD* achieves a balanced trade-off between resource usage and execution time, performing similarly to our HDCU in baseline metrics. For direct comparison, we synthesized the Klessydra core and HDCU on the same Artix-7 FPGA, confirming comparable hardware requirements. However, the HDCU operates at a significantly higher frequency and provides an unprecedented level of flexibility, supporting both prototyping of new HDC algorithms and deployment in resource-constrained edge systems. This capability is not matched by any of the compared accelerators in Table 12.9, making the HDCU an effective general-purpose platform for exploring and accelerating a wide spectrum of HDC-based workloads.

12.7 Summary of Findings

In this Chapter, we introduced the Hyperdimensional Coprocessor Unit, a highly flexible hardware accelerator specifically designed to optimize hyperdimensional computing tasks. The HDCU is configurable at synthesis time in terms of hardware parallelism, supported operations, and local memory sizes, allowing users to trade off execution speed and hardware requirements to meet the demands of different applications, ranging from embedded systems to performance-intensive contexts.

The HDCU is integrated into the execution stage of the RISC-V Klessydra-T03 core, and it is exposed to the software programmers through a custom RISC-V ISE mapping intrinsic functions, constituting a simple API for HDC. The API, including

the possibility to set the HV size at runtime, further enhances the HDCU versatility, enabling the acceleration of different learning tasks in a simple and intuitive way utilizing the same hardware architecture.

To validate the design, we synthesized the Klessydra-T03 core equipped with the HDCU on the Xilinx Zynq UltraScale+ ZCU106 FPGA and analysed its impact on the execution time of learning tasks, varying the HV size and hardware parallelism. Across all benchmarks, the HDCU achieved speedups of up to $1968\times$ and energy reductions of up to $1630\times$ compared to software execution on the baseline RISC-V core, while requiring as few as 1030 LUTs in its minimal configuration.

To the best of our knowledge, the HDCU represents an advancement in hardware-accelerated HDC, providing unprecedented versatility and scalability to tackle diverse learning tasks and application requirements.

Notably, while not presented as a primary contribution of the present dissertation, we subsequently extended the HDCU into a standalone, host-agnostic accelerator compliant with the AMBA AXI4 standard [340]. While the tight coupling between the HDCU and the Klessydra RISC-V core is advantageous for low-latency dispatch and efficient iterative kernels, it inherently limits portability: integrating the coprocessor into a different processor microarchitecture requires modifying the host pipeline, adapting the toolchain, and re-validating the interface. By adopting the AMBA AXI4 standard, the resulting *AXI Plug-and-Play HDC Accelerator* preserves the core functional units and scratchpad memory organization introduced in this chapter, but replaces the ISA-coupled interface with an AXI4-Lite control plane and DMA-driven AXI4-Stream datapaths, enabling seamless integration into any AXI-capable SoC without requiring modifications to the host processor or its toolchain. The interested reader can refer to our paper [340].

Chapter 13

Efficient Hyperdimensional Computing with Modular Composite Representations

Chapter based on Paper 12 [341]: Marco Angioli, Christopher J Kymn, Antonello Rosato, Amy Loutfi, Mauro Olivieri, and Denis Kleyko. Efficient hyperdimensional computing with modular composite representations. arXiv preprint arXiv:2511.09708, 2025

"The limits of my language mean the limits of my world."

Ludwig Wittgenstein

The previous chapter demonstrated how a tightly integrated and flexible accelerator such as HDCU can maximize the performance of BSC for learning on resource-constrained devices. However, despite the attractive simplicity of binary HVs, recent analysis in [241] showed that BSC cannot approximate an arbitrary similarity matrix and, consequently, cannot always learn a Bayes-optimal classifier. This limitation becomes increasingly relevant as we move from simple tasks toward more complex learning problems, and it motivates the adoption of richer representations, including real- or complex-valued hypervectors, that provide greater expressive power [5]. Yet, these high-precision datatypes are typically ill-suited for embedded and low-power systems due to their computational and memory requirements.

To bridge this gap, this chapter explores the *Modular Composite Representation* (MCR), a computing model that represents information using high-dimensional integer vectors and modular arithmetic. Originally proposed as a generalization of BSC, MCR aims to deliver higher representational power while remaining substantially lighter than models based on high-precision components. Despite this promise, MCR has received limited attention in the literature: systematic analyses of its trade-offs and direct comparisons with alternative HDC/VSA models—such as binary spatter codes, multiply-add-permute, and Fourier holographic reduced representation—are still lacking. This gap has sustained the perception that MCR’s added arithmetic complexity may not justify its improved expressivity.

In this chapter, we present the first extensive study of MCR, guided by the following research questions:

❓ Research Question

1. When and why is MCR beneficial?
2. What are its execution time and energy consumption when implemented in dedicated hardware?

We answer these questions across three complementary dimensions. First, in Section 13.4.1, we show that MCR achieves substantially higher *information capacity* than binary and low-precision integer hypervectors, while approaching the expressiveness of complex-valued models at a fraction of the memory footprint. Second, in Section 13.4.2, through large-scale experiments on 123 classification datasets, we demonstrate that *per-component precision matters*: the modular discretized space of MCR consistently outperforms binary and low-precision integer hypervectors under both equal-dimensionality and equal-memory-footprint settings, and can reduce the required hypervector dimensionality by up to $4\times$ while still surpassing binary models. Finally, in Section 13.4.3, we show that the modular arithmetic at the core of MCR maps naturally to digital hardware, and we design the first *dedicated accelerator for MCR*, extending the HDCU (Chapter 12) [209]. Hardware experiments on the basic operations and seven classification datasets provide detailed results on execution time, energy efficiency, and scaling behavior. Moreover, by directly comparing hardware-accelerated MCR with hardware-accelerated binary models (Section 13.4.4), we show that although MCR requires more sophisticated arithmetic, its reduced hypervector dimensionality paired with efficient hardware support can translate into a faster, more energy-efficient, more compact, and higher-precision alternative to existing HDC/VSA models. The rest of this chapter is mainly based on Paper 10 [341].

The chapter is organized as follows. Section 13.1 reviews the MCR model and related work. Section 13.4.1 analyzes information capacity. Section 13.4.2 presents classification experiments across 123 datasets. Section 13.4.3 introduces the MCR-HDCU accelerator. Section 13.4.4 evaluates hardware performance. Finally, Section 13.5 summarizes the findings.

13.1 Modular Composite Representation

MCR is an HDC/VSA model introduced in [3] where information is represented with vectors consisting of D integer-valued components. MCR relies on modular arithmetic to define its operations. Each component of an HV lies on a discretized unit circle $\mathbb{Z}_r$, where r is the chosen modulus, so that values range from 0 to $r - 1$ and incrementing $r - 1$ wraps around to 0. Modular vectors form a *homogeneous space* in the sense that for any two vectors $\mathbf{h}$ and $\mathbf{u}$, there exists a third vector $\mathbf{c}$ such that binding $\mathbf{h}$ with $\mathbf{u}$ produces $\mathbf{c}$. Fig. 13.1 illustrates an example of this discretized space with $r = 16$, where the complement of a value is defined such that their sum equals r , and the opposite is obtained by adding $r/2$.

Within this space, MCR defines binding between two HVs $\mathbf{h}$ and $\mathbf{u}$ as the component-wise modular sum ($c_i = \text{mod}_r(h_i + u_i)$) and unbinding as the component-wise modular subtraction ($c_i = \text{mod}_r(h_i - u_i)$). Similarity is measured by a modular

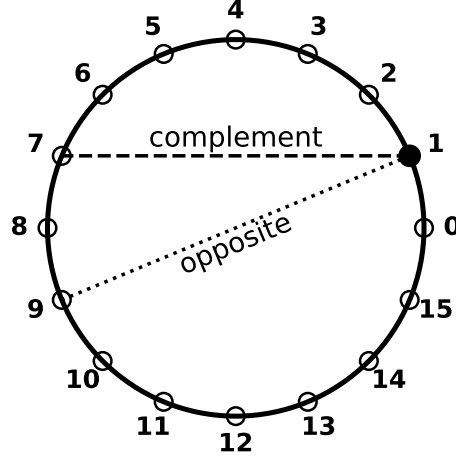


Figure 13.1 Redrawn illustration from [3] showing the possible discretized values in $\mathbb{Z}_{16}$ for a single component of an MCR HV with $r = 16$. The *complement* of a value is defined as the element whose sum with it equals r , while the *opposite* is its antipode, i.e., the value obtained by adding $r/2$.

variant of the Manhattan distance that is defined as follows:

$$\delta(\mathbf{h}, \mathbf{u}) = \sum_{i=1}^D \min(\text{mod}_r(h_i - u_i), \text{mod}_r(u_i - h_i)). \quad (13.1)$$

The superposition (also called bundling) of m HVs $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(m)}$ is carried out in two steps. First, each component $h_i^{(j)} \in [0, r - 1]$ of the j -th HV is projected to an equivalent point on the unit circle (cf. Fig. 13.1), and summed in $\mathbb{C}$:

$$v_i = \sum_{j=1}^m \left(\cos\left(\frac{2\pi}{r} h_i^{(j)}\right) + i \sin\left(\frac{2\pi}{r} h_i^{(j)}\right) \right). \quad (13.2)$$

Next, since the resultant HV $\mathbf{v}$ no longer lies on the discretized unit circle $\mathbb{Z}_r$, its components are mapped back to the nearest elements in $\mathbb{Z}_r$ by finding integers from $[0, r - 1]$ with phases that are nearest to the phases of components of the resultant HV:

$$w_i = \text{mod}_r\left(\left\lfloor \frac{r}{2\pi} \text{mod}_{2\pi}(\text{atan2}(\Im(v_i), \Re(v_i))) \right\rfloor\right). \quad (13.3)$$

If one component happens to be zero (e.g., when an element is superimposed with its inverse), its undefined phase is resolved by assigning it to the integer closest to the arithmetic mean of the original arguments in the integer domain:

$$w_i = \text{mod}_r\left(\left\lfloor \frac{1}{m} \sum_{j=1}^m h_i^{(j)} \right\rfloor\right). \quad (13.4)$$

Notably, each discretization step (hereafter referred to as *normalization*) results in information loss. Therefore, when superimposing multiple HVs, the normalization in Eq. (13.3) should be deferred until the final step [3].

MCR can be seen as a middle-ground between the properties and advantages of BSC and FHRR. It was originally proposed as a generalization of BSC, inheriting its low memory requirements [3]. In fact, when $r = 2$, MCR is equivalent to BSC. For $r > 2$, however, its operations generalize the modular arithmetic of BSC beyond $\mathbb{Z}_2$. At the same time, under the interpretation of integers in $[0, r - 1]$ as phasors on the unit circle (the r th roots of unity), the binding and superposition operations of MCR implement a discretized FHRR, offering comparable expressiveness while being more computationally efficient and easier to implement with LUTs (as we report in Section 13.4.3). In this sense, MCR can be seen as a model covering the whole spectrum between BSC and FHRR, balancing expressiveness and efficient implementation. In line with this discretized-phase view, Yu et al. introduced the Cyclic Group Representation (CGR) model [241], which, similar to MCR implements the binding operation as the component-wise modular sum. Analysis in [241] shows that certain similarity functions cannot be expressed by binary HVs but can be realized within CGR; accordingly, MCR with $r > 2$ also possesses this broader expressivity. Furthermore, modules of MCR/CGR can be combined to form residue number systems [342], which has promising applications to unconventional computing [343] and neuroscience [344, 345].

No prior work has systematically investigated MCR. Beyond being briefly mentioned in a survey [1] and implemented in software [346], its performance on real datasets and potential for hardware implementation have not been thoroughly evaluated. The existing literature has predominantly focused on the two “extreme” points of the design spectrum, BSC and FHRR, while largely overlooking integer-valued alternatives. Perhaps this was because the added computational cost of such designs seemed unjustified relative to their potential gains over binary HVs. Even the original study [3] conjectured that the simpler operations of BSC might be preferable in resource-constrained scenarios with strict timing requirements. In this study, we revisit that conjecture by systematically analyzing the algorithmic performance and hardware complexity of MCR, demonstrating that it offers a favorable trade-off between HV dimensionality and component dynamic range compared to many existing HDC/VSA models.

13.2 Capacity of hypervectors

In HDC/VSA, the *information capacity* is measured as the maximum amount of information (bits) that can be decoded from an HV representing a data structure.

We adopt the setup from [347–349] to evaluate the information capacity of MCR and compare it with the well-known BSC, MAP-I, and FHRR models. Given a codebook matrix Φ of d symbols, where each column Φ_j is a fixed D -dimensional HV representing symbol j , we uniformly sample a sequence $\mathbf{s} = (s_1, \dots, s_m)$ of length m and construct its composite HV as follows:

$$\phi(\mathbf{s}) = \sum_{j=1}^m \rho^{m-j}(\Phi_{s_j}), \quad (13.5)$$

where $\rho(\cdot)$ is a permutation operation representing the position of each symbol within the sequence.

The decoding task then consists of reconstructing the original sequence $\mathbf{s}$ from $\phi(\mathbf{s})$, producing an estimate $\hat{\mathbf{s}}$ as close as possible to $\mathbf{s}$. This can be done using various decoding techniques (see, e.g., [348, 349]). In this study, we use the simplest one – *Codebook decoding*, where each symbol in the codebook is compared to the permuted composite HV $\rho^{-(m-j)}(\phi(\mathbf{s}))$ using an appropriate distance metric, δ , and the symbol with the lowest distance is selected as the estimate $\hat{s}_j$, as in Eq. (13.6). This step is repeated for all positions $j \in [1, m]$ to decode the complete sequence $\hat{\mathbf{s}}$.

$$\hat{s}_j = \arg \min_{c \in \{1, \dots, d\}} \delta(\Phi_c, \rho^{-(m-j)}(\phi(\mathbf{s}))). \quad (13.6)$$

Following the setup in [348, 349], the decoding performance and, thus, the capacity of HVs, are evaluated in terms of two performance metrics. The first metric is the *decoding accuracy*, defined in Eq. (13.7) as the proportion of correctly decoded symbols, averaged over g trials.

$$a = \frac{1}{gm} \sum_{k=1}^g \sum_{j=1}^m \mathbb{I}[s_j^{(k)} = \hat{s}_j^{(k)}]. \quad (13.7)$$

The second metric is the *information rate*. For a given decoding accuracy a and codebook size d , the amount of information decoded per symbol can be computed as follows:

$$I_{\text{symb}}(a, d) = a \log_2(da) + (1 - a) \log_2\left(\frac{d}{d-1}(1 - a)\right). \quad (13.8)$$

The *total information* decoded from a sequence of length m can then be calculated as:

$$I_{\text{tot}} = m I_{\text{symb}}(a, d). \quad (13.9)$$

Normalizing with respect to the dimensionality D yields the *information rate per component* (I_{dim}):

$$I_{\text{dim}} = \frac{I_{\text{tot}}}{D}. \quad (13.10)$$

Furthermore, we also report the *information rate per storage bit* (I_{bit}), Eq. (13.11), where b denotes the number of bits required per HV component. This metric accounts for both accuracy and memory footprint and is, therefore, suitable for comparing models with components requiring different precision, such as binary, integer-, real- or complex-valued.

$$I_{\text{bit}} = I_{\text{tot}}/(Db). \quad (13.11)$$

13.3 Classification with HVs

In this study, we compare the classification performance of different HDC/VSA models across a collection of classification datasets with tabular data. Therefore, we use transformation for key–value pairs to represent feature vectors as HVs. For tabular data, this transformation has been shown to achieve the best trade-off between execution time and accuracy in this setting [218, 317]. Given a d -dimensional feature vector, $\mathbf{x} = [x_1, x_2, \dots, x_d]$, each feature j is assigned a random HV $\mathbf{r}^{(j)}$

acting as a key. This HV is then bound to a value vector ($\psi(x_j)$) that is obtained via the quantization of the corresponding feature value x_j and mapping it into the high-dimensional space using the thermometer code [237, 350]. In MCR, the thermometer code is realized by assigning 0 and $r/2$ as the two extreme values. Such mapping ensures that similar feature values are transformed into similar HVs. The outcome of the bindings is d key-value pairs, which are then superimposed, resulting in the composite HV $\phi(\mathbf{x})$ representing the feature vector:

$$\phi(\mathbf{x}) = \sum_{j=1}^d \mathbf{r}^{(j)} \circ \psi(x_j). \quad (13.12)$$

To train the classifier, we adopt a prototype-based Learning Vector Quantization (LVQ) approach [351, 352]. During the first epoch, prototypes are initialized using a simple centroid classifier [353–355] where all HVs belonging to the same class are aggregated into a class prototype:

$$\mathbf{p}_c = \sum_{k:y^{(k)}=c} \phi(\mathbf{x}^{(k)}). \quad (13.13)$$

From the second epoch onward, LVQ2.1 [351, 356] is used according to Eq. (13.14), where two prototypes are updated per training sample: the correct prototype $\mathbf{p}_+$ and the closest incorrect prototype $\mathbf{p}_-$. For a sample $\phi(\mathbf{x})$ with label c , distance is computed using the appropriate model-specific metric: Hamming for BSC, cosine for MAP, modular Manhattan for MCR, Eq. (13.1).

$$\begin{cases} \mathbf{p}_+ \leftarrow \mathbf{p}_+ + \epsilon(\phi(\mathbf{x}) - \mathbf{p}_+) \\ \mathbf{p}_- \leftarrow \mathbf{p}_- - \epsilon(\phi(\mathbf{x}) - \mathbf{p}_-). \end{cases} \quad (13.14)$$

Here ϵ is the learning rate. The learning rule is triggered only if $\phi(\mathbf{x})$ lies inside the LVQ2.1 window:

$$\min\left(\frac{\delta(\phi(\mathbf{x}), \mathbf{p}_-)}{\delta(\phi(\mathbf{x}), \mathbf{p}_+)}, \frac{\delta(\phi(\mathbf{x}), \mathbf{p}_+)}{\delta(\phi(\mathbf{x}), \mathbf{p}_-)}\right) > s, \quad s = \frac{1 - \omega}{1 + \omega}, \quad (13.15)$$

where in the conducted experiments ω is set to 0.1 following [351]. To prevent the uncontrolled growth of prototypes' norms during iterative updates and ensure the correct operation of LVQ2.1, L2-norms of all class prototypes are reset to one after each training epoch.

To minimize the loss of precision, we maintain high intermediate precision (floating-point) during training and normalize class prototypes only after the end of training. For BSC, this corresponds to the sign function. For MAP-I with low precision, we first rescale the distribution of prototypes' values to match the dynamic range of the targeted b -bit interval $[-2^{b-1}, 2^{b-1} - 1]$, and then apply uniform quantization. For MAP-C32 (32 bits per component), no normalization is applied, maintaining prototypes in floating-point precision. For MCR, prototype updates are accumulated in the complex domain, Eq. (13.2), and then discretized back to $\mathbb{Z}_r$ as described by Eq. (13.3).

During inference, each test sample is represented following Eq. (13.12) and classified using the class label of the nearest prototype as follows:

$$\hat{y}(\phi(\mathbf{x})) = \arg \min_c \delta(\phi(\mathbf{x}), \mathbf{p}_c). \quad (13.16)$$

13.4 Results

13.4.1 Capacity analysis

Following the experiment design introduced in Section 13.2, we perform experiments to measure the information capacity of MCR and several other well-known HDC/VSA models: BSC, MAP-I, and FHRR.

Fig. 13.2 reports the obtained results for three different codebook sizes ($d \in \{5, 15, 100\}$), using HVs of size $D = 500$ and sequence lengths ranging from $m = 10$ to $m = 400$, with analysis performed following the methods reported in [348, 349]. We compare MCR with different quantizations ($r \in \{4, 8, 16\}$, corresponding to 2, 3, and 4 storage bits per component, respectively), BSC (1 bit per component), MAP-I (2, 3, 4, and 32 bits per component), and FHRR (128 bits per component, with 64 bits each for the real and imaginary parts). For MCR, MAP-I, and BSC, to minimize information loss, the composite HV $\phi(\mathbf{s})$ was first constructed using the full-precision accumulation, and only afterwards normalized back to the target precision. The results are averaged over 20 independent random codebooks, with 50 randomly generated sequences for each considered sequence length m .

As expected, the results demonstrate that MCR drastically outperforms BSC as the sequence length increases, with an average improvement in decoding accuracy of 25.5% across sequence lengths from 10 to 400 and the three codebook sizes. More impressively, each MCR variant outperforms the corresponding MAP-I variant with matching precision, yielding average gains of 12.01% for 3 bits, 10.80% for 4 bits, and 8.88% for 5 bits. This positions *the modular space of MCR as a superior choice* over MAP-I quantization for memory-constrained environments. Surprisingly, MCR even performs significantly better than the unconstrained MAP model with 32 bits per component, with an average accuracy gain of 7.63%. Although this may appear counterintuitive, the difference in performance comes from the different properties of superposition adopted by the two models. While MAP relies on a simple integer sum (in effect, only a real part), MCR interprets integers as discretized phasors on the unit circle, and then performs vector addition in $\mathbb{C}$. The greater expressivity in the complex plane preserves more information during superposition and explains why MCR achieves higher capacity. This fact is also consistent with the strong performance of FHRR decoding, which operates in the same manner except for quantization. FHRR has slightly higher decoding accuracy than MCR, although at the cost of requiring many more bits, as we will see. Notably, increasing r yields diminishing returns in MCR: although higher r reduces phase quantization error, the normalization step discards magnitude, preventing MCR from fully converging to FHRR.

When the results are analyzed in terms of information per bit, MCR clearly outperforms all other HDC/VSA models. Despite its high capacity, FHRR has the lowest I_{bit} due to the 128 bits required per component. For the same reason, MAP-I32 is the second lowest. MAP-I variants with low precision improve memory efficiency by using only three to five bits, but they still underperform compared to 1-bit BSC. In contrast, MCR combines high capacity with low precision, consistently achieving the *highest information-per-bit rate* across all settings.

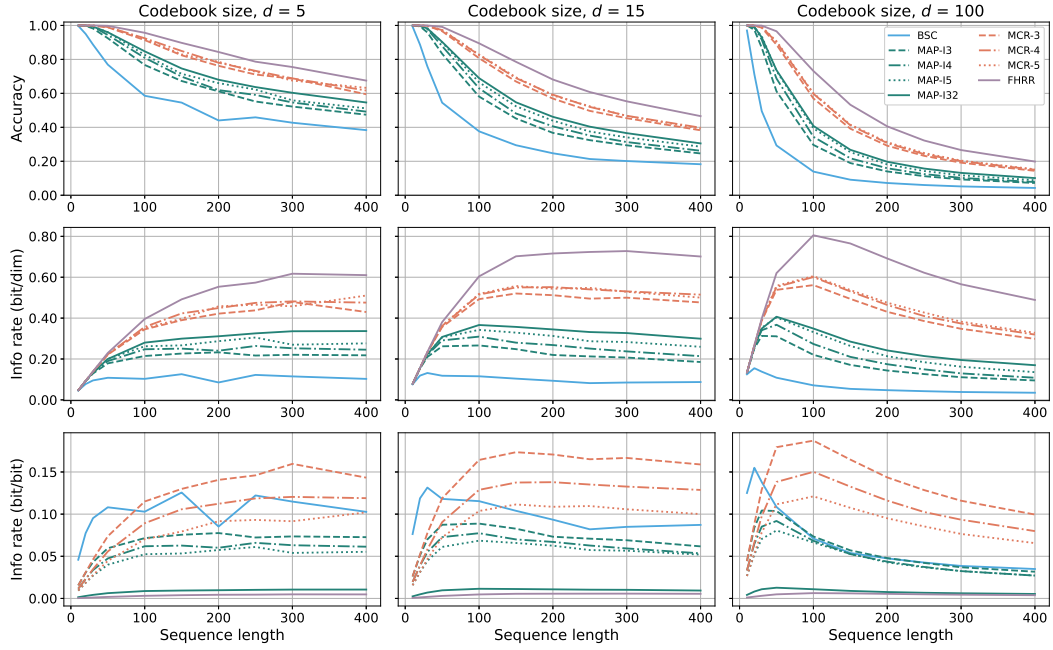


Figure 13.2 Decoding performance and information capacity analysis for MCR compared to BSC, MAP-I, and FHRR. Results are shown for three different codebook sizes ($d = 5, 15, 100$), using HVs of size $D = 500$ and sequence lengths ranging from $m = 10$ to $m = 400$, averaged over 20 independent codebooks with 50 test sequences per length. MCR variants achieve higher capacity than BSC and MAP-I variants and approach the performance of FHRR while maintaining much greater memory efficiency.

13.4.2 Classification performance

To evaluate the performance of MCR on real datasets, we carried out a large-scale study on a collection of 123 classification datasets from [218]. This collection is based on a popular collection from [357] representing a subset of 121 datasets from the UCI Machine Learning repository [358]. We compare MCR-4 to BSC (1 bit), MAP-I4, and MAP-C32. The objective of this analysis is to understand whether the higher per-component precision and higher capacity of MCR translate into tangible improvements in classification accuracy when applied to diverse data, and whether it provides a better trade-off under strict memory constraints.

For our experiments, we used an HV dimensionality $D = 1024$, key-value transformation with 1024 quantization levels generated using the thermometer code, 10 training epochs, and LVQ2.1 with the default parameters: $\omega = 0.1$ and $\epsilon = 0.01$. We averaged the results over 20 independent runs per dataset.

Fig. 13.3 reports the results for MCR-4 compared to the baseline models, plotting the average accuracy versus the memory footprint per class prototype (bits $\times$ dimension). The results confirm that MCR-4, with its higher per-component precision, ($D = 1024$, $b = 4$) outperforms BSC ($D = 1024$, $b = 1$), achieving an average gain of +4.84%. More importantly, MCR-4 also outperforms MAP-I4 ($D = 1024$, $b = 4$) under the same memory footprint with an average gain of +0.61% and approaches the accuracy of full-precision MAP-C32 (-1.57%) while requiring

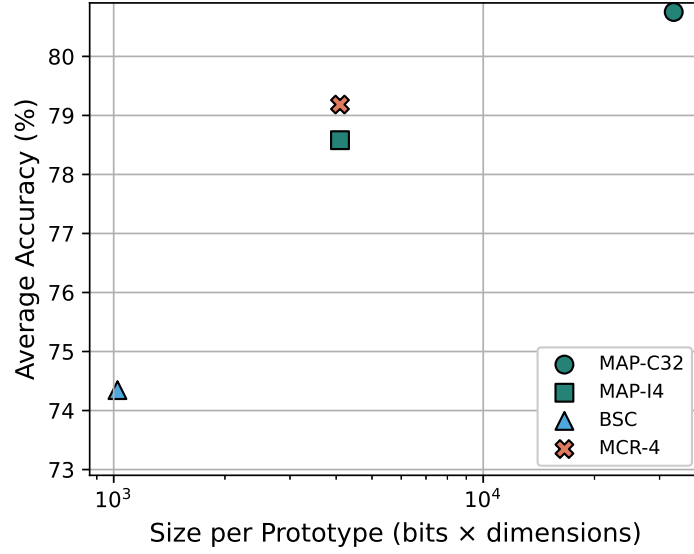


Figure 13.3 Average accuracy across 123 classification datasets for 1-bit BSC, MAP-I4, MCR-4 and MAP-C32, using HVs of size $D = 1024$. MCR outperforms BSC and MAP-I4 and approaches MAP-C32.

only a quarter of its memory footprint. A complementary view of these results is shown in Fig. 13.4, which compares the relative accuracy of MCR-4 and MAP-I4 on each dataset. Most points lie above the diagonal, confirming that MCR provides systematic improvements.

To further investigate the accuracy–memory trade-off, Fig. 13.5 presents the results for MCR-4 with a reduced number of components in HVs. When D is reduced to 256, MCR-4 has the same memory footprint as BSC ($D = 1024$, $b = 1$), MCR-4 still outperforms the binary model with an average improvement of +3.94%. Even more impressively, MCR-4 remains robust when dimensionality is further decreased: with only 64 components, MCR-4 still surpasses BSC (+1.14%) while requiring 75% less memory.

These results clearly demonstrate that *per-component precision matters*: the few-bit quantization defined by MCR enables superior accuracy–memory efficiency compared to simple binarization or restricting the integer range with the clipping function. It also supports a substantial reduction in dimensionality while maintaining competitive classification performance.

13.4.3 Hardware design for MCR

The previous sections demonstrated that MCR achieves a superior trade-off between memory footprint, classification accuracy, and information capacity compared to other well-known HDC/VSA models. However, a natural concern is whether these improvements come at the expense of additional computational costs. Indeed, the original MCR study [3] conjectured that BSC, with its simpler operations, could be preferable for resource-constrained scenarios. This raises an important question:

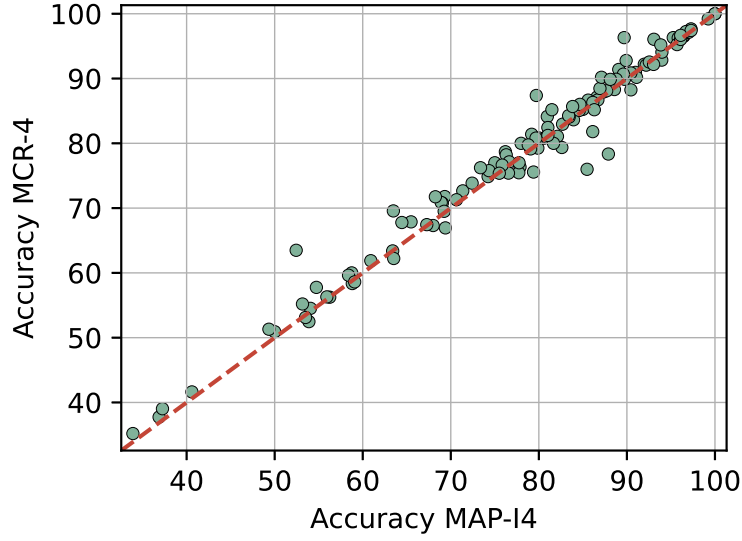


Figure 13.4 Scatter plot comparing classification accuracy of MAP-I4 (x -axis) against MCR-4 (y -axis) across 123 classification datasets. Each point corresponds to one dataset. The dashed diagonal represents equal performance between the two models: the points above the diagonal indicate datasets where MCR-4 outperforms MAP-I4, while the points below indicate the opposite.

can MCR-specific hardware design solve the impact of the additional computational cost of MCR on execution time and energy consumption? In this section, we provide a solution by moving from software simulations to hardware design, showing that the arithmetic of MCR naturally maps onto digital circuits and presenting the first dedicated hardware accelerator for this HDC/VSA model.

Hardware-friendly MCR arithmetic

There are two aspects that might initially be perceived as computationally expensive in MCR: the use of modular reductions and the reliance on trigonometric functions to perform normalization.

Modular reductions As described in Section 13.1, the binding, unbinding, and distance operations in MCR require applying a modular reduction (mod_r) for every HV component to remain in $\mathbb{Z}_r$. This frequent operation might seem costly, since modulo is normally computed in hardware by a divider unit: division is neither associative nor distributive, making parallelization difficult and leading to high area and power costs even with optimized dividers (such as the VLNPD introduced in Chapter 3). However, when implemented in digital hardware with r chosen as a power of 2, modular reduction actually comes for free: if each component is stored using $b = \lceil \log_2(r) \rceil$ bits, the modulo operation is automatically performed by binary overflow. For example, with $r = 16$, storing each component with 4 bits ensures that $14 + 5 = 19$ overflows to 3, which is exactly $\text{mod}_{16}(19)$. As a result, *binding is reduced to integer addition, unbinding to integer subtraction, and distance computation to*

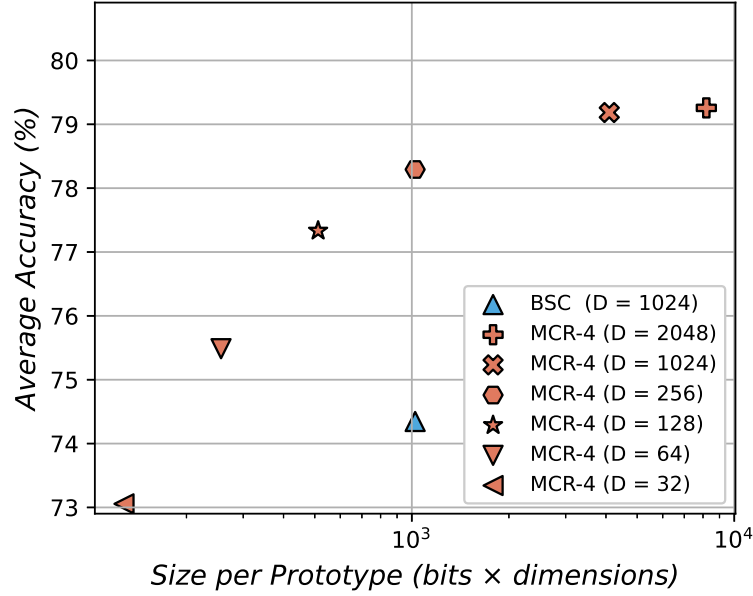


Figure 13.5 Average classification accuracy across 123 classification datasets for MCR-4 with different D compared to BSC ($D = 1024$, $b = 1$). With the same memory footprint, MCR-4 ($D = 256$, $b = 4$) outperforms BSC by +3.94% on average. Even at $D = 64$, requiring only a quarter of BSC’s memory, MCR-4 still achieves a +1.14% improvement, highlighting its superior accuracy–memory trade-off.

two subtractions followed by a min operation. These operations are lightweight, fully parallelizable, and incur no additional cost beyond the standard integer arithmetic.

Trigonometric mappings Another potentially costly step is the involvement of trigonometric functions. First, during superposition, each component $h_i^{(j)} \in [0, r - 1]$ must be mapped to a unit vector in $\mathbb{R}^2$ using cos and sin functions. In software, this requires repeated evaluations of trigonometric functions, which are computationally expensive. In hardware, however, the discretized modular space makes this step efficient: all cos and sin values for all values in $\mathbb{Z}_r$ can be pre-computed and stored in compact fixed-point LUTs, so that each mapping reduces to a single memory lookup followed by a fixed-point accumulation. This completely removes the need for trigonometric evaluation during runtime.

Second, during the normalization, the accumulated HV in $\mathbb{C}$ must be projected back to the nearest integer in $\mathbb{Z}_r$ using atan2, Eq. (13.3). A direct hardware realization of this step is costly, as it requires fixed-point division and nonlinear functions. Division, as already discussed, is slow and difficult to parallelize. Similarly, approximating atan2 by computing the ratio $\Im(v_i)/\Re(v_i)$ and mapping it through a LUT demands a relatively large memory, since the ratio may assume many values depending on fixed-point precision, creating an unfavorable trade-off between accuracy, area, and power consumption. To overcome these limitations in the hardware realization, two approaches can be adopted. The first one is the common CORDIC algorithm, which computes $\text{atan2}(\Im(v_i), \Re(v_i))$ iteratively using only shift-

and-add operations and a small LUT of arctangent constants. The second one, proposed in this work, is a winner-take-all (WTA)-based approach:

$$w_i = \arg \max_{k \in [0, r-1]} (\Re(v_i) \cos\left(\frac{2\pi}{r}k\right) + \Im(v_i) \sin\left(\frac{2\pi}{r}k\right)). \quad (13.17)$$

The key idea is to compare v_i against all r discrete directions $(\cos \frac{2\pi}{r}k, \sin \frac{2\pi}{r}k)$ retrieved from the LUTs and select the one with the highest inner product. Importantly, since the quadrant of v_i can be identified directly from the sign bits of $\Re(v_i)$ and $\Im(v_i)$, the search must cover only $r/4 + 1$ values in the corresponding quadrant, rather than all r directions. This reduces the normalization to $r/4 + 1$ inner product computations (two multiplications and one addition each), followed by the search for an argument of the maximum. In discretized spaces, this normalization procedure is highly efficient and avoids division. Unlike CORDIC, it is fully parallelizable across all candidate directions while being more compact and not requiring additional LUTs.

MCR-HDCU

Building on the foundation of HDCU, we design MCR-HDCU by preserving its integration strategy while replacing the binary datapath with one that is tailored for modular operations, achieving software-controlled and efficient acceleration while retaining full flexibility and programmability. To the best of our knowledge, this represents the first dedicated hardware accelerator for MCR.

Fig. 13.6 depicts a high-level view of the MCR-HDCU microarchitecture. The accelerator features dedicated pipelines and optimized functional units (FUs) for each MCR arithmetic operation. As in the original HDCU, each FU can be selectively enabled or disabled at synthesis time, and its degree of hardware parallelism can be configured through the **SIMD** (Single Instruction Multiple Data) parameter. The design of each functional unit in MCR-HDCU incorporates the optimized arithmetic strategies introduced in Section 13.4.3, ensuring efficient implementation of the modular operations of MCR.

Each HV component is represented using $b = \lceil \log_2(r) \rceil$ bits, where r is a synthesis-time parameter. With this representation, modular wrap-around is naturally handled by binary overflow, eliminating explicit modulo operations and directly wrapping in the $\mathbb{Z}_r$ ring, as discussed in Section 13.4.3.

To sustain parallel HV operations, the accelerator integrates local dedicated Scratchpad Memories (SPMs) that provide low-latency, high-bandwidth access. The number and size of these SPMs are configurable at synthesis time. By default, four SPMs are instantiated: three with bandwidth $b\text{SIMD}$, sustaining the transfer of **SIMD** components per cycle, and one with higher bandwidth $\text{FP} \cdot \text{SIMD}$ dedicated to the Superposition Unit, which requires streaming **SIMD** fixed-point components per cycle for Cartesian accumulation, where **FP** denotes the fixed-point precision. When the target learning task is known in advance, the SPMs dimension can be set precisely to the required number of HVs; for more general-purpose deployments, larger SPMs may be synthesized to preserve flexibility. Data exchange between the host core and the SPMs is supported by dedicated **hvmemld** and **hvmemstr** instructions [209], though in typical workloads, the accelerator operates autonomously on locally stored

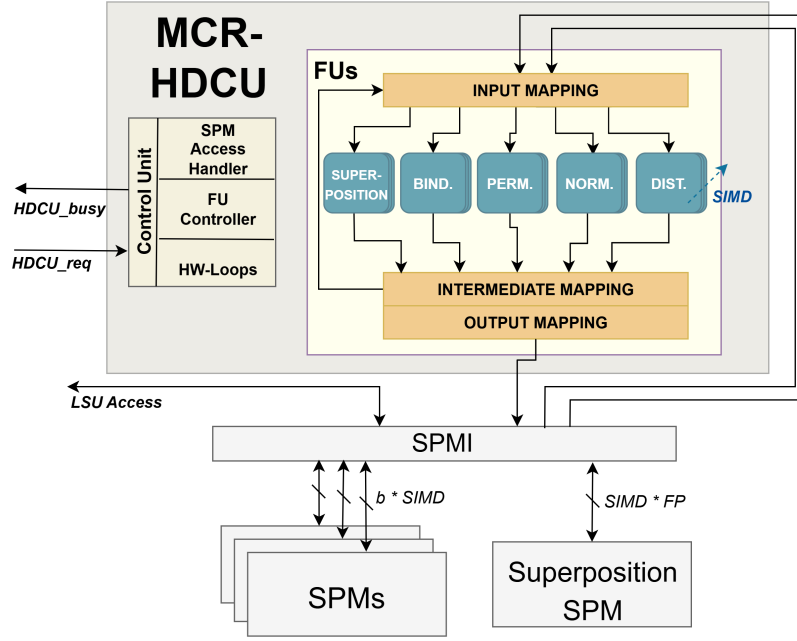


Figure 13.6 High-level schematic of the MCR-HDCU microarchitecture. The design integrates dedicated functional units for each arithmetic operation in MCR, running in parallel over **SIMD** lanes. Specialized Scratchpad Memories ensure low-latency, high-bandwidth access. The hardware parallelism and memory configuration (number and size of Scratchpad Memories) can be configured at synthesis time.

Table 13.1 MCR-HDCU Configuration Parameters

Parameter	Configuration Time
SPM Size	Synthesis
SPM Number	Synthesis
Hardware Parallelism (SIMD)	Synthesis
Functional Unit Enable/Disable	Synthesis
MCR Modulo (r)	Synthesis
Fixed-point Precision (FP)	Synthesis
HVDIM	Runtime
HVCLASS	Runtime

HVs. Table 13.1 summarizes the parameters that can be tuned at the synthesis time and at runtime in MCR-HDCU.

Next, we describe the implementation of each arithmetic operation in the MCR-HDCU accelerator.

Binding By representing HV components using b bits, binding is implemented as a simple modular addition, with the **SIMD** parameter defining how many adders operate in parallel. The same holds for unbinding, which is a modular subtraction. Since additions/subtractions are fully parallelized over the **SIMD** lanes, the total

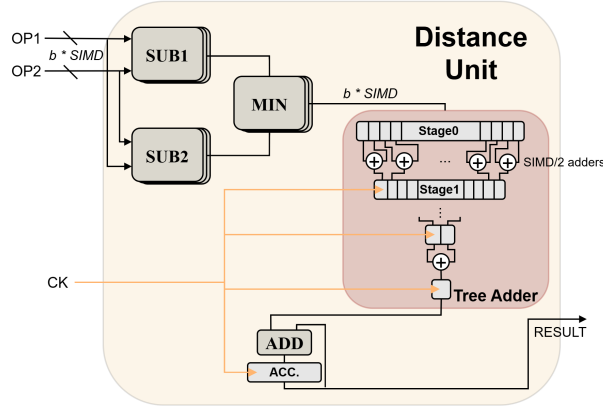


Figure 13.7 Architecture of the Distance Unit. The design computes 2SIMD differences in parallel, selects the modular distance through a `min` circuit, and accumulates results using a parallel tree adder.

latency in clock cycle (L) is

$$L_{\text{bind}} = L_{\text{unbind}} = \frac{\text{HVDIM}}{\text{SIMD}} \text{ cycles}$$

Distance metric In MCR, distance is computed as the angular difference across all components of two HVs, as defined in Eq. (13.1). As in binding, modular reduction is inherently handled by the binary overflow. Accordingly, the unit computes 2SIMD differences in parallel using subtractors, and for each pair of results, a lightweight `min` circuit selects the correct distance. The resulting per-component distances are then accumulated across SIMD lanes by a parallel tree adder of depth $\lceil \log_2(\text{SIMD}) \rceil$, as illustrated in Fig. 13.7. The total latency of the Distance Unit is thus:

$$L_{\text{sim}} = \frac{\text{HVDIM}}{\text{SIMD}} + \lceil \log_2(\text{SIMD}) \rceil \text{ cycles}$$

where the first term accounts for the selected hardware parallelism and the second for the depth of the tree adder.

Superposition This operation requires first mapping each HV component to its equivalent vector, then performing vector addition, and finally normalizing the accumulated result back onto the discrete ring $\mathbb{Z}_r$. As highlighted in Section 13.1, the normalization step should only be applied once after all operands have been accumulated, since repeated quantization would cause excessive loss of stored information. For this reason, in MCR-HDCU we *decouple superposition and normalization* into two separate instructions and FUs, enabling intermediate accumulation in fixed-point precision before the final projection back onto $\mathbb{Z}_r$.

The Superposition Unit is built around two compact LUTs storing precomputed $\cos(\frac{2\pi}{r}k)$ and $\sin(\frac{2\pi}{r}k)$ values for $k \in [0, r-1]$, that as highlighted in Section 13.4.3, are small and hardware-friendly. The FP parameter can be specified at synthesis time to balance performance and hardware cost. To maintain the desired precision

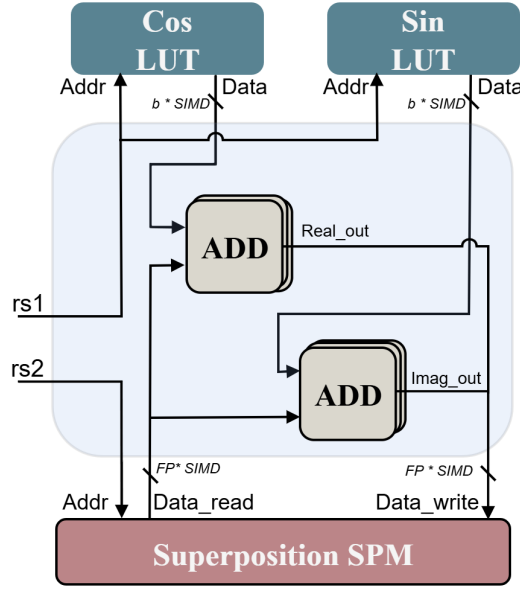


Figure 13.8 Architecture of the Superposition Unit. Each component is mapped to Cartesian form via LUT-based cos / sin values and accumulated in fixed-point precision using dedicated high-bandwidth SPM access.

in cumulative superposition, the unit assumes that the first input HV is already in fixed-point Cartesian form and produces a fixed-point result. The second input, represented in $\mathbb{Z}_r$, is mapped on-the-fly to Cartesian coordinates using cosine and sine LUTs. At each cycle, the unit fetches $\text{SIMD}/2$ real parts and $\text{SIMD}/2$ imaginary parts of the first HV from the dedicated high-bandwidth SPM, while the second HV remains fixed for two cycles. The corresponding cosine values are added to the real components of the first HV, and the sine values to its imaginary components. The results are written back to the SPM at a throughput of $\text{SIMD}/2$ complex components per cycle. The total latency of the operation is therefore:

$$L_{\text{superimpose}} = \frac{2\text{HVDIM}}{\text{SIMD}} \text{ cycles}$$

Normalization The Normalization Unit performs the normalization step, projecting the results of superposition that have accumulated in Cartesian coordinates back onto the discrete ring representing $\mathbb{Z}_r$. As outlined in Section 13.4.3, this is achieved through the WTA-based approach, which avoids the costly division and atan2 computations otherwise required by direct normalization.

Given a superimposed component in Cartesian form $(\Re(v_i), \Im(v_i))$, the unit identifies its quadrant from the sign bits of $\Re(v_i)$ and $\Im(v_i)$ and then compares it against the $r/4 + 1$ directions in that quadrant with coordinates $(\cos \frac{2\pi}{r}k, \sin \frac{2\pi}{r}k)$ precomputed and stored in the same compact LUTs used by the Superposition Unit. Each comparison is implemented as an inner product, requiring two fixed-point multiplications and one addition. The accumulated Cartesian HV is stored in a dedicated high-bandwidth SPM that provides simultaneous access to $\text{SIMD}/2$ real and

Table 13.2 Hardware requirements of MCR-HDCU for different SIMD.

SIMD	Freq.	Module	#LUTs	#FFs	#DSPs	#BRAMs
8	150 MHz	Accelerator	1707	960	9	0
		SPMI	726	282	0	16
16	125 MHz	Accelerator	3139	1671	17	0
		SPMI	1930	486	0	16
32	115 MHz	Accelerator	5823	3098	33	0
		SPMI	3065	887	0	20
64	118 MHz	Accelerator	13609	8564	65	0
		SPMI	7193	1693	0	28

SIMD/2 imaginary parts per cycle. As a result, the total latency of the Normalization Unit is

$$L_{\text{norm}} = \frac{2\text{HVDIM}}{\text{SIMD}}(r/4 + 1) \text{ cycles}$$

Permutation Permutation in HDC/VSA produces HVs that are nearly orthogonal to their input HVs. Implementing arbitrary rotations in hardware while processing SIMD components in parallel would require complex shuffling logic and intermediate buffers, leading to high area and latency overheads. In this work, we use the insight that even structured permutations (e.g., circular shift or block-cyclic permutation) produce HVs that are approximately orthogonal to input HVs. Using this observation, MCR-HDCU performs permutation at the granularity of whole SIMD blocks. Rather than shifting components within each block, the design cyclically reorders entire blocks by remapping memory addresses. With this design choice, permutation reduces to pure address manipulation in the SPM: read addresses are offset by multiples of SIMD lanes, and the HV is streamed directly in permuted form. This yields HVDIM/SIMD distinct permutations, corresponding to all possible block-level cyclic shifts, while keeping the implementation extremely lightweight. The total latency of the permutation is:

$$L_{\text{perm}} = \frac{\text{HVDIM}}{\text{SIMD}} \text{ cycles.}$$

Search The final operation implemented in MCR-HDCU is *search*, which is essential during inference to identify the closest prototype among a set of HVs. Implementing this purely in software would require issuing the distance instruction c times for a c -class problem, limiting the benefits of acceleration. To avoid this overhead, MCR-HDCU integrates a dedicated search instruction that reuses the Distance Unit in a hardware-controlled loop. At runtime, the number of classes is specified as a parameter, enabling the accelerator to iterate over all prototypes in the SPM. The query HV is held constant, while the second operand cycles through each class HV. After each distance computation, the result is compared against the current best match stored in a register, which is updated if necessary. Once the loop completes, the index of the class with the lowest distance is written back to the memory.

Table 13.3 Execution Time in μs and Energy Consumption in μJ per arithmetic operation in MCR

Realization	HVDIM	Binding		Superposition		Normalization		Permutation		Distance	
		Time [μs]	Energy [μJ]	Time [μs]	Energy [μJ]	Time [μs]	Energy [μJ]	Time [μs]	Energy [μJ]	Time [μs]	Energy [μJ]
Software	64	5.90	0.74	13.12	1.64	76.21	9.53	11.90	1.49	4.71	0.59
Software	512	43.63	5.45	102.21	12.78	606.18	75.77	92.90	11.61	35.42	4.43
Software	2048	173.02	21.63	407.58	50.95	2411.27	301.41	380.38	47.55	140.66	17.58
SIMD 8	64	0.15	0.02	0.36	0.04	0.65	0.07	0.15	0.01	0.24	0.02
SIMD 8	512	0.52	0.06	1.85	0.22	4.38	0.53	0.52	0.06	0.99	0.11
SIMD 8	2048	1.80	0.22	6.97	0.93	17.18	2.32	1.80	0.22	3.55	0.44
SIMD 16	64	0.14	0.01	0.31	0.04	0.46	0.05	0.14	0.01	0.23	0.02
SIMD 16	512	0.37	0.04	1.21	0.15	2.70	0.33	0.37	0.04	0.68	0.08
SIMD 16	2048	1.14	0.15	4.28	0.62	10.38	1.53	1.14	0.14	2.22	0.28
SIMD 32	64	0.14	0.02	0.28	0.04	0.32	0.04	0.14	0.02	0.23	0.03
SIMD 32	512	0.26	0.04	0.77	0.12	1.54	0.25	0.26	0.04	0.47	0.06
SIMD 32	2048	0.68	0.10	2.44	0.39	5.71	0.93	0.68	0.10	1.30	0.19
SIMD 64	64	0.13	0.02	0.24	0.04	0.23	0.04	0.13	0.02	0.20	0.03
SIMD 64	512	0.19	0.04	0.48	0.10	0.82	0.17	0.19	0.04	0.33	0.06
SIMD 64	2048	0.39	0.08	1.30	0.27	2.86	0.59	0.39	0.08	0.74	0.15

13.4.4 Performance of MCR-HDCU

This section presents a comprehensive evaluation of the MCR-HDCU accelerator. We begin by reporting hardware synthesis results, including resource utilization and maximum achievable frequency on FPGA, to characterize the hardware requirements of the proposed accelerator and evaluate the cost of different SIMD configurations. We then analyze performance, evaluating the advantages of the proposed MCR-HDCU on both basic HDC/VSA operations and full learning kernels to demonstrate the accelerator’s efficiency and flexibility. Finally, we directly compare MCR and BSC by benchmarking the proposed accelerator against the original binary-only HDCU [209], denoted here as BSC-HDCU.

In the experiments below, all benchmarks are implemented using a custom software library that supports two execution modes: a standard (non-accelerated) configuration compiled for the baseline RISC-V instruction set and executed on the Klessydra-T03 [114] core, and an accelerated configuration compiled with the extended instruction set to exploit the HDCU coprocessor.

Execution time is estimated using cycle-accurate Register-Transfer-Level simulations in QuestaSim, by extracting the number of clock cycles and multiplying them by the post-implementation maximum operating frequency of each configuration. Speedup factors are computed as the ratio between baseline and accelerated execution times. For energy consumption, we perform post-implementation gate-level simulations to generate switching activity files (.saif) for each kernel, which are then analyzed using the Vivado power estimator to derive the dynamic power of both the core and the accelerator.

Hardware realization

To assess the resource requirements and achievable performance of MCR-HDCU, we synthesized the Klessydra-T03 core extended with our MCR-HDCU accelerator on a Xilinx Zynq UltraScale+ ZCU106 (EK-U1-ZCU106-G) device, using Vivado 2023.2. Table 13.2 reports the detailed hardware utilization of MCR-HDCU for SIMD widths ranging from 8 to 64, while fixing the modulus $r = 16$, fixed-point precision FP=16, and using four 2 KB SPMs. The reported metrics include LUTs, Flip-Flops (FFs), Digital Signal Processors (DSPs), Block RAMs (BRAMs), as well as the maximum operating frequency achievable on the FPGA.

This analysis highlights how the accelerator can be adapted to different scenarios by balancing hardware cost against performance. For example, increasing the SIMD width from 8 to 64 raises the LUTs count from 2433 to 20802, underlining the need to carefully select the degree of parallelism depending on the application. In resource-constrained edge devices primarily targeting inference, a smaller SIMD configuration may be preferable. In contrast, when resources are less constrained and training acceleration is also desired, higher parallelism can be exploited to maximize the throughput.

Impact on the basic HDC/VSA arithmetic

Table 13.3 summarizes the execution time and the dynamic energy consumption achieved by MCR-HDCU for each core arithmetic operation. All experiments are conducted across a range of HV dimensionalities and hardware parallelism configurations to explore scalability and design trade-offs. Specifically, we evaluate three representative SIMD configurations {32, 256, 1024} and three HV dimensionalities {64, 512, 2048}.

As the HV dimensionality increases, the hardware loops inside the accelerator allow efficient processing without re-fetching instructions repeatedly, maximizing the execution speed. On the other hand, increasing SIMD values enables greater parallelism, allowing for the simultaneous processing of more HV components, but at the cost of increased hardware complexity, as reported in Table 13.2.

For the *binding* operation, MCR-HDCU achieves speedups from $39\times$ (SIMD = 8, HVDIM = 64) up to $444\times$ (SIMD = 64, HVDIM = 2048), thanks to its lightweight implementation as modular additions with implicit overflow. The more complex *superposition* operation benefits from fixed-point Cartesian accumulation using compact cosine/sine LUTs (see Section 13.4.3), and achieves speedups ranging from $36\times$ up to $314\times$. Similarly, the *normalization* operation, implemented via WTA on the precomputed LUTs, provides speedups from $117\times$ to $843\times$, depending on HV dimensionality and SIMD parallelism. Much larger gains are observed for the *permutation* operation: by reducing it to a simple hardware-friendly block-wise reordering of memory addresses, MCR-HDCU achieves speedups from $79\times$ to $975\times$. Finally, the *distance* metric, that is critical for inference, also benefits strongly from the hardware acceleration, with speedups ranging from $20\times$ to $190\times$, depending on the configuration.

Overall, these results confirm that all core operations of MCR can be executed extremely efficiently in hardware, demonstrating that the additional expressiveness of the modular space does not come at the cost of computational efficiency.

Energy efficiency follows a similar trend. Across all operations, the software implementation on the RISC-V core consumes between $29.5\times$ and $594\times$ more energy than its accelerated counterpart. This reduction highlights the value of specialized hardware, particularly for low-power applications, even when accelerating only individual arithmetic operations.

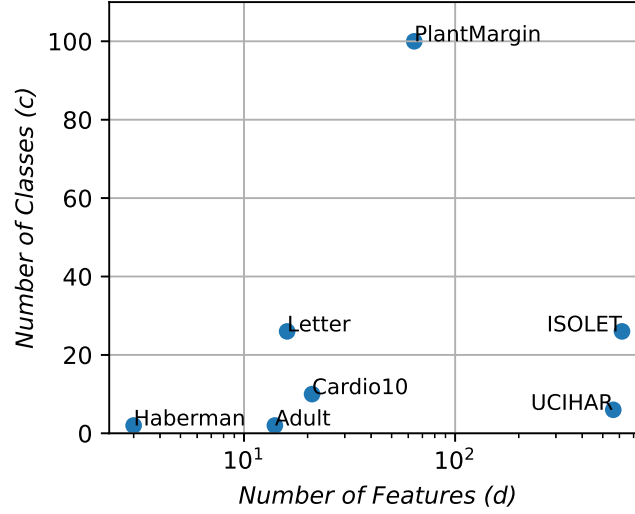


Figure 13.9 Overview of the real datasets selected for the evaluation, showing their complexity in terms of the number of features and classes. The datasets span a wide range of complexities: **HabermanSurvival** ($d = 3$, $c = 2$), **Adult** ($d = 14$, $c = 2$), **Letter** ($d = 16$, $c = 26$), **Cardio10** ($d = 21$, $c = 10$), **PlantMargin** ($d = 64$, $c = 100$), **UCIHAR** ($d = 561$, $c = 6$), and **ISOLET** ($d = 617$, $c = 26$).

Performance on real data

To demonstrate the flexibility of the proposed accelerator and evaluate its performance in realistic scenarios, we selected seven classification datasets from the collection used in Section 13.4.2. The selection captures a wide spectrum of problem complexities in terms of both the number of features and classes. As shown in Fig. 13.9, the selected datasets span from very simple problems such as **HabermanSurvival** and **Adult**, with only a few features and two classes, to demanding problems such as **PlantMargin**, **UCIHAR**, and **ISOLET**, which combine hundreds of features with dozens or even hundreds of classes. Intermediate problems such as **Letter** and **CARDIO10** represent median cases, containing tens of input features with challenging multi-class classification. This variability ensures that the evaluation covers the full range of settings, from lightweight binary problems to high-dimensional, multi-class problems, thereby providing a comprehensive validation of MCR-HDCU. It is important to emphasize that the goal of this analysis is not to assess the classification accuracy, but rather to evaluate the efficiency of the proposed accelerator. Every arithmetic operation implemented in MCR-HDCU was rigorously verified to ensure full consistency with the software model. As a result, the accelerator delivers the exact same accuracy as the reference software implementation reported in Section 13.4.2.

For these experiments, we adopted HVs of size $D = 1024$ and a modulo $r = 16$ (i.e., MCR-4), comparing software execution on the baseline Klessydra-T03 core against accelerated execution with different SIMD configurations $\{8, 16, 32, \text{ and } 64\}$. Table 13.4 reports the obtained execution time for the seven selected datasets, while Fig. 13.10 shows the relative speedups. The reported time corresponds to a single inference iteration. In the MCR-HDCU, latency depends only on the number of

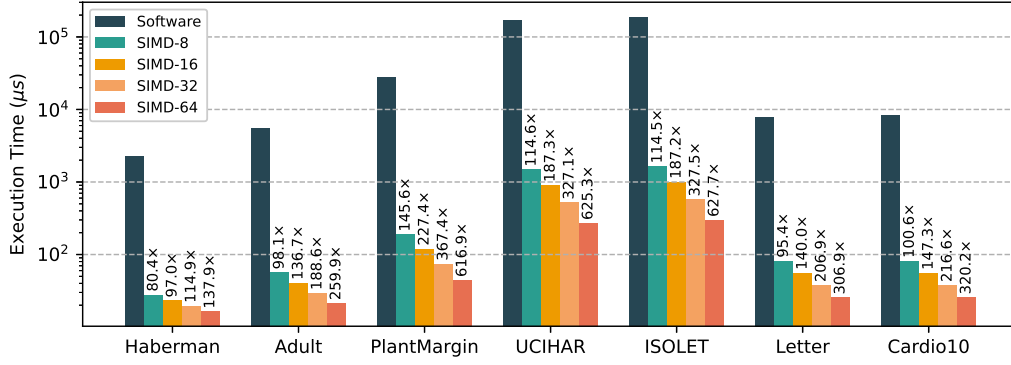


Figure 13.10 Execution time of 7 selected datasets on the baseline Klessydra-T03 core compared to the MCR-HDCU accelerator with different $\text{SIMD} \in \{8, 16, 32, 64\}$.

features and classes and is, therefore, constant across all samples within a dataset.

Across all datasets, the accelerator delivers substantial performance improvements. For the simplest tasks such as *HabermanSurvival* ($d = 3$, $c = 2$), speedups range from $80.4\times$ at $\text{SIMD} = 8$ to $137.9\times$ at $\text{SIMD} = 64$, while *Adult* ($d = 14$, $c = 2$) achieves $98.1\times$ to $259.9\times$. Larger datasets yield even higher gains: *Letter* ($d = 16$, $c = 26$) improves from $95.4\times$ to $306.9\times$ while *Cardio10* ($d = 21$, $c = 10$) from $100.6\times$ to $320.2\times$. For the extreme cases with a large number of features and classes, the impact of the hardware acceleration is even more pronounced: *PlantMargin* ($d = 64$, $c = 100$) shows $145.6\times$ to $616.9\times$, *UCIHAR* ($d = 561$, $c = 6$) $114.6\times$ to $625.3\times$, and *ISOLET* ($d = 617$, $c = 26$) achieves the highest gains with $114.5\times$ to $627.7\times$.

Overall, these results confirm that the benefits of hardware acceleration increase with the dataset complexity. In particular, as the number of features increases, the transformation of feature vectors into HVs dominates the runtime and the acceleration of the superposition and binding operations becomes increasingly beneficial. On the other hand, a larger number of classes amplifies the cost of searching for the closest prototype, making the role of the search and distance units the dominant factor.

Comparison with BSC

The previous sections demonstrated how the arithmetic of MCR can be efficiently implemented in digital hardware devices and how the adoption of a dedicated accelerator can reduce its execution time and energy consumption substantially. At the same time, Section 13.4.2 reported that the higher per-component precision and the modular space allow MCR-4 with $D = 64$ components and $r = 16$ (4 bits per component) to achieve higher average accuracy (75.49% across the collection) than BSC with $D = 1024$ components (74.35%), despite requiring $4\times$ less memory. Building on these findings, we conclude the experiments by revisiting the hypothesis from the original MCR study [3] that *BSC is advantageous over MCR in resource-constrained scenarios with strict requirements on execution time and energy efficiency*. To answer this question, we directly compare MCR-4 ($D = 64$, $r = 16$) against BSC ($D = 1024$), when both are accelerated on their corresponding architectures:

Table 13.4 Execution time in μs comparing software and SIMD hardware realizations across the selected datasets

Dataset	Software	SIMD-8	SIMD-16	SIMD-32	SIMD-64
HabermanSurvival	2258.40	28.08	23.27	19.65	16.38
Adult	5565.15	56.75	40.70	29.50	21.42
PlantMargin	27495.41	188.79	120.92	74.84	44.57
UCIHAR	170264.30	1486.03	909.21	520.56	272.30
ISOLET	188503.79	1647.03	1006.94	575.63	300.32
Letter	7854.92	82.33	56.10	37.97	25.59
Cardio10	8230.95	81.83	55.88	38.00	25.70

Table 13.5 Execution time (μs) and Energy consumption (μJ) for BSC ($D = 1024$) and MCR-4 ($D = 64$), across datasets and SIMD configurations

	SIMD	#LUTs	#FFs	Haberman		Adult		Letter		CARDIO10		Plant		UCIHAR		ISOLET	
				Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
BSC $D = 1024$	32	1886	728	6.418	0.624	21.018	2.158	27.164	2.960	31.473	3.458	87.673	10.185	747.627	87.529	824.864	97.315
	64	3255	1010	4.200	0.428	11.550	1.240	14.632	1.679	16.809	1.929	45.105	5.506	377.345	46.067	416.218	50.812
	128	6547	1671	3.181	0.338	7.070	0.780	8.670	1.042	9.842	1.183	24.819	3.169	200.586	25.615	221.126	28.238
	256	13094	2909	2.605	0.304	4.600	0.553	5.409	0.690	6.019	0.772	13.707	1.847	103.907	13.927	114.437	15.422
MCR $D = 64$	8	2433	1242	2.173	0.235	4.447	0.507	6.140	0.743	6.320	0.771	14.887	1.920	117.720	15.304	130.360	17.077
	16	5069	2157	1.976	0.219	3.736	0.437	4.824	0.603	5.112	0.639	11.800	1.569	91.400	12.156	101.000	13.433
	32	8887	3985	1.750	0.224	3.217	0.428	3.883	0.563	4.283	0.621	9.917	1.527	76.233	11.740	84.033	12.941
	64	20802	10257	1.687	0.270	3.026	0.499	3.470	0.607	3.939	0.693	9.122	1.688	69.661	12.818	76.652	14.181

MCR-HDCU and BSC-HDCU.

For the sake of fairness, we evaluate four SIMD configurations for each accelerator: $\text{SIMD} \in \{8, 16, 32, 64\}$ for MCR-HDCU and $\text{SIMD} \in \{32, 64, 128, 256\}$ for BSC-HDCU. These configurations yield an equivalent number of bits processed per clock cycle, since MCR-4 uses $r = 16$ ($b = 4$ bits per component). Both accelerators are synthesized with four 2 KB SPMs each. Testing SIMD values larger than the dimensionality of HVs for MCR provides no additional insights, as configurations with $\text{SIMD} > \text{HVDIM}$ converge to the same execution time and energy consumption as the case when $\text{SIMD} = \text{HVDIM}$.

Table 13.5 reports the results obtained across the previously selected datasets. Despite the intrinsically more complex arithmetic of MCR, the combination of an optimized accelerator design and the reduced number of components enabled by higher per-component precision allows MCR-HDCU to consistently outperform BSC-HDCU in both execution time and energy consumption. On average, across all datasets and SIMD configurations, MCR-HDCU runs kernels $3.08\times$ faster and with $2.68\times$ lower energy consumption than BSC-HDCU. These advantages are even more pronounced for large datasets such as UCIHAR and ISOLET, where the execution time improves by up to $6.35\times$ while the energy consumption drops by up to $5.72\times$.

Importantly, the reported improvements do not come at the expense of larger accelerators. For example, when comparing MCR-4 with $\text{SIMD} = 8$ against BSC with $\text{SIMD} = 64$, an instance where MCR-HDCU is actually smaller in terms of the required resources, MCR-4 still provides improvements: the execution time is reduced by $1.90\times$ to $3.21\times$ while the energy consumption by $1.82\times$ to $3.01\times$, depending on the dataset.

These results confirm the key insight of this study: although MCR requires more sophisticated arithmetic than BSC, its modular space and higher per-component

precision allow much lower HV dimensionality. When paired with efficient hardware support, this leads to significantly faster execution and lower energy consumption. In other words, MCR is not only more accurate per bit but also more efficient when deployed on dedicated hardware.

13.5 Summary of Findings

In this chapter, we revisited the modular composite representation (MCR) model, providing an extensive analysis of its properties and implementing the first hardware accelerator for this model. Our study demonstrates that the MCR model achieves the best trade-off between information capacity, classification accuracy, and efficiency, positioning it as a compelling alternative to more well-known models such as binary spatter codes [227] and multiply-add-permute [228].

First, we found that the modular discretized space of MCR offers substantially higher information capacity than binary and integer hypervectors, while approaching the information capacity of complex-valued hypervectors at a fraction of their memory footprint. Second, a large-scale evaluation on 123 classification datasets confirmed that MCR achieves higher accuracy than binary and low-precision integer hypervectors and can match the performance of binary hypervectors while requiring up to $4\times$ less memory. These results highlight the importance of per-component precision, showing that the larger dynamic range in non-binary modular arithmetic indeed pays off in practice.

Next, we designed MCR-HDCU – the first hardware accelerator for MCR, demonstrating that its arithmetic maps naturally to digital logic, with modular operations realized implicitly through binary overflow and trigonometric mappings efficiently implemented using compact lookup tables. Experimental results on basic operations and seven classification datasets show up to three orders-of-magnitude speedups and substantial energy savings compared to software execution. When compared to binary hypervectors at iso-accuracy and accelerated with the original HDCU, MCR-HDCU achieves on average $2.7\times$ faster execution and $3.9\times$ lower energy consumption.

Overall, our findings refute the intuition that the additional complexity of MCR outweighs its benefits. On the contrary, it provides a hardware-friendly alternative that delivers superior information capacity, higher classification accuracy, lower memory footprint, and great efficiency when implemented by specialized hardware.

Chapter 14

A Hardware-Friendly Discretized Approach to FHRR

This Chapter presents original work not yet published

"The art of approximation is the art of engineering."

Anonymous

The previous chapters investigated how *dedicated hardware* can unlock the efficiency of different HDC/VSA models on embedded platforms. We began by designing the HDCU (Chapter 12), a flexible hardware accelerator for BSC tightly integrated into a RISC-V core. Subsequently, motivated by the observation in [241] that purely binary hypervectors cannot approximate arbitrary similarity structures, and therefore cannot always learn Bayes-optimal classifiers, we extended the HDCU to support MCR (Chapter 13). Our work repositioned MCR, demonstrating that it provides a more favorable trade-off than BSC for embedded deployment: it achieves higher information capacity and accuracy while remaining faster and more efficient, providing a better balance between representational power and hardware cost.

This chapter focuses on the most complex but expressive HDC/VSA paradigm: the *Fourier Holographic Reduced Representation*. FHRR represents information using complex-valued, unit-modulus hypervectors and has strong links to kernel methods, residual coding, and neuromorphic implementations based on oscillatory or spiking substrates [57, 212, 220, 235]. However, despite its attractive properties, FHRR remains difficult to deploy on real embedded and hardware-constrained systems. Its reliance on high-precision complex arithmetic creates prohibitive overhead in area, memory, and power, widening the gap between its theoretical advantages and the practical constraints of embedded systems.

These considerations motivate the research question guiding this chapter:

❓ Research Question

Can *FHRR* be made practical for resource-constrained embedded systems?

This chapter answers this question by proposing design strategies for a *hardware-friendly* FHRR implementation and introducing the first dedicated accelerator

for this paradigm. Our key insight mirrors the principle that enabled efficient MCR: by discretizing phases on the unit circle, modular arithmetic can be realized without explicit modulo operators, allowing binary overflow to implement wrap-around implicitly. We detail the discretization principle and its implications for each HDC primitive, present the complete accelerator architecture with all its functional units—encoding, binding, unbinding, bundling with CORDIC-based normalization, and permutation—and introduce, to the best of our knowledge, the first hardware implementation of a Resonator Network. We conclude with a preliminary analysis of hardware resource requirements across different configurations. *Note: The accelerator design and preliminary synthesis results presented in this chapter are part of ongoing work. The architecture and functional units are fully specified, but extended validation and comparison with existing HDC accelerators are deferred to the final version of the associated publication.*

14.1 Fourier Holographic Reduced Representations

FHRR [212, 235] defines hypervectors in the frequency domain where each component lies on the complex unit circle. Formally, a hypervector $\mathbf{v} \in \mathbb{C}^D$ is defined as:

$$v_k = e^{i\phi_k}, \quad \phi_k \in (-\pi, \pi], \quad \forall k \in \{1, \dots, D\}, \quad (14.1)$$

where D is the dimensionality and $|v_k| = 1$. Information is encoded strictly in the phase ϕ_k .

Within this space, FHRR defines the *binding* operation between two hypervectors $\mathbf{h}, \mathbf{u} \in \mathbb{C}^D$ as element-wise complex multiplication (Hadamard product):

$$\mathbf{c} = \mathbf{h} \odot \mathbf{u} \implies c_k = h_k u_k = e^{i(\phi_k^{(h)} + \phi_k^{(u)})},$$

so binding corresponds to *adding phases* component-wise. The inverse operation, *unbinding*, is implemented by element-wise multiplication with the complex conjugate:

$$\mathbf{h} \approx \mathbf{c} \oslash \mathbf{u} = \mathbf{c} \odot \mathbf{u}^*, \quad c_k u_k^* = e^{i(\phi_k^{(h)} + \phi_k^{(u)})} e^{-i\phi_k^{(u)}} = e^{i\phi_k^{(h)}},$$

where $(\cdot)^*$ denotes complex conjugation. In the phase domain this is simply phase subtraction. Because all components lie on the unit circle, the exact inverse of a vector is given by its conjugate, and binding/unbinding are perfectly invertible in the absence of noise [57].

Superposition (or bundling) in FHRR is implemented via vector addition followed by re-normalization back to the unit circle. Given a set of hypervectors $\{\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(m)}\}$, their bundle is defined as

$$\mathbf{v} = \sum_{j=1}^m \mathbf{h}^{(j)}, \quad v_k = \sum_{j=1}^m h_k^{(j)},$$

which yields complex components that are no longer unit modulus. To obtain a valid FHRR vector, each component is projected back onto the unit circle by preserving only its phase:

$$w_k = \frac{v_k}{|v_k|} = e^{i \arg(v_k)}, \quad \text{for } v_k \neq 0,$$

and a suitable tie-breaking rule is used when $v_k = 0$ (e.g., random phase or an average of the original arguments) [5, 57].

The similarity between two FHRR vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^D$ is measured by the real part of the normalized Hermitian inner product. This is effectively the mean cosine of the phase differences:

$$\text{sim}(\mathbf{u}, \mathbf{h}) = \Re \left(\frac{1}{D} \sum_{k=1}^D u_k h_k^* \right) = \frac{1}{D} \sum_{k=1}^D \cos(\phi_k^{(u)} - \phi_k^{(h)}). \quad (14.2)$$

Random unit-modulus vectors are nearly orthogonal ($\text{sim} \approx 0$) for large D , while $\text{sim}(\mathbf{x}, \mathbf{x}) = 1$ [1, 57].

An additional advantage of FHRR is the possibility of *fractional binding* or *Fractional Power Encoding (FPE)* [57, 212, 220] introduced in Chapter 2, that allows encoding of continuous quantities (e.g., angles, distances, time) directly in the phase of complex components, extending the expressiveness of the representation. Given a hypervector $\mathbf{v}$ and a scalar $\alpha \in \mathbb{R}$, FPE defines:

$$\mathbf{v}^\alpha = (e^{i\alpha\phi_1}, \dots, e^{i\alpha\phi_D}),$$

i.e., each component's phase is scaled by α . Because information is encoded in a continuous phase on the unit circle, each component in FHRR can carry more information than binary or low-precision integer schemes, enabling lower dimensionality for a given accuracy target [3, 220]. Additionally, many emerging neuromorphic and analog substrates—including coupled oscillators, phase-change materials, photonic interferometers, and spintronic devices—naturally encode information in phases or angles. FHRR aligns well with these substrates, making it an attractive target for brain-inspired hardware where computation is performed by phase interactions rather than digital arithmetic [212, 220]. Recent studies have demonstrated implementations of FHRR on spiking neural networks [212].

14.2 Residue Hyperdimensional Computing

A compelling application that demonstrates the power of FHRR is *Residue Hyperdimensional Computing* (RHC) [342]. RHC is a recent framework that builds directly on the algebra of FHRR and combines the principles of Residue Number Systems (RNS) with hyperdimensional computing to realize a residue-based encoding scheme in high-dimensional space [342].

In a classical RNS, an integer $x \in \mathbb{Z}$ is encoded by its remainders with respect to a set of pairwise coprime moduli $\{m_1, \dots, m_K\}$:

$$\mathbf{r}(x) = [x \bmod m_1, x \bmod m_2, \dots, x \bmod m_K]. \quad (14.3)$$

By the Chinese Remainder Theorem, this representation is unique for $0 \leq x < M := \prod_{k=1}^K m_k$. Addition, subtraction, and multiplication are then performed component-wise on the residues without carries, enabling highly parallel arithmetic.

RHC transfers this idea into the FHRR setting by constructing, for each modulus m , a family of complex hypervectors whose phases are restricted to the m -th roots

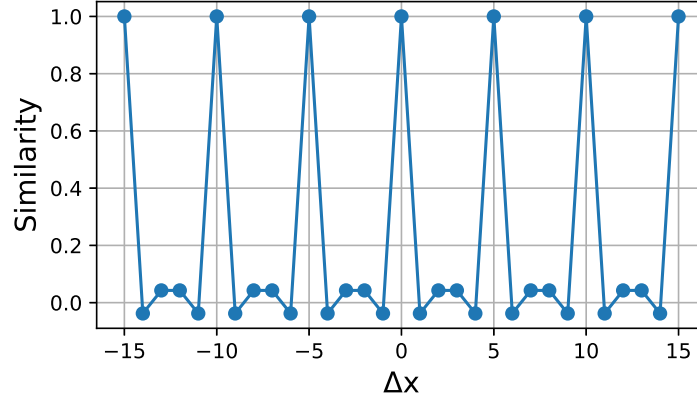


Figure 14.1 Periodicity of the RHC similarity kernel for a single modulus ($m = 5$). The kernel peaks when the difference Δx is a multiple of m , enabling residue-based encoding. Replicated from [342].

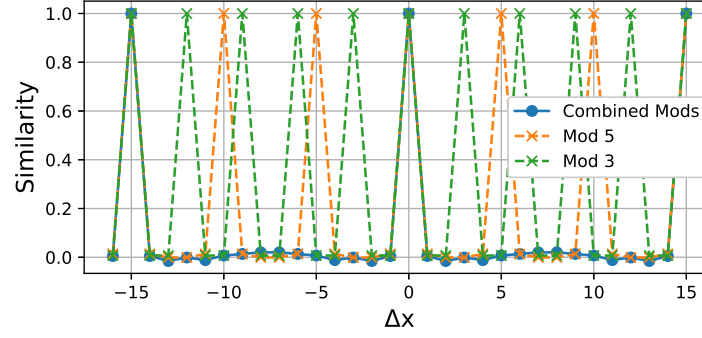


Figure 14.2 RHC similarity kernel for combined moduli ($m_1 = 3, m_2 = 5$). The constructive interference extends the unique range to the product of the moduli ($M = 15$), effectively implementing the Chinese Remainder Theorem in hyperdimensional space. Replicated from [342].

of unity and then using Fractional Power Encoding to map integers to FHRR hypervectors. Formally, for a given modulus m we draw a base hypervector

$$\mathbf{z}_m = [e^{i\Phi_1}, \dots, e^{i\Phi_D}], \quad (14.4)$$

where the support of the phase distribution is constrained to the m -th roots of unity:

$$\Phi_j \in \left\{0, \frac{2\pi}{m}, \dots, \frac{2\pi(m-1)}{m}\right\}, \quad j = 1, \dots, D. \quad (14.5)$$

Using FPE, an integer $x \in \mathbb{Z}$ is then encoded *modulo* m as

$$\mathbf{z}_m(x) = \mathbf{z}_m^x = [e^{i\Phi_1 x}, \dots, e^{i\Phi_D x}]. \quad (14.6)$$

Because each Φ_j is a multiple of $2\pi/m$, adding m to the encoded value leaves the

hypervector unchanged:

$$\begin{aligned}\mathbf{z}_m(x+m) &= [e^{i\Phi_1(x+m)}, \dots, e^{i\Phi_D(x+m)}] \\ &= [e^{i\Phi_1 x} e^{i\Phi_1 m}, \dots, e^{i\Phi_D x} e^{i\Phi_D m}] \\ &= [e^{i\Phi_1 x}, \dots, e^{i\Phi_D x}] = \mathbf{z}_m(x),\end{aligned}\tag{14.7}$$

Crucially, because every base phase Φ_j is a multiple of $2\pi/m$, the encoding is periodic with period m . That is, for any integer k , $e^{i\Phi_j(x+km)} = e^{i\Phi_j x} \cdot (e^{i2\pi})^{k(\dots)} = e^{i\Phi_j x}$. Consequently:

$$\mathbf{z}_m(x) = \mathbf{z}_m(x \bmod m).$$

Thus, $\mathbf{z}_m(x)$ strictly encodes the *residue* of x modulo m .

This periodicity is directly reflected in the induced similarity kernel. The real part of the inner product between the encodings of two integers x and y (where $\Delta x = x - y$) is:

$$K_m(\Delta x) = \frac{1}{D} \Re \langle \mathbf{z}_m(x), \mathbf{z}_m(y) \rangle \approx \begin{cases} 1, & \Delta x \equiv 0 \pmod{m}, \\ 0, & \text{otherwise.} \end{cases}\tag{14.8}$$

As shown in Figure 14.1, the kernel peaks precisely when the integers are congruent modulo m and remains close to zero otherwise.

To encode an integer across multiple moduli $\{m_k\}_{k=1}^K$, RHC combines the individual residue encodings using the FHRR binding operation:

$$\mathbf{z}(x) = \bigodot_{k=1}^K \mathbf{z}_{m_k}(x).\tag{14.9}$$

This single hypervector $\mathbf{z}(x)$ represents the intersection of the individual residue constraints. As illustrated in Figure 14.2, the similarity peaks only when Δx is a multiple of *all* moduli m_k simultaneously. For coprime moduli, this extends the unique dynamic range to $M = \prod m_k$, effectively implementing the Chinese Remainder Theorem in a distributed, holographic format: every dimension carries information about all moduli, in contrast to traditional RNS implementations where residues are stored in separate registers.

This construction inherits the parallel, carry-free arithmetic of RNS while offering four key properties:

1. *Algebraic structure*: simple component-wise operations implement addition and (via a second binding operation) multiplication on encoded integers.
2. *Expressivity*: the dynamic range M grows super-linearly with dimension D , enabling large ranges with moderate hypervector sizes.
3. *Efficient decoding*: decoding can be performed by factorizing $\mathbf{z}(x)$ into its per-modulus components using a resonator network, requiring resources that scale better than linearly with M .
4. *Robustness to noise*: strong robustness to phase noise and perturbations.

14.3 Toward Hardware-Friendly FHRR

Despite the compelling advantages of FHRR, ranging from its high information capacity to its support for powerful algebraic frameworks such as RHC, its adoption in resource-constrained embedded systems is hindered by significant computational complexity. The theoretical elegance of complex-valued hypervectors clashes with the practical realities of digital hardware design. To address this, we must first analyze the available implementation strategies and their specific hardware penalties.

Since FHRR operates on phasors, it can be implemented in two mathematically equivalent but architecturally distinct forms: the *Cartesian* formulation and the *Phase* formulation.

14.3.1 The Cartesian Formulation

In the Cartesian formulation, each hypervector component is stored as a standard complex number $z_k = a_k + ib_k$, with real and imaginary parts represented in fixed- or floating-point formats. This approach allows binding to be defined as component-wise complex multiplication, bundling as vector addition, and similarity as the real part of the normalized dot product.

This representation is highly advantageous for software simulation and is the standard choice for HDC/VSA libraries such as TorchHD [346], leveraging the optimized arithmetic libraries available for modern CPUs and GPUs. However, this choice becomes less attractive when designing custom hardware. Complex components double memory footprint (each component requires storing two real numbers: the real and imaginary parts) and the reliance on complex arithmetic imposes high demands on area and power for floating-point units.

14.3.2 The Phase Formulation

By contrast, the phase formulation exploits the unit-modulus property of FHRR. Each component is represented solely by its angle $\phi_k \in (-\pi, \pi]$ on the unit circle, such that $z_k = e^{i\phi_k}$. This representation is inherently storage-efficient, requiring only a single scalar per dimension.

In this domain, we recall that binding and unbinding reduce to element-wise addition and subtraction of phases, and similarity can be computed as the mean cosine of phase differences. Bundling is more challenging: although it corresponds to vector addition in $\mathbb{C}^D$, maintaining a strictly phase-based representation requires projecting the angles back to Cartesian coordinates on the unit circle, accumulating the corresponding real and imaginary parts across vectors, and converting the result back to a phase via normalization (e.g., $v_k = \text{atan2}(\sum \sin \phi_k, \sum \cos \phi_k)$).

Moreover, in the phase formulation, all operations are defined modulo 2π , which in digital hardware translates into modular arithmetic on phase values. A straightforward implementation requires explicit modulo operations or range reductions, which in turn rely on division or conditional corrections. As discussed in Chapter 2, integer division is among the most expensive arithmetic operations in digital logic, with long critical paths and significant area and energy overhead. Even optimized schemes such as VLNPD alleviate but do not eliminate this cost. Additionally,

the normalization required for bundling relies on trigonometric functions that are notably costly in hardware.

In summary, while phase-based FHRR is conceptually appealing for its memory efficiency, naïve implementations remain burdened by the costs of explicit modulo arithmetic and trigonometric conversions.

14.4 Discretizing the Unitary Circle for Modulo-Free Arithmetic

The central idea explored in this chapter is to bring to FHRR the same insight that underpinned the MCR hardware accelerator in Chapter 13: modular arithmetic can be implemented extremely efficiently in digital hardware if the underlying domain is discretized such that wrap-around is handled implicitly by binary overflow, rather than by explicit modulo operators.

Instead of representing phases ϕ_i as floating- or fixed-point numbers and performing explicit modulo- 2π reductions, we discretize the unit circle into $Q = 2^{Q_\phi}$ uniformly spaced angles. Each phase is encoded as an unsigned integer index q_i :

$$q_i \in \{0, 1, \dots, Q - 1\}, \quad \phi_i \approx \frac{2\pi}{Q} q_i. \quad (14.10)$$

Here, Q_ϕ denotes the bit-width of the phase storage (e.g., $Q_\phi = 8$ yields $Q = 256$ discrete angles with a resolution of $2\pi/256$ radians). The mapping between the continuous angle and the discrete integer index is given by:

$$q_i = \left\lfloor \frac{\phi_i}{2\pi} Q \right\rfloor \pmod{Q}. \quad (14.11)$$

Q_ϕ can be tuned to match the precision required by the target task, yet regardless of the chosen bit-width, the modulo- 2π behavior is realized naturally by unsigned binary overflow on Q_ϕ bits, requiring no explicit modulo logic in hardware. In the remainder of this chapter, we exploit this discretized phase representation to design an efficient and compact hardware accelerator for FHRR.

14.5 Discretized FHRR Accelerator Architecture

Figure 14.3 illustrates the overall architecture of the proposed hardware accelerator, organized as a collection of specialized functional units interconnected through a shared SIMD datapath.

The proposed accelerator implements the complete FHRR pipeline—binding, bundling, permutation, similarity, FPE—and a resonator network, directly in the discretized phase domain. Designed for flexibility, the architecture supports both standard FHRR and RHC configurations through programmable control logic, without requiring changes to the underlying datapath.

To accommodate diverse application constraints, the accelerator exposes a set of synthesis-time parameters that govern the trade-off between precision, throughput, and area. Key configuration knobs include:

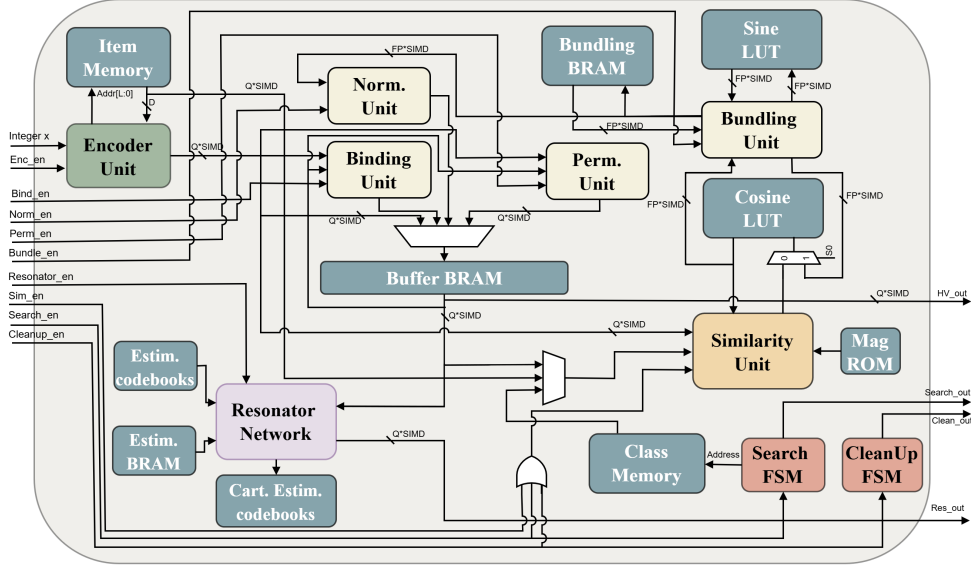


Figure 14.3 Architecture of the proposed accelerator for FHRR, featuring dedicated memories and specialized functional units for each basic FHRR operation connected via a shared bus to an intermediate buffer, enabling efficient chaining of HDC operations.

- Q_ϕ : The bit-width of the phase indices, defining the angular resolution.
- SIMD: The parallelism factor, determining how many vector components are processed per cycle.
- D: The hypervector dimensionality.
- NUM_MODULI: The maximum number of moduli supported in RHC mode.
- FP_INT/FRAC: The precision of the fixed-point datapath used for cartesian accuulation during buindling

All functional units operate on the discretized phase representation, except for the intermediate fixed-point accumulation required during bundling.

To minimize data movement and support efficient operator chaining, the architecture centers around a shared intermediate buffer (Buffer BRAM in Fig. 14.3). This buffer acts as the central exchange point for all operations, eliminating the latency penalty of writing intermediate results back to main memory. Data routing is standardized across all FUs:

- **Operand 1** is multiplexed, sourced either from the *Encoding Unit* (for fresh data) or the *Intermediate Buffer*.
- **Operand 2** is always sourced directly from the *Intermediate Buffer*.

This allows the system to flexibly chain operations, such as binding a newly encoded scalar to an accumulated state vector, under the control of lightweight finite state machines. The following details the design of each functional unit and demonstrates how it has been optimized for performance, resource efficiency, scalability, and flexibility.

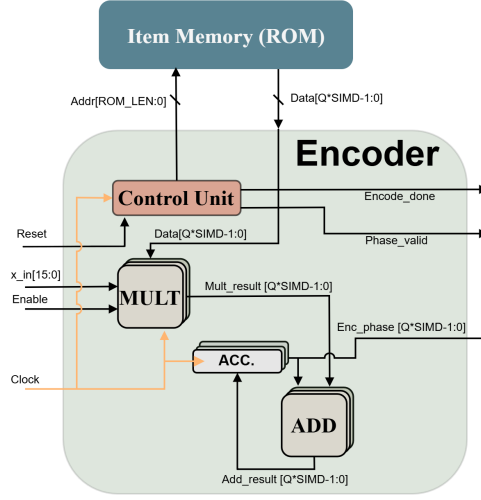


Figure 14.4 Architecture of the Encoding Unit. The unit supports both standard FHRR Fractional Power Encoding and the iterative accumulation required for RHC mode.

Encoding Unit

The encoding unit maps a single scalar value x into a high-dimensional FHRR vector. It supports two operating modes: (i) *Standard FHRR* using FPE [57], and (ii) *RHC mode*, which encodes residues relative to a set of coprime moduli [342].

Standard FHRR Mode

In standard FPE, a scalar $x \in \mathbb{Z}$ is encoded by exponentiating a random complex base vector $\mathbf{z} = [e^{i\phi_1}, \dots, e^{i\phi_D}]$:

$$\mathbf{z}(x) = \mathbf{z}^x = [e^{i\phi_1 x}, \dots, e^{i\phi_D x}]. \quad (14.12)$$

In hardware, the discretized base phases are stored in a ROM packed into SIMD-wide words. For each chunk, the encoder:

1. Fetches SIMD base phases from ROM.
2. Multiplies each phase by the scalar input x (in fixed point).
3. Outputs the result. The modulo- 2π wrap-around occurs naturally due to the Q_ϕ -bit representation.

The latency is dominated by a single pass over the dimension D :

$$L_{\text{enc}}^{\text{FHRR}} \approx \frac{D}{\text{SIMD}}. \quad (14.13)$$

RHC Mode

In RHC mode, the encoder implements the Chinese Remainder Theorem logic. For a set of moduli $\{m_1, \dots, m_K\}$, a scalar x is mapped to residues $r_k = x \bmod m_k$.

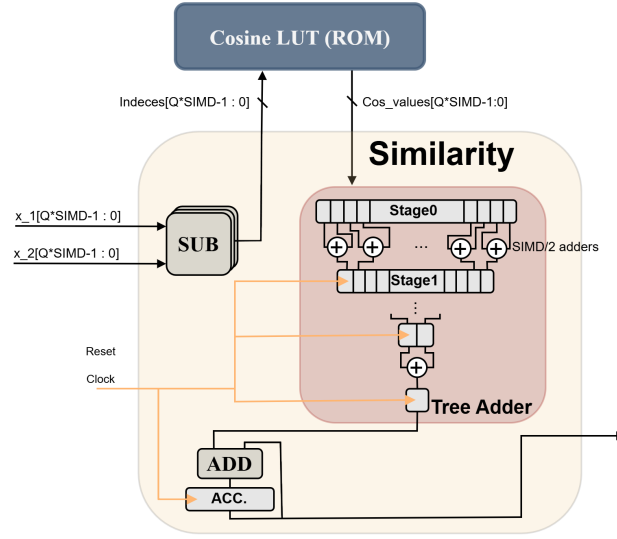


Figure 14.5 Architecture of the Similarity Unit. Discretized phase differences are mapped to cosine similarity scores via a LUT and reduced using a pipelined adder tree.

The final hypervector is the component-wise product (binding) of the base vectors for each modulus raised to the power of the residue. In the discretized hardware domain, the output phase index $q_{out,i}$ is computed by accumulating the discrete phase contributions:

$$\mathbf{z}(x) = \bigodot_{k=1}^K \mathbf{z}_{m_k}^{r_k} \implies q_{out,i} = \left(\sum_{k=1}^K q_{i,k} r_k \right) \bmod 2\pi. \quad (14.14)$$

Thus, even in this residue formulation, *one encoding request still corresponds to one scalar value x* . The key difference is that computing the final phase $q_{out,i}$ now requires combining all K discrete residue contributions.

The hardware handles this by iterating over the moduli. The pipeline iterates K times. In each pass k , it fetches the base phases for modulus m_k , multiplies them by r_k , and accumulates the result into a running phase sum.

The latency scales linearly with the number of moduli:

$$L_{enc}^{RHC} = \frac{D}{SIMD} \cdot K + 1. \quad (14.15)$$

Similarity Unit

The Similarity Unit (Fig. 14.5) performs the following operations for each chunk:

1. A SIMD-wide subtractor computes the discrete phase deltas $\Delta q = q_{b,i} - q_{a,i}$ (overflow handles the modulo).
2. The deltas address a LUT containing pre-computed fixed-point (Q.FP_INT, FP_FRAC) values of cosine.

3. An adder tree reduces the SIMD cosine values into a partial sum.
4. An accumulator aggregates partial sums over all chunks to produce the final score.

Latency is minimized by pipelining the reduction tree: $L_{\text{sim}} = \frac{D}{\text{SIMD}} + \log_2(\text{SIMD})$.

Associative Search and Cleanup

Many HDC tasks rely on prototype-based classification and symbol decoding: given a query hypervector $\mathbf{q}$, the system must find the most similar class prototype (associative search) or the closest item in an item memory (cleanup).

These high-level tasks are structurally identical and are implemented by wrapping the similarity unit with a control FSM that iterates over a memory of stored prototypes $\mathbf{P}$. The nature of $\mathbf{P}$ depends on the operation mode:

- In **Associative Search**, $\mathbf{P} = \{C_1, \dots, C_{N_{\text{class}}}\}$ represents the set of class hypervectors.
- In **Cleanup**, $\mathbf{P} = \{I_1, \dots, I_{N_{\text{item}}}\}$ represents the set of valid symbols in the item memory.

The hardware performs the search by sequentially evaluating:

$$\text{Result} = \arg \max_i \text{sim}(\mathbf{q}, \mathbf{P}_i). \quad (14.16)$$

For N prototypes, the unit streams vectors sequentially, calculating the similarity for each and updating a “current best” register. The total latency is $L_{\text{search}} = N \cdot L_{\text{sim}}$, where L_{sim} is the latency of a single similarity computation.

14.5.1 Binding and Unbinding Units

Binding and Unbinding are computationally trivial in the phase domain. They are implemented as SIMD-wide adders/subtractors. These purely combinational units have a throughput of one chunk per cycle:

$$L_{\text{bind}} = L_{\text{unbind}} = \frac{D}{\text{SIMD}}. \quad (14.17)$$

For the unbinding, the same subtractors already presented in the similarity path are used to minimize hardware overhead.

14.5.2 Bundling Unit and CORDIC-Based Normalization

Performing the superposition operation in the phase domain is non-trivial because the resulting sum does not naturally lie on the unit circle. A naïve attempt to operate directly in phase space (e.g., averaging phases) fails to preserve the correct geometry of the complex plane and degrades quickly as more hypervectors are bundled.

To maintain geometric accuracy, the proposed Bundling Unit (Fig. 14.6) employs a convert-accumulate-normalize strategy:

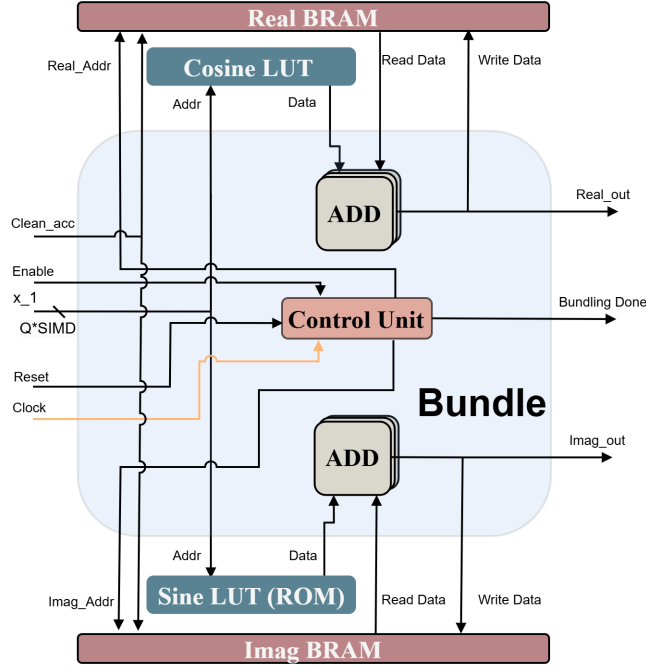


Figure 14.6 Architecture of the Bundling Unit. To ensure geometric accuracy, phases are converted to Cartesian form for accumulation and re-normalized to the unit circle via a shared CORDIC engine.

1. **Cartesian Accumulation:** For each element i in a SIMD chunk, the phase q_i is converted to its Cartesian coordinates $x_i = \cos(q_i)$ and $y_i = \sin(q_i)$ indexing Sine/Cosine LUTs. These values are accumulated into two dedicated accumulation buffers: a Real BRAM and an Imaginary BRAM. Each BRAM serves as a multi-channel accumulator, with the j -th address corresponding to the j -th dimension of the hypervector. For a bundle of N vectors, the accumulation at dimension j is:

$$X_{acc}[j] = \sum_{n=1}^N x_{j,n}, \quad Y_{acc}[j] = \sum_{n=1}^N y_{j,n}$$

2. **Normalization (CORDIC):** Once accumulation is complete, the resulting vectors are projected back onto the unit circle. We employ a CORDIC engine operating in *vectoring mode* to recover the phase angle $q_j = \text{atan2}(Y_{acc}[j], X_{acc}[j])$ without using expensive square roots or divisions.

The CORDIC engine implements an iterative extraction of the angle. For a component with accumulated values $(X_{acc}[j], Y_{acc}[j])$, the algorithm rotates the vector toward the positive real axis. Initializing the internal state as $(x_0, y_0, z_0) = (X_{acc}[j], Y_{acc}[j], 0)$, the k -th iteration proceeds as follows:

$$\begin{cases} x_{k+1} = x_k + \sigma_k(y_k \gg k) \\ y_{k+1} = y_k - \sigma_k(x_k \gg k) \\ z_{k+1} = z_k + \sigma_k \arctan(2^{-k}) \end{cases} \quad \text{where } \sigma_k = \text{sign}(y_k).$$

Here, $\gg k$ denotes an arithmetic right shift, and the $\arctan(2^{-k})$ constants are retrieved from a small ROM. We configure the engine to run for $N_{\text{CORDIC}} = Q_\phi$ iterations to match the hardware's phase discretization. Upon convergence, $y_N \rightarrow 0$ and $z_N \rightarrow \text{atan2}(Y_{\text{acc}}[j], X_{\text{acc}}[j])$. The resulting discrete angle q_j is in Q_ϕ -bit precision.

Crucially, the *accumulation* and *normalization* stages are decoupled and independently controlled. Accumulation can proceed over a large stream of hypervectors (e.g., summing hundreds of items) with high throughput, while normalization is invoked only when a reusable prototype on the unit circle is required (e.g., before storing it in memory or feeding it to the binding, similarity or resonator network units).

The total latency for bundling a set of N_{in} hypervectors of dimension D is:

$$L_{\text{bundle}} = \left(N_{\text{in}} \cdot \frac{D}{\text{SIMD}} \right) + L_{\text{norm}}, \quad (14.18)$$

where the first term represents the pipelined accumulation throughput, and L_{norm} accounts for the CORDIC sweep.

14.5.3 Permutation Unit

Following the optimized strategy introduced for MCR in Chapter 13, the proposed permutation unit implements permutations through address remapping, avoiding explicit barrel shifters.

The unit supports two operating modes, selected via the control signals:

1. **Explicit remapping:** When an index map is provided, SIMD-wide chunks are written to BRAM at the addresses specified by the map. This enables arbitrary permutations at SIMD granularity, for up to $(D/\text{SIMD})!$ distinct permutations.
2. **Cyclic rotation:** When a shift amount is provided, the unit performs the standard right rotation by offsetting the chunk addresses accordingly, thus realizing a cyclic shift of the hypervector at the macro-chunk level. In this case, a maximum of D/SIMD distinct permutations are allowed.

The total latency is:

$$L_{\text{perm}} = \frac{D}{\text{SIMD}}. \quad (14.19)$$

14.5.4 Resonator Network

The FHRR accelerator also includes a dedicated *Resonator Network* unit that implements the iterative cleanup and factor recovery process directly in hardware. To the best of the author's knowledge, this represents the first hardware implementation of a resonator network.

This module receives a composite hypervector $\mathbf{s}$ (the query) and a set of initial candidate factors $\{\hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_K\}$ (which may represent generic factors or, in the case of RHC, the residue components associated with different moduli). These candidates are stored in local BRAMs and updated in-place across iterations. A

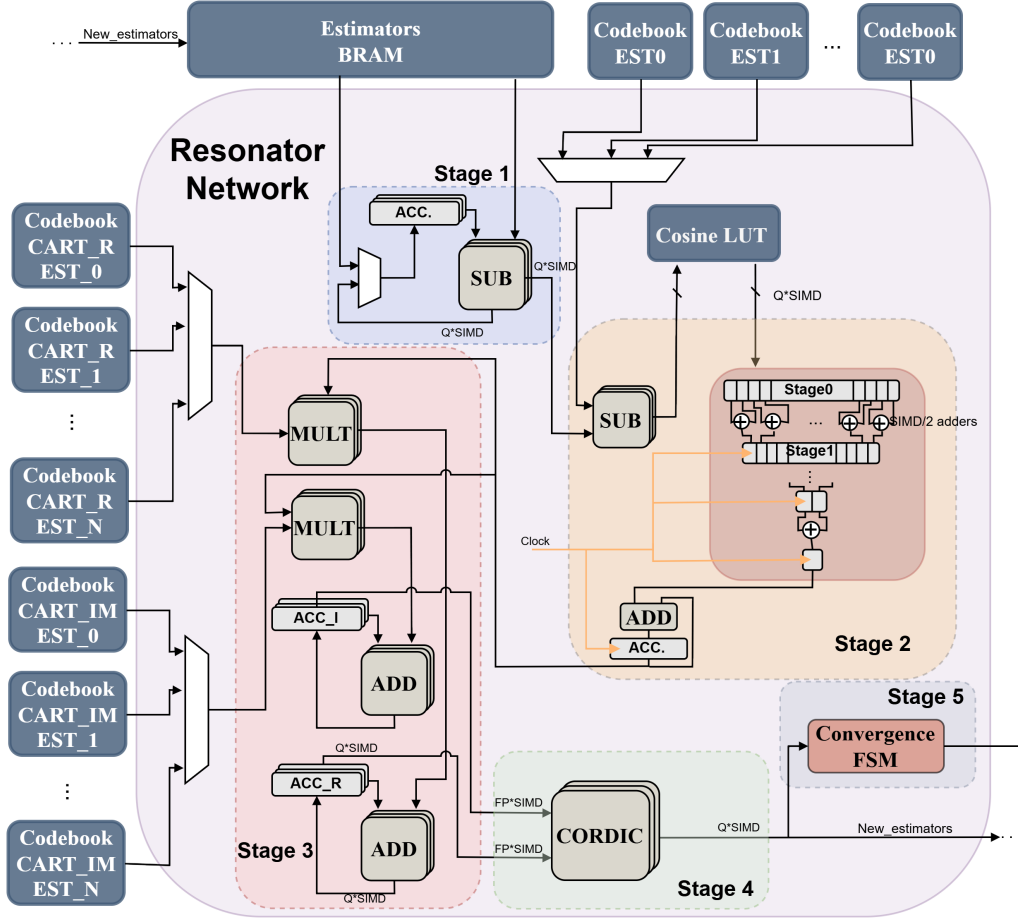


Figure 14.7 Hardware architecture of the Resonator Network Unit. The iterative factor recovery is unrolled into a pipelined datapath comprising residual estimation, codebook similarity search, and weighted update steps.

finite-state controller coordinates the resonance loop: at each iteration, factors are read, processed, and refined. A convergence detector monitors the updates and asserts a *done* signal when a stable configuration is reached or the maximum iteration count is exceeded.

The computation for a single factor update (say, factor j) is decomposed into five pipelined stages, executed by specialized FUs.

1. **Stage 1: Residual estimation in discretized phase.** The first stage estimates the “residual contribution” that factor j must explain by removing the contributions of all other factors $i \neq j$ from the composite vector $\mathbf{s}$. Mathematically, this corresponds to unbinding the other estimates from $\mathbf{s}$:

$$\mathbf{c}_j = \mathbf{s} \odot \left(\bigodot_{i \neq j} \hat{\mathbf{z}}_i \right). \quad (14.20)$$

In the discretized phase domain, this becomes a series of subtractions modulo

2^{Q_ϕ} :

$$q_d^{(\mathbf{c}_j)} = q_d^{(\mathbf{s})} - \sum_{i \neq j} q_d^{(\mathbf{z}_i)} \pmod{2^{Q_\phi}}, \quad d = 1, \dots, D. \quad (14.21)$$

The D components are processed in SIMD fashion over D/SIMD chunks. For each chunk, FU subtracts, lane-wise, the contributions of the $K-1$ other factors. Thus, FU1 performs one SIMD subtraction per factor per chunk. To maximize throughput, the unit implements a *producer-consumer* handshake: a `change_chunk` signal is raised one cycle before a chunk is fully computed. This allows the controller to trigger Stage 2 immediately as the first chunk becomes ready, overlapping the tail of the estimation with the head of the similarity search. The result of this stage is the *current phase estimate* $\mathbf{c}_j$, produced chunk by chunk.

2. **Stage 2: Codebook Similarity Search.** The second stage evaluates the similarity between the estimated residual $\mathbf{c}_j$ and the discrete codebook $Z_j = \{\mathbf{z}_{j,0}, \dots, \mathbf{z}_{j,m_j-1}\}$ containing all valid symbols (or residues) for factor j . Operating in a streaming manner, the unit consumes residual chunks produced by Stage 1 and compares them against the corresponding chunks of the codebook. For each candidate r , it first computes the element-wise phase differences in Q_ϕ -bit precision:

$$\Delta q_d^{(r)} = q_d^{(\mathbf{c}_j)} - q_d^{(\mathbf{z}_{j,r})} \pmod{2^{Q_\phi}}, \quad d = 1, \dots, D, \quad (14.22)$$

and then compute the similarity score, reusing the datapath of the *Similarity Unit*. The phase differences address a shared cosine LUT to map angles to fixed-point values, which are then accumulated via a pipelined adder tree:

$$\text{sim}_r = \frac{1}{D} \sum_{d=1}^D \cos(\Delta q_d^{(r)}). \quad (14.23)$$

This design choice maximizes hardware efficiency by sharing the LUT and reduction logic between the standard similarity instruction and the resonator.

As the pipeline processes each candidate r , it asserts a `sim_valid` pulse and produces a scalar similarity score sim_r . These scalars are immediately forwarded to the next stage, effectively generating the similarity vector

$$\boldsymbol{\sigma}_j = [\text{sim}_0, \dots, \text{sim}_{m_j-1}]^\top.$$

element-by-element in the configured (FP_INT, FP_FRAC) fixed-point format.

3. **Stage 3: Weighted reconstruction in Cartesian space.** To reconstruct the refined factor, stage 3 performs a weighted superposition of the codebook vectors, computing the matrix–vector product between Z_j^{cart} and the similarity vector $\boldsymbol{\sigma}_j$:

$$\tilde{\mathbf{z}}_j^{\text{cart}} = Z_j^{\text{cart}} \boldsymbol{\sigma}_j = \sum_{r=0}^{m_j-1} \text{sim}_r \mathbf{z}_{j,r}^{\text{cart}}. \quad (14.24)$$

This step must be performed in cartesian form. Therefore, to avoid continuous phase–cartesian conversions, a secondary copy of each codebook is stored in *cartesian form* Z_j^{cart} , using the configurable fixed-point format.

In hardware, this is realized by a dedicated SIMD multiply-accumulate unit that maintains a bank of per-lane accumulators. For each incoming similarity pulse sim_r , it multiplies the scalar weight by the cartesian components of $\mathbf{z}_{j,r}^{\text{cart}}$ and accumulates the result. This effectively integrates the contribution of all candidates into the reconstructed vector $\tilde{\mathbf{z}}_j^{\text{cart}}$. Once all similarities have been consumed, the accumulated Cartesian vector for factor j is passed to the final normalization stage.

4. **Stage 4: CORDIC-based phase extraction and normalization.** The Cartesian reconstruction $\tilde{\mathbf{z}}_j^{\text{cart}}$ is generally not unit-modulus. To restore the valid FHRR format, the final stage normalizes each Cartesian pair back to a discretized phase using the shared CORDIC engine (see the bundling unit).
5. **Stage 5: In-place update and convergence.** Finally, the new discrete-phase vector $\hat{\mathbf{z}}_j^{(t+1)}$ is written back to the estimators memory, overwriting the previous estimate $\hat{\mathbf{z}}_j^{(t)}$. The FSM controller advances to the next factor $j + 1$ and repeats the process until global convergence is detected or the maximum iteration count is reached.

Overall, the Resonator Network unit implements the full factor-recovery dynamics in hardware using Q_ϕ -bit phases for all angular quantities and a configurable fixed-point format (FP_INT, FP_FRAC) for similarities and weights. Once the pipeline is filled, the residual estimation stage continuously produces residual chunks, the similarity datapath streams them against the phase codebooks to generate similarities, and the multiply-accumulate unit consumes each similarity as soon as it is available to update the estimate in a SIMD MAC fashion, maximizing overlap between stages and throughput per resonance iteration.

14.5.5 Preliminary Synthesis Results

To evaluate the hardware complexity and scalability of the proposed design, we synthesized the FHRR accelerator targeting the Xilinx Zynq UltraScale+ ZCU106 evaluation board (device part XCZU7EV), using Vivado 2024.1. The design was synthesized with a fixed hypervector dimensionality of $D = 1024$, a phase resolution of $Q = 1024$ (10-bit), and a Q16.16 fixed-point format for internal Cartesian/similarity arithmetic. The choice of a 10-bit phase quantization for these preliminary results was determined by verifying that this resolution is sufficient to successfully model and solve the subset-sum problem using FHRR encoding and resonator network decoding [342]. Future experiments will evaluate the impact of higher resolutions, which may be needed for more complex learning tasks.

Table 14.1 summarizes the post-implementation resource utilization for SIMD parallelism widths ranging from 2 to 16.

The synthesis results confirm that the proposed design is remarkably compact, rendering the deployment of FHRR feasible even on resource-constrained embedded

Table 14.1 Post-Implementation Resource Utilization on ZCU106 (XCZU7EV) for varying SIMD widths ($D = 1024$, $Q_\phi = 10$, Q16.16).

Configuration	LUT	FF	BRAM	DSP
SIMD 2	3645	1315	26	26
SIMD 4	6613	2152	29	52
SIMD 8	12820	3846	34	104
SIMD 16	25821	7459	40	208

systems. This efficiency is a direct consequence of the adopted *discretized phase approach*: by mapping complex-valued algebra to integer modular arithmetic, we avoid the prohibitive cost of wide floating-point units and complex multipliers for the majority of operations.

Furthermore, the results demonstrate that the SIMD parameter serves as an effective knob for design space exploration. As shown in Table 14.1, resource utilization scales linearly with hardware parallelism. This allows designers to trade off execution speed against area, adapting the accelerator precisely to the latency requirements and silicon budget of the specific target application.

14.6 Summary of Findings

This chapter introduced a hardware-friendly discretization strategy for FHRR and presented the first dedicated accelerator for this paradigm. By discretizing phases on the unit circle, we enable modular arithmetic without explicit modulo operators, mirroring the approach that proved successful for MCR.

It is important to highlight that the results presented here are part of an ongoing research effort. At the time of writing, extensive benchmarking is being conducted to further characterize the accelerator’s performance. The final version of this work will include a comprehensive evaluation of execution time speedup and energy consumption across standard datasets, mirroring the rigorous evaluation methodology applied in Chapters 12 and 13.

Chapter 15

Solving 3-SAT Problems Using Resonator Networks and FHRR

"Hard problems are hard because they are worth solving."

Anonymous

The previous chapters developed dedicated hardware accelerators for progressively more expressive HD/VSA models: from binary spatter codes (HDCU, Chapter 12) to modular composite representations (MCR-HDCU, Chapter 13) to Fourier Holographic Reduced Representations (Chapter 14) and the first hardware implementation of a Resonator Network.

This chapter departs from the hardware focus of this part to explore the expressive power of FHRR and Resonator Networks for *combinatorial and symbolic reasoning*. While the investigation is purely algorithmic, it directly informs the capabilities of the resonator network unit designed in the previous chapter and identifies constraints that future accelerators targeting symbolic workloads must satisfy.

Recent works have begun to explore this direction by utilizing hypervectors as state spaces for symbolic search [248, 249, 342]. In these models, a single hypervector encodes an entire structured configuration—such as a visual scene, a subset of integers, or the attributes of a Raven’s matrix panel—and a *Resonator Network* is used to factorize this state. The network runs an iterative dynamics over a set of factors, updating them to maximize the similarity between the current composite estimate and the target hypervector. When the dynamics converge to a fixed point, the resulting state represents a factorization that “explains” the target. This mechanism creates a powerful link between neuro-inspired oscillatory dynamics and discrete optimization, closely paralleling classical models of associative memory in the hippocampal formation and cortex, in which distributed neural populations evolve toward stable activity patterns representing stored memories or candidate interpretations of sensory input.

Recent work [342] has demonstrated this resonator-based approach on the Subset Sum problem, showing that FHRR can indeed solve combinatorial tasks. However, it remains an open question how far this mechanism can be pushed along the axis of computational complexity. This chapter addresses that challenge by applying FHRR

and Resonator Networks to 3-SAT: a canonical NP-complete problem to which all others are reducible, serving as the ultimate stress test for any computational paradigm claiming symbolic reasoning capabilities.

To the best of the author’s knowledge, this is the first exploration of solving 3-SAT using the FHRR/Resonator framework. We guide this investigation through three specific research questions:

? Research Question

1. Can a resonator network operating on FHRR representations reliably solve instances of 3-SAT, and up to which problem sizes?
2. How do architectural and representation choices—such as the number of factors, codebook design, numeric base, and activation functions—affect the probability of converging to a satisfying assignment?
3. How do the observed behaviors relate to the known resonator-network capacity laws as the effective search space grows?

To answer these questions, we exploit a classical polynomial-time reduction from 3-SAT to Subset Sum, encode the problem in the high-dimensional space using the machinery introduced in [342], and leverage a resonator network to search for satisfying assignments.

The results show that resonator networks *can* model and solve 3-SAT problems, but their performance follows the expected scaling with search-space size and resonator capacity. Up to modest instance sizes, the network finds satisfying assignments with a reasonable number of trials; beyond that regime, success rates quickly collapse in a way that is consistent with theoretical capacity limits. Reducing the number of factors, tightening the numeric range, and adjusting activation functions all have measurable impact on success probability, but do not fundamentally overcome the capacity bottleneck—a useful insight for future neuro-inspired approaches to NP-complete problems. These findings also validate the broader applicability of the FHRR accelerator introduced in Chapter 14 for symbolic reasoning workloads.

The chapter is organized as follows. Section 15.1 reviews NP-complete problems and introduces 3-SAT and Subset Sum. Section 15.2 summarizes how FHRR and resonator networks solve Subset Sum in [342]. Section 15.3 presents the reduction from 3-SAT to Subset Sum. Section 15.8 reports experimental results across problem sizes and architectural variations. Finally, Section 15.9 discusses limitations and future directions.

15.1 Complexity: P, NP, and NP-Complete Problems

Many important computational tasks in artificial intelligence, operations research, and formal verification can be framed as *combinatorial* search problems. In these problems, an instance consists of a set of discrete variables and a collection of constraints that determine which variable combinations are valid. Solving the problem typically requires exploring a discrete space that grows exponentially with the number of variables to find a satisfying configuration.

In complexity theory, such tasks are usually formalized as *decision problems*. A

decision problem is identified with a language $L \subseteq \{0, 1\}^*$, where $\{0, 1\}^*$ denotes the set of all finite binary strings. An instance of the problem is a string $x \in \{0, 1\}^*$, and the question is whether $x \in L$ (answer “yes”) or $x \notin L$ (answer “no”). An algorithm *solves* a decision problem L if, for every input x , it outputs the correct yes/no answer.

15.1.1 Complexity Classes

Class P (Polynomial Time) The class **P** collects all decision problems that can be solved in polynomial time by a deterministic algorithm. This means that the running time of the algorithm on inputs of length n is bounded by $p(n)$ for some polynomial p . Intuitively, these are the problems that are considered efficiently solvable (e.g., sorting, shortest path).

Class NP (Nondeterministic Polynomial Time) The class **NP** is defined not by the difficulty of *finding* a solution, but by the efficiency of *verifying* one. Intuitively, a decision problem is in **NP** if, for every “yes” instance, there exists a *witness* (also called certificate) that can be checked in polynomial time.

Formally, a decision problem $L \subseteq \{0, 1\}^*$ belongs to **NP** if there exists a polynomial p and a deterministic polynomial-time algorithm V (the *verifier*) such that for every input $x \in \{0, 1\}^*$:

$$x \in L \iff \exists y \in \{0, 1\}^* \text{ with } |y| \leq p(|x|) \text{ and } V(x, y) \text{ accepts.}$$

In other words, membership in L can be certified by a *short* witness y , and the correctness of this certificate can be checked in polynomial time. For example, for 3-SAT the witness can be a Boolean assignment that satisfies all clauses; for Subset Sum it can be a bit mask indicating which integers are chosen in the subset. In both these cases, finding a solution may be hard, but checking it is trivial.

15.1.2 Reductions and NP-Completeness

To compare the difficulty of decision problems, *polynomial-time reductions* are used. Given two languages L_1 and L_2 , a polynomial-time reduction from L_1 to L_2 is a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that:

- f is computable in time polynomial in $|x|$;
- for all $x \in \{0, 1\}^*$, $x \in L_1 \iff f(x) \in L_2$.

Intuitively, f acts as an efficient “translator.” If $L_1 \leq_p L_2$, then L_1 is *no harder* than L_2 , because any algorithm that solves L_2 can be used to solve L_1 by simply translating the input first.

NP-Completeness Given these notions, a decision problem L is called *NP-hard* if every problem in **NP** can be reduced to it in polynomial time, i.e., for all $L' \in \mathbf{NP}$, $L' \leq_p L$. If, in addition, $L \in \mathbf{NP}$, then L is called *NP-complete*. NP-complete problems are therefore the “hardest” problems in **NP** under polynomial-time reductions. If a polynomial-time algorithm were found for *any* NP-complete

problem, it could be used to solve *every* problem in **NP** efficiently (implying $\mathbf{P} = \mathbf{NP}$).

NP-complete problems are ubiquitous and capture a broad range of practically relevant tasks, including planning, scheduling, graph coloring, and circuit verification. They are therefore natural stress tests for novel computational paradigms. Among the many known NP-complete problems, this chapter focuses on two canonical examples that are tightly connected and will be the focus of the following sections:

- the *Subset Sum* problem, which asks whether a target integer can be represented as the sum of a subset of given integers;
- the *3-SAT* problem, which asks whether there exists a Boolean assignment that satisfies a conjunction of clauses, each clause being a disjunction of exactly three literals.

Crucially for this work, classical complexity theory provides a constructive reduction from 3-SAT to SUBSET-SUM. In the following sections, we leverage this reduction to solve 3-SAT instances using the resonator network originally designed in [342].

15.1.3 Subset Sum and 3-SAT as Canonical NP-Complete Problems

This chapter focuses on two classical NP-complete problems that are tightly connected: *Subset Sum* and *3-SAT*.

Subset Sum Given a multiset of positive integers $S = \{s_1, \dots, s_K\} \subset \mathbb{N}$ and a target integer $T \in \mathbb{N}$, the SUBSET-SUM problem asks whether there exists a submultiset $S^* \subseteq S$ such that:

$$\sum_{s \in S^*} s = T. \quad (15.1)$$

This problem belongs to **NP** because a “yes” answer admits a compact witness: a binary mask $\mathbf{b} \in \{0, 1\}^K$ indicating inclusion in S^* . A verifier can simply compute the dot product $\mathbf{b} \cdot \mathbf{s}$ and check if it equals T in polynomial time.

3-SAT The 3-SAT problem is the canonical restriction of Boolean satisfiability to Conjunctive Normal Form (CNF) where every clause has exactly three literals. Consider n Boolean variables $x_1, \dots, x_n$ and a formula Φ consisting of m clauses:

$$\Phi = C_1 \wedge C_2 \wedge \dots \wedge C_m, \quad (15.2)$$

where each clause is a disjunction of three literals: $C_j = \ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$. Each literal ℓ represents either a variable x_i or its negation $\neg x_i$. The decision problem asks if there exists an assignment $\mathbf{x} \in \{0, 1\}^n$ such that $\Phi(\mathbf{x}) = \text{true}$. Like Subset Sum, 3-SAT is NP-complete because a “yes” instance admits a natural witness: a Boolean assignment that satisfies all clauses. A verifier can check this in time polynomial in the size of Φ by scanning all clauses once.

Both Subset Sum and 3-SAT are not only in **NP** but are also NP-hard under polynomial-time reductions, and are therefore NP-complete. In particular, there

exist classical reductions from 3-SAT to SUBSET-SUM that map any Boolean formula Φ to an integer instance (S, T) such that

$$\Phi \text{ is satisfiable} \iff (S, T) \in \text{SUBSET-SUM}. \quad (15.3)$$

This connection is crucial for this chapter, as it allows converting 3-SAT instances into Subset Sum ones and then apply the encoding and resonator machinery originally developed for the latter in [342].

15.2 Residue Hyperdimensional Computing and Subset Sum

Chapter 14 introduced Residue Hyperdimensional Computing, which combines FHRR with residue number systems to represent integers as complex unit-magnitude hypervectors and to implement arithmetic via simple component-wise operations. The study in [342] demonstrated that a Subset Sum problem can be solved by encoding the target integer T using RHC and then factorizing it with a resonator network:

1. The target integer T is first encoded as a residue hypervector $\mathbf{z}(T)$ using the RHC scheme described in Chapter 14. This hypervector plays the role of the *composite* vector that the resonator network must factorize.
2. For each element $s_k \in S$, a *factor* is introduced with a binary codebook:

$$\mathcal{Z}_k = \{\mathbf{z}(0), \mathbf{z}(s_k)\},$$

where $\mathbf{z}(0)$ corresponds to “element not used” and $\mathbf{z}(s_k)$ corresponds to “element used”. Each factor thus encodes the local decision of including or excluding s_k from the subset.

3. A resonator network is constructed over these K factors. At each iteration, one factor is updated at a time and the dynamics continue until convergence or until a maximum number of iterations is reached.

The state of the resonator at any time corresponds to a candidate subset: for each factor k , the choice between $\mathbf{z}(0)$ and $\mathbf{z}(s_k)$ indicates whether s_k is included. If the network converges to a fixed point in which the product of the selected factor entries exactly matches $\mathbf{z}(T)$ (up to noise and discretization), the resulting subset S^* satisfies

$$\sum_{s \in S^*} s = T,$$

and the Subset Sum instance is a “yes” instance. Conversely, if the resonator fails to find such a fixed point over multiple initializations, the instance is likely a “no” instance or beyond the operational capacity of the network.

The key advantage is that the exponentially large space of subsets (of size 2^K) is explored implicitly by the resonator dynamics in superposition, rather than by explicit enumeration.

15.3 Encoding 3-SAT as Subset Sum

To apply the resonator network to 3-SAT, a 3-SAT instance is first converted into an equivalent Subset Sum instance. This is done through a classical polynomial-time reduction from 3-SAT to Subset-Sum. The construction is best introduced on a small concrete example and then generalized.

15.3.1 Step-by-Step Example: 3 Variables and 3 Clauses

Consider a 3-SAT formula with three variables and three clauses:

$$\Phi = C_1 \wedge C_2 \wedge C_3,$$

over variables x_1, x_2, x_3 , with

$$C_1 = x_1 \vee x_2 \vee x_3, \quad C_2 = x_1 \vee x_2 \vee \neg x_3, \quad C_3 = \neg x_1 \vee x_2 \vee x_3.$$

The aim is to construct a Subset Sum instance (S, T) such that

$$\Phi \text{ is satisfiable} \iff \exists S^* \subseteq S \text{ with } \sum_{s \in S^*} s = T.$$

Step 1: Literal–clause incidence The first step is to record in which clauses each literal appears. Each row corresponds to a literal (x_i or $\neg x_i$) and each column to a clause:

	C_1	C_2	C_3
x_1	1	1	0
$\neg x_1$	1	0	1
x_2	1	1	1
$\neg x_2$	0	0	0
x_3	1	0	1
$\neg x_3$	0	1	0

In this matrix, a “1” indicates that the literal appears in the clause. For instance, x_1 appears in C_1 and C_2 but not in C_3 , giving the pattern $(1, 1, 0)$.

This table captures *which* literals are able to satisfy each clause, but it does not yet enforce variable consistency (a variable must not be both true and false at the same time) or the requirement that *each* clause in the problem is satisfied.

Step 2: Adding a variable block To enforce variable consistency, an additional group of digits (a *variable block*) is introduced. One dedicated digit position is assigned to each variable. For this example, three variable positions are used, denoted by (V_1, V_2, V_3) , one for each of x_1, x_2, x_3 . The idea is that selecting a literal for variable x_i contributes a “1” in the V_i position; selecting both x_i and $\neg x_i$ would contribute a “2” in V_i , which will be incompatible with the chosen target.

The rows now become 6-digit patterns: three variable digits followed by three clause digits $(V_1, V_2, V_3 \mid C_1, C_2, C_3)$.

	V_1	V_2	V_3	C_1	C_2	C_3
x_1	1	0	0	1	1	0
$\neg x_1$	1	0	0	1	0	1
x_2	0	1	0	1	1	1
$\neg x_2$	0	1	0	0	0	0
x_3	0	0	1	1	0	1
$\neg x_3$	0	0	1	0	1	0

Step 3: Adding dummy rows for clauses The next step is to enforce that each clause in the problem is satisfied. Recall that each clause C_j is a disjunction (logical OR) of exactly three literals. This means that C_j is satisfied if *at least one* of its three literals is set to true, and it can be satisfied by one, two, or all three of its literals.

To model this in terms of a subset problem, we fix a target of *exactly 3* per clause (for each column in the clause block), that is the maximum possible number of contributing literals in a 3-literal clause, and we introduce two *dummy* rows for each clause C_j :

$$D_1^{(j)}, D_2^{(j)},$$

The role of these dummies is to “fill up” the clause digit to exactly 3, depending on how many literals for that clause are chosen. In this way:

- if one literal of C_j is selected (contribution 1), choose $D_2^{(j)}$ (dummy 2) to reach $1 + 2 = 3$;
- if two literals are selected (contribution 2), choose $D_1^{(j)}$ to reach $2 + 1 = 3$;
- if all three literals are selected (contribution 3), do not pick any dummy row to keep the total at $3 + 0 = 3$;
- if no literal is selected (contribution 0), none of the available dummies (0, 1, 2) can bring the total to 3, so the target cannot be met and the clause is necessarily unsatisfied.

Row	V_1	V_2	V_3	C_1	C_2	C_3
x_1	1	0	0	1	1	0
$\neg x_1$	1	0	0	1	0	1
x_2	0	1	0	1	1	1
$\neg x_2$	0	1	0	0	0	0
x_3	0	0	1	1	0	1
$\neg x_3$	0	0	1	0	1	0
$D_1^{(1)}$	0	0	0	1	0	0
$D_2^{(1)}$	0	0	0	2	0	0
$D_1^{(2)}$	0	0	0	0	1	0
$D_2^{(2)}$	0	0	0	0	2	0
$D_1^{(3)}$	0	0	0	0	0	1
$D_2^{(3)}$	0	0	0	0	0	2

In this way, the numeric constraint “clause digit must equal 3” is exactly equivalent to the logical requirement that the disjunction of the three literals in C_j is satisfied by at least one selected literal.

Step 4: Defining the target vector Finally, we can choose the target so that:

- the variable block enforces exactly one literal per variable;
- the clause block enforces that each clause receives a total contribution of 3 units.

For this example this corresponds to:

$$T = (1, 1, 1 \mid 3, 3, 3),$$

which can be viewed as a 6-digit integer in base 10. The first three digits (1, 1, 1) enforce that the total number of selected literals per variable is exactly one; the last three digits (3, 3, 3) enforce that each clause receives three units of coverage from literals plus dummies.

The complete set S of integers is obtained by interpreting each row (all literals and all dummies) as a 6-digit vector in base b and mapping it to the corresponding integer.

$$S = \{100110, 100101, 10111, 10000, 1101, 1010, 100, 200, 10, 20, 1, 2\}.$$

15.4 Baseline Formulation

Once the 3-SAT instance has been converted into a Subset Sum instance (S, T) , the next step is to model this Subset Sum problem using high-dimensional representations and a resonator network. The baseline formulation closely follows the strategy used in [342], but here integers are mapped to hypervectors using the FPE of FHRR introduced in Chapter 2, without employing a residue number system.

At a high level, the construction proceeds as follows:

1. **High-dimensional encoding of integers.** The target T and each element $s_k \in S$ (i.e., each row of the Subset Sum table constructed in Section 15.3) are mapped to complex unit-magnitude hypervectors $\mathbf{z}(T)$ and $\mathbf{z}(s_k)$ using the fractional power encoding. Each component of $\mathbf{z}(x)$ lies on the unit circle and represents x as a phase rotation of a random base angle.
2. **Binary codebook per subset element.** Each integer s_k is treated as a separate factor in the resonator network and is associated with a two-entry codebook

$$\mathcal{Z}_k = \{\mathbf{z}(0), \mathbf{z}(s_k)\},$$

corresponding to the choices “do not pick s_k ” and “pick s_k ”. A network configuration obtained by selecting one entry from each $\mathcal{Z}_k$ encodes a candidate subset $S^* \subseteq S$.

3. **Resonator search over all subsets.** A resonator network with $F = K = |S|$ factors is constructed. At each iteration, it updates one factor at a time so as to increase the similarity between the composite estimate

$$\hat{\mathbf{s}}(\mathbf{c}) = \bigodot_{k=1}^K c_k,$$

and the target hypervector $\mathbf{z}(T)$. In this way, the network performs a discrete search over the 2^K possible subsets of S in high-dimensional phase space.

Example: 3 variables and 3 clauses. For the toy 3-SAT instance with three variables and three clauses, the reduction of Section 15.3 produces a Subset Sum instance with

$$K = 2n + 2m = 2 \cdot 3 + 2 \cdot 3 = 12$$

integers: six *literal* rows and six *dummy* rows. Denote them by

$$S = \{s_1, \dots, s_{12}\},$$

ordered, for example, as

$$\begin{array}{ll} s_1 = (x_1), & s_2 = (\neg x_1), \\ s_3 = (x_2), & s_4 = (\neg x_2), \\ s_5 = (x_3), & s_6 = (\neg x_3), \\ s_7 = D_1^{(1)}, & s_8 = D_2^{(1)}, \\ s_9 = D_1^{(2)}, & s_{10} = D_2^{(2)}, \\ s_{11} = D_1^{(3)}, & s_{12} = D_2^{(3)}. \end{array}$$

The resonator network has $K = 12$ factors, one per subset element, and 12 binary codebooks

$$\mathcal{Z}_k = \{\mathbf{z}(0), \mathbf{z}(s_k)\}, \quad k = 1, \dots, 12.$$

A network state chooses, for each row in the table of Section 15.3, whether that row is *active* (picked) or *inactive* (not picked). When the hypervectors corresponding to active rows are bound together, the resulting composite $\hat{\mathbf{s}}(\mathbf{c})$ is intended to match the encoding of the target

$$\mathbf{s} = \mathbf{z}(T), \quad T = (1, 1, 1 \mid 3, 3, 3).$$

A run is declared *successful* if, at convergence, the composite estimate $\hat{\mathbf{s}}(\mathbf{c})$ corresponds to $\mathbf{z}(T)$ and the active subset S^* satisfies

$$\sum_{s \in S^*} s = T.$$

By the construction of Section 15.3, this is equivalent to having found a satisfying assignment for the original 3-SAT formula.

Table 15.1 Baseline success rate of the resonator network on 3-SAT instances using the Subset Sum reduction of Section 15.3 and 2-entry (pick / not-pick) codebooks for each subset element (base-10 representation, $D = 10,000$).

Variables	Clauses	Success rate
3	3	99.2%
3	5	78.6%
3	10	52.0%
10	10	6.0%
20	20	5.0%
20	40	1.5%
20	50	0.4%
20	60	0.0%
21	90	0.0%

Baseline Results The baseline resonator formulation described in this Chapter was evaluated on random satisfiable 3-SAT instances of increasing size. For each pair $(n_{\text{vars}}, n_{\text{clauses}})$, a set of formulas was generated, mapped to Subset Sum as in Section 15.3, and then encoded into hypervectors of dimension $D = 10,000$. A run was counted as successful if, at convergence, the active subset summed to the target T , i.e., the corresponding assignment satisfied all clauses.

Table 15.1 reports the measured success rate of this baseline configuration as a function of the number of variables and clauses.

The resonator reliably solves very small instances (e.g., 3 variables and 3 clauses), achieving success rates close to 100%. As the problem size increases, the success rate decreases, but it is important to interpret these numbers in probabilistic terms. A success rate of, say, 10% still means that a satisfying assignment is typically found after $\mathcal{O}(10)$ independent runs of the resonator, which can be acceptable if each run is fast and can be parallelized in hardware. The critical regime is when the success rate approaches zero: for larger instances such as (20, 60) or (21, 90) essentially no successful runs are observed within the allowed number of iterations, indicating that the correct solution lies outside the effective basin of attraction of the resonator dynamics at dimension $D = 10,000$.

These baseline results motivate a series of modifications to the representation and architecture, aimed at probing whether performance can be improved by:

1. **Reducing the number of factors**, by grouping multiple subset elements into joint codebooks, and thus shrinking the effective search space;
2. **Reducing the magnitude of the integers**, by changing the numeric base or simplifying the digit structure;
3. **Altering the resonator dynamics**, by modifying the initialization strategy or the non-linear decision rule used to choose codebook entries at each update.

The following sections present these variants one by one, and analyze how they shift the empirical success rates relative to the baseline.

15.5 Optimization 1: enforcing variable consistency via codebooks and removing leading digits

The first modification exploits a simple but important property of the resonator formulation: for each factor, the network is *forced* to select exactly one entry from its codebook. This observation can be used to enforce variable consistency directly at the level of codebooks, rather than through the presence of leading "1"s in the integer representation.

As explained, in the classical 3-SAT to Subset Sum reduction, the initial columns of the integer representation enforce that each variable is used exactly once: selecting both x_i and $\neg x_i$ would produce a "2" in the i -th variable position, which cannot match the target. This mechanism is needed when arbitrary subsets of rows can in principle be chosen.

In the resonator setting, however, each factor must always be in exactly one state from its codebook. This makes it natural to merge the two literals of a variable into a single *variable factor* with a 2-entry codebook:

$$\mathcal{Z}_i^{(x)} = \{\mathbf{z}(x_i), \mathbf{z}(\neg x_i)\},$$

so that the resonator explicitly chooses between x_i and $\neg x_i$ for each variable x_i . Under this parametrization, it is impossible to "pick both" x_i and $\neg x_i$, because there is no codebook state that encodes such an inconsistent choice.

Once variable consistency is enforced in this way, the leading presence digits become redundant: the resonator's own constraint (exactly one state per factor) already ensures that each variable is assigned exactly one truth value. This suggests a simple numeric simplification: remove the leading variable-related columns from the Subset Sum representation and keep only the clause-related digits. In the toy example, this corresponds to discarding the (V_1, V_2, V_3) positions and working only with the (C_1, C_2, C_3) digits.

Row	V_1	V_2	V_3	C_1	C_2	C_3		Row	C_1	C_2	C_3
x_1	1	0	0	1	1	0		x_1	1	1	0
$\neg x_1$	1	0	0	1	0	1		$\neg x_1$	1	0	1
x_2	0	1	0	1	1	1		x_2	1	1	1
$\neg x_2$	0	1	0	0	0	0		$\neg x_2$	0	0	0
x_3	0	0	1	1	0	1		x_3	1	0	1
$\neg x_3$	0	0	1	0	1	0		$\neg x_3$	0	1	0
$D_1^{(1)}$	0	0	0	1	0	0	$\longrightarrow$	$D_1^{(1)}$	1	0	0
$D_2^{(1)}$	0	0	0	2	0	0		$D_2^{(1)}$	2	0	0
$D_1^{(2)}$	0	0	0	0	1	0		$D_1^{(2)}$	0	1	0
$D_2^{(2)}$	0	0	0	0	2	0		$D_2^{(2)}$	0	2	0
$D_1^{(3)}$	0	0	0	0	0	1		$D_1^{(3)}$	0	0	1
$D_2^{(3)}$	0	0	0	0	0	2		$D_2^{(3)}$	0	0	2

More generally, for an instance with n variables and m clauses, the target simplifies to

$$T = (3, \dots, 3) \in \mathbb{N}^m,$$

and each integer encodes only its contribution to clause coverage.

This modification reduces the overall magnitude of the integers and the number of significant digits, which could in principle improve the numerical behavior of the fractional power encoding. This optimization was evaluated on random 3-SAT instances. The result demonstrated that removing the leading ones produced essentially *identical* success rates to the baseline, confirming that the resonator choice mechanism already enforces variable consistency and that these columns are not informative in this formulation.

15.6 Optimization 2: grouping codebooks to reduce the number of factors

A second hypothesis is that the baseline formulation may be using “too many” factors, making the dynamics harder to stabilize. In the baseline, every literal and dummy row is an independent factor, creating a search space of size 2^K .

To counteract this, we propose Factor Grouping. This strategy bundles multiple subset elements into a single *joint factor* with a larger codebook. Conceptually, this trades off factor cardinality (number of factors) against codebook size (states per factor), potentially stabilizing the dynamics by reducing the dimensionality of the attractor landscape.

For example, consider grouping two variables x_1 and x_2 . Instead of maintaining two independent binary codebooks:

$$\{x_{1,\text{used}}, x_{1,\text{unused}}\}, \quad \{x_{2,\text{used}}, x_{2,\text{unused}}\},$$

we define a single quaternary codebook that enumerates all joint choices:

$$\{x_{1,\text{unused}}x_{2,\text{unused}}, x_{1,\text{used}}x_{2,\text{unused}}, x_{1,\text{unused}}x_{2,\text{used}}, x_{1,\text{used}}x_{2,\text{used}}\},$$

Here, each entry corresponds to a particular combination of the two decisions (both unused, only x_1 used, only x_2 used, both used). This codebook is constructed by directly encoding the combined state “ (x_1, x_2) ” as the FPE of a single integer representing their sum “ $(x_1 + x_2)$ ”.

Table 15.2 presents the results of this optimization. We define parameters `var_group` and `clauses_group` to denote the number of original elements bundled into each joint factor. Grouping clearly helps in several regimes (e.g., (3, 5) and (10, 10)), boosting success rates by factors of about 2–3× compared to the baseline. Intuitively, fewer factors mean fewer independent “knobs” for the dynamics to coordinate, which can enlarge the basins of attraction of good solutions even if the theoretical search space size is not changed. However, the improvements are not sufficient to fully overcome the exponential growth of the search space: for larger instances, success remains close to zero.

15.7 Optimization 3: changing the numeric base

Another possible challenge for the resonator network is the magnitude of the integers being encoded. In our reduction, each integer is constructed digit-wise from clause

Table 15.2 Success rate with dedicated vs grouped codebooks. Grouping reduces the number of factors and generally improves performance, especially for intermediate problem sizes.

Variables	Clauses	Dedicated codebooks	Grouped codebooks
3	3	99.2%	100% (var_group = 3, clauses_group = 3)
3	5	78.6%	97.6% (var_group = 3, clauses_group = 5)
3	10	52.0%	52.0% (var_group = 3, clauses_group = 2)
10	10	6.0%	20.0% (var_group = 2, clauses_group = 2)
20	20	5.0%	9.0% (var_group = 2, clauses_group = 2)
20	40	1.5%	1.5% (var_group = 2, clauses_group = 2)
20	50	0.4%	0.5% (var_group = 2, clauses_group = 2)
20	60	0.0%	0.2% (var_group = 2, clauses_group = 2)
21	90	0.0%	0.1% (var_group = 2, clauses_group = 2)

contributions and interpreted as a base- b number. With $b = 10$ and m clauses, the target vector has the form:

$$T = (\underbrace{3, \dots, 3}_{m \text{ digits}}) \approx 3 \cdot 10^{m-1}.$$

As m increases, these values grow exponentially. For example, T grows from 333 ($m = 3$) to 333,333 ($m = 6$).

In FHRR’s FPE, an integer x maps to a phase rotation $\theta = \phi x \pmod{2\pi}$. When x is very large, the term ϕx wraps around the unit circle many times. This high frequency of wrapping compresses the distance between distinct integers in the phase domain, reducing the noise margin.

To mitigate this, we observe that the Subset Sum construction only utilizes digits in the range $\{0, 1, 2, 3\}$ (sums of literals and dummies). This implies that base-10 is unnecessarily sparse. We propose replacing it with base-4 representation, the minimal base required to represent these digits without ambiguity.

Concretely, the digit pattern $(3, 3, 3)$ is interpreted as:

$$333_{(\text{base } 10)} = 333 \longrightarrow 333_{(\text{base } 4)} = 3 \cdot 4^2 + 3 \cdot 4 + 3 = 63.$$

The magnitude reduction is dramatic. For $m = 6$:

$$333,333_{(\text{base } 10)} = 3.3 \times 10^5 \quad \text{vs.} \quad 333,333_{(\text{base } 4)} = 4^6 - 1 = 4,095.$$

By reducing the dynamic range of the encoded integers, we reduce the frequency of phase wraps, ensuring that the resulting cosine similarities remain cleaner and more distinct.

Alternative initializations and activation functions Beyond representation, we explored two degrees of freedom in the resonator dynamics to improve convergence reliability.

- **Initialization strategies.** Instead of initializing all factors uniformly at random, some experiments biased the initial state toward particular entries

(e.g., favoring "not pick" states, or favoring certain literal patterns) to test whether a better starting point could increase the probability of converging to a satisfying assignment.

- **Non-linear selection rules.** The standard resonator uses a WTA update, which commits to the single best codebook entry at each step. To avoid getting trapped in local minima early in the search, we experimented with **Softmax** and **Top- k** selection rules. These allow the network to maintain a superposition of multiple likely hypotheses, sharpening the selection only as the system cools.

15.8 Combined optimizations

To investigate the cumulative impact of the proposed architectural and numerical choices, we evaluated a sweep of configurations on a fixed problem size of $(n_{\text{vars}}, n_{\text{clauses}}) = (20, 30)$. The design space included:

- **Grouping:** Variable and Clause group sizes.
- **Representation:** Numeric base (10 vs 4) and presence of leading digits.
- **Dynamics:** Initialization bias and activation rules (WTA, Softmax, Top- k).

Table 15.3 reports a representative subset of these experiments.

Table 15.3 Representative configurations and success rates for 3-SAT instances with $n_{\text{vars}} = 20$ and $n_{\text{clauses}} = 30$ under different combinations of grouping, numeric representation, and activation rule (all at $D = 10,000$).

Grouping (var, clause)	Base / leading digits	Activation rule	Success rate
none	base-10, yes	WTA	0.0%
none	base-4, no	WTA	2.7%
(2, 2)	base-4, no	WTA	3.0%
(2, 5)	base-4, no	WTA	3.3%
(2, 10)	base-4, no	WTA	2.6%
(5, 2)	base-4, no	WTA	2.7%
(5, 5)	base-4, no	WTA	3.8%
(5, 10)	base-4, no	WTA	2.7%
(5, 5)	base-4, no	softmax ($100 \rightarrow 0.2$)	5.8%
(5, 5)	base-4, no	top- k -sort ($k = 18$)	6.5%

The results demonstrate that the adopted strategies definitely play a role. While the baseline configuration fails completely (0.0%) to solve the 3-SAT problem with $(n_{\text{vars}}, n_{\text{clauses}}) = (20, 30)$, the combined optimizations, specifically Factor Grouping and Soft activation, can recover a usable success rate of 6.5%. This confirms that the proposed techniques effectively smooth the energy landscape and improve search. However, the fact that even the most optimized configurations fail in $> 90\%$ of trials suggests that we are battling a fundamental limitation rather than a mere parameter tuning issue.

Overloaded-attractor behavior and oracle initialization. The previous results demonstrated that for small instances, the resonator quickly converges to a satisfying assignment, while as the number of variables and clauses increases, success rates drop sharply. This precipitous drop points to an "overloaded attractor" network whose capacity has been exceeded: the resonator dynamics settle into spurious fixed points because the correct solution is no longer a stable attractor.

To confirm this hypothesis and verify if our 3-SAT formulation is hitting a capacity wall, we conducted a stability analysis using an "oracle initialization" experiment. This experiment aimed to decouple the *search* difficulty from the *storage* capacity. Instead of starting from a random state, the resonator was initialized *exactly* at the factor configuration corresponding to the known satisfying assignment. We then allowed the dynamics to evolve naturally. If the solution lay within a robust basin of attraction, the network should have remained at (or returned to) this state. However, if the network drifts away from the initialized solution, it proves that the satisfying assignment is not a stable fixed point of the system.

The experiments showed a clear threshold behavior:

- with base-10 representation, initializing at the correct solution yields 100% success up to approximately 10 variables and 20 clauses; beyond this size the success rate drops to 0%;
- with base-4 representation, the stable regime extends further, with 100% success up to approximately 20 variables and 40 clauses; again, beyond this point the success rate collapses to 0%.

In other words, once the problem size grows past these thresholds, even the exact solution no longer resides in a robust basin of attraction: the resonator quickly leaves the correct state and converges to spurious fixed points. This confirms that the failure is not simply a search problem, but a stability problem.

Relating the observations to resonator capacity These empirical thresholds align with the theoretical capacity laws derived in [248, 249]. The operational capacity $M_{\max}$ of a resonator is shown to scale approximately as a quadratic function of the hypervector dimension D . For more than three factors ($F \geq 3$), a resonator can factorize a search space of size M with $\gtrsim 95\%$ accuracy as long as

$$M \lesssim \kappa D^2,$$

for a constant κ that depends weakly on the number of factors F . With $D = 10,000$, this quadratic scale is on the order of $D^2 \approx 10^8$.

In our 3-SAT mapping, the effective search space M grows exponentially with problem size:

$$M = 2^{n_{\text{vars}}} \cdot 3^{n_{\text{clauses}}}.$$

Table 15.4 illustrates this scaling.

For small and moderate instances (e.g., (3, 3), (3, 5), (10, 10)), the search-space size M is well below or at most comparable to D^2 , and the resonator achieves good success rates. As the number of variables and clauses increases, M quickly surpasses D^2 by many orders of magnitude. In this regime, success rates collapse in

Table 15.4 Approximate search-space size M for selected 3-SAT configurations under the considered mapping.

Variables	Clauses	Search space M
3	3	3.5×10^1
3	5	2.5×10^2
3	10	5.9×10^4
10	10	6.0×10^4
20	20	3.5×10^9
20	40	1.2×10^{19}
20	50	7.2×10^{23}
20	60	4.2×10^{28}
21	90	8.7×10^{42}

the baseline experiments, and even oracle initialization fails to maintain the correct solution as a stable attractor.

15.9 Summary of Findings

This chapter presented the first exploration of solving NP-complete 3-SAT problems using FHRR and Resonator Networks. Our findings can be summarized as follows:

- **Feasibility:** The reduction from 3-SAT to Subset Sum, combined with FHRR encoding and resonator dynamics, is functionally correct. The resonator reliably solves small combinatorial logic problems, validating the neuro-symbolic mapping.
- **Optimization:** Architectural and numerical optimizations (removing redundant leading digits, grouping codebooks, switching to base-4, and using softmax or top- k activation rules) significantly improve numerical conditioning, extending the solvable regime by effectively compressing the search space and reducing phase noise.
- **Fundamental Limit:** The approach is ultimately bound by the D^2 capacity law. Once the combinatorial search space M exceeds the superposition capacity of the hypervectors, the satisfying assignment ceases to be a stable fixed point and no amount of local tuning in the current framework can fully compensate for this.

These results suggest that while Resonator Networks are powerful tools for factorization, applying them to large-scale NP-complete problems requires strategies to circumvent the exponential growth of M . Future work should investigate new encoding strategies or iterative decomposition approaches that break large 3-SAT instances into smaller sub-problems that fit within the resonator’s quadratic capacity.

Chapter 16

AeneasHDC: An Automatic Framework for Deploying Hyperdimensional Computing Models on FPGAs

Chapter based on Paper 8 [330]: Marco Angioli, Saeid Jamili, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Antonello Rosato, and Mauro Olivieri. Aeneashdc: An automatic framework for deploying hyperdimensional computing models on fpgas. In International Joint Conference on Neural Networks (IJCNN), pages 1–8, 2024. doi: 10.1109/IJCNN60899.2024.10651081

“Progress is made by trial and failure; the failures are generally a hundred times more numerous than the successes.”

William Ramsay

Throughout this thesis, we have demonstrated that HDC provides a compelling paradigm for embedded intelligence: it combines algorithmic simplicity with robustness, inherent parallelism, and low-precision arithmetic inspired by biological computation. We exploited these properties at multiple levels. We introduced HDC-based formulations for online decision-making in contextual bandits and developed lightweight variants tailored to resource-constrained platforms. In parallel, we advanced the hardware dimension of the thesis by designing dedicated hardware architectures for HDC. We began with the HDCU for binary HVs (Chapter 12); we then extended this approach to higher-capacity modular representations with MCR (Chapter 13); and finally explored FHRR (Chapter 14).

However, as HDC models grow in diversity, a fundamental question emerges when developing new algorithms and designing new accelerators while having to satisfy the strict requirements of an embedded application: which model represents the best trade-off between performance and hardware cost under application constraints? How can we navigate the vast HDC design space before committing to the design of a specific and highly optimized hardware accelerator?

Handcrafted accelerators offer maximum performance for final deployment, but they are ill-suited for the exploratory phase of research and development. Determining the optimal combination of hypervector dimensionality, quantization, memory organization, and parallelism for a new task is a labor-intensive process that requires expertise spanning both machine learning and digital design.

This chapter addresses this challenge. We argue that to broaden the adoption of HDC for edge computing, ad-hoc accelerator development must be complemented with *automated design-space exploration*. The approach presented in this chapter is not a replacement for dedicated hardware, but a *bridge* to it: it allows researchers to rapidly prototype, validate, and tune HDC pipelines, generating FPGA implementations that serve primarily as rigorous baselines and design-point evaluations to inform subsequent custom accelerator development. In scenarios where the absolute best efficiency is not required, the same implementations can also be deployed directly.

These considerations motivate the research question guiding this final chapter:

? Research Question

Can we bridge the gap between high-level HDC model specifications and low-level FPGA implementations by automatically generating efficient and configurable hardware accelerators, enabling rapid design-space exploration without requiring expert hardware knowledge?

To answer this question, we introduce *AeneasHDC*, an open-source, end-to-end framework that enables the automatic deployment of HDC models in both software and hardware. AeneasHDC supports classification, clustering, regression, and contextual bandits, and provides a highly configurable workflow—spanning encoding techniques, hypervector types, similarity measures, training and retraining strategies, and memory organization—to explore the accuracy–efficiency trade space in a structured manner.

On the implementation side, the framework automatically generates high-level software implementations (Python, MATLAB, or C++) and synthesizable hardware accelerators via HLS targeting FPGAs. It supports flexible instantiation modes (e.g., inference-only versus full pipeline), selective hardware acceleration of specific pipeline stages, and configurable parallelism to match the constraints of a target device. AeneasHDC also provides detailed evaluation reports through a user-friendly GUI, enabling rapid iteration and fair comparison across design points, and incorporates a Bayesian optimization-based design-space exploration (DSE) strategy that efficiently navigates the combinatorial configuration space, identifying near-optimal operating points with a fraction of the evaluations required by exhaustive search.

We validate AeneasHDC across multiple datasets and learning tasks, illustrating how users can systematically navigate the design space and tune HDC pipelines toward hardware-efficient solutions. Overall, the framework reduces development time substantially (from weeks to minutes in our case studies), enabling reproducible deployment of HDC on embedded FPGA platforms with significantly reduced engineering effort.

Relationship with the published work. This chapter is based on Paper 14 [330], corresponding to the current public release of the framework on GitHub, but includes several substantial extensions that will be published as a follow-up work: support for additional VSA families (MAP-C, FHRR), the random projection encoding technique, CB tasks, the LVQ learning rule, and most notably the Bayesian-guided design-space exploration engine described in Section 16.2.3.

16.1 Related Works

Finding the ideal solution for a specific application adopting a custom design approach can be complex and time-consuming. To alleviate this challenge, the authors in [328] introduced F5-HD, a comprehensive framework designed to automate the deployment of HDC accelerators on FPGAs. F5-HD empowers users to specify problem characteristics, including dataset details, target FPGA, power constraints, and desired trade-offs in accuracy. The framework then automatically configures the architecture by exploiting pre-defined processing elements.

Similarly, the work in [329] introduces HD2FPGA, an automated tool for generating HDC hardware accelerators connected to a host CPU. In addition to what is described for [328], this framework supports clustering problems, different HV sizes, and automatically calibrates hardware parallelism to match user-specific requirements.

However, some limitations persist within these automated models. Neither of these frameworks allows users to customize the details of the HDC model, as they adhere to fixed and predetermined design choices. The chosen encoding technique does not allow for the generation of accelerators that can work with temporal sequences or different input data, nor is it flexible enough to support a variety of applications. The hardware is consistently instantiated for training, inference, and retraining without the flexibility to synthesize only a specific block or explore the effect of hardware parallelism at different levels of the learning process.

On the algorithmic side, part of the design choices available for HDC models [216, 297, 359, 360] has been explored in [305] and [361]. The work in [305], presents HDTorch, a software library with CUDA extensions for testing the impact of different strategies on the model accuracy. In HwAwHDC [361], a framework based on RL optimization is used to search for the optimal trade-off between accuracy, model size, execution time, memory usage, and energy consumption of HDC software models implemented on embedded devices.

Nonetheless, there is a noticeable gap in the literature. Existing work does not provide a straightforward method for systematically testing and exploring the influence of different model design choices on hardware implementations, along with their impact on accuracy. This work seeks to fill this gap by presenting a fully automated, highly flexible environment for deploying HDC models in hardware.

16.2 AeneasHDC

This work presents AeneasHDC, an automatic, open-source platform for simple and fast deployment of HDC models in both software and hardware. The framework is

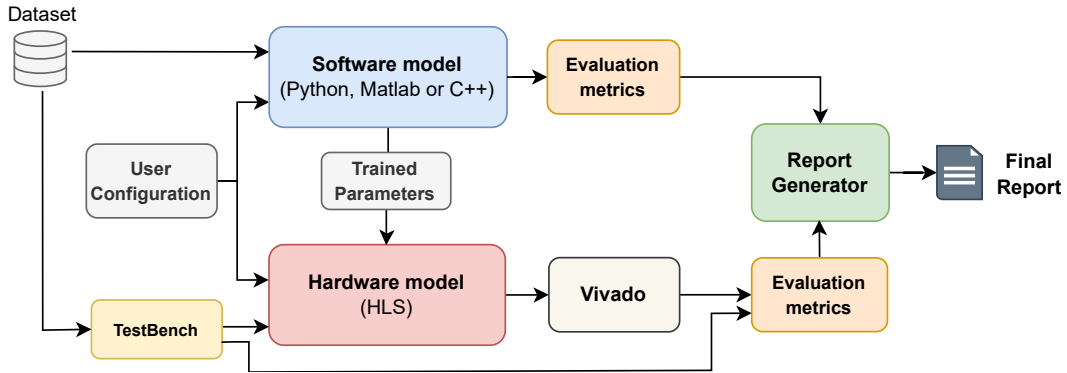


Figure 16.1 Overview of the Aeneas Hypercompute framework

extremely customizable, supporting a wide range of the most common techniques adopted in the literature for learning problems, enabling users to easily assess the impact of different design choices on model accuracy, memory usage, execution time, power consumption, and area requirements.

Fig. 16.1 illustrates the workflow of AeneasHDC. An intuitive Graphic User Interface (GUI) allows specifying the problem characteristics, including dataset details, the number of features and classes, the learning task, the desired programming language, the target FPGA, and a comprehensive set of configuration options for the model. An interactive mechanism lets the user decide how to distribute the workload between the software and the hardware implementations. The hardware design, in fact, is highly customizable, as discussed in Section 16.2.2, and can be generated to accelerate the entire process (train, inference, retrain) or just certain stages. For example, users might choose to train and retrain the model in software and use hardware only for inference, as is commonly done in embedded applications.

Given these input parameters, the system automatically generates all the configuration files required by the following entities and instantiates the model in software and hardware. On the software side, the model can be implemented in Python, MATLAB, or C++, thanks to the versatile AeneasHDC library. The model is executed, and the accuracy metrics are collected. Additionally, the data required for inference is extracted and passed to the hardware model.

The hardware architecture is generated using the Xilinx Vitis High-Level Synthesis (HLS) tool, and it is then synthesized, simulated, and tested using Vivado 2023. Data on power consumption, execution time, and resource allocation is gathered from the design.

The report-generation module generates PDF summaries from software and hardware results, detailing model configuration, dataset, programming language, performance metrics, execution time, memory usage, power consumption, and resource utilization. These reports are stored locally and easily accessible via a user-friendly GUI for reviewing and comparing test runs.

In addition to automatic single-point evaluation, AeneasHDC provides an *automated design-space exploration* (DSE) engine that leverages Bayesian optimization to efficiently navigate the combinatorial configuration space and identify Pareto-optimal design points.

Table 16.1 Explorable design space in the AeneasHDC framework. Parameters are grouped into model architecture, encoding, training, and inference categories.

	Parameter	Explorable Space
Model	Learning Task	[Classification, Clustering, Regression, Contextual Bandits]
	VSA Model	[BSC, MAP-B, MAP-I, MAP-C, FHRR]
	HV Dimension D	Any positive integer, e.g. {128, 256, 512, 1024, 2048, 10000}
	HV Density	[Dense, Sparse]
	HV Sparsity p	Any $p \in (0, 0.5]$, e.g. {0.01, 0.02, 0.05, 0.1, 0.2}
Encod.	Level HV Generation	[Random Flipping, Random Selection, Thermometer]
	Number of Level HVs L	Any positive integer, e.g. {5, 10, 20, 30, 50}
	Encoding Technique	[Record-based, N-gram-based, Random Projection]
	Temporal Encoding	[Enabled, Disabled]
	Temporal Window Size	Any positive integer, e.g. {2, 3, 4, 5}
Train.	Learning Rule	[Centroid, LVQ] (classification only)
	Retraining	[Enabled, Disabled]; epochs: any, e.g. {5, 10, 20, 50}
	Learning Rate Schedule	[Data-dependent, Iteration-dependent]
Inf.	Similarity Measure	[Cosine, Dot Product]
	Quantization	[Binary, Ternary, Power-of-Two, Quantized, Integer, Fixed Point]

The entire described process is engineered to be fully automated, abstracting complex or time-consuming tasks associated with coding models or designing custom hardware architectures. The GUI is an efficient singular point of control that enables users to manage the model at every stage.

AeneasHDC is completely open-source, with all its libraries and scripts available on GitHub [330]. We offer comprehensive documentation that explains basic usage, theoretical concepts behind parameter selections, and detailed insights into the code and library functions. This encourages community collaboration to enhance and evolve the framework over time. Additionally, our GitHub repository includes a suite of tests for assessing the effects of various parameters on hardware implementation in generic learning problems, facilitating practical application and continuous improvement.

16.2.1 Explorable Design Space

AeneasHDC provides a comprehensive range of configuration options for developing HDC models, encompassing most techniques commonly found in the literature and covering the principal VSA families explored throughout this thesis. Table 16.1 lists all the currently supported parameters, organized into four groups: *model architecture*, *encoding*, *training*, and *inference*.

AeneasHDC supports the principal VSA families studied in this thesis and in the broader literature. At the time of writing, supported models include BSC, $\mathcal{H} = \{0, 1\}^D$, MAP-B, $\mathcal{H} = \{\pm 1\}^D$, MAP-I, $\mathcal{H} = \mathbb{Z}^D$, MAP-C, $\mathcal{H} = \mathbb{R}^D$, and FHRR, $\mathcal{H} = \mathbb{C}^D$. This breadth of representation support allows AeneasHDC to serve as a unified experimental platform for comparing the accuracy-efficiency trade-offs across VSA families on the same dataset and target FPGA, a capability that, to the best of our knowledge, no prior framework offers.

AeneasHDC supports the three standard learning tasks—classification, clustering, and regression—as well as contextual bandits, reflecting the central application

domain of this thesis.

AeneasHDC supports three of the encoding strategies introduced in Chapter 2: record-based, N-gram-based and random projection. Three techniques are available for generating level HVs and representing scalar numbers: Random Flipping, Random Selection and Thermometer. The Random Flipping method offers the highest accuracy at the expense of high memory usage. Conversely, the Random Selection and Thermometer methods reduce memory needs and power consumption, respectively.

For temporal data, the framework enables sliding-window encoding by applying n-gram composition over consecutive time steps, with a configurable window size that determines how many consecutive samples are jointly encoded into a single HV.

For classification tasks, two learning rules are available. The *Centroid* rule (introduced in Chapter 2), which updates class HVs by accumulating correctly and incorrectly classified training samples through addition and subtraction, following the standard single-pass and multi-epoch HDC training procedure [362]; and the LVQ rule (extensively introduced in Chapter 13), which instead applies an adaptive step size to both the winning and competing class HVs, providing finer control over the learning dynamics and often yielding improved accuracy.

In the centroid rule, the framework supports iterative retraining over a configurable number of epochs. Two learning rate strategies are available [359]: a data-dependent schedule that adapts the step size to the magnitude of the current prediction error, and an iteration-dependent schedule that decays the learning rate over epochs. Retraining can be conducted on the same training set, as is common in the HDC literature, or on any user-specified dataset.

After training, the class HVs can optionally be quantized to reduce their bit-width, a key strategy for minimizing hardware area, memory, and dynamic power. Available options include binary clipping (1-bit sign), ternary quantization ($\{-1, 0, +1\}$), power-of-two quantization (which replaces multiplications in encoding and similarity with arithmetic shifts), generic q -bit quantized formats, full-precision integer, and fixed-point representation with configurable integer and fractional widths. The choice of quantization directly affects the arithmetic complexity of the generated hardware accelerator.

During inference, the encoded query HV is compared against the stored class (or action) HVs using one of two similarity measures: cosine similarity and dot product. While the similarity measure is in theory fixed and related to the VSA model, as demonstrated in the previous chapters there are cases where the norm of the hvs is known to be constrained or non-relevant, allowing to replace cosine similarity with a lighter alternative that avoids the normalization step.

Notably, AeneasHDC can be easily extended to other applications by modifying the open-source code.

16.2.2 Hardware Architecture

The AeneasHDC platform is designed to transition from software-defined models to practical hardware implementations seamlessly. All the configurations discussed in Section 16.2.1 can be implemented in hardware, on any target FPGA, empowering users to directly assess how different design choices impact hardware performance and efficiency.

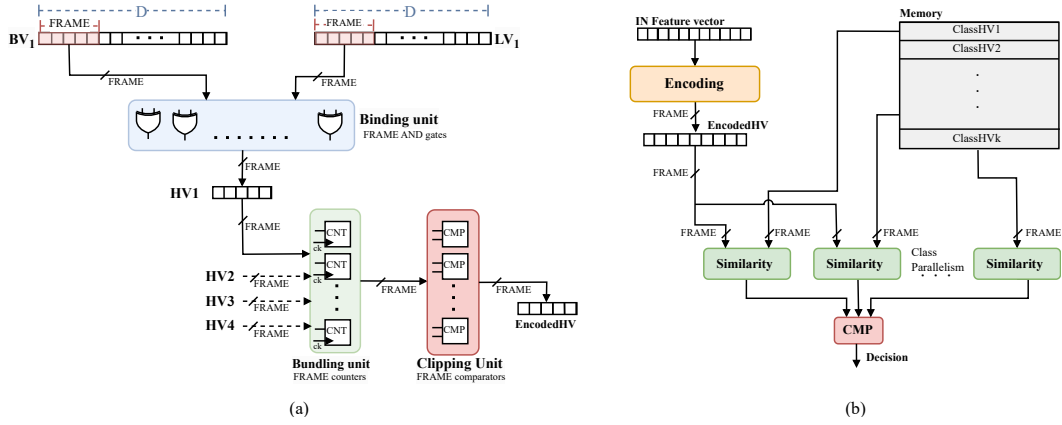


Figure 16.2 Reconfigurable Hardware parallelism applied to a Record-based encoding example case. (a) Element parallelism, SIMD elements are computed in parallel. (b) Class Parallelism.

The accelerators synthesized and implemented via AeneasHDC feature a standard AXI (Advanced eXtensible Interface) connection with the processor and the dedicated memories that ensures straightforward integration and broad compatibility. In these designs, the encoding process follows a pipeline approach that enables simultaneous processing of various input elements at different stages, optimizing hardware resource utilization, operating frequency, and throughput. Additionally, our framework introduces two pivotal features that add significant flexibility to the hardware architectures, ensuring efficient solutions in diverse environments.

- **Selective Model Instantiation:** the user can choose which operational stages of the HDC learning process are synthesized in hardware. The architecture can be configured to accelerate the full pipeline (training, retraining, and inference) on high-resource FPGAs, or to instantiate only the inference stage on resource-constrained systems, the most common embedded deployment scenario, which minimizes area and simplifies the control logic;
- **Hardware Parallelism:** a *partial processing* mechanism to control the degree of hardware parallelism at different stages of the learning process, allowing fine-grained trade-offs between processing speed, hardware requirements, and power consumption. This control can be applied at three independent levels.
 1. **Element parallelism:** the *SIMD* parameter controls the number of HV elements processed in parallel. Fig. 16.2(a) illustrates element parallelism applied to a record-based encoding: all functional units (binding, bundling, clipping, comparators) are replicated *SIMD* times, and at each processing step *SIMD* elements are computed simultaneously, resulting in a total of $\lceil D/SIMD \rceil$ iterations per HV. Setting *SIMD* = *D* yields a fully unrolled single-cycle-per-stage design; setting *SIMD* = 1 produces the most compact architecture at the cost of *D* cycles per stage;
 2. **Feature parallelism:** a reconfigurable level of parallelism that enables concurrent processing of multiple features. When set to $f_p > 1$, the

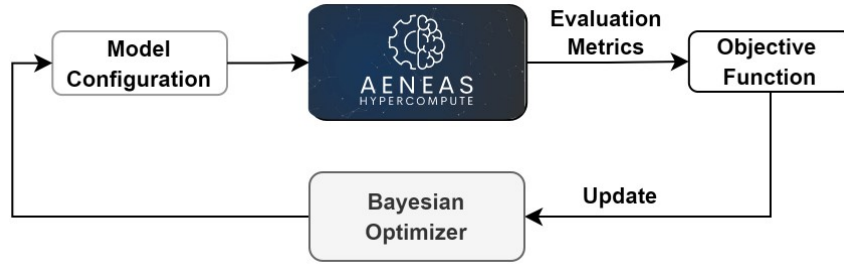


Figure 16.3 Architecture of the AeneasHDC design-space exploration engine. The Bayesian optimizer proposes a model configuration, AeneasHDC evaluates it in software and hardware collecting accuracy, resource utilization, power, and throughput metrics, and the resulting objective value is fed back to update the surrogate model. The loop repeats for a user-specified number of iterations.

pipeline instantiates f_p independent encoding paths operating on different features simultaneously. This approach is particularly effective for datasets with a large number of input features, where sequential feature-by-feature encoding would otherwise dominate the inference latency;

3. **Class parallelism:** reconfigurable hardware parallelism for the similarity search, depicted in Fig. 16.2(b). The user can choose to compare the encoded HV against each prototype HV sequentially, or to enable parallel computation over more prototypes by replicating the similarity unit and the corresponding class memory banks. This is instrumental for multi-class problems and can be combined freely with the previous two levels.

16.2.3 Automated Design-Space Exploration

The design space defined in Table 16.1 is combinatorially large: even a modest sweep over five HV dimensions, four clipping strategies, and three SIMD parallelism levels yields $5 \times 4 \times 3 = 60$ distinct hardware configurations, each requiring synthesis and evaluation.

To address this challenge, AeneasHDC incorporates an *automated design-space exploration* (DSE) engine based on a *Bayesian optimization* strategy that replaces manual trial-and-error with a principled, closed-loop optimization process.

Fig. 16.3 illustrates the architecture of the DSE engine. The process operates as a closed feedback loop comprising four stages:

1. **Model Configuration.** The Bayesian optimizer proposes a candidate configuration and passes it to AeneasHDC.
2. **Model Evaluation.** AeneasHDC automatically generates the software model and, when hardware metrics are requested, the corresponding HLS-based hardware accelerator. The framework executes the model and collects a comprehensive set of *evaluation metrics*: accuracy (or other task-specific metrics such as NMI for clustering and MSE for regression), hardware resource utilization (LUTs, FFs, DSPs, BRAMs), operating frequency, inference throughput (samples per second), and dynamic power consumption.

3. **Objective Function.** The collected metrics are combined into a scalar objective value according to the active optimization mode (described below). In constrained modes, configurations that violate the specified hardware budgets receive a heavy penalty, moving the optimizer away from infeasible regions of the design space.
4. **Surrogate Update.** The objective value is fed back to the Bayesian optimizer, which updates its internal surrogate model—a Gaussian process that estimates the objective surface from all configurations evaluated so far—and selects the next candidate by maximizing an acquisition function (expected improvement) that balances exploration of unexplored regions with exploitation of promising areas.

This loop repeats for a user-specified number of iterations.

The DSE engine supports two principal operating modes, each tailored to a different design scenario.

- **Mode 1, Accuracy optimization:** In this mode, the objective function is set to maximize model accuracy (or minimize error) without any constraint on hardware resources. The optimizer is free to explore large dimensions, high-precision representations, and computationally intensive encoding techniques, making this mode suitable for the initial phase of model development, where the goal is to establish an upper bound on achievable accuracy for a given dataset and learning task before introducing hardware constraints.
- **Mode 2, Hardware-constrained optimization:** In this mode, the objective function still targets accuracy maximization, but subject to explicit hardware constraints. Configurations that violate the specified budgets are heavily penalized in the objective, effectively excluding infeasible designs from the search. Three sub-modes are available, which can also be combined:
 - **Area-constrained** (LUT and FF budgets): the optimizer ensures that the synthesized accelerator fits within a user-specified number of LUTs and FFs, allowing the designer to target a specific FPGA or a specific portion of a larger device.
 - **Power-constrained:** the optimizer ensures that the dynamic power consumption of the generated accelerator remains below a specified threshold, a critical requirement for battery-powered or thermally constrained edge devices.
 - **Speed-constrained** (throughput or latency budget): the optimizer ensures that the inference throughput exceeds a minimum number of samples per second (or, equivalently, that the per-sample latency remains below a specified threshold), addressing real-time processing requirements.

The DSE results are presented through the same report-generation pipeline described above: for each evaluated configuration, the framework records accuracy, LUT count, flip-flop count, DSP utilization, BRAM usage, dynamic power, and inference throughput. Summary plots and tabular comparisons are generated automatically, enabling the user to identify the most efficient design point.

Table 16.2 Summary of HDC model configurations and datasets for the single-configuration validation experiments.

Test-id	Dataset	Task	HV VSA Dimension	Encoding Technique	Temporal Encoding	Level HV Generation	#LVs	Clipping / Quantization	Similarity Measure
Test 1	CARDIO	Classification	512	N-gram	No	Flipping	10	None	Cosine
Test 2	EMG	Classification	256	Record	Yes (n-gram=4)	Selection	21	Binary	Hamming
Test 3	Hepta	Clustering	1024	Record	No	Thermometer	20	Quant. (6 bits)	Cosine
Test 4	Boston	Regression	512	Record	No	Flipping	20	None	DOTP

This capability transforms AeneasHDC from a single-point code generator into a true co-design exploration tool, and, to the best of our knowledge, makes it the first HDC framework to offer Bayesian-guided hardware DSE that spans multiple VSA families.

16.3 Experiments

To validate the AeneasHDC platform and demonstrate its versatility in generating HDC models in software and hardware, we conducted two sets of experiments. The first set (Section 16.3.1) consists of single-configuration tests that showcase the platform’s ability to generate correct and efficient accelerators across different learning tasks, datasets, and model configurations. The second set (Section 16.3.2) demonstrates the automated design-space exploration engine introduced in Section 16.2.3, showing how Bayesian-guided optimization can discover superior configurations compared to manual selection.

16.3.1 Single-Configuration Validation

In these experiments, the HDC models were first trained and retrained over 50 iterations using the software model. We then deployed the hardware for the inference stage on the Zynq UltraScale+ ZCU102 board, using a target frequency of 100 MHz and a 64-element SIMD size. For each experiment, we evaluated the software and hardware accuracy, the area occupation, power consumption, and execution time.

Table 16.2 provides a detailed summary of the configuration chosen for the HDC model in each test, while Table 16.3 shows the corresponding outcomes. These tests were designed to showcase the capability and flexibility of the platform across varied scenarios; the search for optimal configurations for each dataset is addressed in Section 16.3.2 through the automated DSE engine.

The first dataset, *CARDIO* [332], consists of 3 classes and spatial sequences described by 21 features. For this test, we employed 512-dimensional MAP-I hypervectors encoded using the n-gram technique, random flipping level HVs, no class clipping, and cosine similarity. The obtained results confirm the capability of the platform, showing an accuracy of 88.78% for both the software and the hardware models generated by AeneasHDC. In terms of hardware requirements, the synthesized architecture necessitated 12 766 LUTs, 19 744 FFs, and 80 DSPs. The power consumption was measured at 206 mW for dynamic power, with a processing speed of 5 542.92 classifications per second. Notably, the high hardware demand for this accelerator is attributed to the cosine similarity computation on unclipped class HVs, which requires 17 bits for each element. Clipping or quantizing the class HVs

would be a key strategy for minimizing the area requirements of the architecture, as confirmed by the DSE results in Section 16.3.2.

Table 16.3 Test Configurations

Test-id	Software Metrics	Hardware Metric	#LUTs	#FFs	#DSPs	Dynamic Power	Task/s
Test 1	88.78% Acc.	88.78% Acc.	12766	19744	80	206	5542.92
Test 2	94.79% Acc.	94.79% Acc.	2996	6616	0	6	3219.22
Test 3	100% NMI	100% NMI	14090	19536	90	161	80042.68
Test 4	28.79 MSE	28.8 MSE	4330	4800	0	30	5543.85

The second classification dataset, *EMG* [337], is a temporal encoding problem involving four features distributed across five classes. We employed 256-dimensional BSC hypervectors, using record-based encoding in conjunction with n-gram composition to capture temporal windows of 4 elements. Level HVs generated with the random selection method were used, and binary clipping was applied to the trained class HVs. This configuration achieved 94.79% accuracy for both the software and the hardware models. The hardware architecture synthesized for this configuration is notably efficient, comprising 2 996 LUTs, 6 616 FFs, and no DSPs, and consuming only 6 mW of dynamic power while producing 3 219.22 classifications per second. This remarkable efficiency in hardware resources and power consumption, despite the model’s complexity, can be attributed to the binary clipping: reducing class HVs to single-bit representations transforms similarity computation into simple popcount operations, eliminating the need for DSPs and substantially reducing the datapath width.

For the clustering task, we employed the *Hepta* [363] dataset, which comprises 3 features and 7 clusters. The HDC model was configured with 1024-dimensional MAP-B (bipolar) HVs, record-based encoding, and the thermometer technique for level HV generation. We applied quantized clipping with 6-bit precision to the trained class HVs and cosine similarity for inference. This approach was chosen to balance hardware simplicity with effective performance. Remarkably, both the software and hardware models achieved an NMI score of 1.00, indicating perfect cluster recovery. The hardware configuration required 14 090 LUTs, 19 536 FFs, and 90 DSPs, while consuming 161 mW of dynamic power. Similar to Test 1, the use of cosine similarity led to substantial hardware overhead, but employing a 6-bit quantized format significantly simplified the logic compared to the unclipped case, reducing the bit requirement for each element of the class HVs from eighteen to just six. Additionally, the small number of features in this dataset enables a throughput of 80 042.68 samples per second.

Finally, for regression, we analyzed the *Boston Housing* dataset [364], comprising 12 features. The model was configured with 512-dimensional BSC hypervectors, record-based encoding, no clipping. Both the software and hardware models achieved a Mean Squared Error (MSE) of 28.79 and 28.80, respectively. The synthesized accelerator required 4 330 LUTs, 4 800 FFs, and zero DSPs, while consuming 30 mW of dynamic power and processing 5 543.85 samples per second.

Across all four tests, the hardware model produces results that are identical (or differ only at the last decimal place) to the software model, confirming the functional

correctness of the AeneasHDC hardware generation pipeline. The results also illustrate the strong influence of design choices on hardware efficiency: Test 2 (binary clipping, cosine similarity) requires 77% fewer LUTs than Test 1 (no clipping, cosine similarity) despite using a more complex temporal encoding scheme, highlighting how clipping can dominate the area–accuracy trade-off. These observations motivate the automated exploration presented next, where the DSE engine navigates such interactions systematically.

16.3.2 Design-Space Exploration Results

The single-configuration experiments above demonstrate the correctness and versatility of AeneasHDC, but each test uses a manually selected configuration. A natural question arises: can automated design-space exploration discover configurations that outperform manual selection? To answer this, we applied the Bayesian DSE engine described in Section 16.2.3 to two of the datasets from the previous experiments—EMG for accuracy-oriented optimization and CARDIO for area-constrained optimization—demonstrating both operating modes of the DSE.

Accuracy-Oriented DSE: EMG Classification

We first applied the DSE in *accuracy optimization mode* (Mode 1) to the EMG classification task, running the Bayesian optimizer for 100 iterations. In this mode, no hardware constraints are imposed; the objective function is set to maximize classification accuracy, and the optimizer is free to explore the full parameter space of Table 16.1.

Fig. 16.4 shows some of the parameter values explored over the 100 iterations. The scatter plots reveal the characteristic behavior of Bayesian optimization: during the initial iterations, the optimizer samples diverse configurations across the full range of each parameter. As the surrogate model becomes more accurate, the sampling progressively concentrates on high-performing regions.

Fig. 16.5(a) shows the cumulative average score progression over the 100 iterations. The curve rises steeply during the first 20 iterations as the optimizer identifies effective encoding and training configurations, and continues to improve gradually thereafter as the surrogate model refines its estimate of the objective surface. The best configuration discovered by the DSE achieves an accuracy of **96.10%**, representing a +1.31 percentage-point improvement over the 94.79% obtained by the manually selected configuration in Test 2. This improvement confirms that automated exploration can discover design points that are difficult to reach through manual tuning: the optimizer identified a combination of encoding technique, number of level HVs, clipping strategy, and learning rate schedule that jointly outperforms the expert-selected baseline.

Area-Constrained DSE: CARDIO Classification

We then applied the DSE in *hardware-constrained mode* (Mode 2, area sub-mode) to the CARDIO classification task. The hardware budgets were set to 10 000 LUTs and 10 000 FFs, reflecting a scenario where the accelerator must fit within a limited portion of the target FPGA. The optimizer ran for 30 iterations, a smaller budget

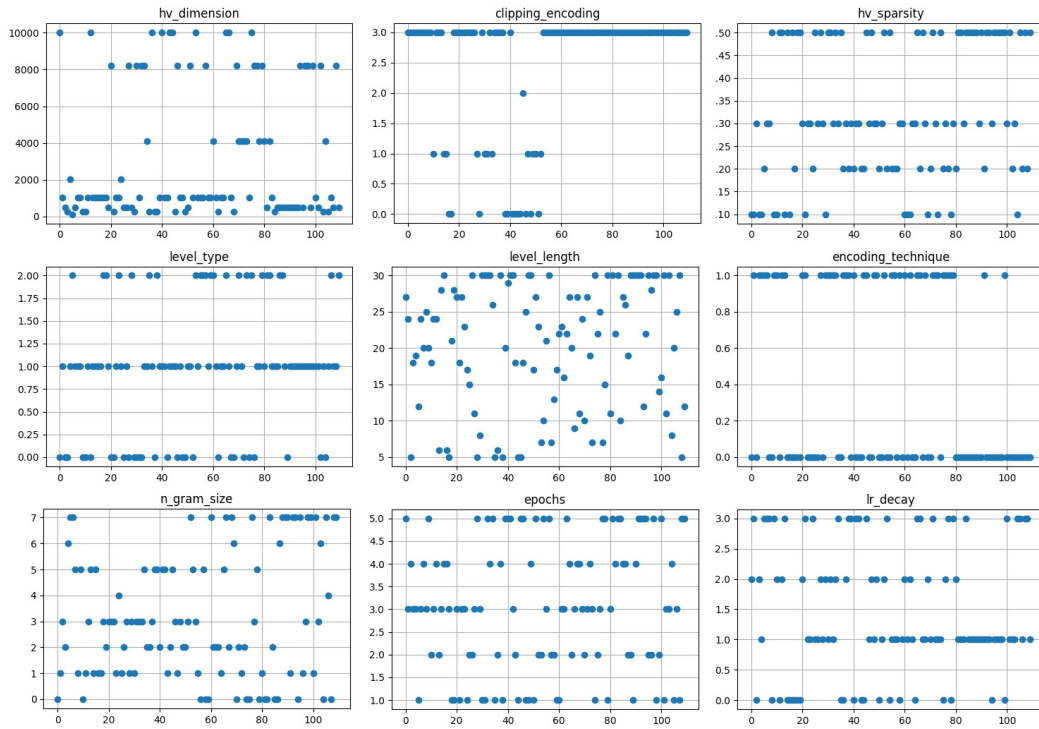


Figure 16.4 Parameter values explored by the Bayesian optimizer over 100 iterations on the EMG classification task (accuracy-optimization mode). Each subplot shows the sampled values of one parameter as a function of the iteration index. The optimizer initially explores the full range of each parameter and progressively concentrates on high-performing regions as the surrogate model improves.

Table 16.4 CARDIO classification: comparison between the manually configured model (Test 1) and the best configuration identified by the area-constrained DSE (budget: 10 000 LUTs, 10 000 FFs).

Configuration	Accuracy	LUTs	FFs	DSPs	Dyn. Power [mW]	Speed [samples/s]
Manual (Test 1)	88.78%	12 766	19 744	80	206	5 542.92
Area-constrained DSE	90.41%	4 713	8 274	0	51	19 525.52
<i>Improvement</i>	<i>+1.63 pp</i>	<i>−63%</i>	<i>−58%</i>	<i>−100%</i>	<i>−75%</i>	<i>+3.5×</i>

chosen to demonstrate that the Bayesian approach can deliver meaningful results even with limited evaluation budget.

Fig. 16.5(b) shows the cumulative average score progression. The dashed vertical line marks the end of the random initialization phase (approximately iterations 0–10), during which the optimizer builds its initial surrogate model by sampling diverse configurations without guidance. After this phase, the Bayesian-guided search begins to exploit the learned surrogate, and the cumulative score increases steadily as the optimizer discovers configurations that simultaneously satisfy the area constraints and maximize accuracy.

Table 16.4 compares the DSE-optimized configuration against the manually designed accelerator from Test 1. The results are striking: the DSE configuration not only satisfies the area constraints (4 713 LUTs and 8 274 FFs, well within the

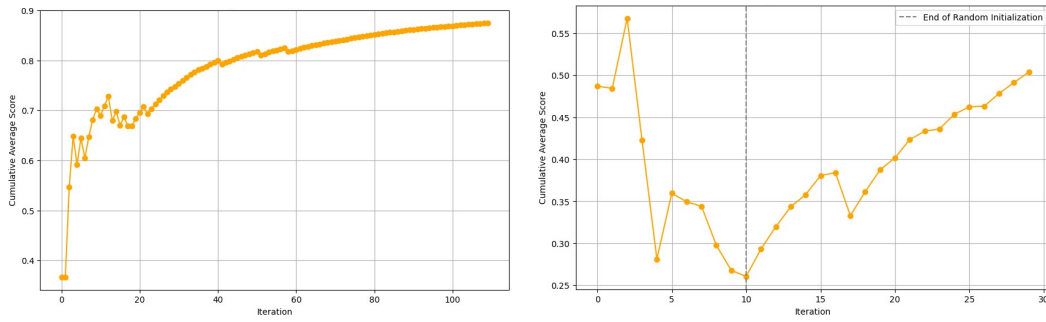


Figure 16.5 Cumulative average score progression during design-space exploration for two representative scenarios. (a) Unconstrained accuracy maximization on EMG. (b) Area-constrained optimization on CARDIO.

10 000/10 000 budget) but simultaneously improves *every* metric compared to the manual baseline. Accuracy increases from 88.78% to 90.41% (+1.63 percentage points), LUT usage drops by 63%, FF usage drops by 58%, DSP utilization is completely eliminated (from 80 to 0), dynamic power is reduced by 75% (from 206 mW to 51 mW), and inference throughput increases by $3.5\times$ (from 5 543 to 19 526 samples per second).

These dramatic improvements stem from the optimizer’s ability to jointly explore the interaction between encoding technique, clipping strategy, similarity measure, level HV generation, and hardware parallelism.

16.4 Summary of Findings

This chapter introduced AeneasHDC, a comprehensive and open-source framework designed for the automated generation of HDC learning models in both software and hardware. AeneasHDC supports a broad spectrum of design choices available in the literature—spanning five VSA families (BSC, MAP-B, MAP-I, MAP-C, and FHRR), multiple encoding techniques, configurable quantization strategies, and learning rules—enabling rapid and systematic exploration of the impact of different configurations on accuracy, power consumption, memory requirements, hardware overhead, and execution time.

The proposed framework integrates an intuitive GUI that streamlines the specification of problem parameters (dataset, number of input features, classes, and target FPGA), the required task (classification, clustering, regression, or contextual bandits), and the full model configuration. The hardware accelerators generated by AeneasHDC through Xilinx Vitis HLS can be reconfigured to support different operational stages and levels of parallelism, allowing designers to trade execution time for hardware resource allocation according to the constraints of the target deployment platform.

We validated the platform through two complementary sets of experiments. First, a suite of four single-configuration tests demonstrated that the automatically generated hardware accelerators reproduce the software model’s accuracy across all learning tasks. Second, we demonstrated the Bayesian-guided design-space exploration engine, which transforms AeneasHDC from a single-point code generator

into a principled co-design optimization tool. In accuracy-optimization mode on the EMG task, the DSE discovered a configuration achieving 96.10% accuracy, a +1.31% improvement over the manually selected baseline. In area-constrained mode on the CARDIO task (budget: 10 000 LUTs and 10 000 FFs), the DSE simultaneously improved accuracy by +1.63% while reducing LUT usage by 63%, FF usage by 58%, dynamic power by 75%, eliminating all DSP usage, and increasing inference throughput by $3.5\times$. These results confirm that automated exploration can discover design points that dominate manual configurations across all objectives.

AeneasHDC provides a straightforward yet powerful method for exploring the influence of algorithmic design choices on hardware implementations, representing, to the best of our knowledge, the first HDC framework to offer Bayesian-guided hardware design-space exploration spanning multiple VSA families.

Chapter 17

Conclusions and Future Perspectives

“The end of all our exploring will be to arrive where we started and know the place for the first time.”

Thomas Stearns Eliot

The proliferation of IoT networks and embedded devices has created an urgent demand for on-device intelligence. These systems must sense, decide, and react to their environment in real time, without relying on cloud connectivity. Yet traditional artificial intelligence, dominated by resource-intensive models, is fundamentally misaligned with the power, memory, and latency constraints of the edge. As more than half of all new AI workloads are now expected to be deployed on edge platforms, the gap between the computational requirements of modern AI and the resources available on embedded hardware has become the defining challenge of the field.

This dissertation addressed the AI efficiency crisis in the specific context of online decision-making at the edge, adopting contextual bandit algorithms as the central abstraction and algorithmic-hardware co-design as the guiding methodology. The overarching research question asked whether the computational power, energy, and memory bottlenecks of edge AI can be mitigated by combining full-stack co-design with neuro-inspired representations, thereby decoupling adaptive intelligence from brute-force hardware scaling.

The work progressed across three complementary dimensions—arithmetic and algorithmic optimization of linear models, a neuro-inspired paradigm shift via hyperdimensional computing, and the development of dedicated hardware accelerators—each targeting a different layer of the efficiency stack and each motivated by the limitations revealed by the preceding one. Importantly, while contextual bandits served as the unifying thread throughout this journey, the contributions delivered along the way extend well beyond the bandit setting to the broader landscape of embedded AI and hyperdimensional computing.

The following sections summarize the principal contributions of each part, discuss their limitations, and outline future research directions.

17.1 Summary of Contributions

17.1.1 Part I: Arithmetic and Algorithmic Optimization

The first part of this thesis investigated and targeted the dominant bottlenecks of traditional linear bandit algorithms on embedded processors. Profiling revealed that integer division can dominate execution time, accounting for up to 70% of the total runtime in bandit inference workloads on the edge. The research question driving this part was: *How far can arithmetic, algorithmic, and microarchitectural co-design push the efficiency of classical linear bandit algorithms?*

To address the arithmetic bottleneck, Chapter 3 introduced the Variable-Latency Non-Performing Divider, a novel data-dependent hardware division unit that exploits the statistical relationship between input operands to reduce average latency by up to $20.65\times$ via early termination and iteration skipping. Four hardware variants were presented, offering distinct trade-offs between latency, area, and energy efficiency, all remaining compact and suitable for embedded implementations. The design was validated on both FPGA (Xilinx Virtex UltraScale+) and ASIC (GF22FDX) technology, achieving the lowest average execution time per division and dynamic energy consumption reported in the literature.

Chapter 4 integrated VLNPD into the vector accelerator of the Klessydra-T13 RISC-V core, demonstrating that operand-dependent division is particularly beneficial under vector (SIMD) execution. This integration reduced the contribution of division to the overall runtime of linear bandits at the edge from 70% to approximately 12%.

Chapter 5 attacked the algorithmic bottleneck by reformulating the learning update of both the Disjoint and Hybrid LinUCB algorithms via the Sherman–Morrison–Woodbury identity, eliminating explicit matrix inversion and enabling efficient vectorization. Numerical stability was assessed through controlled error analysis, confirming that the incremental formulation remains well-conditioned over extended learning horizons. Combined with the vectorized kernels and hardware optimizations of the preceding chapters, this co-design achieved up to $58\times$ speedup and $50\times$ energy reduction on three embedded platforms (Cortex-M4 STM32, Cortex-A72 Raspberry Pi 4, and RISC-V Klessydra-T13), while reducing the per-round update complexity from cubic to quadratic.

These results demonstrated that substantial efficiency gains are achievable through cross-layer co-design. However, they also exposed a fundamental ceiling: the quadratic complexity and memory overhead with respect to the context dimension remain inherent limitations of dense linear algebra, motivating the paradigm shift explored in Part II.

17.1.2 Part II: A Paradigm Shift to Neuro-Inspired Contextual Bandits

Rather than further optimizing the execution speed of the same workload, the second part of this dissertation changed the nature of the computation itself by adopting hyperdimensional computing, a neuro-inspired paradigm based on high-dimensional distributed representations. The guiding research question was: *Instead*

of further optimizing the execution of classical kernels, can we change the nature of the computation itself to bypass the quadratic scaling limit?

Chapter 6 addressed this question through a systematic ablation study that isolated the role of the precision matrix in the linear bandit estimator and tested whether it could be bypassed by encoding contexts in a high-dimensional space. Experiments across three encoding regimes—original low-dimensional features, dense Gaussian random projections, and biologically-inspired sparse projections with k -Winners-Take-All sparsification—established that nonlinear encoding is the critical ingredient and that under this setting the covariance matrix reduces to a scaled copy of the reward vector plus cross-talk noise, allowing it to be omitted entirely. These findings provided the theoretical and empirical foundation for the transition to Hyperdimensional Computing in the subsequent chapters.

Chapter 7 introduced HD-CB, the first framework that models and solves contextual bandit problems directly in high-dimensional space. By encoding contexts as high-dimensional vectors and representing actions as vector prototypes, HD-CB reduced per-step complexity from quadratic (after the optimizations of Part I) to linear. Four variants were proposed, each addressing specific challenges in exploration–exploitation trade-offs and resource constraints. Experiments across multiple synthetic environments and the MovieLens-100k dataset demonstrated competitive learning performance with improved scalability and substantially lower computational cost.

Chapter 8 developed HD-CB_{BIN}, a fully binarized variant that restricts all hypervectors to binary representations. This specialization drastically reduces memory requirements compared to real-valued HD-CB and enables similarity computation via Hamming distance. When accelerated on hardware, HD-CB_{BIN} achieved up to $141\times$ speedup over matrix-based algorithms, establishing its suitability for extreme edge deployment.

Chapter 9 introduced Hybrid HD-CB, extending the framework to settings where arms share a common feature representation. By encoding context-action interactions in high-dimensional space through binding operations, this variant learns shared structure across actions while preserving linear complexity. Evaluation on both synthetic benchmarks and the MovieLens-100k dataset demonstrated significant speedups and memory reductions compared to traditional Hybrid LinUCB.

Chapter 10 presented two engineering refinements that improve the execution efficiency and memory footprint of HD-CB. First, matrix-form reformulations of all HD-CB variants were derived, exposing data parallelism and enabling BLAS-level optimization with $2.3\times$ – $3.6\times$ CPU speedup. Second, memory-optimized variants based on permutation-based channel multiplexing reduced storage from $\mathcal{O}(ND)$ to $\mathcal{O}(D)$, making deployment feasible on platforms where per-action hypervector storage would exceed on-chip memory.

Chapter 11 advanced the learning foundations of HD-CB along several converging directions. Probabilistic HD-CB replaced deterministic updates with bounded stochastic updates governed by time-decaying probabilities, enabling deployment with as few as 3 bits per element while preserving competitive reward performance. A theoretical analysis of the baseline update rule then identified the implicit loss function it minimizes and revealed an ensemble structure in which each hypervector component acts as an independent weak learner at a distinct reward threshold,

diagnosing the specific limitations that the subsequent formulation addresses. Loss-Based HD-CB replaced this implicit objective with explicit minimization of a well-defined loss function, bridging the gap between heuristic updates and optimization-grounded formulations and improving accuracy by up to 3 percentage points over standard HD-CB. Loss-Probabilistic HD-CB combined the error-driven loss-based updates with the bounded stochastic mechanics of the probabilistic variant, retaining principled learning within fixed-precision integer arithmetic. Finally, the entire HD-CB pipeline was formally mapped onto a Random Vector Functional Link network. A comprehensive cross-family comparison across all variants revealed that the 4-bit Loss-Probabilistic configuration outperforms the 32-bit LinEPS baseline, making it a strong candidate for embedded platforms.

Altogether, the contributions of Part II established hyperdimensional contextual bandits as an ideal candidate for resource-constrained systems and for dedicated hardware acceleration—the subject of Part III.

17.1.3 Part III: Hardware Acceleration and Co-Design

The third part of this thesis bridged algorithms and silicon by designing domain-specific hardware architectures for hyperdimensional computing. The driving goal was to develop a family of general accelerators capable of supporting not only the full range of HD-CB variants, but also any HDC learning task, lowering the barrier to adoption of hyperdimensional computing on embedded platforms. The research question was: *What efficiency gains can be unlocked by translating neuro-inspired learning algorithms into dedicated hardware accelerators?*

Chapter 12 introduced the HDCU, a general-purpose configurable hardware accelerator for binary HDC primitives. The HDCU features optimized functional units for each basic arithmetic operation with synthesis-time configurability in parallelism, supported operations, and local memory size. A custom RISC-V Instruction Set Extension, fully integrated into the GCC toolchain, exposes the accelerator to software via simple intrinsics. Integrated into the Klessydra-T03 core as a tightly coupled coprocessor, the HDCU achieved speedups of up to $1968\times$ and energy reductions of up to $1630\times$ compared to software execution, while requiring as few as 1030 LUTs in its minimal configuration.

Chapter 13 extended hardware support to Modular Composite Representations, repositioning this model in the literature through large-scale experiments on 123 classification datasets. The results demonstrated that MCR provides a more favorable trade-off than binary spatter codes: higher information capacity and classification accuracy, with reduced memory overhead for equivalent representational power. The MCR-HDCU accelerator—the first dedicated hardware for this model—exploited the natural mapping of modular arithmetic to digital logic through implicit binary overflow, achieving on average $2.7\times$ faster execution and $3.9\times$ lower energy consumption than binary models accelerated with the original HDCU at equivalent accuracy.

Chapter 14 addressed the most expressive model in the HDC family: the FHRR. A hardware-friendly discretization strategy was proposed that quantizes phases on the unit circle into uniformly spaced angles, enabling modular arithmetic without explicit modulo operators. Building on this insight, the first dedicated FHRR accelerator was introduced, featuring functional units for all primitive operations as

well as the first hardware implementation of a Resonator Network.

Chapter 15 explored the computational boundaries of the FHRR paradigm by shifting focus from learning to symbolic reasoning. Resonator Networks were applied to NP-complete 3-SAT problems via a classical polynomial-time reduction to Subset Sum. The results demonstrated that the same neuro-inspired primitives used for pattern matching can support exact combinatorial search, while revealing and quantifying the fundamental D^2 capacity limit that governs the approach’s scalability. This proof-of-concept validated the broader applicability of FHRR beyond learning tasks while providing crucial insights for future algorithm and hardware design.

Chapter 16 introduced AeneasHDC, an automated open-source framework for deploying hyperdimensional computing models on FPGAs. Supporting five VSA families, multiple encoding techniques, configurable quantization strategies, and Bayesian-guided design-space exploration, AeneasHDC reduces accelerator development time from weeks to minutes. In area-constrained experiments, the automated exploration discovered configurations that simultaneously improved accuracy by +1.63% while reducing LUT usage by 63%, FF usage by 58%, dynamic power by 75%, and increasing inference throughput by $3.5\times$ compared to manually selected baselines.

17.2 Limitations

While the contributions presented in this dissertation advance the state of the art across multiple fronts, each comes with specific limitations that should be acknowledged. The following discussion organizes them by part, noting where later chapters address specific shortcomings.

17.2.1 Part I: Arithmetic and Algorithmic Optimizations

The VLNPD (Chapter 3) achieves significant average-case latency reductions, but its performance is inherently data-dependent: the speedup varies with the statistical distribution of operands, and worst-case latency remains comparable to conventional dividers. This property, while advantageous on average, makes execution time non-deterministic, which is a potential concern for safety-critical real-time systems that require guaranteed worst-case latency bounds. Furthermore, the VLNPD was designed for integer divisions. While a common choice for embedded systems, future work should explore the extension of this approach to floating-point pipelines. The integration of VLNPD into the Klessydra-T13 vector coprocessor (Chapter 4) demonstrated the benefits of variable-latency division under SIMD execution, but the evaluation was limited to a single RISC-V microarchitecture with a specific vector lane configuration. The interaction with different vector execution models (e.g., out-of-order and decoupled pipelines) was not explored.

17.2.2 Part II: Neuro-Inspired Contextual Bandits

The foundational HD-CB framework (Chapter 7) has two principal limitations. First, the accumulation-based update causes action representatives to grow unboundedly, leading to precision issues on low-precision hardware—a limitation addressed by the

Probabilistic HD-CB variant in Chapter 11. Second, the update rule is heuristic rather than derived from a formal optimization objective, and the implicit exploration threshold at 0.5 lacks theoretical justification—a gap closed by the Loss-Based formulation in Chapter 12. Additionally, the framework lacks formal regret guarantees comparable to those of linear bandits; the RVFL interpretation (Chapter 11) provides a path toward such results but does not yet deliver them. Finally, all experiments assumed stationary reward distributions; the framework’s behavior under concept drift or non-stationary environments remains unexplored.

HD-CB_{BIN} (Chapter 8) achieves extreme memory efficiency through binarization, but periodic hard resetting of accumulated counters to binary values discards graded information, potentially degrading decision quality in complex environments. This accuracy penalty was partially masked by the use of the confidence-based HD-CB_{UNC2} variant and controlled synthetic environments, but became more apparent under simpler ϵ -greedy exploration, as demonstrated in Chapter 11. An extensive evaluation with different exploration strategies is required.

Hybrid HD-CB (Chapter 9) successfully captures shared structure across actions but inherits the heuristic update limitations of the base framework. The structured updates based on loss functions developed for the baseline algorithm in Chapter 11 should be extended to the hybrid setting.

The memory-compressed variants in Chapter 10 drastically reduce storage but introduce cross-talk between superposed action channels. Also, this optimization was developed for the baseline HD-CB and has not been extended or tested on the more advanced variants yet.

Beyond the chapter-specific limitations described above, some cross-cutting themes deserve mention. The best encoding technique for HD-CB remains an open question: no extensive analysis has been conducted on the choice of encoding method, which could improve performance substantially depending on the environment. Moreover, while these algorithms have been accelerated in hardware using the general-purpose HDCU for the binary case, no dedicated hardware accelerator for contextual bandits has been realized yet, although one would maximize performance and energy efficiency. The experimental evaluation throughout this dissertation relied predominantly on standardized yet synthetic bandit environments, benchmarks from the UCI/OpenML repositories, and the MovieLens-100k dataset. While these are standard and widely used, they do not capture the full complexity of real-world deployment scenarios, including non-stationary environments, adversarial contexts, delayed or batched feedback, and more complex reward structures. Extending the proposed methods to address these non-idealities and demonstrating them in actual embedded systems operating in the field (such as robots) would substantially strengthen the practical significance of the results. Finally, all work in this thesis focused on single-agent settings. The extension to multi-agent or federated bandit learning, where multiple edge devices collaboratively learn while respecting communication and privacy constraints, was not addressed and will be of high interest.

17.2.3 Part III: Hardware Acceleration and Co-Design

The HDCU (Chapter 12) achieved impressive speedups and flexibility, but its tight coupling with the Klessydra-T03 core limits portability—a limitation subsequently mitigated by an AXI4-compliant standalone version described in the chapter.

The FHRR accelerator (Chapter 14) represents the first dedicated hardware for this paradigm, but comprehensive benchmarking of execution time and energy across standard datasets was still ongoing at the time of writing.

The application of resonator networks to 3-SAT (Chapter 15) demonstrated feasibility but is fundamentally bound by the D^2 capacity law: once the combinatorial search space exceeds the superposition capacity of the hypervectors, success rates collapse to zero, limiting practical applicability to small instances.

AeneasHDC (Chapter 16) currently relies on Xilinx Vitis HLS, limiting deployment to a single FPGA vendor and excluding other vendors as well as ASIC flows. The design-space exploration operates over a discrete parameter grid; the joint optimization of continuous algorithmic parameters and hardware-level decisions remains unexplored.

17.3 Future Work

The contributions and limitations discussed above naturally suggest several research directions that could extend and strengthen the work presented in this dissertation.

17.3.1 Algorithmic Extensions

Several directions could broaden the applicability and performance of the HD-CB family. The adoption of more sophisticated exploration strategies from the bandit literature could improve convergence time and address more complex environments. Closely related is the extension to non-stationary and adversarial settings, where forgetting mechanisms would make HD-CB suitable for dynamic real-world deployments. The bounded representations of Probabilistic HD-CB offer a natural substrate for such mechanisms, as they already limit the influence of distant past observations.

Beyond exploration and non-stationarity, richer action-space formulations deserve investigation: extending HD-CB to contextual bandits under budget constraints (where the learner must maximize reward subject to a cumulative cost), to continuous action spaces where the action is a real-valued vector rather than a discrete choice. Multi-task and transfer learning across related bandit problems is a natural fit for the hyperdimensional framework, where the representational similarity between environments could be captured through hypervector operations, enabling knowledge transfer. Looking further ahead, extending the same approach to full reinforcement learning with temporal credit assignment, multi-agent settings, or safe exploration under constraints would connect HD-CB to the broader RL landscape.

17.3.2 Hardware Evolution

The most promising hardware direction for the proposed HD-CB algorithms is neuromorphic implementation. Hyperdimensional computing has natural affinities

with neuromorphic hardware, where spiking neurons and event-driven computation can implement HDC operations with extreme energy efficiency. The RVFL interpretation of HD-CB establishes a bridge between hyperdimensional representations and spiking neural networks: implementing HD-CB on neuromorphic platforms (e.g., Intel Loihi, SynSense Xylo, AKIDA) could enable always-on, ultra-low-power, bandwidth-efficient learning for autonomous embedded systems.

A second direction is the development of a custom ASIC accelerator for contextual bandit workloads. The Loss-Probabilistic HD-CB variant is an appealing target, thanks to its superior performance at just 3 bits per element. Taping out such an accelerator in a modern technology node (e.g., 22 nm) would enable direct comparison with existing neural network accelerators and establish the energy-efficiency frontier for CB at the edge.

Third, unifying the HDCU, MCR-HDCU, and FHRR architectures into a single multi-model accelerator with runtime model selection would maximize flexibility while sharing common infrastructure (scratchpad memories, parallel processing elements, control plane). Fourth, introducing in-memory computing in the designed HDC accelerators—mapping similarity computations directly into emerging non-volatile memory arrays—could eliminate the data-movement bottleneck and enable orders-of-magnitude improvements in energy efficiency.

17.4 Closing Remarks

This dissertation set out to answer whether the computational bottlenecks of online decision-making can be overcome on resource-constrained embedded platforms through hardware-algorithm co-design and neuro-inspired representations. The answer, supported by contributions across three progressive parts, is affirmative.

Starting from conventional linear models and systematically eliminating bottlenecks at the arithmetic, algorithmic, and microarchitectural levels, we achieved up to $58\times$ speedup and $50\times$ energy reduction. Recognizing the inherent scaling limits of linear algebra, we changed the nature of the computation itself by introducing the HD-CB framework, the first application of hyperdimensional computing to contextual bandit problems, demonstrating significant speedups and memory reductions while providing competitive learning performance. The resulting framework is uniquely aligned with the constraints of edge devices and the strengths of dedicated hardware. Closing the loop between algorithms and silicon, we designed a family of domain-specific hardware accelerators spanning binary, modular, and complex-valued representations, achieving orders-of-magnitude improvements in both speed and energy efficiency.

Together, these contributions demonstrate that algorithmic-hardware co-design, grounded in neuro-inspired representations, can decouple adaptive intelligence from brute-force hardware scaling, making autonomous Edge AI practically achievable. The path from arithmetic optimization to neuro-inspired computing traced in this thesis reflects a broader principle: that the most impactful efficiency gains come not from optimizing individual layers in isolation, but from rethinking the entire computational stack, from the mathematical formulation of the learning problem to the logic gates that execute it.

This vision aligns with a growing consensus in the community. As recently articulated by Google DeepMind [365]: *“In this epoch, AI is not merely an application; it has become the core software layer that redefines system design. [...] Co-design must become an integral part of the research culture. Every proposal for a new model should include a consideration of its hardware requirements, and every hardware innovation should clearly articulate the new algorithms.”* This thesis contributes to that direction and reflects the conviction that the path forward lies in rethinking how we execute AI, drawing inspiration from biological computation and, most importantly, fostering collaboration across disciplines to unlock capabilities that neither community could achieve alone.

Bibliography

- [1] Denis Kleyko, Dmitri A Rachkovskij, Evgeny Osipov, and Abbas Rahimi. A survey on hyperdimensional computing aka vector symbolic architectures, part i: Models and data transformations. *ACM Computing Surveys*, 55(6):1–40, 2022.
- [2] Eric Matthews, Alec Lu, Zhenman Fang, and Lesley Shannon. Rethinking integer divider design for fpga-based soft-processors. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 289–297. IEEE, 2019.
- [3] Javier Snaider and Stan Franklin. Modular composite representation. *Cognitive Computation*, 6(3):510–527, 2014.
- [4] Sérgio Branco, André G Ferreira, and Jorge Cabral. Machine learning in resource-scarce embedded systems, fpgas, and end-devices: A survey. *Electronics*, 8(11):1289, 2019.
- [5] Kenny Schlegel, Peer Neubert, and Peter Protzel. A comparison of vector symbolic architectures. *Artificial Intelligence Review*, 55(6):4523–4555, 2022.
- [6] Nils J Nilsson. *The quest for artificial intelligence*. Cambridge University Press, 2009.
- [7] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006.
- [8] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, Boston, MA, 4th edition, 2020. ISBN 978-0134610993.
- [9] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing test: Philosophical and methodological issues in the quest for the thinking computer*, pages 23–65. Springer, 2007.
- [10] Reza Toorajipour, Vahid Sohrabpour, Ali Nazarpour, Pejvak Oghazi, and Maria Fischl. Artificial intelligence in supply chain management: A systematic literature review. *Journal of business research*, 122:502–517, 2021.
- [11] Samantha Tyler, Matthew Olis, Nicole Aust, Love Patel, Leah Simon, Catherine Triantafyllidis, Vijay Patel, Dong Won Lee, Brendan Ginsberg, Hiba

- Ahmad, et al. Use of artificial intelligence in triage in hospital emergency departments: a scoping review. *Cureus*, 16(5), 2024.
- [12] Sabina-Cristiana Necula and Vasile-Daniel Păvăloaia. Ai-driven recommendations: A systematic review of the state of the art in e-commerce. *Applied Sciences*, 13(9):5531, 2023.
- [13] Dana Remus and Frank Levy. Can robots be lawyers: Computers, lawyers, and the practice of law. *Geo. J. Legal Ethics*, 30:501, 2017.
- [14] Darko B Vuković, Senanu Dekpo-Adza, and Stefana Matović. Ai integration in financial services: A systematic review of trends and regulatory challenges. *Humanities and Social Sciences Communications*, 12(1):1–29, 2025.
- [15] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–46, 2021.
- [16] Kate Crawford. *The Atlas of AI: Power, Politics, and the Planetary Costs of Artificial Intelligence*. Yale University Press, New Haven, CT, 2021. ISBN 9780300209570. doi: 10.12987/9780300252392. URL <https://yalebooks.yale.edu/book/9780300209570/atlas-of-ai/>.
- [17] Luca Benini. Enabling sustainable AI: an open computing platform perspective. Keynote talk, slides available via PULP Platform, 2025. URL <https://pulp.ethz.ch/docs/alibaba2025/BeniniAB02-25.pdf>. Accessed: 2025-12-01.
- [18] Adnan Mehonic and Anthony J Kenyon. Brain-inspired computing needs a master plan. *Nature*, 604(7905):255–260, 2022.
- [19] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. In *2022 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2022.
- [20] International Energy Agency. Energy and ai. World energy outlook special report, International Energy Agency, Paris, 2025. URL <https://iea.blob.core.windows.net/assets/b8a83930-5c77-4da7-b795-270ab6a6c272/EnergyandAI.pdf>. IEA. CC BY 4.0.
- [21] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE access*, 6:6900–6919, 2017.
- [22] Hesham El-Sayed, Sharmi Sankar, Mukesh Prasad, Deepak Puthal, Akshansh Gupta, Manoranjan Mohanty, and Chin-Teng Lin. Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. *IEEE Access*, 6:1706–1717, 2017.
- [23] Adam Wright. Worldwide idc global datasphere forecast, 2024–2028: Ai everywhere, but upsurge in data will take time, 2024.

- [24] Satyajit Sinha. State of iot 2025: Number of connected iot devices growing 14% to 21.1 billion globally. <https://iot-analytics.com/number-connected-iot-devices/>, October 2025. IoT Analytics research blog.
- [25] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.
- [26] Rodrigo Roman, Jianying Zhou, and Javier Lopez. On the features and challenges of security and privacy in distributed internet of things. *Computer networks*, 57(10):2266–2279, 2013.
- [27] Brian Krzanich. Data is the new oil in the future of automated driving. Intel Editorial, November 2016. URL <https://download.intel.com/newsroom/2021/archive/2016-11-15-editorials-krzanich-the-future-of-automated-driving.pdf>. Accessed: 2025-12-01.
- [28] Amira Bourechak, Ouarda Zedadra, Mohamed Nadjib Kouahla, Antonio Guerrieri, Hamid Seridi, and Giancarlo Fortino. At the confluence of artificial intelligence and edge computing in iot-based applications: A review and new perspectives. *Sensors*, 23(3):1639, 2023.
- [29] Pengfei Li, Mohammad J. Islam, and Shaolei Ren. A case study of environmental footprints for generative ai inference: Cloud versus edge. *SIGMETRICS Perform. Eval. Rev.*, 53(2):21–26, August 2025. ISSN 0163-5999. doi: 10.1145/3764944.3764950. URL <https://doi.org/10.1145/3764944.3764950>.
- [30] International Energy Agency. Electricity 2024: Analysis and forecast to 2026, 2024. <https://www.iea.org/reports/electricity-2024>.
- [31] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai. *Commun. ACM*, 63(12):54–63, November 2020. ISSN 0001-0782. doi: 10.1145/3381831. URL <https://doi.org/10.1145/3381831>.
- [32] International Data Corporation (IDC). Three forces shaping the future of it leadership, December 2025. URL <https://www.idc.com/resource-center/blog/three-forces-shaping-the-future-of-it-leaderships/>. IDC FutureScape 2026 IT Predictions blog; forecasts that by 2030, 50% of enterprise AI inference workloads will run on endpoints or edge nodes.
- [33] International Data Corporation (IDC). IDC estimates global spending on edge computing to grow at 13.8% reaching nearly \$380 billion by 2028, mar 2025. URL <https://my.idc.com/getdoc.jsp?containerId=prUS53261225>. Press Release, Doc #prUS53261225.
- [34] Maurizio Capra, Riccardo Peloso, Guido Masera, Massimo Ruocco, and Maurizio Martina. Edge computing: A survey on the hardware requirements in the internet of things world. *Future Internet*, 11(4):100, 2019.

- [35] Himanshu Gauttam, Garima Nain, KK Pattanaik, and Paulo Mendes. Edge-ai: A systematic review on architectures, applications, and challenges. *Journal of Network and Computer Applications*, page 104375, 2025.
- [36] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9):2533, 2020.
- [37] Bill Dally. Hardware for deep learning, 2023.
- [38] Djallel Bouneffouf, Irina Rish, and Charu Aggarwal. Survey on applications of multi-armed and contextual bandits. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, page 1–8. IEEE Press, 2020. doi: 10.1109/CEC48606.2020.9185782. URL <https://doi.org/10.1109/CEC48606.2020.9185782>.
- [39] Guillaume Chacun, Mehdi Akeddar, Thomas Rieder, Bruno Da Rocha Carvalho, and Marina Zapater. Dronebandit: Multi-armed contextual bandits for collaborative edge-to-cloud inference in resource-constrained nanodrones. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, page 98–104, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706059. doi: 10.1145/3649476.3658720. URL <https://doi.org/10.1145/3649476.3658720>.
- [40] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1st edition, 1998. ISBN 978-0-262-19398-6.
- [41] Amirhosein Toosi, Andrea G Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A brief history of ai: how to prevent another winter (a critical review). *PET clinics*, 16(4):449–469, 2021.
- [42] Marvin Lee Minsky. *Semantic information processing*, volume 1. MIT press Cambridge, MA, 1968.
- [43] John McCarthy. What is artificial intelligence? <http://www-formal.stanford.edu/jmc/whatisai.pdf>, 2007. Defines AI as “the science and engineering of making intelligent machines, especially intelligent computer programs.”.
- [44] Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.
- [45] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346, 1990.
- [46] Thomas Hobbes and Hermann Klenner. *Leviathan*. Felix Meiner Verlag, 2005.
- [47] Gottfried Wilhelm Leibniz. Characteristica universalis and calculus ratiocinator. In L. E. Loemker, editor, *Philosophical Papers and Letters*, pages 221–235. Kluwer Academic, Dordrecht, 1989. Leibniz outlines his project of a universal logical calculus and symbolic language for all human knowledge.
- [48] David Hume. *A treatise of human nature*. Oxford University Press, 2000.

- [49] Immanuel Kant. *Critique of pure reason*. Cambridge university press, 1999.
- [50] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 43–98. Princeton University Press, Princeton, NJ, 1956. Lectures delivered at Caltech, Jan. 1952 (posthumously edited from notes).
- [51] A Aldo Faisal, Luc PJ Selen, and Daniel M Wolpert. Noise in the nervous system. *Nature reviews neuroscience*, 9(4):292–303, 2008.
- [52] Geoffrey E. Hinton, James L. McClelland, and David E. Rumelhart. Distributed representations. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, pages 45–93. MIT Press, Cambridge, MA, 1986. ISBN 9780262681170. doi: <https://doi.org/10.7551/mitpress/5236.003.0006>.
- [53] Simon Thorpe. Localized versus distributed representations. In *The handbook of brain theory and neural networks*, pages 549–552. 1998.
- [54] R Quian Quiroga, Leila Reddy, Gabriel Kreiman, Christof Koch, and Itzhak Fried. Invariant visual representation by single neurons in the human brain. *Nature*, 435(7045):1102–1107, 2005.
- [55] Tim van Gelder. Distributed vs. local representation. In Robert Andrew Wilson and Frank C. Keil, editors, *MIT Encyclopedia of the Cognitive Sciences*. MIT Press, 1999.
- [56] Tony Plate. Distributed representations. *Encyclopedia of Cognitive Science*, 33, 2002.
- [57] Tony A Plate. *Holographic Reduced Representation: Distributed representation for cognitive structures*, volume 150. CSLI Publications Stanford, 2003.
- [58] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
- [59] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [60] Marvin Minsky and Seymour A Papert. *Perceptrons, reissue of the 1988 expanded edition with a new foreword by Léon Bottou: an introduction to computational geometry*. MIT press, 2017.
- [61] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- [62] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing, volume 1: Explorations in the microstructure of cognition: Foundations*. The MIT press, 1986.

- [63] Jerry A Fodor and Zenon W Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.
- [64] Ray Jackendoff. Précis of foundations of language: Brain, meaning, grammar, evolution. *Behavioral and brain sciences*, 26(6):651–665, 2003.
- [65] Ross W Gayler. Vector symbolic architectures answer jackendoff’s challenges for cognitive neuroscience. *arXiv preprint cs/0412059*, 2004.
- [66] Dmitri A Rachkovskij and Ernst M Kussul. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Computation*, 13(2):411–452, 2001.
- [67] Christoph Von Der Malsburg. Am i thinking assemblies? In *Brain Theory: Proceedings of the First Trieste Meeting on Brain Theory, October 1–4, 1984*, pages 161–176. Springer, 1986.
- [68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
- [69] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [70] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0070428072.
- [71] Marco Angioli, Marcello Barbirotta, Antonio Mastrandrea, Saeid Jamili, and Mauro Olivieri. Automatic hardware accelerators reconfiguration through linearcb algorithms on a risc-v processor. In *2023 18th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)*, volume 1, pages 169–172, Valencia, Spain, 2023. IEEE. doi: 10.1109/PRIME58259.2023.10161944.
- [72] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences*, 11(11):4948, 2021.
- [73] Michael L Littman. Reinforcement learning improves behaviour from evaluative feedback. *Nature*, 521(7553):445–451, 2015.
- [74] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web, WWW ’10*, page 661–670, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587998. doi: 10.1145/1772690.1772758. URL <https://doi.org/10.1145/1772690.1772758>.
- [75] Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 297–306, 2011.

- [76] ZOZO Research. zr-obp: Open bandit pipeline, 2021. URL <https://github.com/st-tech/zr-obp/tree/master>. Accessed: 2024-07-12.
- [77] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.
- [78] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.
- [79] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International conference on machine learning*, pages 127–135. PMLR, 2013.
- [80] Robin Allesiardo, Raphaël Féraud, and Djallel Bouneffouf. A neural networks committee for the contextual bandit problem. In *International Conference on Neural Information Processing*, pages 374–381. Springer, 2014.
- [81] Dongruo Zhou, Lihong Li, and Quanquan Gu. Neural contextual bandits with ucb-based exploration. In *International conference on machine learning*, pages 11492–11502. PMLR, 2020.
- [82] Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Efficient implementation of linearucb through algorithmic improvements and vector computing acceleration for embedded learning systems. *ACM Transactions on Embedded Computing Systems*, 24(4):1–23, 2025.
- [83] Audrey Durand, Charis Achilleos, Demetris Iacovides, Katerina Strati, Georgios D Mitsis, and Joelle Pineau. Contextual bandits for adapting treatment in a mouse model of de novo carcinogenesis. In Finale Doshi-Velez, Jim Fackler, Ken Jung, David Kale, Rajesh Ranganath, Byron Wallace, and Jenna Wiens, editors, *Machine learning for healthcare conference*, volume 85 of *Proceedings of Machine Learning Research*, pages 67–82, Palo Alto, California, 17–18 Aug 2018. PMLR. URL <https://proceedings.mlr.press/v85/durand18a.html>.
- [84] Hamsa Bastani and Mohsen Bayati. Online decision making with high-dimensional covariates. *Operations Research*, 68(1):276–294, 2020.
- [85] Ambuj Tewari and Susan A. Murphy. *From Ads to Interventions: Contextual Bandits in Mobile Health*, pages 495–517. Springer International Publishing, Cham, 2017. ISBN 978-3-319-51394-2. doi: 10.1007/978-3-319-51394-2_25. URL https://doi.org/10.1007/978-3-319-51394-2_25.
- [86] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Miro Dudík, Luong Hoang, John Langford, Lihong Li, Dan Melamed, Gal Oshri, Siddhartha Sen, et al. Multiworld testing decision service: A system for experimentation, learning, and decision-making. *Whitepaper of Microsoft*, pages 1–40, 2016.
- [87] Anna Coenen. How The New York Times is experimenting with recommendation algorithms, 2019. URL <https://tinyurl.com/536zku2f>. Accessed: 2026-01-13.

- [88] Fernando Amat, Ashok Chandrashekar, Tony Jebara, and Justin Basilico. Artwork personalization at netflix. In *Proceedings of the 12th ACM conference on recommender systems*, RecSys '18, page 487–488, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359016. doi: 10.1145/3240323.3241729. URL <https://doi.org/10.1145/3240323.3241729>.
- [89] Rojeena Bajrachrya and Haejoon Jung. Contextual bandits approach for selecting the best channel in industry 4.0 network. In *2021 International Conference on Information Networking (ICOIN)*, pages 13–16, Jeju Island, Korea, 2021. IEEE.
- [90] Yingyan Zeng, Xiaoyu Chen, and Ran Jin. Ensemble active learning by contextual bandits for ai incubation in manufacturing. *ACM Transactions on Intelligent Systems and Technology*, 15(1):1–26, 2023.
- [91] Lixing Chen, Jie Xu, Shaolei Ren, and Pan Zhou. Spatio-temporal edge service placement: A bandit learning approach. *IEEE Transactions on Wireless Communications*, 17(12):8388–8401, 2018. doi: 10.1109/TWC.2018.2876823.
- [92] Chien-Sheng Yang, Ramtin Pedarsani, and A Salman Avestimehr. Edge computing in the dark: Leveraging contextual-combinatorial bandit and coded computing. *IEEE/ACM Transactions on Networking*, 29(3):1022–1031, 2021.
- [93] Wali Ullah Khan, Muhammad Awais Javed, Sherali Zeadally, Eva Lagunas, and Symeon Chatzinotas. Intelligent and secure radio environments for 6g vehicular aided hetnets: Key opportunities and challenges. *IEEE Communications Standards Magazine*, 7(3):32–39, 2023. doi: 10.1109/MCOMSTD.0007.2200065.
- [94] Rohan Banerjee, Rajat Kumar Jenamani, Sidharth Vasudev, Amal Nanavati, Katherine Dimitropoulou, Sarah Dean, and Tapomayukh Bhattacharjee. To ask or not to ask: Human-in-the-loop contextual bandits with applications in robot-assisted feeding. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1378–1384, 2025.
- [95] Marilyn Wolf. *Computers as components: principles of embedded computing system design*. Elsevier, 2012.
- [96] Fortune Business Insights. Ai inference market size, share & growth forecast, January 2026. URL <https://www.fortunebusinessinsights.com/ai-inference-market-113705>. Accessed February 3, 2026.
- [97] Atmel Corporation. ATmega328P 8-bit AVR microcontroller with 32k bytes in-system programmable flash. Datasheet, 2014. URL <https://tinyurl.com/pwlodel>. p. 268, Accessed: 2026-02-24.
- [98] Atmel Corporation. ATmega640/V, ATmega1280/V, ATmega1281/V, ATmega2560/V, ATmega2561/V microcontrollers. Datasheet, 2014. URL <https://tinyurl.com/y23wkvbe>. p. 376, Accessed: 2026-02-24.
- [99] STMicroelectronics. STM32L073x8, STM32L073xB ultra-low-power 32-bit MCU Arm Cortex-M0+. Datasheet, 2018. URL <https://tinyurl.com/22guh2ks>. p. 67, Accessed: 2026-02-24.

- [100] Atmel Corporation. SAM D21E, SAM D21G, SAM D21J 32-bit Cortex-M0+ microcontrollers. Datasheet, 2014. URL <https://tinyurl.com/266m8nxz>. p. 942, Accessed: 2026-02-24.
- [101] Atmel Corporation. SAM3X, SAM3A Series 32-bit Cortex-M3 microcontrollers. Datasheet, 2014. URL <https://tinyurl.com/29ndvdbf>. p. 1391, Accessed: 2026-02-24.
- [102] STMicroelectronics. STM32F215xx, STM32F217xx high-performance Arm Cortex-M3 32-bit MCUs. Datasheet, 2012. URL <https://tinyurl.com/238qydw>. p. 78, Accessed: 2026-02-24.
- [103] STMicroelectronics. STM32F469xx high-performance Arm Cortex-M4 32-bit MCUs. Datasheet, 2016. URL <https://tinyurl.com/26g3gdm7>. p. 100, Accessed: 2026-02-24.
- [104] Jeff Geerling. Power consumption benchmarks — Raspberry Pi Dramble. Online resource, 2022. URL <https://tinyurl.com/y4p24yra>. Accessed: 2026-02-24.
- [105] Li Minn Ang and Kah Phooi Seng. Gpu-based embedded intelligence architectures and applications. *Electronics*, 10(8):952, 2021. doi: 10.3390/electronics10080952. URL <https://www.mdpi.com/2079-9292/10/8/952>.
- [106] Esteban Anides, Guillermo Salinas, Eduardo Pichardo, Juan G Avalos, Giovanni Sánchez, Juan C Sánchez, Gabriel Sánchez, Eduardo Vazquez, and Linda K Toscano. A real-time fpga-based metaheuristic processor to efficiently simulate a new variant of the pso algorithm. *Micromachines*, 14(4):809, 2023. doi: 10.3390/mi14040809. URL <https://www.mdpi.com/2072-666X/14/4/809>.
- [107] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual. volume I: user-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. URL <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf>.
- [108] Hyun Woo Oh and Seung Eun Lee. The design of optimized risc processor for edge artificial intelligence based on custom instruction set extension. *IEEE Access*, 11:49409–49421, 2023. doi: 10.1109/ACCESS.2023.3276411. URL <https://doi.org/10.1109/ACCESS.2023.3276411>.
- [109] Vaibhav Verma and Mircea R Stan. Ai-pim—extending the risc-v processor with processing-in-memory functional units for ai inference at the edge of iot. *Frontiers in Electronics*, 3:898273, 2022. doi: 10.3389/felec.2022.898273. URL <https://doi.org/10.3389/felec.2022.898273>.
- [110] Karthikeyan Kalyanasundaram Balasubramanian, Mirco Di Salvo, Walter Rocchia, Sergio Decherchi, and Marco Crepaldi. Designing risc-v instruction set extensions for artificial neural networks: An llvm compiler-driven perspective.

- IEEE Access*, 12:55925–55944, 2024. doi: 10.1109/ACCESS.2024.3389673. URL <https://doi.org/10.1109/ACCESS.2024.3389673>.
- [111] Vasileios Titopoulos, Kosmas Alexandridis, Christodoulos Peltekis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Indexmac: A custom risc-v vector instruction to accelerate structured-sparse matrix multiplications. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.
- [112] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, August 2014. URL <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf>.
- [113] Abdallah Cheikh, Gianmarco Cerutti, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. The microarchitecture of a multi-threaded risc-v compliant processing core family for iot end-nodes. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES 2017)*, volume 512 of *Lecture Notes in Electrical Engineering*, pages 89–97, Cham, Switzerland, 2017. Springer. doi: 10.1007/978-3-319-93082-4_12. URL https://doi.org/10.1007/978-3-319-93082-4_12.
- [114] Abdallah Cheikh, Stefano Sordillo, Antonio Mastrandrea, Francesco Menichelli, Giuseppe Scotti, and Mauro Olivieri. Klessydra-t: Designing vector coprocessors for multithreaded edge-computing cores. *IEEE Micro*, 41(2):64–71, 2021.
- [115] PULP Platform. Pulpino: An open-source microcontroller system based on 32-bit RISC-V cores. <https://github.com/pulp-platform/pulpino>, 2016. Accessed: 2026-02-23.
- [116] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Marco Ottavi, and Mauro Olivieri. Evaluation of dynamic triple modular redundancy in an interleaved-multi-threading risc-v core. *Journal of Low Power Electronics and Applications*, 13(1):2, 2022.
- [117] Advanced Micro Devices, Inc. Performance and resource utilization for floating-point v7.1, 2025. URL https://download.amd.com/docnav/documents/ip_attachments/floating-point.html. Vivado Design Suite Release 2025.1. Accessed: 2025-12-26.
- [118] Mark Horowitz. 1.1 computing’s energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.
- [119] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

- [120] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [121] Udayan S Patankar, Miguel E Flores, and Ants Koel. Division algorithms-from past to present chance to improve area time and complexity for digital applications. In *2020 IEEE Latin America Electron Devices Conference (LAEDC)*, pages 1–4. IEEE, 2020.
- [122] Rainer Karl Ludwig Trummer. *A high-performance data-dependent hardware integer divider*. Citeseer, 2005.
- [123] Udayan S Patankar and Ants Koel. Review of basic classes of dividers based on division algorithm. *IEEE Access*, 9:23035–23069, 2021.
- [124] Stuart F Obermann and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on computers*, 46(8):833–854, 2002.
- [125] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd annual design automation conference*, pages 1–6. ACM, 2015. doi: 10.1145/2744769.2751163. URL <https://doi.org/10.1145/2744769.2751163>.
- [126] Zhenhong Liu, Amir Yazdanbakhsh, Taejoon Park, Hadi Esmaeilzadeh, and Nam Sung Kim. Simul: An algorithm-driven approximate multiplier design for machine learning. *IEEE Micro*, 38(4):50–59, 2018.
- [127] Mohsen Imani, Ricardo Garcia, Andrew Huang, and Tajana Rosing. Cade: Configurable approximate divider for energy efficiency. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 586–589. IEEE, 2019.
- [128] Alessandro De Gloria and Mauro Olivieri. Completion-detecting carry select addition. *IEE Proceedings-Computers and Digital Techniques*, 147(2):93–100, 04 2000. doi: 10.1049/ip-cdt:20000451.
- [129] Mauro Olivieri and Antonio Mastrandrea. A general design methodology for synchronous early-completion-prediction adders in nano-cmos dsp architectures. *VLSI Design*, 2013(1):785281, 2013. doi: 10.1155/2013/785281.
- [130] Swaroop Ghosh, Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. Voltage scalable high-speed robust hybrid arithmetic units using adaptive clocking. *IEEE transactions on very large scale integration (VLSI) systems*, 18(9):1301–1309, 2009.
- [131] Mauro Olivieri. Design of synchronous and asynchronous variable-latency pipelined multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):365–376, 2002. doi: 10.1109/92.924058.

- [132] Ray Andraka. A survey of cordic algorithms for fpga based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, page 191–200, New York, NY, USA, 1998. ACM. ISBN 0897919785. doi: 10.1145/275107.275139.
- [133] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. doi: 10.1109/JPROC.2017.2761740.
- [134] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.
- [135] Ismail Lamaakal, Chaymae Yahyati, Ibrahim Ouahbi, Khalid El Makkaoui, and Yassine Maleh. A survey of model compression techniques for tinyml applications. In *2025 International Conference on Circuit, Systems and Communication (ICCSC)*, pages 1–6. IEEE, 2025.
- [136] Jacob Sander, Achraf Cohen, Venkat R Dasari, Brent Venable, and Brian Jalalian. On accelerating edge ai: Optimizing resource-constrained environments. *arXiv preprint arXiv:2501.15014*, 2025.
- [137] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.
- [138] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.
- [139] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- [140] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*, pages 291–326. Chapman and Hall/CRC, 2022.
- [141] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [142] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.
- [143] Sinong Wang, Belinda Z Li, Madian Khabisa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.
- [144] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

- [145] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.
- [146] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.
- [147] Behrouz Farhang-Boroujeny. *Adaptive filters: theory and applications*. John wiley & sons, 2013.
- [148] Ali H Sayed. *Fundamentals of adaptive filtering*. John Wiley & Sons, 2003.
- [149] Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.
- [150] William W Hager. Updating the inverse of a matrix. *SIAM review*, 31(2): 221–239, 1989.
- [151] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.
- [152] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU – datasheet. Technical report, NVIDIA, 2020. URL <https://tinyurl.com/2n94ssom>. Accessed: 2026-02-23. Page 2: "Max Thermal Design Power (TDP): 400W*** (SXM4 module)".
- [153] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2): 292–308, 2019.
- [154] Norman P Jouppi et al. TPU v4: Google’s fourth-generation tensor processing unit, 2023. Google Blog, <https://tinyurl.com/26cc5yvg>.
- [155] Google Coral Team. Google edge TPU: Performance benchmarks and technical documentation, 2022. Original whitepaper URL discontinued; technical specs at <https://tinyurl.com/2y4v67ao>.
- [156] Arm Ltd. Arm Ethos-U55 NPU, 2021. <https://tinyurl.com/28lrxq6r>.
- [157] Hailo Technologies Ltd. Hailo-8 AI processor: Product brief, 2023. <https://tinyurl.com/284m4p6e>.
- [158] Apple Inc. A17 Pro - neural engine, 2023. Available: <https://tinyurl.com/yqq6abvd>.

- [159] Apple Inc. Apple M2 chip technical overview, 2022. <https://tinyurl.com/26petrk5>.
- [160] NVIDIA Corporation. NVIDIA Grace Hopper superchip architecture, 2023. <https://tinyurl.com/2eq5cq8n>.
- [161] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi: 10.1038/nature14539.
- [162] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- [163] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13): 3521–3526, 2017.
- [164] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1(2):139–159, 2009.
- [165] Olaf Sporns. *Networks of the Brain*. MIT press, 2016.
- [166] David Attwell and Simon B Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, 21(10):1133–1145, 2001.
- [167] Simon B Laughlin and Terrence J Sejnowski. Communication in neuronal networks. *Science*, 301(5641):1870–1874, 2003.
- [168] William B Levy and Victoria G Calvert. Communication consumes 35 times more energy than computation in the human cortex, but both costs are needed to predict synapse number. *Proceedings of the National Academy of Sciences*, 118(18):e2008173118, 2021.
- [169] Daniela Gandolfi, Jonathan Mapelli, and Francesco Maria Puglisi. Brain-inspired computing: from neuroscience to neuromorphic electronics for new forms of artificial intelligence, 2025.
- [170] Sohan Shankar, Yi Pan, Hanqi Jiang, Zhengliang Liu, Mohammad R Darbandi, Agustin Lorenzo, Junhao Chen, Md Mehedi Hasan, Arif Hassan Zidan, Eliana Gelman, et al. Bridging brains and machines: A unified frontier in neuroscience, artificial intelligence, and neuromorphic systems, 2025.
- [171] Alexander Ororbia, Ankur Mali, Adam Kohan, Beren Millidge, and Tommaso Salvatori. A review of neuroscience-inspired machine learning. *arXiv preprint arXiv:2403.18929*, 2024.
- [172] John Von Neumann and Ray Kurzweil. *The computer and the brain*. Yale university press, 2012.

- [173] James C Knight, Philip J Tully, Bernhard A Kaplan, Anders Lansner, and Steve B Furber. Large-scale simulations of plastic neural networks on neuromorphic hardware. *Frontiers in neuroanatomy*, 10:37, 2016.
- [174] Steve Furber. Biologically-inspired massively-parallel computing. In Teerachot Siriburanon, Mohammed A. Maktoomi, Taro Miyatani, Toru Ishii, Hitoshi Yamaguchi, and Masahiro Yamaguchi, editors, *Many-Core Computing: Hardware and Software*, pages 459–478. IET, 2019. ISBN 978-1-78561-922-7. doi: 10.1049/PBPC022E_ch22. URL https://digital-library.theiet.org/doi/abs/10.1049/PBPC022E_ch22.
- [175] Alejandro Rodriguez-Garcia, Jie Mei, and Srikanth Ramaswamy. Enhancing learning in spiking neural networks through neuronal heterogeneity and neuromodulatory signaling. *arXiv e-prints*, pages arXiv–2407, 2024.
- [176] Bruno B Averbeck, Peter E Latham, and Alexandre Pouget. Neural correlations, population coding and computation. *Nature reviews neuroscience*, 7(5):358–366, 2006.
- [177] Dmitrii Zendrikov, Sergio Solinas, and Giacomo Indiveri. Brain-inspired methods for achieving robust computation in heterogeneous mixed-signal neuromorphic processing systems. *Neuromorphic Computing and Engineering*, 3(3):034002, 2023.
- [178] Ahsan Adeel, Mario Franco, Mohsin Raza, and Khubaib Ahmed. Context-sensitive neocortical neurons transform the effectiveness and efficiency of neural information processing. *arXiv preprint arXiv:2207.07338*, 2022.
- [179] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
- [180] Timoleon Moraitis, Abu Sebastian, and Evangelos Eleftheriou. Optimality of short-term synaptic plasticity in modelling certain dynamic environments. *arXiv preprint arXiv:2009.06808*, 2020.
- [181] Wolfgang Maass. Noise as a resource for computation and learning in networks of spiking neurons. *Proceedings of the IEEE*, 102(5):860–880, 2014.
- [182] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and post-synaptic cell type. *Journal of neuroscience*, 18(24):10464–10472, 1998. doi: 10.1523/JNEUROSCI.18-24-10464.1998.
- [183] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997. doi: 10.1126/science.275.5306.1593.
- [184] Peter Dayan and Yael Niv. Reinforcement learning: the good, the bad and the ugly. *Current opinion in neurobiology*, 18(2):185–196, 2008.

- [185] Guangzhi Tang, Neelesh Kumar, Ioannis Polykretis, and Konstantinos P Michmizos. Biograd: biologically plausible gradient-based learning for spiking neural networks. *arXiv preprint arXiv:2110.14092*, 2021. doi: 10.48550/arxiv.2110.14092.
- [186] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.
- [187] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.
- [188] Elizabeth S Spelke and Katherine D Kinzler. Core knowledge. *Developmental science*, 10(1):89–96, 2007.
- [189] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.
- [190] Daniel J Felleman and David C Van Essen. Distributed hierarchical processing in the primate cerebral cortex. *Cerebral cortex (New York, NY: 1991)*, 1(1): 1–47, 1991. doi: 10.1093/cercor/1.1.1.
- [191] Bernard J Baars. *A cognitive theory of consciousness*. Cambridge University Press, 1993.
- [192] Stanislas Dehaene and Jean-Pierre Changeux. Experimental and theoretical approaches to conscious processing. *Neuron*, 70(2):200–227, 2011.
- [193] György Buzsáki. *Rhythms of the Brain*. Oxford university press, 2006.
- [194] Giacomo Indiveri, Bernabé Linares-Barranco, Robert Legenstein, George Deligeorgis, and Themistoklis Prodromakis. Integration of nanoscale memristor synapses in neuromorphic computing architectures. *Nanotechnology*, 24(38): 384010, 2013.
- [195] D Ielmini, Z Wang, and Y Liu. Brain-inspired computing via memory device physics. *APL Materials*, 9(5), 2021.
- [196] Damien Querlioz. Memory-centric artificial intelligence with nanodevices. In *ECS Meeting Abstracts*, volume 237 of *Electrochemical Society Meeting Abstracts*, pages 1387–1387. The Electrochemical Society, Inc., ECS, 2020. doi: 10.1149/MA2020-01371387mtgabs. URL <https://ecs.confex.com/ecs/237/webprogram/Paper376214.html>.
- [197] Robert A Nawrocki, Richard M Voyles, and Sean E Shaheen. A mini review of neuromorphic architectures and implementations. *IEEE Transactions on Electron Devices*, 63(10):3819–3829, 2016.

- [198] Priyadarshini Panda and Kaushik Roy. Invited talk: Re-engineering computing with neuro-inspired learning: Devices, circuits, and systems. In *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, pages 1–18. IEEE, 2020.
- [199] Dhireesha Kudithipudi, Catherine Schuman, Craig M Vineyard, Tej Pandit, Cory Merkel, Rajkumar Kubendran, James B Aimone, Garrick Orchard, Christian Mayr, Ryad Benosman, et al. Neuromorphic computing at scale. *Nature*, 637(8047):801–812, 2025.
- [200] Stephen Ornes. Can neuromorphic computing help reduce ai’s high energy cost? *Proceedings of the National Academy of Sciences*, 122(44):e2528654122, 2025.
- [201] Giacomo Indiveri. Neuromorphic is dead. long live neuromorphic. *Neuron*, 113(20):3311–3314, 2025.
- [202] Pentti Kanerva. *Sparse distributed memory*. MIT press, 1988.
- [203] Mohsen Imani, Abbas Rahimi, Deqian Kong, Tajana Rosing, and Jan M Rabaey. Exploring hyperdimensional associative memory. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pages 445–456. IEEE, 2017.
- [204] Abbas Rahimi, Sohum Datta, Denis Kleyko, Edward Paxon Frady, Bruno Olshausen, Pentti Kanerva, and Jan M Rabaey. High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2508–2521, 2017.
- [205] Manuel Schmuck, Luca Benini, and Abbas Rahimi. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(4):1–25, 2019.
- [206] Behnam Khaleghi, Hanyang Xu, Justin Morris, and Tajana Šimunić Rosing. tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 408–413. IEEE, 2021.
- [207] Denis Kleyko, Edward Paxon Frady, and Friedrich T Sommer. Cellular automata can reduce memory requirements of collective-state computing. *IEEE Transactions on Neural Networks and Learning Systems*, 33(6):2701–2713, 2022.
- [208] Sandy A Wasif, Miran Wael, Paul R Genssler, Eman Azab, Maggie Mashaly, Mohamed A Abd El Ghany, and Hussam Amrouch. Domain-specific hyperdimensional risc-v processor for edge-ai training. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.
- [209] Rocco Martino, Marco Angioli, Antonello Rosato, Marcello Barbirotta, Abdallah Cheikh, and Mauro Olivieri. Configurable hardware acceleration for

- hyperdimensional computing extension on risc-v. *IEEE Transactions on Computers*, 2025.
- [210] E Paxon Frady and Friedrich T Sommer. Robust computation with rhythmic spike patterns. *Proceedings of the National Academy of Sciences*, 116(36):18050–18059, 2019.
- [211] Graham Bent, Chris Simpkin, Yuhua Li, and Alun Preece. Hyperdimensional computing using time-to-spike neuromorphic circuits. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.
- [212] Jeff Orchard, P Michael Furlong, and Kathryn Simone. Efficient hyperdimensional computing with spiking phasors. *Neural Computation*, 36(9):1886–1911, 2024.
- [213] Abbas Rahimi, Pentti Kanerva, Luca Benini, and Jan M Rabaey. Efficient biosignal processing using hyperdimensional computing: Network templates for combined learning and classification of exg signals. *Proceedings of the IEEE*, 107(1):123–143, 2018.
- [214] Magnus Sahlgren. An introduction to random indexing. In *Methods and applications of semantic indexing workshop at the 7th international conference on terminology and knowledge engineering*, 2005.
- [215] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *science*, 338(6111):1202–1205, 2012.
- [216] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi. A survey on hyperdimensional computing aka vector symbolic architectures, Part II: Applications, cognitive models, and challenges. *ACM Computing Surveys*, 55(9):1–52, 2023.
- [217] Denis Kleyko, Mike Davies, Edward Paxon Frady, Pentti Kanerva, Spencer J Kent, Bruno A Olshausen, Evgeny Osipov, Jan M Rabaey, Dmitri A Rachkovskij, Abbas Rahimi, and F. T. Sommer. Vector symbolic architectures as a computing framework for emerging hardware. *Proceedings of the IEEE*, 110(10):1538–1571, 2022.
- [218] Pere Vergés, Mike Heddes, Igor Nunes, Denis Kleyko, Tony Givargis, and Alexandru Nicolau. Classification using hyperdimensional computing: A review with comparative analysis. *Artificial Intelligence Review*, 58(6):173, 2025.
- [219] Alejandro Hernandez-Cano, Cheng Zhuo, Xunzhao Yin, and Mohsen Imani. Reghd: Robust and efficient regression in hyper-dimensional learning system. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 7–12. IEEE, 2021.
- [220] E Paxon Frady, Denis Kleyko, Christopher J Kymn, Bruno A Olshausen, and Friedrich T Sommer. Computing on functions using randomized vector representations (in brief). In *Proceedings of the 2022 Annual Neuro-Inspired Computational Elements Conference*, pages 115–122, 2022.

- [221] Mohsen Imani, Yeseong Kim, Thomas Worley, Saransh Gupta, and Tajana Rosing. Hdcluster: An accurate clustering using brain-inspired high-dimensional computing. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1591–1594, 2019. doi: 10.23919/DATE.2019.8715147.
- [222] Tharindu Bandaragoda, Daswin De Silva, Denis Kleyko, Evgeny Osipov, Urban Wiklund, and Damminda Alahakoon. Trajectory clustering of road traffic in urban environments using incremental machine learning in combination with hyperdimensional computing. In *2019 IEEE intelligent transportation systems conference (ITSC)*, pages 1664–1670. IEEE, 2019.
- [223] Denis Kleyko, Evgeny Osipov, Ross W Gayler, Asad I Khan, and Adrian G Dyer. Imitation of honey bees’ concept learning processes using vector symbolic architectures. *Biologically Inspired Cognitive Architectures*, 14:57–72, 2015.
- [224] Yang Ni, Mariam Issa, Danny Abraham, Mahdi Imani, Xunzhao Yin, and Mohsen Imani. Hdpg: Hyperdimensional policy-based reinforcement learning for continuous control. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1141–1146, 2022.
- [225] Marco Angioli, Antonello Rosato, Marcello Barbirotta, Rocco Martino, Francesco Menichelli, and Mauro Olivieri. Hd-cb: The first exploration of hyperdimensional computing for contextual bandits problems. *IEEE Open Journal of the Computer Society*, pages 1–12, 2025. doi: 10.1109/OJCS.2025.3642108.
- [226] Alexander N Gorban and Ivan Yu Tyukin. Blessing of dimensionality: mathematical foundations of the statistical physics of data. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376(2118):20170237, 2018.
- [227] Pentti Kanerva. A family of binary spatter codes. In *International Conference on Artificial Neural Networks (ICANN)*, pages 517–522, 1995.
- [228] RW Gayler. Multiplicative binding, representation operators, and analogy [abstract of poster]. *Advances in analogy research: Integration of theory and data from the cognitive, computational, and neural sciences*, ed. K. Holyoak, D. Gentner & B. Kokinov. New Bulgarian University. Available at: <http://cogprints.org/502/>. [RWG, FTS], 1998.
- [229] Manuel Eggimann, Abbas Rahimi, and Luca Benini. A 5 μ w standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(10):4116–4128, 2021. doi: 10.1109/TCSI.2021.3100266.
- [230] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. In-memory hyperdimensional computing. *Nature Electronics*, 3(6):327–337, 2020.
- [231] Vidura Sumanasena, Daswin De Silva, Evgeny Osipov, Dmitri Rachkovskij, and Ross W Gayler. Implementing holographic reduced representations for

- spiking neural networks. *IEEE Access*, 13:116606–116620, 2025. doi: 10.1109/ACCESS.2025.3580582.
- [232] Christopher Kymn, Connor Bybee, Zeyu Yun, Denis Kleyko, and Bruno Olshausen. Oscillator associative memories facilitate high-capacity, compositional inference. In *New Frontiers in Associative Memories*, 2025.
- [233] Jan Gosmann and Chris Eliasmith. Vector-derived transformation binding: An improved binding operation for deep symbol-like processing in neural networks. *Neural computation*, 31(5):849–869, 2019.
- [234] Stephen I Gallant and T Wendy Okaywe. Representing objects, relations, and sequences. *Neural computation*, 25(8):2038–2078, 2013.
- [235] Tony A Plate. *Distributed representations and nested compositional structure*. University of Toronto, Department of Computer Science, 1994.
- [236] P Andrew Penz. The closeness code: an input integer to binary vector transformation suitable for neural network algorithms. In *Proceedings of IEEE First International Conference on Neural Networks*, pages 515–522, 1987.
- [237] Dmitriy A Rachkovskiy, Sergey V Slipchenko, Ernst M Kussul, and Tatyana N Baidyk. Sparse binary distributed encoding of scalars. *Journal of Automation and Information Sciences*, 37(6):12–23, 2005. ISSN 1064-2315.
- [238] Dominic Widdows and Trevor Cohen. Reasoning with vectors: A continuous model for fast robust inference. *Logic Journal of the IGPL*, 23(2):141–173, 2015.
- [239] Abbas Rahimi, Simone Benatti, Pentti Kanerva, Luca Benini, and Jan M. Rabaey. Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016. doi: 10.1109/ICRC.2016.7738683.
- [240] Igor Nunes, Mike Heddes, Tony Givargis, and Alexandru Nicolau. An extension to basis-hypervectors for learning from circular data in hyperdimensional computing. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.
- [241] Tao Yu, Yichi Zhang, Zhiru Zhang, and Christopher M De Sa. Understanding hyperdimensional computing for parallel single-pass learning. *Advances in neural information processing systems*, 35:1157–1169, 2022.
- [242] Shijin Duan, Xiaolin Xu, and Shaolei Ren. A brain-inspired low-dimensional computing classifier for inference on tiny devices. *arXiv preprint arXiv:2203.04894*, 2022.
- [243] Mike Heddes, Igor Nunes, Tony Givargis, Alexandru Nicolau, and Alex Veidenbaum. Hyperdimensional hashing: A robust and efficient dynamic hash table. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 907–912, 2022.

- [244] Anthony Thomas, Sanjoy Dasgupta, and Tajana Rosing. A theoretical perspective on hyperdimensional computing. *Journal of Artificial Intelligence Research*, 72:215–249, 2021.
- [245] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.
- [246] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20, 2007.
- [247] Lulu Ge and Keshab K. Parhi. Robust Clustering Using Hyperdimensional Computing. *IEEE Open Journal of Circuits and Systems*, 5:102, 2024. doi: 10.1109/ojcas.2024.3381508.
- [248] E Paxon Frady, Spencer J Kent, Bruno A Olshausen, and Friedrich T Sommer. Resonator networks, 1: An efficient solution for factoring high-dimensional, distributed representations of data structures. *Neural computation*, 32(12): 2311–2331, 2020.
- [249] Spencer J Kent, E Paxon Frady, Friedrich T Sommer, and Bruno A Olshausen. Resonator networks, 2: Factorization performance and capacity compared to optimization-based methods. *Neural computation*, 32(12):2332–2388, 2020.
- [250] Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Saeid Jamili, and Mauro Olivieri. Design, implementation and evaluation of a new variable latency integer division scheme. *IEEE Transactions on Computers*, 73(7):1767–1779, 2024. doi: 10.1109/TC.2024.3386060.
- [251] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.
- [252] Evgeny Milanov. The rsa algorithm. *RSA laboratories*, 1(11):1–11, 2009.
- [253] Kristina P Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE Access*, 8:80716–80727, 2020.
- [254] RC Mittal and A Al-Kurdi. Lu-decomposition and numerical structure for solving large sparse nonsymmetric linear systems. *Computers & Mathematics with Applications*, 43(1-2):131–155, 2002.
- [255] Walter Gander. Algorithms for the qr decomposition. *Res. Rep*, 80(02): 1251–1268, 1980.
- [256] Xiaojun Wang. Variable precision floating-point divide and square root for efficient fpga implementation of image and signal processing algorithms. *Doctoral Thesis-Northeastern University*, 2007.
- [257] Donald G Bailey. Space efficient division on fpgas. In *Electronics New Zealand Conference (EnzCon’06)*, pages 206–211, 2006.

- [258] Rohini S Hongal and DJ Anita. Comparative study of different division algorithms for fixed and floating point arithmetic unit for embedded applications. *International Journal of Computer Sciences and Engineering*, 4(9):48–54, 2016.
- [259] Xin Fang and Miriam Leeser. Open-source variable-precision floating-point library for major commercial fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(3):1–17, 2016.
- [260] Xilinx. Microblaze processor reference guide, Oct 2021. URL https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf.
- [261] Intel. Nios ii processor reference guide, Oct 2020. URL <https://www.intel.com/content/www/us/en/docs/programmable/683836/current/introduction.html>.
- [262] Milos D Ercegovic and Tomas Lang. *Digital arithmetic*. Elsevier, 2004.
- [263] Nitish Aggarwal, Kartik Asooja, Saurabh Shekhar Verma, and Sapna Negi. An improvement in the restoring division algorithm (needy restoring division algorithm). In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 246–249. IEEE, 2009.
- [264] Firas Hassan, Ahmed Ammar, and Hayden Drennen. A 32-bit integer division algorithm based on priority encoder. In *2020 27th IEEE international conference on electronics, circuits and systems (ICECS)*, pages 1–4. IEEE, 2020.
- [265] Mauro Olivieri, Abdallah Cheikh, Gianmarco Cerutti, Antonio Mastrandrea, and Francesco Menichelli. Investigation on the optimal pipeline organization in risc-v multi-threaded soft processor cores. In *2017 New Generation of CAS (NGCAS)*, pages 45–48. IEEE, 2017.
- [266] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors. *IEEE Access*, 10:126074–126088, 2022.
- [267] M. Morris R. Mano and Charles R. Kime. *Logic and Computer Design Fundamentals*. Pearson Prentice Hall, Upper Saddle River, NJ, 2007. ISBN 978-0131989269. URL <https://www.pearson.com>.
- [268] Ioannis Tsiokanos, Lev Mukhanov, and Georgios Karakonstantis. Low-power variation-aware cores based on dynamic data-dependent bitwidth truncation. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 698–703. IEEE, 2019.
- [269] Stefania Perri, Fanny Spagnolo, Fabio Frustaci, and Pasquale Corsonello. Design of leading zero counters on fpgas. *IEEE Embedded Systems Letters*, 15(3):149–152, 2022.

- [270] Ahmed Ammar, Hayden Drennen, and Firas Hassan. High-precision priority encoder based integer division algorithm. In *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 494–497. IEEE, 2021. doi: 10.1109/MWSCAS47672.2021.9531809.
- [271] J. M. Torres-Palma. Sc-rsa: Basic and minimal implementation of rsa-32 in c. Online: <https://github.com/jmtorrespalma/sc-rsa>, 2016. GitHub repository.
- [272] Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, and Mauro Olivieri. Exploring variable latency dividers in vector hardware accelerators. In *2024 19th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)*, volume 1, pages 1–4, Larnaca, Cyprus, 2024. IEEE, Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/PRIME61930.2024.10559734.
- [273] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Marco Angioli, Saeid Jamili, and Mauro Olivieri. Fault-tolerant hardware acceleration for high-performance edge-computing nodes. *Electronics*, 12(17):3574, 2023. ISSN 2079-9292. doi: 10.3390/electronics12173574. URL <https://www.mdpi.com/2079-9292/12/17/3574>.
- [274] Matthew Johns and Tom J Kazmierski. A minimal risc-v vector processor for embedded systems. In *2020 forum for specification and design languages (FDL)*, pages 1–4. IEEE, 2020.
- [275] Abdallah Cheikh, Stefano Sordillo, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Efficient mathematical accelerator design coupled with an interleaved multi-threading risc-v microprocessor. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society*, pages 529–539. Springer, 2019.
- [276] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Analysis of a fault tolerant edge-computing microarchitecture exploiting vector acceleration. In *2022 17th Conference on Ph. D Research in Microelectronics and Electronics (PRIME)*, pages 237–240. IEEE, 2022.
- [277] David Cortes. contextualbandits: Contextual bandit algorithms, 2021. URL <https://github.com/david-cortes/contextualbandits/tree/master>. Accessed: 2024-07-12.
- [278] Radu Ciucanu, Marta Soare, and Sihem Amer-Yahia. Implementing linear bandits in off-the-shelf sqlite. In *EDBT*, volume 2, pages 388–2, 2022.
- [279] Timothy A Mann, Hugo Penedones, Shie Mannor, and Todd Hester. Adaptive lambda least-squares temporal difference learning. *arXiv preprint arXiv:1612.09465*, 2016.
- [280] Shicheng Guo, Xueyu Wei, Yun Xu, Wei Xue, Xuangou Wu, and Bo Wei. A survey of linear value function approximation in reinforcement learning. In

- International Symposium on Intelligence Computation and Applications*, pages 266–280. Springer, 2021.
- [281] Elias D Nino Ruiz, Adrian Sandu, and Jeffrey Anderson. An efficient implementation of the ensemble kalman filter based on an iterative sherman–morrison formula. *Statistics and Computing*, 25(3):561–577, 2015.
- [282] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990. doi: 10.1109/2.44900.
- [283] Marcello Barbirotta, Francesco Menichelli, Abdallah Cheikh, Antonio Mastrandrea, Marco Angioli, and Mauro Olivieri. Dynamic triple modular redundancy in interleaved hardware threads: An alternative solution to lockstep multi-cores for fault-tolerant systems. *IEEE Access*, 12:95720–95735, 2024. doi: 10.1109/ACCESS.2024.3425579.
- [284] Dmitri A Rachkovskij, Ivan S Misuno, and Sergey V Slipchenko. Randomized projective methods for the construction of binary sparse vector representations. *Cybernetics and Systems Analysis*, 48(1):146–156, 2012.
- [285] Sanjoy Dasgupta, Charles F. Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. *Science*, 358(6364):793–796, 2017. doi: 10.1126/science.aam9868. URL <https://www.science.org/doi/abs/10.1126/science.aam9868>.
- [286] Dmitri A Rachkovskij. Vector data transformation using random binary matrices. *Cybernetics and Systems Analysis*, 50(6):960–968, 2014.
- [287] Dmitri A Rachkovskij. Formation of similarity-reflecting binary vectors with random binary projections. *Cybernetics and Systems Analysis*, 51(2):313–323, 2015.
- [288] Sophie JC Caron, Vanessa Ruta, Larry F Abbott, and Richard Axel. Random convergence of olfactory inputs in the drosophila mushroom body. *Nature*, 497(7447):113–117, 2013.
- [289] Andrew C Lin, Alexei M Bygrave, Alix De Calignon, Tzumin Lee, and Gero Miesenböck. Sparse, decorrelated odor coding in the mushroom body enhances learned odor discrimination. *Nature neuroscience*, 17(4):559–568, 2014.
- [290] Teuvo Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, 100(4):353–359, 1972.
- [291] James A Anderson. A simple neural network generating an interactive memory. *Mathematical biosciences*, 14(3-4):197–220, 1972.
- [292] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

- [293] Mika Laiho, Jussi H Poikonen, Pentti Kanerva, and Eero Lehtonen. High-dimensional computing with sparse vectors. In *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4, 2015.
- [294] E Paxon Frady, Denis Kleyko, and Friedrich T Sommer. Variable binding for sparse distributed representations: Theory and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 34(5):2191–2204, 2023.
- [295] Denis Kleyko and Dmitri A Rachkovskij. On design choices in similarity-preserving sparse randomized embeddings. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2024.
- [296] Cheng-Yang Chang, Yu-Chuan Chuang, Chi-Tse Huang, and An-Yeu Wu. Recent progress and development of hyperdimensional computing (hdc) for edge intelligence. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 13(1):119–136, 2023. doi: 10.1109/JETCAS.2023.3242767.
- [297] Eman Hassan, Yasmin Halawani, Baker Mohammad, and Hani Saleh. Hyperdimensional computing challenges and opportunities for ai applications. *IEEE Access*, 10:97651–97664, 2021.
- [298] Yang Ni, Danny Abraham, Mariam Issa, Yeseong Kim, Pietro Mercati, and Mohsen Imani. Qhd: A brain-inspired hyperdimensional reinforcement learning algorithm. *arXiv preprint arXiv:2205.06978*, 2022.
- [299] Yuta Saito, Shunsuke Aihara, Megumi Matsutani, and Yusuke Narita. Open bandit dataset and pipeline: Towards realistic and reproducible off-policy evaluation. *arXiv preprint arXiv:2008.07146*, 2020.
- [300] Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms*. Cambridge University Press, 2020. ISBN 9781108571401. doi: 10.1017/9781108571401. URL <https://doi.org/10.1017/9781108571401>.
- [301] Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020. doi: 10.21105/joss.02174. URL <https://doi.org/10.21105/joss.02174>.
- [302] Andrew Collins, Dominika Tkaczyk, and Joeran Beel. One-at-a-time: A meta-learning recommender-system for recommendation-algorithm selection on micro level. *arXiv preprint arXiv:1805.12118*, 2018.
- [303] Sercan Aygun, Mehran Shoushtari Moghadam, M Hassan Najafi, and Mohsen Imani. Learning from hypervectors: A survey on hypervector encoding. *arXiv preprint arXiv:2308.00685*, 2023.
- [304] Marco Angioli, Antonello Rosato, Marcello Barbirotta, Rocco Martino, Andrea Marcelli, Antonio Mastrandrea, and Mauro Olivieri. Hd-cb bin: A lightweight approach for contextual bandit learning in real-time applications. In *2025 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2025.

- [305] William Andrew Simon, Una Pale, Tomas Teijeiro, and David Atienza. Hd-torch: Accelerating hyperdimensional computing with gp-gpus for design space exploration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–8, 2022.
- [306] Manar Abu Talib, Sohaib Majzoub, Qassim Nasir, and Dina Jamal. A systematic literature review on hardware implementation of artificial intelligence algorithms: Ma talib et al. *The Journal of Supercomputing*, 77(2):1897–1938, 2021.
- [307] Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Dario Natale, Marco Re, and Sergio Spanò. “mr q-learning” algorithm for efficient hardware implementations. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, pages 1186–1190. IEEE, 2021.
- [308] Kofi Appiah, Andrew Hunter, Hongying Meng, Shigang Yue, Mervyn Hobden, Nigel Priestley, Peter Hobden, and Cy Pettit. A binary self-organizing map and its fpga implementation. In *2009 International Joint Conference on Neural Networks*, pages 164–171. IEEE, 2009.
- [309] Denis Kleyko, Evgeny Osipov, Daswin De Silva, Urban Wiklund, and Daminda Alahakoon. Integer self-organizing maps for digital hardware. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.
- [310] Moritz Hardt and Benjamin Recht. *Patterns, predictions, and actions: Foundations of machine learning*. Princeton University Press, 2022.
- [311] Chae Young Lee, Sara Achour, and Zerina Kapetanovic. Navhd: Low-power learning for micro-robotic controls in the wild. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 13399–13405. IEEE, 2025.
- [312] Boris Igel'nik and Yoh-Han Pao. Stochastic choice of basis functions in adaptive function approximation and the functional-link net. *IEEE transactions on Neural Networks*, 6(6):1320–1329, 1995.
- [313] Yoh-Han Pao, Gwang-Hoon Park, and Dejan J Sobajic. Learning and generalization characteristics of the random vector functional-link net. *Neurocomputing*, 6(2):163–180, 1994.
- [314] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.
- [315] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang. Extreme learning machine for regression and multiclass classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):513–529, 2011.
- [316] Guang-Bin Huang, Lei Chen, and Chee-Kheong Siew. Universal approximation using incremental constructive feedforward networks with random hidden nodes. *IEEE transactions on neural networks*, 17(4):879–892, 2024.

- [317] Denis Kleyko, Mansour Kheffache, E Paxon Frady, Urban Wiklund, and Evgeny Osipov. Density encoding enables resource-efficient randomly connected neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(8):3777–3783, 2020.
- [318] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.
- [319] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee micro*, 38(1):82–99, 2018.
- [320] Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.
- [321] Stefano Bertazzoni, Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Marco Re, and Sergio Spanò. Design space exploration for edge machine learning featured by mathworks fpga dl processor: a survey. *IEEE Access*, 12:9418–9439, 2024.
- [322] Neftali Watkinson, Divya Devineni, Victor Joe, Tony Givargis, Alexandru Nicolau, and Alexander Veidenbaum. Using hyperdimensional computing to extract features for the detection of type 2 diabetes. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 149–156. IEEE, 2023.
- [323] Alessio Burrello, Lukas Cavigelli, Kaspar Schindler, Luca Benini, and Abbas Rahimi. Laelaps: An energy-efficient seizure detection algorithm from long-term human ieeg recordings without false alarms. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 752–757. IEEE, 2019.
- [324] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 115–120. IEEE, 2020.
- [325] Tianyang Yu, Bi Wu, Ke Chen, Gong Zhang, and Weiqiang Liu. Fully learnable hyperdimensional computing framework with ultratiny accelerator for edge-side applications. *IEEE Transactions on Computers*, 73(2):574–585, 2023.
- [326] Tianyang Yu, Bi Wu, Ke Chen, Gong Zhang, and Weiqiang Liu. Lahdc: Logic-aggregation-based query for embedded hyperdimensional computing accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(1):119–129, 2024.

- [327] Behnam Khaleghi, Jaeyoung Kang, Hanyang Xu, Justin Morris, and Tajana Rosing. Generic: highly efficient learning engine on edge using hyperdimensional computing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1117–1122, 2022.
- [328] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana Rosing. F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 53–62, 2019.
- [329] Tinaqi Zhang, Sahand Salamat, Behnam Khaleghi, Justin Morris, Baris Aksanli, and Tajana Simunic Rosing. Hd2fpga: Automated framework for accelerating hyperdimensional computing on fpgas. In *2023 24th International Symposium on Quality Electronic Design (ISQED)*, pages 1–9. IEEE, 2023.
- [330] Marco Angioli, Saeid Jamili, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Antonello Rosato, and Mauro Olivieri. Aeneashdc: An automatic framework for deploying hyperdimensional computing models on fpgas. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2024. doi: 10.1109/IJCNN60899.2024.10651081.
- [331] Farhad Taheri, Siavash Bayat-Sarmadi, and Shahriar Hadayeghparast. Risc-hd: Lightweight risc-v processor for efficient hyperdimensional computing inference. *IEEE Internet of Things Journal*, 9(23):24030–24037, 2022. doi: 10.1109/JIOT.2022.3191717.
- [332] D. Campos and J. Bernardes. Cardiotocography. UCI Machine Learning Repository, 2010. DOI: <https://doi.org/10.24432/C51S4N>.
- [333] Jorge Reyes-Ortiz, Davide Anguita, Alessandro Ghio, Luca Oneto, and Xavier Parra. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository, 2013. Accessed: 2024-05-20.
- [334] Iouliia Skliarova. Accelerating population count with a hardware co-processor for microblaze. *Journal of Low Power Electronics and Applications*, 11(2):20, 2021. doi: 10.3390/jlpea11020020.
- [335] Rocco Martino and Marco Angioli. HDCU: Hyperdimensional Coprocessor Unit, 2024. URL <https://github.com/klessydra/HDCU>. GitHub.
- [336] Justin Morris, Roshan Fernando, Yilun Hao, Mohsen Imani, Baris Aksanli, and Tajana Rosing. Locality-based encoder and model quantization for efficient hyper-dimensional computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(4):897–907, 2021.
- [337] A. Rahimi. Emg dataset. <https://github.com/abbas-rahimi/HDC-EMG/blob/master/dataset.mat>, January 2024. URL <https://github.com/abbas-rahimi/HDC-EMG/blob/master/dataset.mat>.
- [338] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936. doi: <https://doi.org/10.1111/j.1469-1809.1936>.

- tb02137.x. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x>.
- [339] Mohsen Imani, Samuel Bosch, Sohum Datta, Sharadhi Ramakrishna, Sahand Salamat, Jan M Rabaey, and Tajana Rosing. Quanthd: A quantization framework for hyperdimensional computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2268–2278, 2019.
 - [340] Rocco Martino, Marco Pisani, Marco Angioli, Marcello Barbirotta, Antonio Mastrandrea, Antonello Rosato, and Mauro Olivieri. A general-purpose AXI plug-and-play hyperdimensional computing accelerator. *Electronics*, 15(2):489, 2026. doi: 10.3390/electronics15020489.
 - [341] Marco Angioli, Christopher J Kymn, Antonello Rosato, Amy Loutfi, Mauro Olivieri, and Denis Kleyko. Efficient hyperdimensional computing with modular composite representations. *arXiv preprint arXiv:2511.09708*, 2025.
 - [342] Christopher J Kymn, Denis Kleyko, E Paxon Frady, Connor Bybee, Pentti Kanerva, Friedrich T Sommer, and Bruno A Olshausen. Computing with residue numbers in high-dimensional representation. *Neural Computation*, 37(1):1–37, 2024.
 - [343] Amos R Omondi and A Benjamin Premkumar. *Residue number systems: theory and implementation*, volume 2. World Scientific, 2007.
 - [344] Ila R Fiete, Yoram Burak, and Ted Brookings. What grid cells convey about rat location. *Journal of Neuroscience*, 28(27):6858–6871, 2008.
 - [345] Christopher J Kymn, Sonia Mazelet, Anthony Thomas, Denis Kleyko, Edward Paxon Frady, Friedrich T Sommer, and Bruno A Olshausen. Binding in hippocampal-entorhinal circuits enables compositionality in cognitive maps. *Advances in neural information processing systems*, 37:39128–39157, 2024.
 - [346] Mike Heddes, Igor Nunes, Pere Vergés, Denis Kleyko, Danny Abraham, Tony Givargis, Alexandru Nicolau, and Alexander Veidenbaum. Torchhd: An open source python library to support research on hyperdimensional computing and vector symbolic architectures. *Journal of Machine Learning Research*, 24(255):1–10, 2023.
 - [347] E Paxon Frady, Denis Kleyko, and Friedrich T Sommer. A theory of sequence indexing and working memory in recurrent neural networks. *Neural Computation*, 30(6):1449–1513, 2018.
 - [348] Michael Hersche, Stefan Lippuner, Matthias Korb, Luca Benini, and Abbas Rahimi. Near-channel classifier: Symbiotic communication and classification in high-dimensional space. *Brain Informatics*, 8(1):16, 2021.
 - [349] Denis Kleyko, Connor Bybee, Ping-Chen Huang, Christopher J Kymn, Bruno A Olshausen, E Paxon Frady, and Friedrich T Sommer. Efficient decoding of compositional structure in holistic representations. *Neural Computation*, 35(7):1159–1186, 2023.

- [350] P Andrew Penz. The closeness code: an input integer to binary vector transformation suitable for neural network algorithms. In *Proceedings of IEEE First International Conference on Neural Networks*, pages 515–522, 1987.
- [351] David Nova and Pablo A Estévez. A review of learning vector quantization classifiers. *Neural Computing and Applications*, 25(3):511–524, 2014.
- [352] Cameron Diao, Denis Kleyko, Jan M Rabaey, and Bruno A Olshausen. Generalized learning vector quantization for classification in randomized neural networks and hyperdimensional computing. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. IEEE, 2021.
- [353] Denis Kleyko, Evgeny Osipov, Nikolaos Papakonstantinou, Valeriy Vyatkin, and Arash Mousavi. Fault detection in the hyperspace: Towards intelligent automation systems. In *2015 IEEE 13th international conference on industrial informatics (INDIN)*, pages 1219–1224. IEEE, 2015.
- [354] Abbas Rahimi, Pentti Kanerva, and Jan M Rabaey. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. In *Proceedings of the 2016 international symposium on low power electronics and design*, pages 64–69, 2016.
- [355] Denis Kleyko, Evgeny Osipov, Nikolaos Papakonstantinou, and Valeriy Vyatkin. Hyperdimensional computing in industrial systems: the use-case of distributed fault isolation in a power plant. *IEEE Access*, 6:30766–30777, 2018.
- [356] Teuvo Kohonen. Improved versions of learning vector quantization. In *1990 ijcn international joint conference on Neural networks*, pages 545–550. IEEE, 1990.
- [357] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The journal of machine learning research*, 15(1):3133–3181, 2014.
- [358] Arthur Asuncion, David Newman, et al. Uci machine learning repository, 2007.
- [359] Mohsen Imani, Justin Morris, Samuel Bosch, Helen Shu, Giovanni De Micheli, and Tajana Rosing. Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing. In *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4. IEEE, 2019.
- [360] Denis Kleyko, Abbas Rahimi, Dmitri A Rachkovskij, Evgeny Osipov, and Jan M Rabaey. Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics. *IEEE Transactions on Neural Networks and Learning Systems*, 29(12):5880–5898, 2018.
- [361] Junhuan Yang, Venkat Kalyan Reddy Yasa, Yi Sheng, Dayane Reis, Xun Jiao, Weiwen Jiang, and Lei Yang. Hardware-aware automated architecture search for brain-inspired hyperdimensional computing. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 352–357. IEEE, 2022.

- [362] Alejandro Hernández-Cano, Namiko Matsumoto, Eric Ping, and Mohsen Imani. Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 56–61. IEEE, 2021.
- [363] Alfred Ultsch and Jörn Lötsch. The fundamental clustering and projection suite (fcps): A dataset collection to test the performance of clustering and data projection algorithms. *Data*, 5(1), 2020. ISSN 2306-5729. URL <https://www.mdpi.com/2306-5729/5/1/13>.
- [364] Boston housing dataset, 2024. URL <https://www.kaggle.com/datasets/vikrishnan/boston-house-prices>.
- [365] Amir Yazdanbakhsh. Beyond moore’s law: harnessing the redshift of generative ai with effective hardware-software co-design. *arXiv preprint arXiv:2504.06531*, 2025.