# Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications

FRANCESCO MINERVINI, OSCAR PALOMAR, OSMAN UNSAL, ENRICO REGGIANI, JOSUE QUIROGA, JOAN MARIMON, CARLOS ROJAS, ROGER FIGUERAS, ABRAHAM RUIZ, ALBERTO GONZALEZ, JONNATAN MENDOZA, IVAN VARGAS, CÉSAR HERNANDEZ, JOAN CABRE, LINA KHOIRUNISYA, MUSTAPHA BOUHALI, JULIAN PAVON, FRANCESC MOLL, and MAURO OLIVIERI, Barcelona Supercomputing Center
MARIO KOVAC, MATE KOVAC, and LEON DRAGIC, University of Zagreb, FER
MATEO VALERO and ADRIAN CRISTAL, Barcelona Supercomputing Center

The maturity level of RISC-V and the availability of domain-specific instruction set extensions, like vector processing, make RISC-V a good candidate for supporting the integration of specialized hardware in processor cores for the High Performance Computing (HPC) application domain. In this article,[1] we present Vitruvius+, the vector processing acceleration engine that represents the core of vector instruction execution in the HPC challenge that comes within the EuroHPC initiative. It implements the RISC-V vector extension (RVV) 0.7.1 and can be easily connected to a scalar core using the Open Vector Interface standard. Vitruvius+ natively supports long vectors: 256 double precision floating-point elements in a single vector register. It is composed of a set of identical vector pipelines (lanes), each containing a slice of the Vector Register File and functional units (one integer, one floating point). The vector instruction execution scheme is hybrid in-order/out-of-order and is supported by register renaming and arithmetic/memory instruction decoupling. On a stand-alone synthesis, Vitruvius+ reaches a maximum frequency of 1.4 GHz in typical conditions (TT/0.80V/25°C) using GlobalFoundries 22FDX FD-SOI. The silicon implementation has a total area of 1.3 $mm^2$ and maximum estimated power of ~920 mW for one instance of Vitruvius+ equipped with eight vector lanes.

[1]This is a new article, not an extension of a conference paper.

This research received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No. 800928 (European Processor Initiative) and Specific Grant Agreement No. 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project PCI2022-132935 is also co-funded by MCIN/AEI/10.13039/501100011033 and by the UE NextGenerationEU/PRTR. This work was also partially supported by the Spanish Ministry of Science and Innovation (PID2019-107255GB-C21/AEI/10.13039/501100011033).

Authors' addresses: F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. Gonzalez, J. Mendoza, I. Vargas, C. Hernandez, J. Cabre, L. Khoirunisya, M. Bouhali, J. Pavon, F. Moll, M. Olivieri, M. Valero, and A. Cristal, Barcelona Supercomputing Center, Placa E. Guell 1-3, Barcelona, Spain; emails: {francesco.minervini, oscar.palomar, osman.unsal, enrico.reggiani, josue.quiroga, joan.marimon, carlos.rojas, roger.figueras, abraham.ruiz, alberto.gonzalez, jonnatan.mendoza, ivan.vargas, cesar.hernandez, joan.cabre, lina.khoirunisya, mustapha.bouhali, julian.pavon, francesc.moll, mauro.olivieri, mateo.valero, adrian.cristal}@bsc.es; M. Kovac, M. Kovac, and L. Dragic, University of Zagreb, FER, Unska 3, 10000, Zagreb, Croatia; emails: {mario.kovac, mate.kovac, leon.dragic}@fer.hr.



1544-3566/2023/02-ART28
https://doi.org/10.1145/3575861

CCS Concepts: • **Computer systems organization → Single instruction, multiple data;** • **Hardware → Hardware accelerators**;

Additional Key Words and Phrases: RISC-V, vector accelerator, SIMD, HPC

**ACM Reference format:**
Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, César Hernandez, Joan Cabre, Lina Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. 2023. Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications. *ACM Trans. Arch. Code Optim.* 20, 2, Article 28 (February 2023), 25 pages.
https://doi.org/10.1145/3575861

## 1 INTRODUCTION

The Covid-19 pandemic remarked the importance of scientific research. The heavy amount of computation needed to characterize the SARS-CoV-2 virus' genome [33] proves that there is a tangible need for investing in **High Performance Computing (HPC)** technologies to fit the computation requirements of the "race to Exascale" [18]. Generally speaking, Exascale computing refers to the capability of a machine to execute at least $10^{18}$ operations per second [16]. Among the commitments with these objectives [14, 16, 17, 22], the **European Processor Initiative (EPI)** aims to create a sustainable hardware/software ecosystem that could sign the independence of Europe on computing systems [15]. Nonetheless, the challenge to build Exascale machines within a 20-MW power envelope has led to a focus away from peak performance to energy-efficient performance. For instance, the 59th edition of the TOP500 list [44] revealed the Frontier system at the Oak Ridge National Laboratory (ORNL) to be the first true Exascale machine, yet ranking in the second position of the Green500 list [19]. This shows that energy efficiency is becoming a top priority for High Performance Computing (HPC) facilities [1, 20, 26]. The renewed interest in vector architectures due to their characteristic of efficiently exploiting **Data-Level Parallelism (DLP)** perfectly fits with the requirements of the Exascale challenges.

Historically, vector processing has always been associated with supercomputing. The golden era of vector processors started with the introduction of the CRAY-1 [35] in 1976, which broke up with the memory-to-memory philosophy of precedent machines like TI-ASC [45] and STAR-100 [7], instead introducing a **Vector Register File (VRF)** and interconnect to allow data movement between the functional units and the vector registers [13]. Vector machines dominated the supercomputing market for about 15 years, when they were extirpated by parallel machines based on multiple out-of-order microprocessors, as the advances in CMOS VLSI technology allowed more transistors to fit on a die. Although multicore architectures represent a valid approach to data-parallel problems, they still have efficiency issues due to their high instruction fetch and decode overheads. The renaissance of vector processing is a direct consequence of the slowdown of Moore's law and the limitations on energy efficiency imposed by the physics of CMOS circuit scaling [9, 11].

Vector processors operate on arrays of data, where a single datum of the array is referred to as a vector element [10]. A dedicated **Instruction Set Architecture (ISA)** defines the vector architectural parameters, such as the number of vector registers and the **Maximum Vector Length (MVL)**. Particular features like reductions use common arithmetic operations to *reduce* a vector register to a scalar value. They are also characterized by unique memory operations like *strided* loads and stores, where the stride defines the increment, expressed in bytes, of memory locations

marking the beginnings of new vector elements, and *gather-scatter* operations, which locate vector elements by accessing memory through a set of indices, represented by elements of another vector. When compared to **Single Instruction Multiple Data (SIMD)** architectures, vector processors offer a higher level of abstraction. Single Instruction Multiple Data (SIMD) architectures, like the ARM Neon [32] or the Intel AVX-512 [8], are characterized by the fact that more elements are packed in the same register, which can be computed by the available functional units. To exploit Data Level Parallelism (DLP), the software needs to know how many functional units, also called *SIMD lanes*, are available to produce effective code. Additionally, the maximum number of elements that can be processed in parallel is limited by the size of the registers. Any attempt to increase the size of the registers and/or the number of functional units implies the introduction of new dedicated instructions, reducing the portability of the Instruction Set Architecture (ISA). Ottavi et al. [29] solve this limitation by encapsulating the number of elements to process in the instruction encoding and controlling it through a **Control and Status Register (CSR)**. Although this solution is feasible for specific Machine Learning (ML) workloads, the number of maximum elements within one operation is still limited by the size of the scalar registers. If the size of the scalar registers increases, new combinations of mixed-width operations are possible, and the ISA needs to be modified at least to specify the new setting of the Control and Status Register (CSR) that holds the SIMD width. On the contrary, vector ISAs are agnostic of the number of available functional units, and the amount of elements to be processed is only limited by the defined Maximum Vector Length (MVL). Advances in ISA offer vector architectures the opportunity to expand beyond HPC to other market segments such as Digital Signal Processing (DSP) and multimedia applications. Examples of it are the vector extensions for NEC [27], the ARM's **Scalable Vector Extension (SVE)** [42], and the RISC-V vector extension (RVV) [34]. The latter is currently gaining importance both in the academic and the industrial world [24]. RVV declares two implementation-specific parameters [34]. The maximum size in bits of a vector element (ELEN), with ELEN $\geq 8$; the number of bits in a single vector register (VLEN). Additionally, it includes CSRs that can be modified through specific instructions to change the operational vector length, *vl*, the Selected Element Width (SEW), and the vector register group multiplier (LMUL), which defines the number of vector registers to form a wider vector register group.

In this context, this article presents Vitruvius+, a RISC-V decoupled **Vector Processing Unit (VPU)** that represents the core of vector instruction execution in the HPC challenge that comes within the EuroHPC initiative. Our Vector Processing Unit (VPU) is based on RVV-0.7.1 and targets HPC applications using long vectors. Accordingly, the MVL is 256 **Double Precision (DP)**-elements, or 16,384 bits. By setting LMUL = 8, Vitruvius+ can achieve an upper bound MVL of 2,048 Double Precision (DP)-elements. To the best of our knowledge, this is the longest hardware vector length produced by a vector architecture. Vitruvius+ features an efficient hybrid in-order/out-of-order architecture boosted by vector register renaming, vector memory-to-arithmetic operation chaining, dedicated support for reductions, and reconfiguration of the inter-lane interconnect. Vitruvius+ is also the first VPU supporting the **Open Vector Interface (OVI)** [37] standard. It has been successfully taped out using GlobalFoundries 22FDX (GF22FDX) as part of the European Processor Initiative (EPI) project. On a stand-alone synthesis, it reaches a maximum frequency of 1.4 GHz in typical conditions (TT/0.80V/25°C).

The article is organized as follows. Section 2 presents the state of the art and illustrates the baseline architecture. Section 3 describes the vector microarchitecture in detail. Section 4 presents the outstanding features implemented in Vitruvius+. Section 5 presents the approach we follow for our evaluations. In Section 6, we report the results of our experiments. Finally, Section 7 reports future plans and concludes the article.

## 2 BACKGROUND AND BASELINE ARCHITECTURE

### 2.1 State of the Art

The resurgence of vector processors is proved by several recent works delivered by both academic and industrial organizations. Hwacha [21] is a single-lane decoupled vector accelerator that implements vector instructions as a custom extension. Several Hwacha versions have been disclosed, with the work of Colin et al. [6] drastically improving the energy efficiency. Ara [23] implements a subset of RVV-0.5 and was taped out featuring an MVL of 256 DP-elements. Arrow [4] targets a **Field-Programmable Gate Array (FPGA)** implementation and supports a subset of RVV-0.9. Similarly, Vicuna [30] was also designed for Field Programmable Gate Array (FPGA) and implements RVV-0.10. RISC-V vector processors have also been released by industrial entities. The Alibaba T-Head Xuantie910 [5] is a multi-core 12-stage out-of-order processor that supports RVV-0.7.1 with a variable number of vector pipelines each operating on 128-bit vector registers. Andes' NX27V [2] is the first vector processor to implement RVV-1.0, and the VLEN can be configured from 128 to 512 bits. SiFive's X280 [39] and P270 [40] also implement RVV-1.0, with 512-bit and 256-bit supported VLEN, respectively. Finally, among the non-RISC-V VPUs, the NEC SX-Aurora **Vector Engine (VE)** [27] and the SVE Vector Engine (VE) of A64FX from Fujitsu [28] are the most popular.

### 2.2 Baseline Architecture

Vitruvius+ is the next generation of Vitruvius,[2] the VPU of the first phase tapeout of EPI [43]. Therefore, Vitruvius is the baseline architecture that Vitruvius+ extends upon. The main design challenges are the following:

- Interface with the scalar core
- Definition of the MVL, in particular, to justify the long-vector-oriented design
- Implementation of the Vector Register File (VRF), to support long vectors and vector register renaming
- Out-of-order execution of vector operations
- Definition of the lane interconnect.

The following sections present the state space exploration of the aforementioned design features.

*2.2.1 Interface with the Scalar Core.* In EPI, Vitruvius is coupled to the Semidynamics[3] Avispado scalar core [38]. They communicate through the Open Vector Interface (OVI) standard [37]. Figure 1 shows the transaction groups composing OVI. One of the characteristics of OVI is that the information is transmitted through a credit-based system. In this context, a credit represents an available resource, like a FIFO queue entry, to allow the transmission of certain information. For example, when an instruction is granted execution, Vitruvius frees a slot in its instruction queue, and a credit is returned to Avispado to notify it that Vitruvius is ready to receive a new vector instruction. In the following, we briefly describe the main components, whereas more details can be found elsewhere [37]:

- *ISSUE*: Vitruvius receives instructions from the scalar core on the signal *inst*, together with the 64-bit scalar operand *scalar_opnd*, the instruction identifier *sb_id*, and a *valid*. Whenever

[2]To the best of our knowledge, details of the Vitruvius microarchitecture have never been published. We presented an overview in one of the RISC-V events with no proceedings.
[3]https://semidynamics.com/.

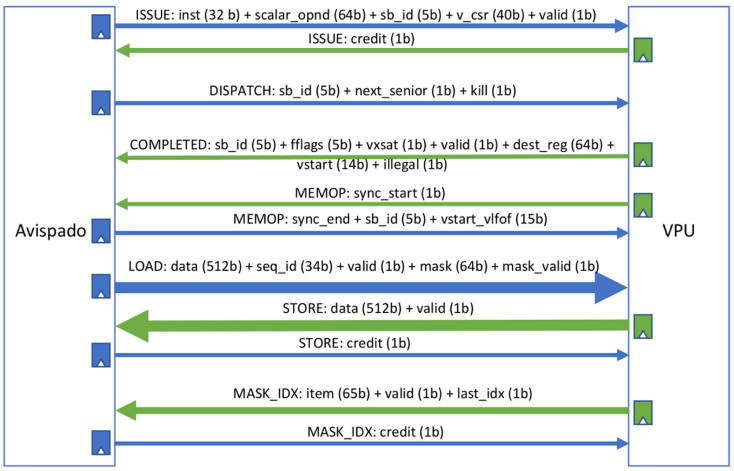



Fig. 1. OVI overview. Retrieved from Semidynamics [37].

Vitruvius consumes the instructions, it returns a *credit* to get ready for receiving a new instruction from Avispado.

- *DISPATCH*: This is used for marking an instruction as non-speculative (it is *next_senior*), or to *kill*.
- *COMPLETED*: Upon completing an instruction, it is marked as *valid*, together with the eventually generated floating-point exception flags *fflags*, the eventual saturation bit for fixed-point operations *vxsat*, the eventual scalar result *dest_reg*, the *vstart* corresponding to the last valid vector element processed by a vector load, and eventually the *illegal* bit if the instruction was decoded as illegal.
- *MEMOP*: In OVI, it is the scalar core that generates the memory requests to execute vector memory operations. Unlike the work of Espasa and Valero [12], Vitruvius does not have any access to the memory hierarchy. The *sync_start* is set by Vitruvius to allow the start of memory requests. Upon completing the requests, the *sync_end* is set together with the memory instruction identifier *sb_id*, and the eventual vector element whose related memory request caused an exception, like a page fault, provided on *vstart_vlfof*.
- *LOAD*: On executing a vector load, Avispado sends the data in the shape of a whole cache line (512 bits),[4] provided on *data* and flagged by *valid*, together with metadata provided on *seq_id*, to locate vector elements in the cache line. On a vector masked load, the *mask* is applied if *mask_valid* is set.
- *STORE*: On executing a vector store, Vitruvius sends the 512-bit *data* flagged by a *valid* only if there are credits available. By setting *credit*, Avispado allows Vitruvius to send new data.
- *MASK_IDX*: This is used in case of masked and/or indexed memory operations. The *item* can represent 64 mask bits, a 64-bit index, or a mask bit placed in the most significant bit with the others carrying a 64-bit index, for masked, indexed, and masked-indexed memory operations, respectively. Indexed memory operations set the *last_idx* for the last index to send.

*2.2.2 Long-Vector Design.* Hardware-supported vector sizes depend on the characteristics of the target applications. For example, short vectors are common in stencil and graph processing

[4]Avispado is not open source, so we do not know the exact size of the cache, although Semidynamics [38] reports values between 8 and 32 kB.

kernels, whereas HPC, physical simulation, and financial analysis applications feature long vectors. Therefore, the applications can affect the decision on the MVL to support. We analyzed the target applications of the EPI project and designed an architecture that supports long vectors, with an MVL of 256 64-bit elements. Using long vectors is beneficial for the following reasons:

- Hide memory latency by combining spatially parallel with temporally parallel execution
- Improve efficiency by avoiding fetches, especially for codes with many loops
- Reduce code size and dynamic instruction count.

Vitruvius is intended to accelerate HPC applications showing high DLP. We quantified "long vectors" based on the study reported by Ramírez et al. [31]. It shows that for applications featuring regular DLP, when using long vectors, the number of total instructions dramatically drops, not only because of the many scalar instructions being replaced by a single vector instruction but also due to the reduced loop counts and control instructions. Another important aspect is the start-up time, which represents the latency in clock cycles to fill the vector execution pipeline. The start-up time is mainly determined by the execution latency of the vector functional units, and the design of the VRF. For long vectors, the initial start-up time can be amortized over the several cycles of execution for the operations, whereas for short-vector implementations, which typically complete operations in less than a dozen cycles, the start-up time can drastically decrease performance. For these reasons, we designed Vitruvius to support long vectors, where each vector register has an MVL of 256 64-bit floating-point elements, like SX-Aurora VE [36] and Ara [23].

*2.2.3 VRF.* The VRF design and the prior microarchitecture state space exploration was driven by the following high-level goals:

- Support for long vectors, where each vector register can hold up to 256 DP-elements, as explained in Section 2.2.2
- Renaming capabilities, allowing for lightweight out-of-order execution mechanism
- The technology node of the EPI project: GF22FDX, targeting a nominal frequency of 1 GHz.

To allow a more aggressive scalar-vector decoupling empowered by lightweight out-of-order execution eliminating **Write After Write (WAW)** and **Write After Read (WAR)** vector dependencies, we designed the VRF with 40 physical registers to support vector register renaming. Therefore, the whole VRF size, expressed in bytes, is given by

$$Tot\_Bytes\_VRF = Num\_VRegs \cdot VReg\_Max\_Elements \cdot Element\_Bytes,$$

where *Num_VRegs* indicates the 40 physical vector registers, *VReg_Max_Elements* represents the MVL of 256 DP-elements, and *Elements_Bytes* is the size in bytes of each vector element. With these design parameters, the size of the VRF amounts to 80 kB. We organized Vitruvius in a lane-based fashion, where each lane contains a slice of the VRF and functional units. The eight-lane configuration splits the VRF into 10-kB slices, one such slice per lane. Therefore, we explored the available memory instances included in the GF22FDX register file portfolio and conducted an experimental study on the type of memory to use. We generated different configurations using the foundry-compatible memory compilers and synthesized them for a clock period of 800 ps. Table 1 summarizes the results. It shows that multi-ported register file configurations are not desirable either because of the large area overhead or because of the timing violations and the high estimated power consumption. Therefore, we opted for a VRF configuration that instantiates five 2-kB 1RW SRAM banks in each lane. Beside the lower area overhead, this design choice is also driven by another observation. To achieve a peak performance of one DP-**Fused Multiply-Add (FMA)** and one DP-memory-access per cycle, the lane local **Finite State Machine (FSM)** orchestrates the

Table 1. Early State Space Exploration Results for Different VRF Configurations

| **Clock Period:** 800 ps (1.25 GHz) | | | | | |
|---|---|---|---|---|---|
| **Configuration** | **Ports** | **Cell Type** | **Area ($\mu m^2$)** | **Slack SS[a]/TT[b] (ps)** | **Power (mW)** |
| 5 × 2 kB | 1RW | SRAM | 21,005.726 | 0/193 | 25.2674 |
| 1 × 10 kB | 3R2W | Latch | 338,725.864 | 0/0 | 119.572 |
| 1 × 8 kB | 3R2W | Latch | 273,288.375 | 0/0 | 107.601 |
| 1 × 10 kB | 1R2W | Latch | 239,603.614 | 0/0 | 98.6253 |
| 1 × 8 kB | 1R2W | Latch | 184,466.954 | 0/0 | 78.5065 |
| 5 × 2 kB | 1R1W | SRAM | 35,763.728 | −429/−68 | 32.6354 |

[a]Slow corner conditions (SS/0.72V/125°C).
[b]Typical corner conditions (TT/0.80V/25°C).
*Note:* 1RW = shared read/write port; 1R1W = one read port and one write port; 1R2W = one read port and two write ports; 3R2W = three read ports and two write ports.

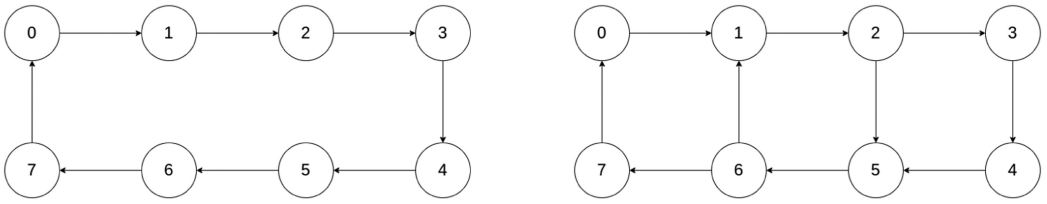
vector instruction execution over five states, three of which are for reading the source operands of an Fused Multiply-Add (FMA), one for the memory access, and another for the arithmetic write-back. This means that when the pipeline operates at full speed, the functional units produce five results in five cycles, hence the minimum number of banks to support write-back is 5. This way, the VRF can sustain the target throughput yet with the lowest impact on area.

*2.2.4 Out-of-Order Execution.* Vitruvius schedules the execution using an out-of-order execution mechanism. First, the instructions pass through the renaming stage that eliminates the Write After Write (WAW) and Write After Read (WAR) dependencies. The renamed instructions are then split into two concurrent streams by placing them either in the memory or the arithmetic instruction queue. This equips Vitruvius with lightweight out-of-order execution capabilities. As an example, imagine a sequence of vector-vector add operations, *vadd.vv*, followed by a sequence of vector strided loads, *vlse.v*. The renaming unit eliminates every WAW dependency in the instruction flow. The sequence of *vadd.vv* is placed in the arithmetic queue, whereas the loads go to the memory queue. This allows for the first vector load to overlap with the execution of the first *vadd.vv*, thus reducing the time for completion.

*2.2.5 Lane Interconnect.* The modular VPU design features independent vector lanes, which need to exchange operand data for certain RVV instructions such as reductions or permutations. Therefore, a lane interconnect needs to be implemented. The lane interconnect design and the prior microarchitecture state space exploration was driven by the following goals:

- High level of determinism, so that it is possible to know exactly the latency of a packet in the lane interconnect and schedule new transfers accordingly (note that this requires a contention-free network even at peak load)
- Ease of routing and simplified flow control scheme
- Low power and area impact while being able to support the most common transfers with an acceptable throughput.

Therefore, considering that we target up to eight vector lanes, we selected a ring topology as the lane interconnect. This well-known topology has the property of being completely deterministic, meaning that the latency of a packet traveling in the network can be calculated up front and relies only on the distance between the sender and the receiver. There is no packet deflection in this network. Therefore, it does not need any complex routing algorithm nor a centralized controller. A router in this network can either accept the incoming data or let it pass to the direct neighboring router. For these reasons, we preferred a ring topology over other structures, such as the popular




(a) Ring topology (b) Mesh topology

Fig. 2. Configuration of the evaluated lane interconnect topologies.

mesh topology. A mesh has typically more links than a ring, thus reducing the maximum distance between nodes in the network. However, more links contributes to higher area and power consumption, due to the internal routers now having three ports. Within the set of applications we target, listed in Section 6, some of the most common RISC-V instructions that need inter-lane communication are *vslideup* and *vslidedown*, which move elements in a vector register up and down by a specified offset, respectively. For example, when executing a *vslide1up*, element 0 ends up in the position of element 1, element 1 ends up in the position of element 2, and so on. In other words, lane 0 sends elements to lane 1, lane 1 sends elements to lane 2, and so on. To justify our choice of using a ring interconnect, we developed an in-house cycle-accurate high-level simulator modeling a mesh and a ring. We modeled the mesh as a $2 \times 4$ configuration for the eight lanes in Vitruvius, using the common XY routing algorithm. The data movement happens in only one direction in both configurations. Figure 2 represents the analyzed configurations. In this simple model, the intermediate nodes of the mesh always try to use the shortest path to reach the destination, if available. Both networks satisfy the following criteria:

- There are no buffers in the network. Buffers are one of the most power- and area-consuming elements in a network, and their inclusion conflicts with the requirement of low power and area impact.
- There is only one physical link connecting a node to another, with reconfiguration capabilities. This means that the data movement is dynamically set according to the type of instruction to execute and the related offset, to reach the destination with fewer cycles. For example, executing a *vslideup* with offset 5 in the clockwise direction, with an eight-lane configuration, is the same as executing a *vslidedown* with offset 3 in the counterclockwise direction.[5] This reduces the cases to study to offset values between 1 and 4.

We simulated both configurations using patterns for the *vslideup* operation. Table 2 reports the results of the simulations for the different offsets, assuming vectors of 256 elements. Results show that there is no clear advantage in using a mesh interconnect for the evaluated traffic patterns. In particular, it can be noted that the mesh performs slightly worse than the ring for offset values 3 and 4. This is due to the fact that in this analysis the mesh always tries to use the shortest path for the data transfer. For these cases, the usage of the intermediate direct links causes other nodes to delay the packet injection which finally increases the total latency. A disclaimer that this analysis is not suggesting that a ring topology is better than a mesh. This study shows that the design of the inter-lane interconnect depends on the target traffic patterns. For instance, the mesh is actually a ring with the intermediate connections, and by using the same links as for the ring makes the throughput the same for both configurations. However, this confirms that there is no usage of

[5]In this case, lane 0 targets lane 5 as the final destination of its packets. Therefore, by moving data in the counterclockwise direction, the ring takes three cycles to complete the transfer.

Table 2. Performance Comparison Between the Ring and the Mesh Lane Interconnect for the Execution of *vslideup* with Different Offset Values

| Operational $vl$ = 256 | | | | |
|---|---|---|---|---|
| **Configuration** | **Offset** | **Packets** | **Cycles** | **Throughput (DP-element/cycle)** |
| Ring | 1 | 255 | 33 | 7.76 |
| | 2 | 254 | 65 | 3.94 |
| | 3 | 253 | 97 | 2.64 |
| | 4 | 252 | 129 | 1.98 |
| Mesh | 1 | 255 | 33 | 7.76 |
| | 2 | 254 | 65 | 3.94 |
| | 3 | 253 | 99 | 2.59 |
| | 4 | 252 | 131 | 1.95 |

*Note:* The column *Packets* represents the number of elements to transfer and is obtained by subtracting the offset from the *vl*.

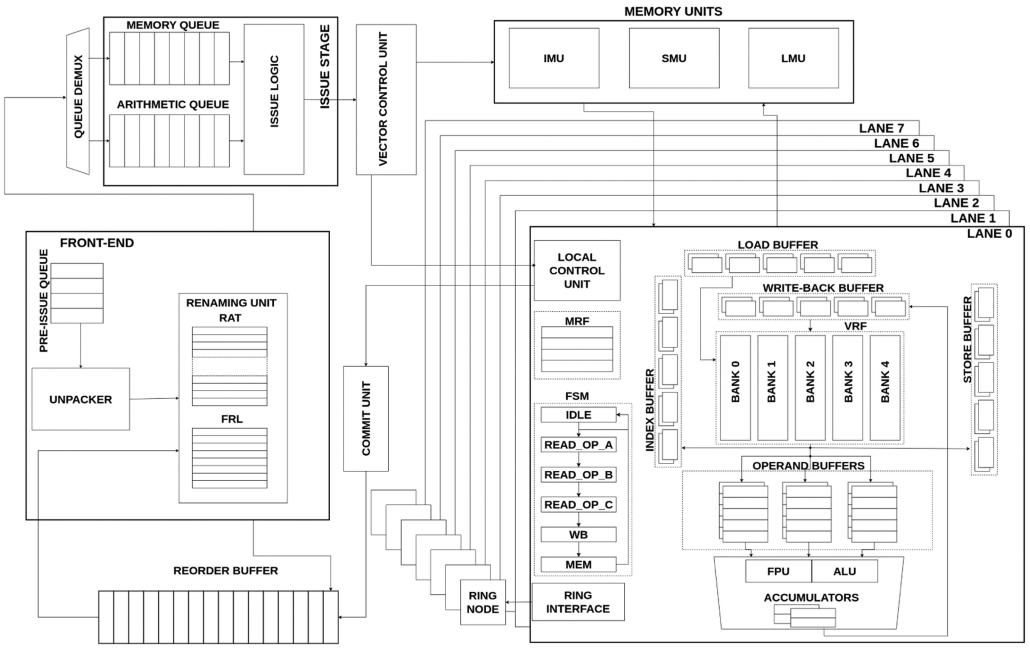



Fig. 3. Vitruvius+ top-level block diagram for the eight-lane configuration.

the intermediate links for the target traffic. The ring is more area- and power-efficient than the mesh because of the easier control flow to route the packets and the absence of the intermediate connections.

## 3 MICROARCHITECTURE

In this section, we describe the microarchitecture of our VPU, giving details on the most important blocks. A high-level block diagram is shown in Figure 3.

### 3.1 Front-End

The front-end processes the instructions received from the scalar core. It includes the *pre-issue queue*, which uses a credit-based system to get new instructions from OVI. The *unpacker* classifies

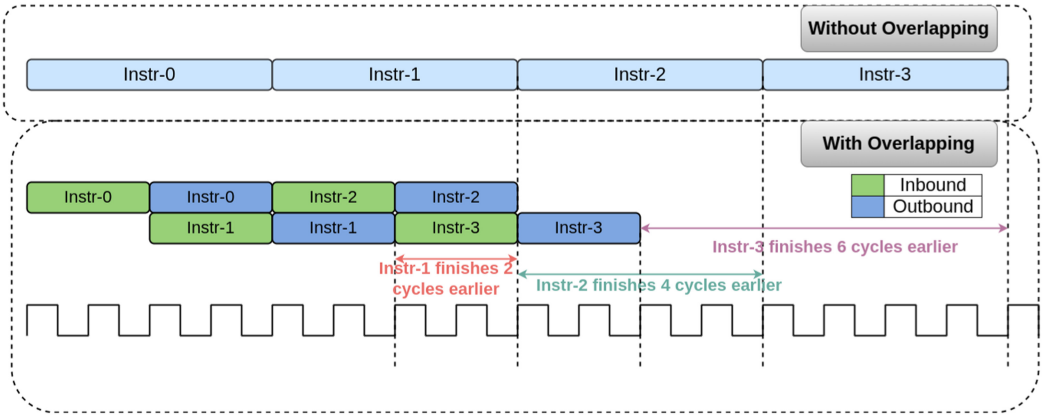



Fig. 4. Overlapping of vector instructions example.

the instruction according to its type and performs a fine-grain decoding, providing on the output a vector of control bits used by the lanes to understand what kind of operation to apply. The *renaming unit*, already mentioned in Section 2.2.4, resolves the WAR and WAW dependencies, empowering the lightweight out-of-order execution. It holds the **Register Alias Table (RAT)**, which maps each logical register to the last assigned physical register, and the **Free Register List (FRL)**, which keeps track of the available physical registers. The *queue demultiplexer* splits the memory and the arithmetic instruction streams onto different queues, enabling parallelism.

### 3.2 Issue Stage

Instructions from the queue demultiplexer are split and stored in the issue FIFO queues according to their type. The number of queue entries is parameterized. Control logic in the issue stage continuously checks the availability of the resources to be used by the instructions at the head of the queues. This block analyzes the decoded information of arithmetic operations to enable back-to-back execution. This feature, which we term *overlapping*, allows for opportunistically starting a new instruction before the previous one has finished, to fully exploit the vector arithmetic pipeline. This mechanism is shown in Figure 4. We name *inbound* an instruction at the execution stage that is reading the vector operands from the VRF. As soon as all the vector source elements have been read, the instruction becomes *outbound*. The overlapping enabled by the issue stage control logic allows, at this point, another instruction to enter the execution phase by making it *inbound*, so as to start reading the source operands. By the time the first instruction exits the *outbound* phase, which happens when it has written back all the results, the second one may have already generated the first results. Therefore, the effect of overlapping is that the second instruction advances faster in writing back its results. Figure 4 depicts this scenario, showing the reduction in the number of cycles to complete the instructions when overlapping is enabled. Although in a certain way overlapping resembles the traditional instruction pipelining, it actually enables different lanes to start the execution of an instruction earlier, in case the operational *vl* is higher than the one of the previous operation. This corresponds to concurrent execution of instructions in different lanes.

### 3.3 Memory Units

Vitruvius+ supports vector memory operations through a dedicated set of units. These units use the OVI signals while also communicating with the vector lanes. As explained in Section 2.2.1, Vitruvius+ does not have access to the memory hierarchy, and it is the scalar core that performs

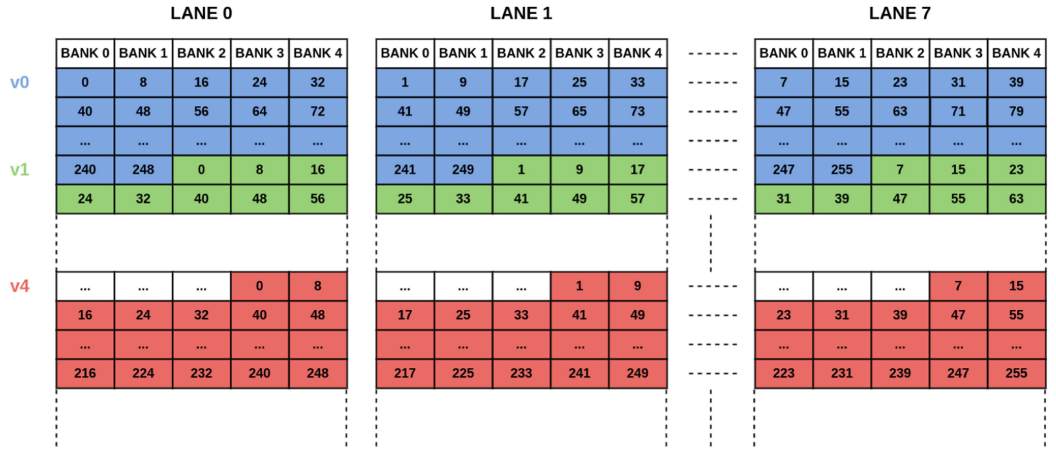



Fig. 5. VRF 64-bit element mapping showing the interleaved distribution.

memory accesses for vector memory operations. The **Load Management Unit (LMU)** processes data coming from OVI when executing a vector load. Data reaches the vector unit at the granularity of a cache line, as explained in Section 2.2.1. The **Store Management Unit (SMU)** interacts with the scalar core through the credit system described in Section 2.2.1. Finally, the **Item Management Unit (IMU)** is in charge of managing the transactions for the *MASK_IDX* bus of OVI in Figure 1, thus it is involved in the execution of masked and/or indexed vector memory operations.

### 3.4 Vector Lane

*3.4.1 VRF.* Each lane holds a slice of the VRF, with the slices being composed of five single-port 2-kB SRAM banks, as explained in Section 2.2.3. The vector registers are organized in an interleaved fashion, as shown in Figure 5. In other words, the first eight bytes are always in lane 0, the next eight bytes are always in lane 1, and so on. The distribution of the vector elements in the VRF is a microarchitecture decision. The interleaving has some benefits. When executing a stride-1 vector load, each lane can receive exactly one DP-element per cycle from the input 512-bit cache line. This guarantees that no lane gets starved from not receiving elements in case there is an arithmetic instruction waiting for data to arrive from a vector load. The VRF banks can be accessed independently, but, on a read, a full set of five elements, one per bank, will be tentatively gathered. In RVV, the mask register is implicitly vector register *v0*. However, the mask registers are kept separated from the VRF, to avoid possible bank conflicts when executing predicated operations.

*3.4.2 Finite State Machine (FSM).* Each lane features an FSM in charge of managing the reads and writes from/to the VRF. It is based on a five-state structure (plus an idle state to avoid unnecessary state transitions and contributing to energy efficiency). The FSM leaves the idle state upon having received a start signal from the local control unit and/or having detected that some memory operation is executing. Then, the FSM cycles over the five active stages. Elements read from the VRF are buffered and then provided to the functional units. Since each read provides five 64-bit values, and the FSM state repeats every five cycles, arithmetic operations can fully exploit the functional unit pipelines, providing a 64-bit result per cycle, after paying the initial start-up time. However, as presented in Section 2.2.2, the start-up time is amortized over the many operations executed by managing long vectors. Moreover, the FSM capability of addressing each bank independently is leveraged to write to the VRF combining data from different in-flight vector loads.

*3.4.3 Execution Wrapper.* The source buffers receive vector elements from the VRF in preparation toward the functional units. Each lane features a **Floating-Point Unit (FPU)** developed by the University of Zagreb,[6] that performs floating-point operations, and an **Arithmetic Logic Unit (ALU)** that manages integer and fixed-point computations. All the units in the vector lanes are fully pipelined (except square root and division) and have a throughput of 64-bit/cycle, working in SIMD fashion when the element width is less than 64 bits. The Floating Point Unit (FPU) supports all classes of operations, including FMA, division, square root, comparison, and other types like widening and narrowing operations. Integer operations executed in the Arithmetic Logic Unit (ALU) also range from the most common ones, like FMA, multiplication, addition, and bit manipulation, to others like narrowing, widening, and fixed point. Additionally, our vector unit supports all type of vector reduction operations specified by the RVV specifications.

### 3.5 Ring Interconnect

The class of *permutation instructions* in the RISC-V V-extension involves vector elements shuffling and manipulation. Due to the mapping presented in Figure 5, the lanes need a medium to transfer/receive data to/from other lanes. As discussed in Section 2.2.5, a unidirectional ring topology is used to interconnect the lanes. It is designed to have a single-cycle latency for one-hop transfers (i.e., to transfer data from one lane to its direct neighbor). Reduction operations can benefit from leveraging this type of interconnect for transferring partial results to the neighboring lanes. Additionally, the ring interconnect executes the slide operations, which move elements of a vector register given an offset, and vector register gather operations, which use one vector register as a set of indices and realize any permutation of the vector register used as source of data. The maximum injection rate of the ring interconnect is eight DP-elements per cycle.

### 3.6 Reorder Buffer

As the execution of arithmetic and memory instructions can complete in an out-of-order way, a **Reorder Buffer (ROB)** is used to keep the instruction commit order. While receiving information about any in-flight instruction, only one instruction per cycle is marked as completed on the *COMPLETED* bus of OVI in Figure 1. If any exception occurs, the Reorder Buffer (ROB) notifies the scalar core about this event through specific signals of the interface, and internally triggers the roll-back process in case the vector unit needs to go back to a previous safe state. While doing so, the ROB disables any possibility for new instructions to proceed in the front-end until the roll-back phase is finalized.

## 4 VITRUVIUS+ OUTSTANDING FEATURES

In the following, we describe the outstanding features that Vitruvius+ implements, which distinguish it from the majority of state-of-the-art solutions:

- Implements memory-to-arithmetic vector instruction *out-of-order chaining*
- Optimizes the execution of vector-vector move operations, by introducing what we call the *fast move* operations
- Features reconfiguration capabilities in the inter-lane ring interconnect
- Introduces dedicated support for accelerating the execution of vector reduction operations.

The impact of these new implemented features is highlighted in Section 6, whereas an extensive description of their usage is provided in the next paragraphs.

[6] https://www.fer.unizg.hr/en.

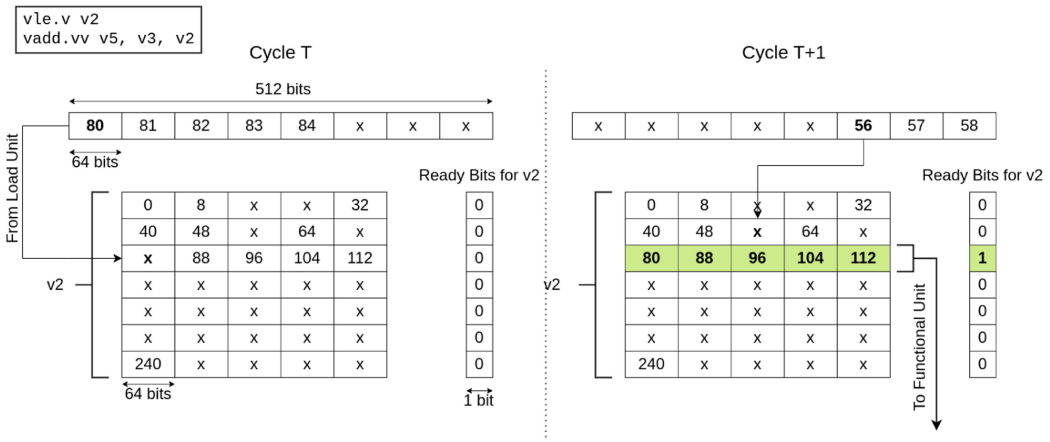



Fig. 6. Detail of the out-of-order chaining mechanism. By keeping track of the availability of the element groups, the operations can generate results in an out-of-order manner.

### 4.1 Vector Out-of-Order Chaining

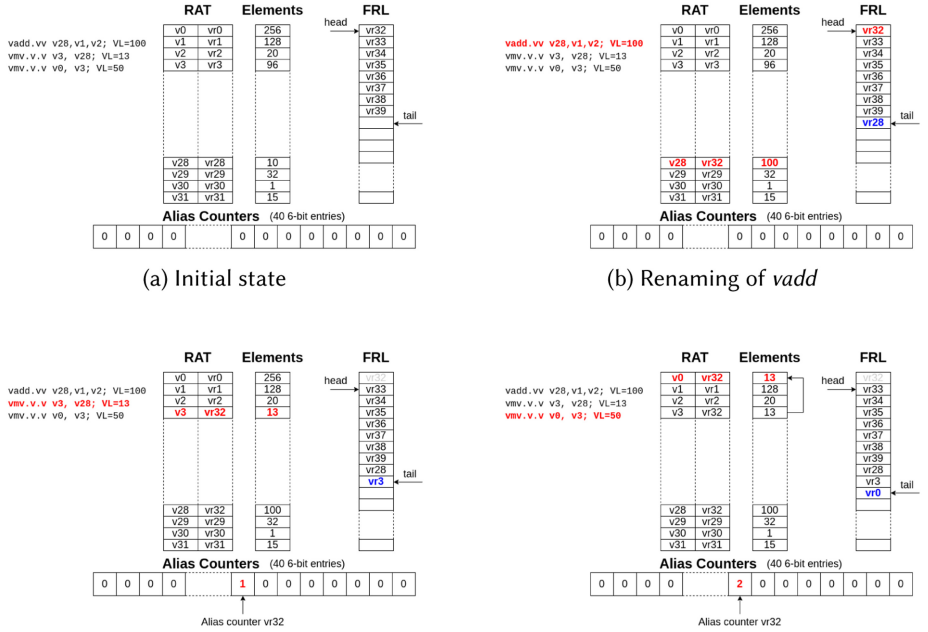
A common features of vector processors is *chaining*. Chaining forwards the results of a vector operation to another operation that uses them as source operands. Practically, it executes a bypass of the vector elements produced as results of a functional unit to another functional unit that uses them as input for its operation. If we consider arithmetic-to-arithmetic chaining, Vitruvius+ does not benefit from it, because only one arithmetic instruction at a time can run in a lane. The type of vector chaining that is beneficial to our VPU is the memory-to-arithmetic chaining. As explained in Section 2.2.1, OVI specifies that the scalar core accesses memory on behalf of a vector load operation. This makes the order of arrival of the vector elements unpredictable. Referring to the VRF organization in Figure 5, it is possible, for example, that lane 0 receives the element group 40–72 before it receives the group 0–32. To handle this case, we implement an optimization to keep track of the availability of the element groups and start the dependent arithmetic operation if at least a group is ready. The availability of the vector element groups is controlled by a specific structure inside the lanes, which we call the *ready bits* table. This structure is composed of single bit per group of elements for each vector register. The implementation of this mechanism, which we call *vector out-of-order chaining*, helps overcoming the limitation of the OVI standard to disallow to serve the memory requests on the VPU side. Figure 6 explains through an example how the out-of-order chaining works. A *vadd.vv* uses *v2* as one of its source vector operands, and *v2* is also the destination of a previous *vle.v*. Vector register *v2* is represented with some elements already written in the VRF, and some not ready elements marked with the undefined value *x*. The ready bits for *v2* are also depicted. For simplicity, we show only the VRF slice of lane 0, therefore the element mapping presented in Section 3.4.1. At cycle *T*, a new 512-bit cache line is received with five valid elements, with element 80 belonging to lane 0. This element completes the third element group from the top in the VRF, and the corresponding ready bit is set to 1. At cycle *T+1*, this element group is read from the VRF and sent to the functional unit to start the *vadd.vv*. Note that, as discussed, this element group is computed before groups 1 and 2, thus allowing for out-of-order write-back of the results.

### 4.2 Fast Moves

In RVV, the vector-vector move is encoded as *vmv.v.v vd, vs1*, and copies values of *vs1* into *vd*, up to the current *vl*. We made a preliminary analysis on a subset of the target applications where

Table 3. Percentage of Vector-Vector Moves over the Total Arithmetic Instructions for Some of the Target Benchmarks

| Benchmark | Total Arithmetic Instructions | Total vmv.v.v Instructions | Percentage |
|---|---|---|---|
| Jacobi-2D | 20,402 | 6,163 | 30.2% |
| Streamcluster | 745,236 | 165,608 | 28.6% |
| LavaMD | 37,376 | 2,048 | 5.5% |
| Blackscholes | 672,000 | 9,600 | 1.4% |
| MMUL | 65,793 | 256 | 0.4% |




(a) Initial state

(b) Renaming of *vadd*

(c) Renaming of the first fast move

(d) Renaming of the second fast move

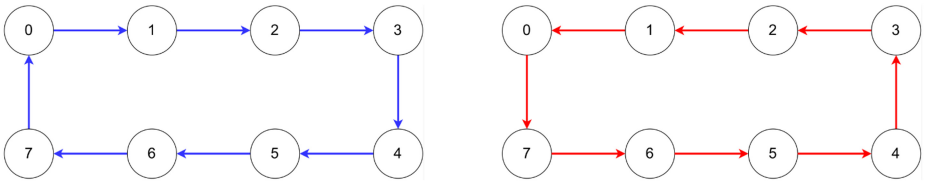
Fig. 7. Fast move mechanism at the renaming stage.

*vmv* instructions are executed. Table 3 shows the number of *vmv* compared to the total number of arithmetic instructions. In *Jacobi-2D* and *Streamcluster*, this type of instruction represents a considerable percentage of the overall arithmetic operations. With no optimizations, the execution of this instruction proceeds in the same way as for other arithmetic instructions: it consumes one physical register due to renaming, and accesses the VRF to read the source vector and to write it into the destination vector. The overall effect is to create a copy of a vector into another. Therefore, we designed an optimization in Vitruvius+ that enables a smart execution of *vmv* instructions, which we call *fast move*. In particular, the instruction is completely resolved at renaming. Figure 7 illustrates the mechanism. To support the optimization, we included two additional structures in the renaming unit. The *element table* refers to the elements assigned to a vector register the last time it was the destination of a vector operation. The 40 *alias counters*, one per physical register, keep track of the number of times the same physical register is allocated to multiple logical registers. When the renaming unit works in the standard mode, associating one physical register to a unique logical register, the alias counter of that physical register is 0. On the contrary, whenever

a fast move is executed, one physical register is associated with multiple logical registers, and the alias counter is increased according to the number of fast moves that rename to that physical register. We assume the initial state is the one represented in Figure 7(a), and the instructions enter the renaming stage in the shown order. In Figure 7(b), the *vadd* renames its destination register *v28* to the physical register *vr32*. The instruction will be issued to the lanes, freeing the old physical register *vr28* when retired. Next, a *vmv* enters the renaming stage as shown in Figure 7(c). This instruction can be executed in the fast mode. In one cycle, it accesses the Register Alias Table (RAT) to read the last physical registers assigned to *v3* and *v28*, which are *vr3* and *vr32*, respectively. In the next cycle, it writes *vr32* to the RAT entry corresponding to *v3*, and updates the alias counter of *vr32* and the new value of assigned elements in the element table. In the next cycle, the instruction is completed, as it does not need to be executed in the vector lanes. From now on, *v3* is mapped to *vr32* with vector length 13. We say *vr32* is now an "alias" of *v3*. Therefore, the fast move optimization reduces the latency of execution of the *vmv* to just three cycles, and reduces power dissipation by avoiding unnecessary accesses to the VRF. Similarly, in Figure 7(d), another fast move is executed. The process is the same as the one described before, with the alias counter of *vr32* increased to 2. However, note that this time the element table entry for *v0* is updated with the elements from *v3*, instead of the *vl* value that comes with the instruction. The lowest number of valid elements between the last assigned to the physical vector register used as alias and *vl* is selected when executing the fast move at renaming. In this way, when accessing *v0*, it will read valid elements only until $vl = 13$. When a fast move is retired from the ROB, the alias counters are decreased. The physical register is written back to the Free Register List (FRL) when the alias counter is 0. If we imagine a long sequence of *vmv*, by executing them in the fast mode we could potentially have all 32 logical registers renamed to the same physical register. This means that only one out of the 40 physical registers is allocated. The other 39 are available for renaming new instructions. In this way, the vector instruction window can be expanded as more instructions enter the vector pipeline.

### 4.3 Switched Ring Reconfiguration

The inter-lane interconnect of our VPU is built over a unidirectional ring topology. This is an extremely low-power and area-efficient interconnect, while providing sufficient support for the data movement operations needed by our applications. As explained in Section 2.2.5, *vslideup* and *vslidedown* are the most common data movement operations in our set of applications. Therefore, Vitruvius+ implements an optimization in the inter-lane ring to enhance the execution of these instructions. In the baseline unidirectional ring topology, like the one shown in Figure 2(a), in the eight-lane configuration, eight packets can be injected into the ring where each hop takes one cycle, until they reach their final destination. Therefore, in the worst case, assuming the data movement occurs in the clockwise direction, a packet reaches its destination after a maximum of seven cycles, which happens when executing a *vslideup* with offset 7, for example, since lane 0 targets lane 7 as the final destination of its packets. By introducing limited reconfiguration capabilities in the ring, the maximum latency in the worst-case scenario is reduced to four cycles, at the cost of just a 2% of area overhead and no timing issues. The reconfigurability of the links depends on the type of instruction to execute and the related offset value. By taking the previous example of the *vslideup* with offset 7, with this optimization the ring recognizes that is more convenient to move data in the counterclockwise direction to reduce the latency of the operation. In this way, packets sent by lane 0 reach lane 7 in only one cycle. Figure 8 presents the reconfiguration of the inter-lane ring to move data in either one or the other direction. In particular, the clockwise direction is selected when executing *vslideup* with offset values whose *mod*8 is between 1 and 3, and *vslidedown* with offset values




(a) Data movement in the clockwise direction

(b) Data movement in the counterclockwise direction

Fig. 8. Inter-lane ring configurations for the data movement.

whose *mod*8 is between 5 and 7. On the contrary, the counterclockwise direction is selected when executing *vslideup* with offset values whose *mod*8 is between 5 and 7, and *vslidedown* with offset values whose *mod*8 is between 1 and 3. The expression *mod*8 is the modulo of the number of lanes.[7] If the result of *mod*8 is 4, the direction of movement is selected by the operation itself—that is, by default, clockwise for *vslideup* and counterclockwise for *vslidedown*.

### 4.4 Vector Reductions Enhancement

RVV-0.7.1 includes vector reduction instructions. These operations take a vector register and a scalar held in element 0 of a second vector register, and perform a reduction using some binary operator. The result is then stored in element 0 of the destination vector register. The ISA distinguishes between *ordered* and *unordered* reduction operations. Ordered vector reductions operate on the element values in order, starting with the scalar value held in element 0 of the second vector operand. Unordered reductions provide some flexibility on how to operate with the vector elements. Vitruvius+ supports all the reduction instructions listed in RVV-0.7.1. The FPU and the ALU share a common structure for the execution of reductions, called the *reduction handler*. The reduction handler splits the execution of reduction operations into two phases:

- *Intra-lane reduction*: This refers to the initial steps to reduce the vector elements locally in each lane, generating intermediate results for the second phase.
- *Inter-lane reduction*: Reduce the partial results from the first phase to the final scalar result to be placed in element 0 of the destination vector register.

Because of the element distribution observed in Section 3.4.1, the ordered reduction operations effectively use only the inter-lane phase, since each value has to be transmitted to the neighbor lane for each step of the computation. Ordered and unordered reductions use the inter-lane ring for the inter-lane phase. However, the flexibility to compute the source vector elements in the unordered reductions allows for optimizing both phases. Assume the operation is a floating-point reduction sum, *vfredsum.vs vd, vs2, vs1*, with the source vector operands being *vs2* and *vs1*, with the latter providing the element 0 as the initial scalar value. The final result is given by $vd[0] = vs1[0] + vs2[0] + vs2[1] + \cdots + vs2[vl-1]$. Being an unordered reduction, the sequence of operations can be optimized. Figure 9 shows the optimization for the intra-lane phase. It depicts a set of $N$ 64-bit accumulators that are used as inputs to the functional unit, with the other input being elements of *vs2*. This structure is placed in all the lanes. We define the array *vsrc* for each lane as $vsrc[0] = vs2[0], vsrc[1] = vs2[8], vsrc[2] = vs2[16], \ldots$ for lane 0, $vsrc[0] = vs2[1], vsrc[1] = vs2[9], vsrc[2] = vs2[17], \ldots$ for lane 1, and so on, according to the element mapping in Figure 5. The first $N$ elements of *vsrc* are directly placed as initialization

[7]The mechanism holds for a generic power of 2 *Num_Lanes*. The maximum latency in ring is therefore *Num_Lanes*/2.

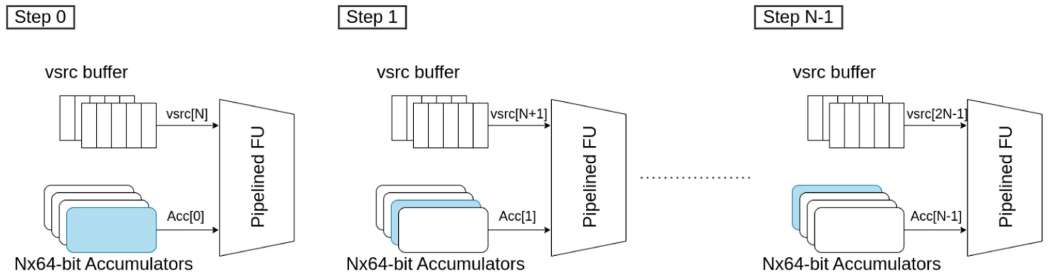



Fig. 9. Usage of the accumulators for the optimization of the intra-lane phase of a vector reduction.

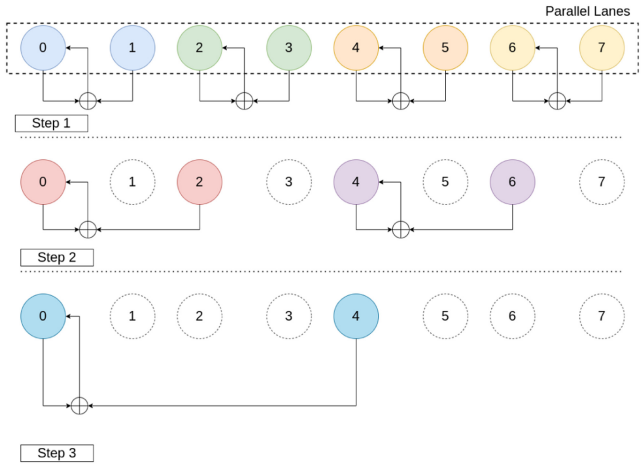



Fig. 10. Tree-like scheduling of the operations for the inter-lane phase of a vector reduction.

value of the accumulators, with the exception of lane 0, where the first accumulator is initialized with $vs1[0] + vsrc[0]$. At step 0, the first pair of operands to enter the pipelined functional unit is $(Acc[0]; vsrc[N])$. At step 1, although this pair is already in the execution pipeline, a new accumulator is selected and a new pair of operands feeds the functional unit. The mechanism keeps selecting a different accumulator in a round-robin fashion. When all the elements of *vsrc* local to each lane are computed, the final step combines the $N$ results in the accumulators to produce a single value. After that, the partial results of the lanes need to be combined to finalize the reduction operation.

To combine the partial results of the lanes, the inter-lane ring is used. A naive implementation would pass the result of lane 0 to lane 1, compute a new partial result, then pass it to lane 2 and repeat the process, until all the partial results are computed to generate the final scalar result. This mechanism is strictly sequential and does not benefit from the parallelism of the multi-lane organization. Therefore, we design an additional optimization for the execution of unordered vector reductions with the objective of parallelizing the computation of the final result. By orchestrating the computation steps between pairs of lanes, the latency of the inter-lane phase can be reduced, as well as the number of data transfers in the inter-lane ring. This is done by scheduling the operations in a *tree-like* fashion, as represented in Figure 10. In step 1, all the lanes are involved in the computation of partial results by operating in pairs. Thus, three computations are parallelized. In step 2, the results generated as the outcome of step 1 are further processed by only two pairs of lanes. Then, step 3 operates on the last two partial results and generates the final scalar result

to be stored in element 0 of the destination vector register, held in lane 0 as per the VRF mapping shown in Section 3.4.1.

## 5 METHODOLOGY

### 5.1 Experimental Setup

Vitruvius+ is fully described at the **Register Transfer Level (RTL)** using the SystemVerilog language. The generated Register Transfer Level (RTL) code passes through RTL simulation for functional verification, synthesis, and physical design. Functional verification, driven by randomly generated binaries and directed tests, is performed through a dedicated Universal Verification Methodology environment. It is based on a co-simulation scenario with the Spike [41] RISC-V ISA simulator as a golden reference for the vector instructions. We use QuestaSim-64 2021.3_2 for the RTL simulation. The tests are executed on both Spike and Vitruvius+. The check with Spike is done on a per-instruction basis. In case of any mismatch, the simulation stops and information of the discrepancies is generated to ease debugging. Then, we synthesized Vitruvius+ targeting GF22FDX using Cadence Genus Synthesis Solution 19.11. The generated netlist is then used for the physical implementation phase with Cadence Innovus 19.11. The netlist is also used in the aforementioned Universal Verification Methodology environment to run our set of vectorized benchmarks, with back-annotation on timing information to estimate the switching activity for each of the selected applications. Then, we used this information in Cadence Joules RTL Power Solution to extract power metrics.

### 5.2 Benchmarks Description

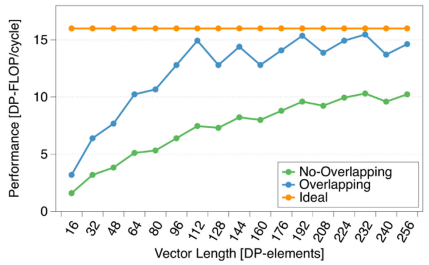
We benchmarked several long-vector compatible kernels from the HPC applications and other domains [31]. Among the HPC applications, *axpy* represents a common kernel in applications based on BLAS (Basic Linear Algebra Subprograms). Because of its low arithmetic intensity (i.e., three memory operations for each arithmetic), it is a typical memory-bound kernel. The second is **Matrix-Matrix Multiplication (MMUL)**, a compute-bound Basic Linear Algebra Subprograms (BLAS) kernel highly used for benchmarking, as it gives insights on the top performance of the system. The **Fast Fourier Transform (FFT)** kernel is an efficient method for computing the discrete Fourier transform of a sequence of complex numbers. From the molecular dynamics domain, we have *LavaMD*. This application calculates the particle potential and relocation due to mutual forces between particles within a large 3D space. Another kernel is *Blackscholes*, which represents the broad field of analytic PDE solvers and their application in computational finance. *Jacobi2D* is part of the PolyBench suite and implements an iterative algorithm for calculating the solutions of a diagonally dominant system of linear equations. *Pathfinder* uses the ghost zone optimization technique to find the shortest path on a 2D grid. Finally, *Streamcluster* solves an online clustering problem by assigning each of the input points to its nearest center.

## 6 EXPERIMENTAL RESULTS

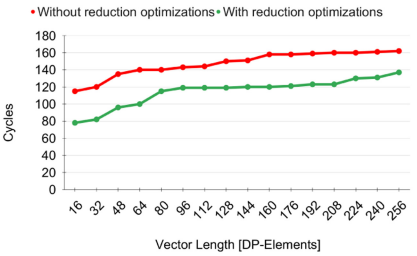
In this section, we present the experimental results of our VPU, including comparisons with state of the art vector processors.

### 6.1 FPGA Evaluation

Vitruvius+ was synthesized for the preliminary FPGA evaluation using Vivado 2020.1 on a VCU128 FPGA from Xilinx. We used specific micro-kernels for evaluating the impact of the overlapping and the vector reductions enhancement. To characterize the impact of overlapping, we created a sequence of back-to-back FMA operations first for a version of the VPU that does not enable




(a) Performance comparison of the instructions overlapping optimization




(b) Performance improvements of the vector reductions after the optimizations

Fig. 11. Evaluation of the overlapping and the vector unordered reductions optimizations.

overlapping, then enabling this feature. We ran this test for a considerable range of *vl* values. Figure 11(a) compares the performance in terms of DP-FLOP/cycle for the two cases. Because each lane has an FPU, the ideal throughput of Vitruvius+ equipped with eight lanes amounts to 16 DP-FLOP/cycle.[8] Despite the low performance in the short-vector region, affected by the non-negligible start-up time, the overlapping optimization performs significantly better than the non-overlapping case. The low performance in the short-vector region is due in part to the VRF accesses generated by the FSM. When the number of elements assigned to each lane is low, such as the case of $vl = 16$ where each lane has only 2 DP-elements, the FSM accesses only two out of the five banks of the VRF. A smart operation scheduler may detect this case and pack the access to the VRF of the following instruction with the one of the previous instruction, trying to optimize the usage of the resource, provided there are no bank conflicts. However, the necessary control logic could have a non-negligible impact on area. On the contrary, the overlapping optimization drastically increases performance in the long-vector region. Overall, this optimization gives an average speedup of 1.7X. We also evaluated the optimizations for the execution of vector unordered reduction operations presented in Section 4.4. In particular, we executed a micro-kernel composed of *vfredsum* instructions, for different vector length values. Figure 11(b) compares the performance obtained by enabling both the usage of the multiple accumulators and the tree-based execution scheme, explained in Section 4.4. It shows the decrease of the cycle count due to the optimizations. We measured a peak reduction of 40 cycles.

### 6.2 Synthesis Results

Vitruvius+ was synthesized for the GF22FDX technology. We selected the 8-Tracks (8T) standard cell library. In the synthesis process, we used a clock period of 700 ps, a bit lower than the target clock corresponding to the frequency of 1.2 GHz, to force the synthesizer to optimize the output netlist and give some margin to the **Place and Route (PnR)** tool to meet the timing constraints. We enabled the retiming feature to deal with the timing critical paths of our design. We set the synthesizer to use both **Super Low $V_t$ (SLVT)** and **Low $V_t$ (LVT)** cells, where $V_t$ represents the transistor *threshold voltage*, to use the fast SLVT cells in the timing critical paths of the design, whereas using slower LVT cells in the parts of the design with no timing issues, contributing to power efficiency. We also enable the clock-gating feature. In the typical corner (TT/0.80 V/25°C), Vitruvius+ successfully meets the timing constraints, reaching an estimated maximum frequency of 1.4 GHz. In the slow corner (SS/0.72 V/125°C), the maximum frequency is around 1.2 GHz, with

[8]Each FMA accounts for two operations: one multiplication and one addition. Therefore, each lane has a peak throughput of 2 DP-FLOP/cycle.

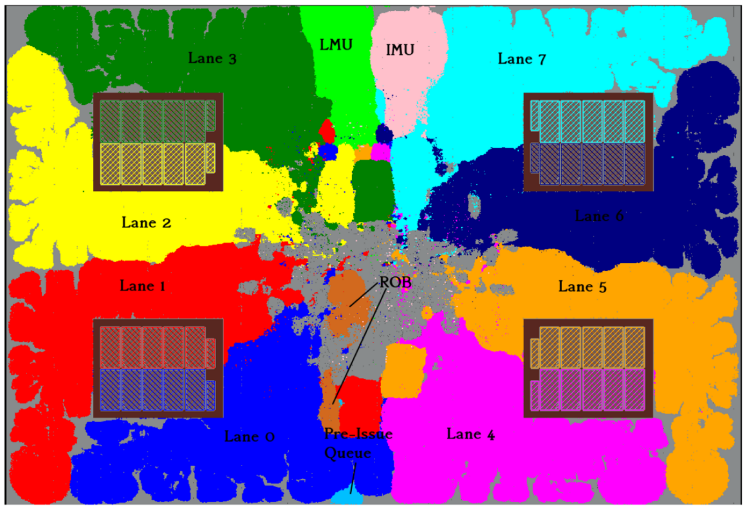



Fig. 12. Floorplan of Vitruvius+ with eight lanes.

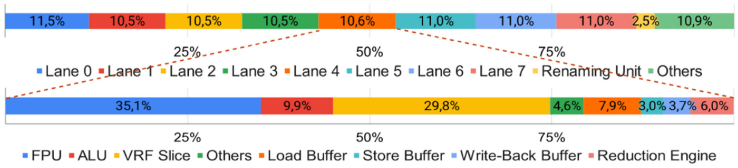



Fig. 13. Breakdown of the area in Vitruvius+. The area distribution of a single lane is also shown.

the critical path being in the Load Management Unit (LMU), one of the memory units presented in Section 3.3. The final synthesis report shows that 45.7% of the total cells are LVT cells and 54.3% are SLVT cells.

### 6.3 Physical Design

The netlist obtained from the synthesis entered the Place and Route (PnR) phase. We tried different placements of the synthesized netlist, and we ended up with the configuration shown in Figure 12, where the different lanes and their VRF slices are highlighted. The floorplan fits in a $1{,}600 \times 1{,}100$ $\mu m^2$ rectangle, where 20 µm on each side are left for the power ring. Therefore, a total area of 1.7064 $mm^2$ is left for the design. Our results show that Vitruvius+ occupies 76% of the available area, giving a total of 1.3 $mm^2$. Power estimation in the typical corner reported by the PnR tool shows a maximum power of ~920 mW. Figure 13 shows the area breakdown of Vitruvius+ and zoom-in on the area distribution of the internal module of a single lane, with the percentage of area consumed by the internal blocks. As depicted, the renaming unit, which enables the lightweight out-of-order execution in Vitruvius+, accounts for only the 2.5% of the total area of the VPU. As for the area breakdown of the single lane, the FPU and the VRF represents the most area-consuming components, occupying 35.1% and 29.8% of the total lane area, respectively. The reduction handler and related improvements described in Section 4.4 accounts for the 6% of the lane area. Figure 14 reports the breakdown of the maximum estimated power consumption. As shown, the impact of the renaming unit is negligible and accounts for only the 1.63% of the total estimated power.

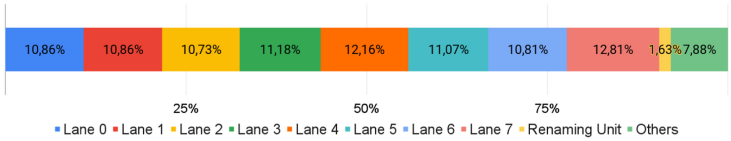



Fig. 14. Breakdown of the maximum power consumption in Vitruvius+.

Table 4. Performance Evaluation for the Set of Benchmarks and Speedup over Vitruvius

| **Frequency:** 1.4 GHz | | | | | |
|---|---|---|---|---|---|
| **Application** | **FLOP/cycle** | | **Power [mW]** | **GFLOPS/W** | **Speedup** |
| | **Vitruvius** | **Vitruvius+** | | | |
| Axpy | 4.1 | 6.1 | 347 | 24.6 | 1.5X |
| MMUL | 14 | 15.5 | 459 | 47.3 | 1.1X |
| FFT | 2.5 | 3.1 | 348 | 12.5 | 1.2X |
| Jacobi2D | 7.4 | 8.2 | 400 | 28.7 | 1.1X |
| Blackscholes | 5 | 6 | 427 | 19.7 | 1.2X |
| LavaMD | 6 | 6.6 | 446 | 20.7 | 1.1X |
| Pathfinder | 2.5 | 3.0 | 365 | 11.5 | 1.2X |
| Streamcluster | 3.9 | 7.1 | 516 | 19.3 | 1.8X |

### 6.4 Benchmarks

Table 4 shows performance, power, and efficiency results, as well as the speedup over the baseline architecture, Vitruvius, for the target benchmarks. Power data were obtained by simulating the back-annotated netlist of the whole design on QuestaSim-64 2021.3_2. The resulting activity was processed in Cadence Joules RTL Power Solution for each of the evaluated benchmarks. Overall, Vitruvius+ scores better than Vitruvius for all the benchmarks due to the optimizations presented in Section 4. In particular, *axpy* benefits from the out-of-order chaining mechanism, as the functional unit can start processing data not strictly from the first element group, as discussed in Section 4.1. Similarly, Fast Fourier transform (FFT) includes indexed operations, and partially benefits from the out-of-order chaining while encountering the bottleneck on the memory interface. FFT also has *vrgather*, one of the longest-latency operations in Vitruvius+. *Streamcluster* is the benchmark that most benefits from the optimizations described in Section 4. In particular, both the optimizations of vector reductions and the fast-move mechanism contribute to the considerable speedup over the previous design. *Jacobi-2D* also improves a bit after the fast-move optimization and the reconfiguration of the inter-lane ring, with a 10% increase on the FLOP/cycle. *Pathfinder* reports a 1.2X speedup due to the switched-ring capabilities. For MMUL, we observe a light speedup. The performance is very close to the peak, showing an efficiency of 97% of FPU utilization.

### 6.5 Comparison with State-of-the-Art Vector Units

We compare the results obtained from the evaluation of Vitruvius+ with relevant state-of-the-art VPUs. Results are collected in Table 5. We consider metrics of vector units in isolation as much as possible. We also believe that for a decoupled unit such as ours, the final efficiency would depend on the type and design of the scalar core.

*6.5.1 Comparison with RISC-V Vector Units.* Hwacha [21] is a single-lane vector unit that implements a custom vector ISA and was taped out with an MVL of 512 bits. The authors report a peak energy efficiency of 16.7 GFLOPS/W at 0.65 V, running at 250 MHz. Hwacha V4 [6] increases

Table 5. Comparison with Relevant State-of-the-Art Vector Units

| RISC-V Vector Processors | | | | | |
|---|---|---|---|---|---|
| **Name** | **RVV version** | **VLEN (bits)** | **Area ($mm^2$)** | **Frequency (GHz)** | **Peak Efficiency (GFLOPS/W)** |
| Vitruvius+ (this work) | 0.7.1 | 16,384 | 1.3 | 1.4 | 47.3 |
| [21] Hwacha | Non-Standard | 512 | 1.31[a] | 1.0 | 16.7 |
| [6] HwachaV4 | Non-Standard | 512 | 4.06[a] | N/A[b] | 40+ |
| [23] Ara | 0.5 | 16,384 | 0.6 | 1.3 | 46.4 |
| [5] Xuantie910 VPU | 0.7.1 | 256 | 0.2[a] | 2.0 | N/A |
| [2] Andes NX27V | 1.0 | 512 | N/A | 1.4 | N/A |
| [40] SiFive P270 | 1.0rc | 256 | N/A | N/A | N/A |
| [39] SiFive X280 | 1.0 | 512 | N/A | N/A | N/A |
| **Non-RISC-V Vector Processors** | | | | | |
| **Name** | **Vector ISA** | **VLEN (bits)** | **Area ($mm^2$)** | **Frequency (GHz)** | **Peak Efficiency (GFLOPS/W)** |
| [27] SX-Aurora VE | NEC Vector ISA | 16,384 | 30.22[a] | 1.6 | N/A |
| [28] A64FX | Scalable Vector Extension (SVE) | 128–2,048 | 1.22[a] | 1.8 | 26.8[a] |

[a]Estimated from publicly available resources.
[b]Not publicly available.
*Note:* The reported values come from taped-out implementations.

the peak energy efficiency up to 40+ GFLOPS/W. Vitruvius+ gets higher peak energy efficiency of 47.3 GFLOPS/W when running a 256 × 256 MMUL kernel, and also runs at higher frequency than the reported Hwacha implementations. Ara [23] targets long vectors with 256 DP-elements and was taped out in a four-lane configuration. This implementation reports an efficiency of 46.4 GFLOPS/W when running a 256 × 256 MMUL kernel. The peak energy efficiency is similar to Vitruvius+. However, Ara supports only a subset of the RVV-0.5, and is missing the support for vector reductions and vector register grouping. Xuantie-910 [5] was taped out with an MVL of 256 bits, way lower than the vector length supported in Vitruvius+. We did not find any other publicly available information for a deeper analysis. Although Table 5 reports ASIC implementations, there are vector units oriented at FPGA implementation, like Arrow [4] and Vicuna [30]. Arrow implements a subset of RVV-0.9, with no support for reductions and permutation instructions, including vector register gather and slide operations, as well as memory indexed operations. Vitruvius+ implements all these features. Platzer and Puschner [30] report a peak efficiency of +90% for compute-bound tasks and different configuration with up to 2,048 bits of MVL. Vitruvius+ reaches a peak efficiency of +97% for compute-bound tasks like the MMUL kernel or the sequence of back-to-back FMAs as reported in Section 6.1. With these works being FPGA-based implementations, we cannot make a fair comparison in terms of area and power efficiency.

*6.5.2 Comparison with Non-RISC-V Vector Units.* We consider two of the most popular non-RISC-V vector units. A disclaimer that by no means the following paragraphs are advocating Vitruvius+ is at the maturity level of the commercial vector units. We report the following considerations as a qualitative analysis to better understand how Vitruvius+ positions itself when compared to the best-in-class vector processors. Because of the lack of publicly available information on power analysis of the NEC SX-Aurora TSUBASA VE [27], we considered performance per area for a qualitative analysis. We retrieved information about the fabrication aspects of SX-Aurora from Schor [36]. By removing the estimated area due to caches from the reported area information, and by equally splitting the remaining area of the single vector core between the

scalar pipeline and the VPU, we estimated an area of 30.22 $mm^2$ for the SX-Aurora VPU. This gives ≈10.16 GFLOP/$mm^2$ at 1.6 GHz (16 nm). Calculating the same metrics for Vitruvius+, we obtain 17.23 GFLOP/$mm^2$ at 1.4 GHz, as a proof of its efficient resource utilization. We believe that Vitruvius+ would be even more efficient when scaling to the same technology node.

For the SVE VPU of the A64FX core from Fujitsu [28], we used the results from the work of Arima et al. [3]. Again, by eliminating the area contribution due to caches, we extrapolated an area per core of 2.44 $mm^2$. We equally distributed the calculated per-core area among its scalar and vector pipelines. This favors the VE, which gets 1.22 $mm^2$ and results in 47 GFLOPS/$mm^2$ at 1.8 GHz (7 nm). Vitruvius+ yields a total of 17.23 GFLOPS/$mm^2$ at 1.4 GHz (22 nm). We believe that technology scaling from 22 nm down to 7 nm will place Vitruvius+ efficiency closer to the result of A64FX. From the work of Arima et al. [3], we could also retrieve information on power metrics. In particular, it reports that the power consumption of a core group composed of 12 cores when running DGEMM is ≈26 W. This gives an estimated power per core of 2.1 W. From Moss [25], where the peak performance of DGEMM is revealed, we deduced a peak performance per core of around 56.2 GFLOPS, which corresponds to almost 97% of efficiency of FPU utilization. Based on this, we calculate a peak power efficiency of 26.8 GFLOPS/W (1.8 GHz, 7 nm). Based on the results of MMUL we measure in Section 6.4, for Vitruvius+ we calculate an efficiency of 97% of FPU utilization, with a peak power efficiency of 47.3 GFLOPS/W (1.4 GHz, 22 nm).

## 7 CONCLUSION AND FUTURE WORK

This work presented Vitruvius+, a VPU with 256-DP-element vectors targeting HPC applications. It implements RVV-0.7.1 and adopts a hybrid in-order/out-of-order execution scheme supported by register renaming and arithmetic/memory instruction decoupling. In its next generation, Vitruvius+ will support RVV-1.0, the latest version of RVV. We identify mainly two challenging features to support. The first feature is the new mask layout, which always maps bit *i* of the mask register to bit *i* of the vector register *v0*, regardless of the element size and the LMUL settings. Due to the interleaving of the vector elements shown in Figure 5, this implies the need of a mechanism to distribute accordingly the mask bits to the vector lanes when executing predicated instructions. Another feature supported in RVV-1.0 is what the ISA calls *fractional LMUL*. RVV-1.0 allows LMUL to assume fractional values, specifically 1/2, 1/4, and 1/8. The overall effect of this feature is the reduction of the vector length within a single vector register. We believe that this feature does not have a critical impact on the current design and can be implemented with a few changes in the front-end.

## REFERENCES

[1] Rob Aitken. 2021. Performance per Watt Is the New Moore's Law. Retrieved December 15, 2022 from https://www.arm.com/blogs/blueprint/performance-per-watt.

[2] AndesCore. 2020. AndesCore NX27V Processor 64-bit CPU with RISC-V Vector Extension. Retrieved May 29, 2022 from http://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/.

[3] Eishi Arima, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, and Mitsuhisa Sato. 2021. Power/performance/area evaluations for next-generation HPC processors using the A64FX chip. In *Proceedings of the 2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS'21)*. 1–6. DOI: https://doi.org/10.1109/COOLCHIPS52128.2021.9410320

[4] Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen A. R. Saghir. 2021. Arrow: A RISC-V vector accelerator for machine learning inference. arxiv:2107.07169 (2021).

[5] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, et al. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. 52–64.

[6] Schmidt Colin, Ou Albert, and Asanović Krste. 218. Hwacha V4: Decoupled Data Parallel Custom Extension. Retrieved December 15, 2022 from https://riscv.org/wp-content/uploads/2018/12/Hwacha-A-Data-Parallel-RISC-V-Extension-and-Implementation-Schmidt-Ou-.pdf.

[7] Control Data Corporation. 1975. *Control Data STAR-100 Computer*. Control Data Corporation. http://bitsavers.trailing-edge.com/pdf/cdc/cyber/cyber_200/60256000_STAR-100hw_Dec75.pdf.

[8] Marius Cornea. 2015. *Intel AVX-512 Instructions and Their Use in the Implementation of Math Functions. Intel Corporation.*

[9] R. G. Dreslinski, M. Wieckowski, D. Blauw, D. Sylvester, and T. Mudge. 2010. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE* 98 (2010), 253–266.

[10] Ralph Duncan. 1990. A survey of parallel computer architectures. *Computer* 23, 2 (1990), 5–16.

[11] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA'11)*. 365–376.

[12] Roger Espasa and Mateo Valero. 1996. Decoupled vector architectures. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*.

[13] R. Espasa, M. Valero, and J. E. Smith. 1998. Vector architectures: Past, present and future. In *Proceedings of the International Conference on Supercomputing (ICS'98)*. ACM, New York, NY, 425–432. https://doi.org/10.1145/277830.277935

[14] EuroHPC. 2018. The European High Performance Computing Joint Undertaking (EuroHPC JU). Retrieved May 29, 2022 from https://eurohpc-ju.europa.eu/.

[15] European Processor Initiative. 2019. EPI Accelerator. Retrieved May 29, 2022 from https://www.european-processor-initiative.eu/accelerator/.

[16] Exascale Computing Project. 2018. Exascale Computing Project. Retrieved May 29, 2022 from https://www.exascaleproject.org/.

[17] Fujitsu Post-K. 2019. Fujitsu Begins Production of Post-K. Retrieved December 15, 2022 from https://www.fujitsu.com/global/about/resources/news/press-releases/2019/0415-01.html.

[18] Fabrizio Gagliardi, Miquel Moreto, Mauro Olivieri, and Mateo Valero. 2019. The international race towards Exascale in Europe. *CCF Transactions on High Performance Computing* 1 (2019), 3–13.

[19] GREEN500 List. 2022. GREEN500 List—June 2022. Retrieved December 15, 2022 from https://www.top500.org/lists/green500/2022/06/.

[20] Jonathan Koomey. 2016. Our Latest on Energy Efficiency of Computing over Time, Now Out in Electronic Design. Retrieved December 15, 2022 from https://www.koomey.com/post/153838038643.

[21] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. 2014. A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *Proceedings of the 40th European Solid State Circuits Conference (ESSCIRC'14)*. 199–202. DOI: https://doi.org/10.1109/ESSCIRC.2014.6942056

[22] Y. Lu. 2019. Paving the way for China exascale computing. *CCF Transactions on High Performance Computing* 1 (2019), 63–72.

[23] Matheus Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. 2020. Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 02 (Feb. 2020), 530–543. DOI: https://doi.org/10.1109/TVLSI.2019.2950087

[24] Kim McMahon. 2022. Intel Corporation Makes Deep Investment in RISC-V Community to Accelerate Innovation in Open Computing. Retrieved May 30, 2022 from https://riscv.org/blog/2022/02/intel-corporation-makes-deep-investment-in-risc-v-community-to-accelerate-innovation-in-open-computing.

[25] Sebastian Moss. 2018. Fujitsu Reveals Specs of A64FX, Its Post-K Supercomputer CPU. Retrieved December 15, 2022 from https://www.datacenterdynamics.com/en/news/fujitsu-reveals-specs-a64fx-its-post-k-supercomputer-cpu/.

[26] Sam Naffziger and Jonathan Koomey. 2016. Energy Efficiency of Computing: What's Next? Retrieved December 15, 2022 from https://www.electronicdesign.com/technologies/microprocessors/article/21802037/energy-efficiency-of-computing-whats-next.

[27] NEC. 2020. SX-Aurora TSUBASA. Retrieved December 15, 2022 from https://www.nec.com/en/global/solutions/hpc/sx/architecture.html.

[28] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. 2020. Supercomputer Fugaku CPU A64FX realizing high performance, high-density packaging, and low power consumption. *Fujitsu Technical Review* 2020 (2020), 1–9.

[29] Gianmarco Ottavi, Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. 2020. A mixed-precision RISC-V processor for extreme-edge DNN inference. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'20)*. 512–517. DOI: https://doi.org/10.1109/ISVLSI49217.2020.000-5 arxiv:2010.04073

[30] Michael Platzer and Peter Puschner. 2021. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems (ECRTS'21)*. 1–18. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[31] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. A RISC-V simulator and benchmark suite for designing and evaluating vector architectures. *ACM Transactions on Architecture and Code Optimization* 17, 4 (Nov. 2020), Article 38, 30 pages. DOI: https://doi.org/10.1145/3422667

[32] Venu Gopal Reddy. 2008. *Neon Technology Introduction. ARM Corporation.*

[33] Antonio Regalado. 2022. MIT Technology Review: Covid Variant Tracking. Retrieved May 30, 2022 from https://www.technologyreview.com/2022/02/23/1044975/covid-19-variant-tracking-scientists/.

[34] RISC-V V-extension. 2022. RISC-V V-extension. Retrieved May 29, 2022 from https://github.com/riscv/riscv-v-spec.

[35] Richard M. Russell. 1978. The CRAY-1 computer system. *Communications of the ACM* 21, 1 (Jan.1978), 63–72. https://doi.org/10.1145/359327.359336

[36] David Schor. 2018. SX-Aurora-Microarchitectures-NEC. Retrieved December 15, 2022 from https://en.wikichip.org/wiki/nec/microarchitectures/sx-aurora.

[37] GitHub. 2020. OVI: Open Vector Interface. Retrieved December 15, 2022 from https://github.com/semidynamics/OpenVectorInterface.

[38] Semidynamics. 2021. Semidynamics High Bandwidth RISC-V IP Core Avispado. Retrieved December 15, 2022 from https://semidynamics.com/products/avispado.

[39] SiFive. 2022. SiFive Intelligence X280. (2022). Retrieved August 21, 2022 from https://www.sifive.com/cores/intelligence-x280.

[40] SiFive. 2022. SiFive Performance P270. Retrieved August 21, 2022 from https://www.sifive.com/cores/performance-p270.

[41] GitHub. 2014. Spike RISC-V ISA Simulator. Retrieved May 30, 2022 from https://github.com/riscv/riscv-isa-sim.

[42] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, et al. 2017. The ARM scalable vector extension. *IEEE Micro* 37, 2 (2017), 26–39.

[43] European Processor Initiative. 2021. EPI EPAC 1.0 RISC-V Test Chip Taped-out. Retrieved December 15, 2022 from https://www.european-processor-initiative.eu/epi-epac1-0-risc-v-test-chip-taped-out/.

[44] TOP500 List. 2022. TOP500 List—June 2022. Retrieved December 15, 2022 from https://www.top500.org/lists/top500/2022/06/.

[45] W. J. Watson. 1972. The TI ASC: A highly modular and flexible super computer architecture. In *Proceedings of the Fall Joint Computer Conference, Part I (AFIPS'72, Fall, Part I)*. 221–228. https://doi.org/10.1145/1479992.1480022

Received 31 May 2022; revised 30 September 2022; accepted 10 November 2022