Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

ISMatch: A real-time hardware accelerator for inexact string matching of DNA sequences on FPGA

Alberto Marchisio [a,*], Federico Teodonio [b], Antonello Rizzi [b], Muhammad Shafique [c]

[a] Technische Universität Wien (TU Wien), Vienna, Austria
[b] University of Rome "La Sapienza", Rome, Italy
[c] eBrain Lab, Division of Engineering, New York University Abu Dhabi, United Arab Emirates

ARTICLE INFO

Keywords:
FPGA
DNA sequence
Inexact string matching
Hardware design
Hardware accelerator
High Level Synthesis
Levenshtein distance
Energy efficiency
Performance

ABSTRACT

Since DNA strings suffer from variations like mutation, noisy sampling, and transmission, instead of searching for the exact match, the inexact string matching (ISM) of DNA sequences is preferred. Due to the large amount of data and massive data-dependency, the ISM algorithm is not suitable for being implemented into a general-purpose hardware. Towards this, we propose *ISMatch*, a novel specialized hardware architecture for computing the ISM in a fast and energy-efficient way. Our implementation on a Xilinx Ultrascale+ FPGA shows up to 70× and 2.2× clock cycles reduction compared to the ARM-based and the HLS implementations, respectively.

1. Introduction

The inexact string matching (ISM) is an important problem in fields such as bioinformatics [1], signal processing [2,3], and text retrieval [4], where the exact string matching is not necessarily feasible. Since the genome is usually composed of millions or billions of base pairs that can have several variations during the DNA replication or genetic recombination [5], finding an exact match in the DNA string is highly unlikely, because all the similar strings that had got variations during DNA processes are discarded [6]. Moreover, the current trend is to process a large amount of such data [7]. For instance, due to the large collection of words, a mistaken word cannot be recovered with a standard string matching. Hence, for text retrieval, an ISM algorithm is needed [8]. The ISM algorithm can find hits with a level of inexactness that can be decided, and can yield to different shadow hits that would not have been detected by an exact string matching algorithm. Especially in the bioinformatics field, the possible variations that can occur in the DNA protein chains [5] are Single Nucleotide Polymorphisms (SNPs), insertion/deletions of small fragments, inversion, Copy Number Variations (CNVs). All these variations can lead to different diseases that can be detected with a research in the DNA protein chain using an ISM algorithm.

For the ISM algorithm, the distance between strings needs to be properly defined. The most common definitions are based on the Burrows-Wheeler Transform [9] and the Levenshtein distance [10]. Due to its relatively simple algorithmic constructs, the data-dependency that perfectly fits the HDL development, and the low memory requirement, the Levenshtein distance is adopted in this work. However, the computation of the ISM algorithm grows exponentially with the length of the pattern to search; it is $O(|p||s|)$ in time, where $|p|$ and $|s|$ are the length of the two strings. Hence, this can lead to an unacceptably large execution time. For this reason, the CPU implementations pose a huge bottleneck as they cannot compute large pattern search with a high degree of parallelism, thereby rendering the overall system inefficient and extremely slow. Although an HDL implementation can provide a high speedup, the level of parallelism for the computational units that search different patterns is limited by the resource usage. The aim of this work is to increase the compute resource efficiency and throughput of the ISM algorithm by developing an optimized hardware accelerator, and to demonstrate its efficacy by comparing it with the CPU implementation, an High Level Synthesis (HLS)-based hardware implementation, and other state-of-the-art accelerators.

1.1. Key scientific challenges and motivational case study

The target problem is to optimize the resource usage with the minimum desired speedup factor. For the High Level Synthesis (HLS),

* Corresponding author.
E-mail address: alberto.marchisio@tuwien.ac.at (A. Marchisio).

[1] Note, this is an illustrative example to make the computations simple, while in the real world the strings can be thousand or millions of characters long. However, these lengths are significant for comparison purposes.

https://doi.org/10.1016/j.micpro.2023.104763
Received 30 March 2022; Received in revised form 3 August 2022; Accepted 4 January 2023
Available online 9 January 2023

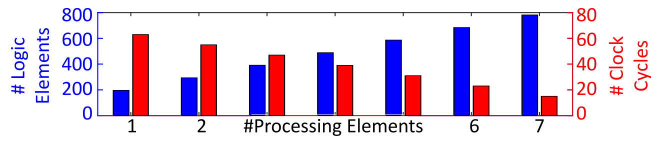


Fig. 1. Clock cycles and resource usage with more processing elements are used to process an 8-characters long string. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

this can be done by choosing different target optimizations (e.g., loop unrolling, pipelining, etc.). The specialized HDL design is composed of the Levenshtein distance computation block that computes the distance based on the Levenshtein algorithm (see Section 2.1). Its inner data-dependency allows for designing the block with a systolic architecture composed of different processing elements, each of them computing every anti-diagonal in one clock cycle through the wavefront approach [11]. The throughput of the system is incremented when more processing elements (PEs) are introduced to compute the Levenshtein distance. To motivate the need for having multiple PEs in parallel, Fig. 1 shows tradeoffs between the latency and the resource usage for the hardware-based processing of the Levenshtein distance between strings of 8-characters length.[1] The resource usage grows w.r.t. the number of PEs of the architecture, as the throughput improves with a linear relationship. The key challenges that are addressed in this paper are:

- *How to efficiently compute the Levenshtein distance?*
- *Which hardware platform provides a fast execution to process the ISM algorithm in real-time?*
- *How can we develop efficient an accelerator for this task on FPGA?*

To address the above problems, we propose ISMatch, a novel hardware architecture design that enables efficient computation of the Levenshtein distance for the ISM algorithm, and compare its execution with other implementations (see Fig. 2).

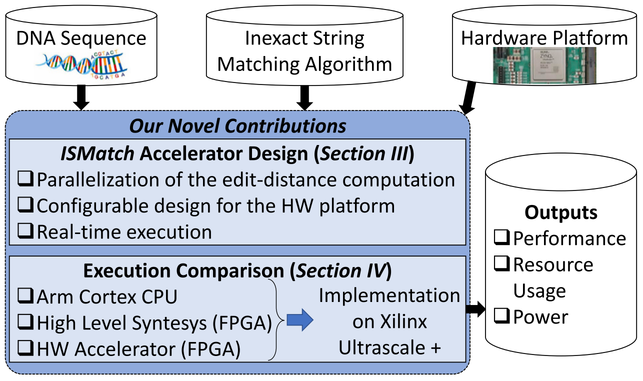
1.2. Our novel contributions

- **ISMatch: Optimized Hardware Architecture Design** (Section 3) for accelerating the ISM algorithm, realized with a specialized HDL design and implemented on a Xilinx Ultrascale + FPGA. It supports a parallel execution of the algorithm for computing the edit-distance and fast PCIe interface with the host. It can be configured for executing the algorithm with different parallelism granularity, by choosing the matching threshold and the number of PEs of the architecture.
- **High Level Synthesis Implementation** (Section 4.3) on FPGA, with different optimization levels, for comparison purposes.
- **Performance Comparison** (Section 4) of the Levenshtein algorithm for different implementations, which are:
  - An optimized ARM CPU-based implementation
  - An optimized GPU-based implementation [11]
  - High Level Synthesis for FPGA implementation
  - State-of-the-art Accelerator [12] for FPGA implementation
  - ISMatch Accelerator for FPGA implementation

  The measured output metrics that are compared across the hardware platforms are performance, resource usage, and power consumption.

2. Overview of inexact string matching

The ISM algorithm searches and validates a pattern in a long string like DNA protein sequences. When the pattern is found an occurrence is



DNA Sequence
Inexact String Matching Algorithm
Hardware Platform
Our Novel Contributions
ISMatch Accelerator Design (Section III)
❑Parallelization of the edit-distance computation
❑Configurable design for the HW platform
❑Real-time execution
Execution Comparison (Section IV)
❑Arm Cortex CPU
❑High Level Syntesys (FPGA)
❑HW Accelerator (FPGA)
Implementation on Xilinx Ultrascale +
Outputs
❑Performance
❑Resource Usage
❑Power

Fig. 2. Overview of our novel contributions (shown in blue boxes).

created. The exact matching search cannot detect the small variations that can occur in the DNA protein chains and a huge number of diseases cannot be detected. For this reason, an inexact string searching algorithm is employed, since it can detect inexact matches with a variation level that can be controlled by a threshold.

2.1. The Levenshtein distance computation

The ISM algorithm searches in a long string a pattern with a grade of dissimilarity. The search of the pattern is realized by comparing the pattern with a moving window that scans the whole string. The dissimilarity between the string inside the window s and the pattern p are calculated with the Levenshtein distance [10], also called Edit distance, which is the minimum unitary cost operation necessary for the transformation of a given string into another one. Each element of the Levenshtein matrix is computed as in Eq. (1).

$$lev_{\mathrm{p,s}}(i,j) = \begin{cases} i & \text{if j=0 and i>0} \\ j & \text{if i=0 and j>0} \\ min(lev_{\mathrm{p,s}}(i-1,j)+1, & \\ lev_{\mathrm{p,s}}(i,j-1)+1, & \\ lev_{\mathrm{p,s}}(i-1,j-1)+[p_{\mathrm{i}} \neq s_{\mathrm{j}}]) & \end{cases} \tag{1}$$

where $lev_{\mathrm{p,s}}(i,j)$ represents the $(i$th, jth$)$ element of the Levenshtein matrix between the pattern p and the string inside the window s, the indexes i and j go from 0 to the length of the strings $|p|$ and $|s|$, respectively. p_{i} and s_{j} represent the ith element of the string p and the jth element of the string s, respectively.

The Levenshtein distance computation leads to a matrix as the one in Fig. 3. The matrix construction follows the procedure of Algorithm 1, whose complexity is O($|p||s|$), since on lines 9 and 10 of Algorithm 1 there are two nested for loops, where each loop index goes from 0 to the length of the string. Fig. 3 illustrates a simple yet comprehensive example, where the matrix is computed with the two strings s = “CTTAC” and p = “CTGA”. Note that the edit distances between the substrings and the pattern are obtained without extra computation. E.g., the distance between s = “CTTAC” and p = “CTGA” in Fig. 3 is obtained without extra computation and results with a distance of 1. If any of the distances between the substrings is below the threshold K, an occurrence will be generated. The Levenshtein matrix construction has data-dependencies, where each element computation depends only on its upper, its left, and its upper-left neighbors. Going more into detail of the example, in Fig. 3a, the computation of the value at this coordinate depends on its neighbors, whose values are 0, 1, and 1. Since $s[i] = p[j]$, the cost is equal to 0, as indicated in line 12 of Algorithm 1. As a consequence, the final value results to be 0, following the computations in line 19 of Algorithm 1. The algorithm proceeds to the next value, as in Fig. 3b, and assigns the value to 1. Afterward, all the values in the

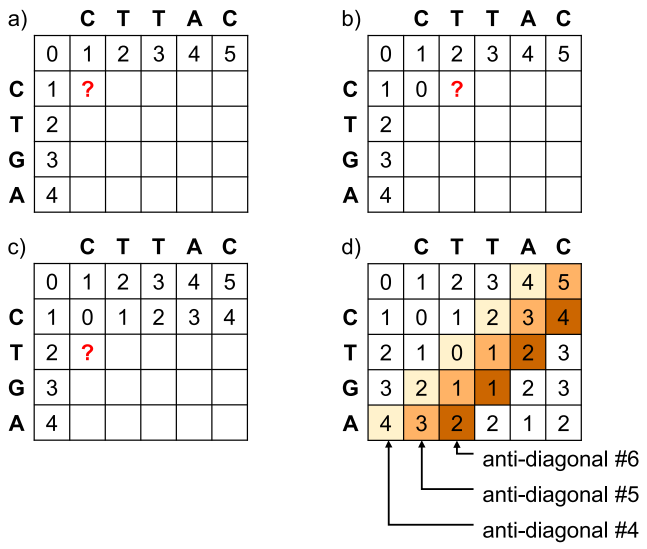


Fig. 3. Data-dependency across the matrix for computing the Levenshtein distance.

row are processed, before moving to the next row (see Fig. 3c). Finally, all the values of the matrix are shown in Fig. 3d. Note that, since the computation of each element depends only on the three neighbors, in practice, the algorithm can be implemented on a systolic architecture, where each anti-diagonal can be computed concurrently. For instance, as shown in Fig. 3d, all the values in the anti-diagonal #5 can be computed concurrently after the anti-diagonal #4, all the values in the anti-diagonal #6 can be computed concurrently afterward, and so on. Such a systolic architecture is implemented in the proposed ISMatch accelerator.

Algorithm 1 Matrix construction for the Levenshtein distance computation.

```
1: int LevenshteinDistance(char $s[1..m]$, char $p[1..n]$)
   {// d is a table with m+1 rows and n+1 columns}
2: declare int $d[0..m, 0..n]$
3: for $i$ from 0 to $m$ do
4:     $d[i, 0] := i$
5: end for
6: for $j$ from 0 to $n$ do
7:     $d[0, j] := j$
8: end for
9: for $i$ from 1 to $m$ do
10:    for $j$ from 1 to $n$ do
11:        if $s[i] = p[j]$ then
12:            $cost := 0$
13:        else
14:            $cost := 1$
15:        end if
16:        $d[i, j] := minimum($
17:                $d[i-1, j] + 1,$
18:                $d[i, j-1] + 1,$
19:                $d[i-1, j-1] + cost)$
20:    end for
21: end for
22: return $d[m, n]$
```

2.2. Validation of the pattern

When the Levenshtein distance between the pattern and the string coming from the moving window is below the threshold, an occurrence is generated. The occurrence needs to be validated before being saved into the memory. This allows the algorithm to discard the duplicate hits or substrings that are part of a longer occurrence. The validation process ensures that the occurrence is not part of a longer occurrence, and relies on the following variables:

- The length of the occurrence p.
- The distance d between the occurrence p and the string s.
- The threshold K.
- The validation of another occurrence with higher priority.

Algorithm 2 Validation of an occurrence.

```
1: if $d < K$ and not busy(more priority) then
2:     $p :=$ occurrence
3:     $busy[d] := 1$
4: end if
5: if $busy[d] := 1$ then
6:     $cnt := cnt + 1$
7: end if
8: if $cnt = |p|$ and not busy(more priority) then
9:     p is valid
10: else if $cnt > |p|$ and not busy(more priority) then
11:     if $cnt - |last\ occurrence\ validated| > |p|$ then
12:         p is valid
13:     else
14:         p is not valid
15:     end if
16: end if
```

The validation process of a string is described in Algorithm 2. It relies on different priorities, from 0 to K. The lower the Levenshtein distance is, the higher the priority is. In line 1, an occurrence begins a validation process only if there is not another occurrence of the same priority or higher. If there is an ongoing validation process for the same or higher priority, the occurrence is discarded. When a validation process starts, a counter starts counting, and it is increased every time the window slides one character into the text. In line 8, if the counter reaches the length of the occurrence $|p|$ and concurrently there is no other validation process with higher priority, the occurrence becomes valid, which means that the counter has reached the length of the current occurrence without finding any other occurrence. If there is a validation process with higher priority during the same time, the algorithm first validates the higher priority occurrence. In line 11, when the higher priority occurrence is validated, a simple inequality checks if the low priority occurrence is a substring of another higher priority occurrence. If it is, the occurrence is discarded, otherwise the occurrence is valid. It is important to notice that the higher priority occurrences block the lower priority occurrences to be validated. Indeed, an occurrence with maximum priority corresponds to an exact match. When the validation succeeds, the occurrence is saved into the memory with its length, Levenshtein distance and index.

2.3. Major related works in hardware accelerator designs

Many related studies focus on the read alignment problem, most of them implementing the Smith–Waterman algorithm [13]. The work in [14] proposed an architecture based on PE arrays. The architecture proposed in [15] can integrate up to 110 processing elements. Darwin [16] is an efficient ASIC accelerator for genomic sequence alignment. GateKeeper [17] and SneakySnake [18] proposed specialized architectures for pre-alignment of DNA sequences. The architecture in [19] supports both exact and approximate alignment computations. An OpenCL-based design is proposed in [20]. The work in [21] accelerates the pre-alignment filters on FPGA using HLS toolchains that can be executed through the OpenCL runtime. The architectures proposed in [22,23] accelerate the Smith–Waterman algorithm through

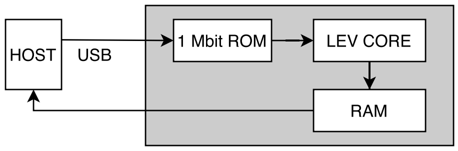


Fig. 4. Unoptimized design of [12].

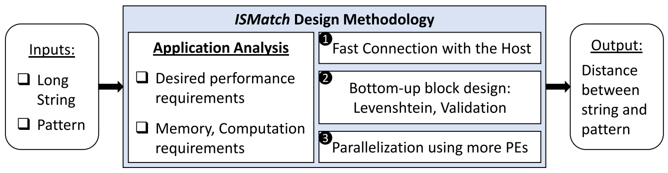

ISMatch Design Methodology
Inputs:
Long String
Pattern
Application Analysis
Desired performance requirements
Memory, Computation requirements
1 Fast Connection with the Host
2 Bottom-up block design: Levenshtein, Validation
3 Parallelization using more PEs
Output:
Distance between string and pattern

Fig. 5. ISMatch design methodology.

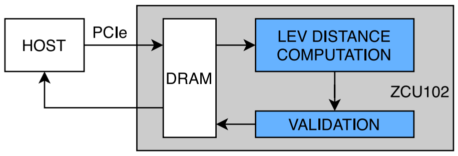


Fig. 6. High-level scheme of our ISMatch architecture.

systolic arrays. The ASAP architecture [24] executes the read alignment algorithm and interfaces to the host through a CAPI interface [25]. Minimap2 [26] is an efficient alignment algorithm for mapping DNA or long mRNA sequences against a large reference database. GenASM [27] accelerates read alignment through a specialized architecture based on systolic-array-based compute units and on-chip SRAMs. BWA-Mich [28] proposed a novel indexing data structure based on Enumerated Radix Tree and designed a custom architecture based on it to accelerate the seeding step of read alignment. SeGraM [29] is an algorithm/hardware co-design framework for accelerating the seeding and alignment steps of genome sequences. However, all the aforementioned architectures focus on the *alignment* problem. Since the primary focus of our paper is on string *matching*, the additional computations for the alignment are not needed, and existing hardware architectures for alignment can be integrated with our design. The works published in [11,30] proposed specialized optimizations for executing approximate string matching on GPUs, while our work focuses on FPGA.

The work of [12] proposed an online approximate string matching with a specialized HDL implementation. It consists of a Levenshtein core block (LEV CORE) for computing the distance, and a RAM memory to store the validated occurrences, connected to the host via USB. However, the study did not emphasize the throughput variations between a CPU implementation and the HDL design, and the speedup factor that the HDL design has introduced. This work also lacks of results with parallel search of different patterns and a proper fast interface with the host PC that enables a fast transfer (i.e., 1Gb/s or higher) of the data to the FPGA RAM, thus preventing its usage for a real-time execution. This aspect is extremely important, since the long DNA sequences consist of millions or billions of elements, which cannot be stored on-chip, and need to be streamed from the host PC. Indeed, in [12], the host is interfaced with a low-speed USB link, which limits the potential speedup guaranteed by the specialized HDL design. Fig. 4 illustrates a summary scheme of the system proposed in [12], with its system setup. The LEV CORE block is responsible for the computation and validation of the occurrences. The 1 Mbit ROM is loaded with a fraction of the data and then the algorithms starts, the occurrences are saved and then transferred to the PC with a USB interface. The lack of a fast interface for data transfer is a bottleneck for the entire ISM algorithm accelerator system.

The study proposed in [31] realized a parallel inexact string matching accelerator using the Burrows-Wheeler Transform (BWT). Each processing element contains a storage unit allocated into the BRAM, which leads to a heavy memory resource usage that limits the number of parallel PEs that can be synthesized into the FPGA. Moreover, it poses a limitation on the length of the pattern to search, since, due to the BWT process, the longer the pattern is, the larger the BRAM allocation is needed. *Our work aims at overcoming the memory challenge of [31] by using a distance computation of strings based on the Levenshtein algorithm, which does not require massive memory usage like in [31] and implements parallel pattern search.*

3. ISMatch: Hardware accelerator design

The online execution complexity of the Levenshtein distance computation is O($|p||s|$) in time and requires a fast execution of the algorithm. With an hypothetical data transfer of 100 Mbit/s between the host and the board, considering the 100 MHz clock frequency of the board, a DRAM organized in words of 32 bytes, a 8-characters window, the algorithm should not exceed 32 clock cycles for the computation. If the algorithm requires more clock cycles to compute the Levenshtein distance, the speed between the host and the board can be reduced. Considering this application analysis, we develop the ISMatch design methodology, which is depicted in Fig. 5. To achieve real-time execution, it is important to design a fast connection to the host, to stream the long string that needs to be processed. In particular, its real-time system execution is constrained that it should not exceed the latency of 100 clock cycles. Exceeding this threshold can cause the algorithm to miss data to scan into the DRAM. The main blocks, computing the Levenshtein distance and the validation, are designed bottom-up, with a modular approach. In this way, it is possible to build an architecture having multiple PEs that can process concurrently.

3.1. Research highlights and innovations

Our proposed ISMatch architecture computes the distance between the occurrence p and the string s. These data are fed into the architecture by a Host PC, as illustrated in Fig. 6. To guarantee high-speed connection between the PC and the board, the DRAM is constantly filled with data from the PC. The DRAM is organized in 32-bits wide words and each memory location stores one character. The Levenshtein distance computation block, which will be discussed more in detail in Section 3.2, reads the characters and computes the Edit distance by asserting a 1-bit hit signal when an occurrence is found. Afterward, the validation block (see Section 3.3) controls to avoid duplicate occurrences and validates the output.

In summary, the following research highlights and innovations can be identified:

- We define the desired performance, memory, and computation requirements.
- We devise fast communication between the host PC and the board through a PCIe connection interface.
- We design the hardware architectures of the processing blocks that compute the Levenshtein distance and validate the occurrences.
- The proposed ISMatch architecture supports different degrees of parallelism, thus making it highly scalable.
- We compare our ISMatch design with CPU and HLS implementation, and we conduct performance comparisons with related works, including GPU and FPGA implementations.

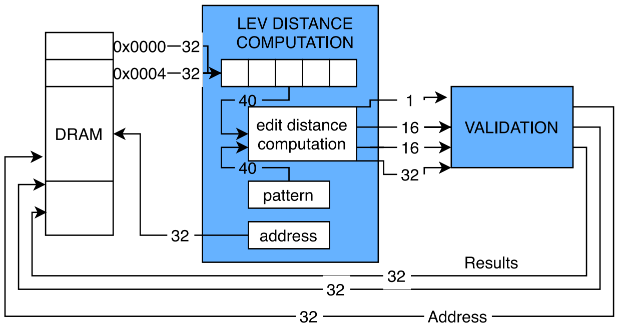


Fig. 7. Circuit diagram of the Levenshtein distance computation block.

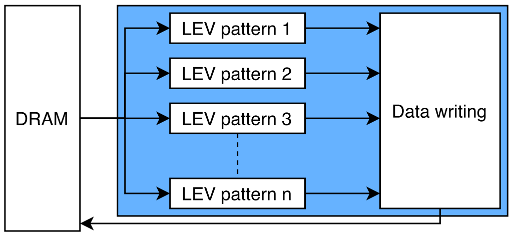


Fig. 8. Parallelization of the Levenshtein distance computation blocks.

3.2. Levenshtein distance computation

Fig. 7 shows the details of the block that computes the Levenshtein distance. The DRAM memory locations fill the window that has a variable length (e.g., it has a length of 5 characters in Fig. 7). This block stores the pattern that we want to compare with the string. The DRAM is scanned with an address that is incremented every time the Levenshtein distance computation block computes the result. The Levenshtein distance computation block computes the distance and asserts the 1-bit *hit* signal when an occurrence is found. Note, the threshold for the assertion of such a signal can be configured by the user. When the hit signal is asserted, the output data are valid. The output data are the following:

- 16 bits for the Edit distance
- 16 bits for the length of the occurrence
- 32 bits for the index of the text loaded into the DRAM

The validation block reads the output data and validates the occurrence. If the occurrence is valid (i.e., it is not a substring of a longer occurrence) the block writes two 32-bit words into the DRAM. In the first word the 16-bit Occurrence Length and the 16-bit Levenshtein distance are stored, while in the second word the 32-bit index of the occurrence is saved.

The ISMatch design is scalable, because it allows to introduce a configurable number of Levenshtein computation blocks in parallel, to better utilize the computational hardware resources. The parallel search of different patterns can be conducted without computation time overhead. As shown in Fig. 8, each block receives the same data from the DRAM and computes the distance between the window and the pattern that belongs to the block. To avoid simultaneous DRAM accesses from different Levenshtein blocks, the Data writing block collects all the occurrences and sequentially accesses the DRAM for avoiding collisions.

The optimized design relies on a systolic architecture composed of n Processing Elements (PEs), where n is the length of the window and the pattern (Fig. 9). The systolic architecture allows to execute the Levenshtein distance in $|p| + |s| - 1$ clock cycles, where p represents the pattern and s the string coming from the window. In Fig. 10 are shown

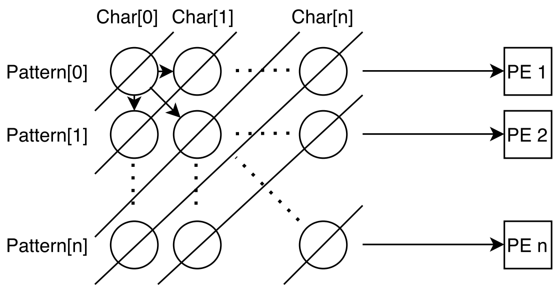


Fig. 9. Linear mapping on a Systolic architecture with n Processing Elements.

the first 2 PEs of the edit distance computation block, whose complete block is composed of n PEs. The Char Comparison block is a comparator that returns the logic 0 if the characters are the same, or the logic 1 if they are different. These PEs build the Levenshtein matrix, thus saving each result in the registers that compose the array. When the matrix is complete, if the result is below the threshold K, an occurrence will be generated, and the index, the length, and the Levenshtein distance will be transmitted to the validation block.

3.3. Validation of the occurrence

One of the most important parts of the entire algorithm is the validation procedure. The task of this block is to ensure that multiple occurrences that are part of a longer occurrence are not saved, thus avoiding unnecessary memory usage and communications. When the Levenshtein distance computation block finds an occurrence, the *hit* signal is asserted, and the output data is valid. Fig. 11 shows the architectural diagram of the validation process.

The Validation architecture is composed of $K + 1$ blocks, where K is the desired threshold. Each block validates the occurrence for a specific Levenshtein distance. When a hit is generated, the data is transmitted to the block of the corresponding Levenshtein distance. For instance, if the occurrence has a Levenshtein distance of 1, the data will be sent to the validation block 1. The validation algorithm explained in Algorithm 2 is realized with a hierarchical implementation. Each validation block, except for the last one, has a 1-bit busy signal that is asserted when a validation is running. The validation block 0, which is the first one, validates the exact match and inhibits the other blocks for the validation. I.e., the block i inhibits all the blocks between $i + 1$ and K. Fig. 12 reveals the inside of a validation block. The block is based on a counter, and the occurrence is validated when the counter has reached the end and all the busy signals from blocks with higher priorities are done.

4. Execution comparison

4.1. Experimental setup

The target board for implementing and comparing the different hardware platforms is the Zynq UltraScale + ZCU102 MPSoC (see Fig. 13), whose specifications are reported in Table 1. This board features a quad-core Arm® Cortex®-A53, dual-core Cortex-R5F real-time processors, and a Mali™-400 MP2 graphics processing unit based on Xilinx's 16 nm FinFET+ programmable logic fabric. The board has a built-in Ethernet connection, an optical small form factor pluggable (SFP), and a PCIe for ideal fast communication with the Host computer. It can improve the resource allocation in the Zynq board, avoiding huge BRAMs allocations that free the space for more PEs that perform parallel pattern search. The ZCU102 board allows to have a bare metal execution of the code that is ideal for our application.

The software environment that surrounds the FPGA is based on Vivado and the Xilinx SDK, that jointly work for programming the

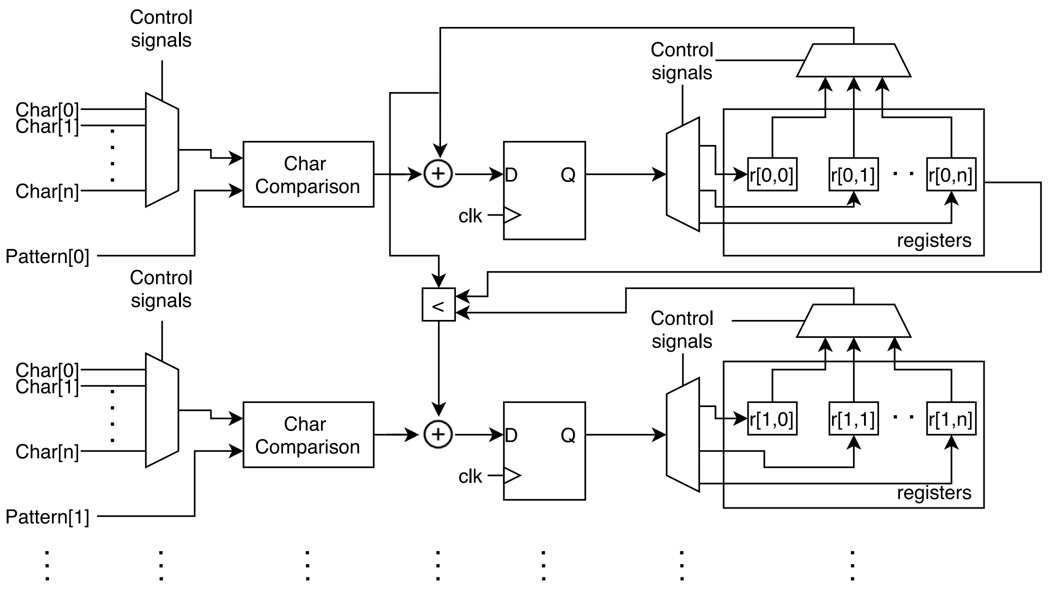


Fig. 10. First 2 processing elements inside the Edit distance computation block.

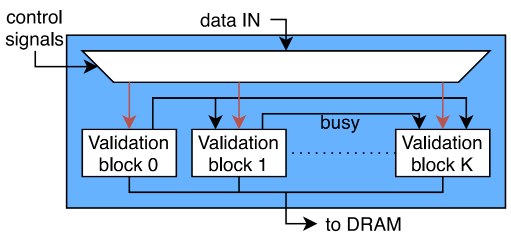


Fig. 11. Validation scheme composed of K + 1 blocks.

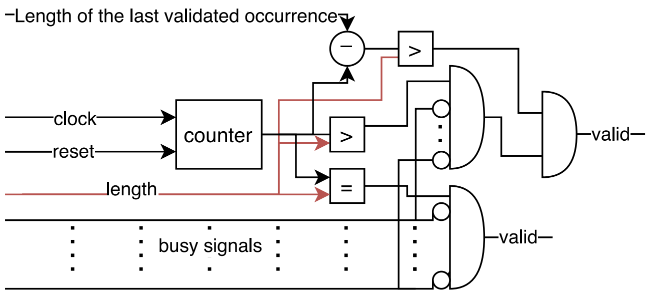


Fig. 12. Circuit diagram of a validation block.

Fig. 13. ZCU102 board.

Table 1
Specifications of the ZCU102 board.

I/O Pin Count	1156
Available IOBs	328
LUT Elements	274 080
FlipFlops	548 160
Block RAMs	912
DSPs	2520
Gb Transceivers	24

board. The Vivado 2018.2 software is the environment for the HDL part, where all the blocks are designed and placed. Moreover, Vivado synthesizes the bitstream, which is loaded onto the FPGA. The Xilinx SDK is required for the development of the code for the ARM processor, as well as for the serial communication and the debug. The Vivado High Level synthesis software builds a HDL block from the source C code, with user-defined optimizations (e.g., pipelining of loop functions, loop unrolling, BRAM interfaces, etc.). The tools also reports the usage of the FPGA resources.

The synthesized HDL is imported in Vivado, and an example of the synthesized layout is shown in Fig. 14. The board is connected to the computer through a PCIe interface. The PCIe interface in the Zynq Ultrascale+ can be composed of 4 lanes, works up to 5.0 Gb/s with a max payload size of 256 bytes. This allow the algorithm to fast transfer the data between the host and the ZCU102 DRAM. Hence, the algorithm can execute in real-time without bottlenecks.

4.2. ARM-based CPU implementation

The CPU implementation is developed in C code in the Xilinx SDK environment, and executed by the Arm® Cortex®-A53 processor without any optimization in bare metal. The computation of the edit distance with a double for loop, as shown in lines 9–10 of Algorithm 1, leads to a time-complexity of the algorithm of $O(|p||s|)$. The system configuration is illustrated in Fig. 15. The DNA string is loaded into the DRAM of the board from the host PC, then the algorithm is executed and the results are saved on a file. The computational execution time

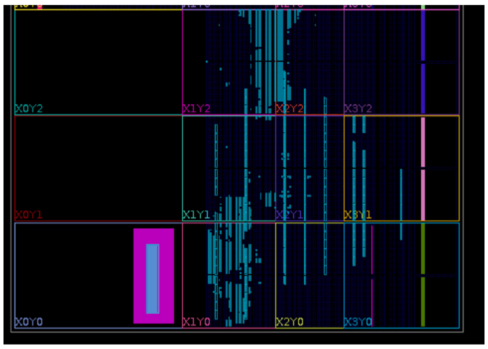


Fig. 14. Synthesized layout of the ISMatch architecture on the ZCU102 board.

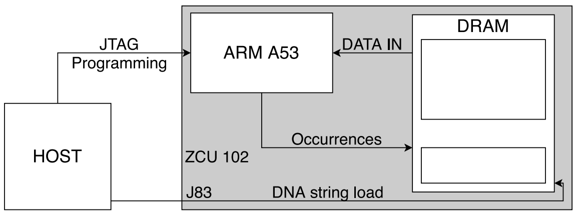


Fig. 15. ARM configuration.

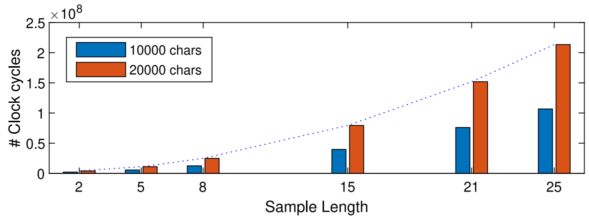


Fig. 16. Non-linear time increment with different sample lengths. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

is calculated with the counter value from the Physical Counter in the A53 core.

Fig. 16 shows the time increment w.r.t. the length of the sample. The clock cycles for computing a 10,000-characters file length is shown in blue, and the ones for computing a 20,000-characters file length in red. The implementation reveals that due to the complexity $O(|p||s|)$ of the algorithm there is a non-linear time increment with the length of the pattern, and a linear time increment with the length of the DNA string. This implementation highlights the inefficiency of the ARM-based execution of the algorithm, revealing that CPUs are not optimized for the target application. The algorithm itself is $O(|p||s|)$ in time and grows up fast with the length of the pattern. Computing a Levenshtein distance between a long pattern and a long window is extremely time-demanding and it constitutes the main bottleneck for the entire algorithm.

4.3. High level synthesis implementation

For overcoming the problem of the time demanding Levenshtein distance computation of the CPU implementation, a HDL project is designed with the aim of reducing the clock cycles to compute the Levenshtein distance. The HDL block that computes the Levenshtein distance is synthesized by the Vivado High Level Synthesis (HLS) tool, with different optimizations that can accelerate the computation and reduce the latency of the computation. The software synthesized a block that can be imported in Vivado and be interfaced with specialized blocks that controls its behavior. Fig. 17 illustrates the HLS block with all its signals. The hardware starts to compute the edit distance when the signal “start” is asserted to logic 1, and during the computation the signal “idle” is asserted to logic 0 to indicate that the design is operating. When the function is terminated, the signal “done” is asserted to 1 and the data “return” is valid. The Levenshtein algorithm is composed of a double for loop that provides the complexity of $O(|p||s|)$ in time. Hence, we study the impact of different optimization criteria on the HLS implementation, and compare them with the CPU and the custom accelerator. While the ARM-based implementation took 1200 clock cycles to compute the distance between two strings of 8-chars length, the HLS block without any optimization takes only 308 cycles to compute the distance between the same two strings. The specialized purpose of the HDL explains the 75% reduction of clock cycles latency. Table 2 its resource utilization.

The next step is to study the impact of the HLS-level optimizations on the performance and resource usage. The first optimization done in the HLS design is to pipeline the external loop. It leads to a latency of 162 clock cycles, which constitutes a 47,4% reduction compared to the not optimized block. This optimization increases the utilization of the LUT by 276% and of the FFs by 268%. Note that the increment of resources is not linear with the clock cycles decrease. The second optimization realizes a joint loop unrolling of the internal loop and pipeline of the external loop. Such a double optimization leads to a 37 clock cycles latency for a edit distance computation between strings of 8-chars length. Hence, compared to the ARM-based implementation, the speedup factor achieved by the HLS implementation is 32.43×.

Vivado HLS synthesizes the Levenshtein distance computation block that needs proper control signals to correctly execute. For the validation part, instead, the custom block described in Section 3.3 is developed in HDL and synthesized by Vivado. The validation block validates the occurrence and saves it into the BRAM. Since the ZCU102 board is interfaced with the host PC, all the occurrences are transferred from the board to the host. The Levenshtein distance block synthesized by Vivado HLS can be replicated for different patterns for a parallel search of different patterns.

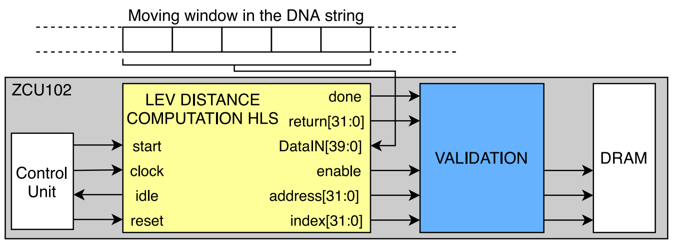


Fig. 17. Detailed HLS implementation and respective signals.

4.4. ISMatch architecture implementation in FPGA

Our proposed ISMatch architecture has been synthesized and implemented into the ZCU102 MPSoc. The Levenshtein distance computation block, which is the core of the design, computes the Levenshtein matrix with the wavefront approach, computing the elements of each anti-diagonal at the same time, leading to a time of execution of $|p|+|s|-1$. Since the window is set to have the same length of the pattern, the execution time is $2|p|-1$. Table 2 compares the ISMatch Levenshtein block resource usage configured with $|p|=8$, and the HLS Levenshtein block resource usage.

Due to the flexibility of the Vivado environment, the ISMatch is also synthesized for the Zedboard Zynq-7000 with the resource usage reported in Table 3. This table also compares the complete resource usage with all the control and validation blocks of the fastest HLS

Table 2
Resource usage of HLS and ISMatch Levenshtein blocks on the ZCU102.

	HLS without optimizations	HLS pipelining	HLS pipelining & loop unrolling	ISMatchL
FFs	144 (0.03%)	531 (0.10%)	3309 (0.60%)	220 (0.04%)
LUTs	628 (0.23%)	2363 (0.86%)	8456 (3.08%)	781 (0.28%)

Table 3
Resource usage of the complete system implemented on the ZCU102 and Zedboard implementation.

	HLS (ZCU102)	ISMatch (ZCU102)	ISMatch (Zedboard)
FFs	5815 (1.06%)	13 756 (2.5%)	2551(2.4%)
LUTs	7591 (2.77%)	9875 (3.6%)	1888 (3.55%)

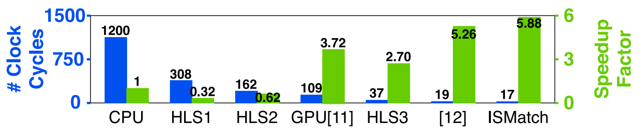


Fig. 18. In blue, clock cycles latency. In green, speedup factor normalized w.r.t. the CPU implementation. HLS1, HLS2, and HLS3 represent the HLS implementations without optimization, with pipelining, and with pipelining & loop unrolling, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Levenshtein block and the ISMatch implementation. The Zedboard implementation features less Logic Elements (LEs), mostly due to the less demanding DMA structure for the DDR3 of the board, compared with the DDR4-based DMA of the ZCU102.

4.5. Comparison results: Performance, resource usage, power

The Levenshtein algorithm evidences the performance benefit that a specialized hardware design can achieve with a strong data-dependency algorithm. This leads to different optimization aiming for a speedup. The ARM-based implementation of the Levenshtein algorithm highlights the need of a parallel computation with its 1200 clock cycles for computing the Levenshtein distance between two strings of 8 characters each. The fastest HLS implementation leads to a 2.70× speedup factor with a combined resource usage of 11 765 LEs. The ISMatch specialized hardware architecture leads to a 5.88× speedup factor with a combined resource usage of 1001 LEs. The Levenshtein algorithm is also implemented in [11] with a GPU optimization that reaches a speedup factor of 3.72× compared to the CPU performance.

The state-of-the-art designs of [12,31] do not analyze the speedup factor introduced w.r.t. a CPU implementation. Fig. 18 reports the clock cycles latency and speedup factors of the implementations proposed in this paper, compared to the works in [11,12] with a threshold of K = 2. The speedup factor takes into account the different clock frequencies of CPU, GPU and FPGA implementations, which are 1200 MHz, 405 MHz, and 100 MHz, respectively. Identifying good comparison metrics among systems implemented on different platforms is not trivial. For instance, the HLS3 implementation, i.e., with pipelining and loop unrolling, takes 37 clock cycles, while the GPU-based implementation in [11] requires 109 clock cycles. However, considering that different platforms have different operating clock frequencies, the latter has a higher speedup factor, which is 3.72×, compared to a value of 2.70× for the HLS3 implementation. Overall, our proposed custom ISMatch architectural design not only requires few clock cycles (only 17), but also achieves a speedup factor of 5.88× w.r.t. the CPU implementation.

Table 4
Power consumption for the ISMatch, HLS, and CPU implementations on the ZCU102.

		ISMatch		HLS		CPU	
D	Clocks	0.062 W	2.960 W	0.021 W	2.873 W	0.001 W	2.773 W
	Signals	0.032 W		0.017 W		0 W	
	Logic	0.093 W		0.012 W		0 W	
	BRAM	0.002 W		0.053 W		0 W	
	PS	2.770 W		2.770 W		2.772 W	
S	PL	0.624 W	0.723 W	0.625	0.724 W	0.623 W	0.722 W
	PS	0.099 W		0.099		0.099 W	

These values are also better than the state-of-the-art architecture proposed in [12], which measures 19 clock cycles and a speedup factor of 5.26×.

The resource usage of [31] is affected from the extensive use of the BRAM resources due to the BWT implementation. The reported resource usage of [31] is 112 384 LEs for computing the search of 20 patterns. The power consumption of the ISMatch, HLS, and CPU-based designs implemented on the Xilinx ZCU102 is listed in Table 4. The Processing System (PS) in the Dynamic part corresponds to the ARM processor power consumption. Clocks, Logic and Signals represent the power consumption of the ISM HDL part. While the static power (S) is dominated by the Programmable Logic (PL), the dynamic power (D) is dominated by the PS. The overall low power consumption is due to the essential PEs implemented in hardware. Note that, considering the performance improvement, our ISMatch design achieves higher energy-efficiency than the HLS and CPU-based designs.

4.6. Key observations deriving from the results

Based on the results obtained by different implementations, the following key observations can be noted:

- Our proposed ISMatch architecture achieves the *highest performance* among all the state-of-the-art designs and HLS implementations. E.g., the ISMatch design can compute the Levenshtein distance for a 8-characters string in 17 clock cycles, thereby enabling real-time execution of the ISM algorithm.
- The design can be configured for different hardware platforms and resources, and configured with different levels of parallelism.
- The ISMatch architecture implemented on the ZCU102 board executes the algorithm in a *power-efficient way, without wasting the hardware resources*. The whole system, composed of the ISMatch architecture, the BRAM and all the signals, consume less than 4 W, and use only 2.5% FFs and 3.6% LUTs.

5. Conclusion

In this paper, we propose ISMatch, the design of a hardware accelerator for the Inexact String Matching algorithm, and its implementation on the Xilinx Ultrascale+ FPGA. Due to its fast execution and connection with the Host, it can handle with high performance the streaming and comparison of long DNA sequences, thus enabling real-time DNA sequence matching. The degree of parallelism of the main computational block, which is the Processing Element that computes the Levenshtein distance, can be reconfigured for leveraging the tradeoff between performance and FPGA resource usage. Our design can be employed not only for DNA sequencing purposes, but also for similar applications like data processing [32] or document similarity [33]. Our ISMatch architecture has been compared with an ARM-based CPU implementation, a High Level Syntesis implementation, and the state-of-the-art designs of [11,12], showing up to 70× clock cycles reduction.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has been supported in part by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien. The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

References

[1] G.H. Gonnet, Some string matching problems from Bioinformatics which still need better solutions, J. Discrete Algorithms 2 (1) (2004) 3–15, http://dx.doi.org/10.1016/S1570-8667(03)00062-5.

[2] M. Bertini, A.D. Bimbo, W. Nunziati, Video clip matching using MPEG-7 descriptors and edit distance, in: H. Sundaram, M.R. Naphade, J.R. Smith, Y. Rui (Eds.), Image and Video Retrieval, 5th International Conference, CIVR 2006, Tempe, AZ, USA, July 13-15, 2006, Proceedings, in: Lecture Notes in Computer Science, 4071, Springer, 2006, pp. 133–142, http://dx.doi.org/10.1007/11788034_14.

[3] R. Typke, F. Wiering, R.C. Veltkamp, A survey of music information retrieval systems, in: ISMIR 2005, 6th International Conference on Music Information Retrieval, London, UK, 11-15 September 2005, Proceedings, 2005, pp. 153–160, URL http://ismir2005.ismir.net/proceedings/1020.pdf.

[4] K. Kukich, Techniques for automatically correcting words in text, ACM Comput. Surv. 24 (4) (1992) 377–439, http://dx.doi.org/10.1145/146370.146380.

[5] J. Schleif, Genetics and Molecular Biology, 2nd ed., Johns Hopkins Univ. Press, Baltimore, 1993, (10).

[6] S. Hakak, A. Kamsin, P. Shivakumara, G.A. Gilkar, W.Z. Khan, M. Imran, Exact string matching algorithms: Survey, issues, and future research directions, IEEE Access 7 (2019) 69614–69637, http://dx.doi.org/10.1109/ACCESS.2019.2914071.

[7] E.W. Sayers, M. Cavanaugh, K. Clark, K.D. Pruitt, C.L. Schoch, S.T. Sherry, I. Karsch-Mizrachi, GenBank, Nucleic Acids Res. 49 (Database-Issue) (2021) D92–D96, http://dx.doi.org/10.1093/nar/gkaa1023.

[8] J. Kawulok, Approximate string matching for searching DNA sequences, Int. J. Biosci., Biochem. Bioinf. (2013) 145–148.

[9] S. Neuburger, The Burrows-Wheeler transform: data compression, suffix arrays, and pattern matching by Donald Adjeroh, Timothy Bell and Amar Mukherjee Springer 2008, SIGACT News 41 (1) (2010) 21–24, http://dx.doi.org/10.1145/1753171.1753177.

[10] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Sov. Phys. Dokl. 10 (1965) 707–710.

[11] K. Balhaf, M. Shehab, W. Sarayrah, M. Al-Ayyoub, M. Al-Saleh, Y. Jararweh, Using GPUs to speed-up levenshtein edit distance computation, in: 2016 7th International Conference on Information and Communication Systems (ICICS), 2016, http://dx.doi.org/10.1109/IACS.2016.7476090.

[12] A. Cinti, F.M. Bianchi, A. Martino, A. Rizzi, A novel algorithm for online inexact string matching and its FPGA implementation, Cogn. Comput. 12 (2) (2020) 369–387, http://dx.doi.org/10.1007/s12559-019-09646-y.

[13] T. Smith, M. Waterman, Identification of common molecular subsequences, J. Mol. Biol. (1981) http://dx.doi.org/10.1016/0022-2836(81)90087-5.

[14] Y. Chen, J. Cong, J. Lei, P. Wei, A novel high-throughput acceleration engine for read alignment, in: 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015, IEEE Computer Society, 2015, pp. 199–202, http://dx.doi.org/10.1109/FCCM.2015.27.

[15] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, N. Sun, Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator, in: 2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012, 29 April - 1 May 2012, Toronto, Ontario, Canada, IEEE Computer Society, 2012, pp. 184–187, http://dx.doi.org/10.1109/FCCM.2012.39.

[16] Y. Turakhia, G. Bejerano, W.J. Dally, Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly, in: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 199–213, http://dx.doi.org/10.1145/3173162.3173193.

[17] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, C. Alkan, GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping, Bioinform 33 (21) (2017) 3355–3363, http://dx.doi.org/10.1093/bioinformatics/btx342.

[18] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, O. Mutlu, SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs, Bioinformatics 36 (22–23) (2021) 5282–5290, http://dx.doi.org/10.1093/bioinformatics/btaa1015.

[19] J. Arram, T. Kaplan, W. Luk, P. Jiang, Leveraging FPGAs for accelerating short read alignment, IEEE ACM Trans. Comput. Biol. Bioinf. 14 (3) (2017) 668–677, http://dx.doi.org/10.1109/TCBB.2016.2535385.

[20] L.D. Tucci, K. O'Brien, M. Blott, M.D. Santambrogio, Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL, in: D. Atienza, G.D. Natale (Eds.), Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017, IEEE, 2017, pp. 716–721, http://dx.doi.org/10.23919/DATE.2017.7927082.

[21] D. Castells-Rufas, S. Marco-Sola, J.C. Moure, Q. Aguado-Puig, A. Espinosa, FPGA acceleration of pre-alignment filters for short read mapping with HLS, IEEE Access 10 (2022) 22079–22100, http://dx.doi.org/10.1109/ACCESS.2022.3153032.

[22] P. Chen, C. Wang, X. Li, X. Zhou, Hardware acceleration for the banded Smith-Waterman algorithm with the cycled systolic array, in: 2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013, IEEE, 2013, pp. 480–481, http://dx.doi.org/10.1109/FPT.2013.6718421.

[23] X. Fei, Z. Dan, L. Lina, M. Xin, Z. Chunlei, FPGASW: Accelerating large-scale Smith–Waterman sequence alignment application with backtracking on FPGA linear systolic array, Interdiscipl. Sci.: Comput. Life Sci. 10 (2017) 176–188.

[24] S.S. Banerjee, M. El-Hadedy, J.B. Lim, Z.T. Kalbarczyk, D. Chen, S.S. Lumetta, R.K. Iyer, ASAP: accelerated short-read alignment on programmable hardware, IEEE Trans. Comput. 68 (3) (2019) 331–346, http://dx.doi.org/10.1109/TC.2018.2875733.

[25] J. Stuecheli, B. Blaner, C.R. Johns, M.S. Siegel, CAPI: a coherent accelerator processor interface, IBM J. Res. Dev. 59 (1) (2015) http://dx.doi.org/10.1147/JRD.2014.2380198.

[26] H. Li, Minimap2: pairwise alignment for nucleotide sequences, Bioinformatics 34 (18) (2018) 3094–3100, http://dx.doi.org/10.1093/bioinformatics/bty191.

[27] D.S. Cali, G.S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J.S. Kim, R. Ausavarungnirun, M. Alser, J. Gómez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, O. Mutlu, GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis, in: 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020, IEEE, 2020, pp. 951–966, http://dx.doi.org/10.1109/MICRO50266.2020.00081.

[28] A. Subramaniyan, J. Wadden, K. Goliya, N. Ozog, X. Wu, S. Narayanasamy, D.T. Blaauw, R. Das, Accelerated seeding for genome sequence alignment with enumerated radix trees, in: 48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021, IEEE, 2021, pp. 388–401, http://dx.doi.org/10.1109/ISCA52012.2021.00038.

[29] D.S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G.S. Kalsi, Z. Zuo, C. Firtina, M.B. Cavlak, J.S. Kim, N. Mansouri-Ghiasi, G. Singh, J. Gómez-Luna, N.A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, O. Mutlu, SeGraM: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping, in: V. Salapura, M. Zahran, F. Chong, L. Tang (Eds.), ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022, ACM, 2022, pp. 638–655, http://dx.doi.org/10.1145/3470496.3527436.

[30] T. Ho, S.-R. Oh, H. Kim, A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations, PLOS ONE 12 (10) (2017) 1–15, http://dx.doi.org/10.1371/journal.pone.0186251.

[31] Y. Xin, B. Liu, B. Min, W.X.Y. Li, R.C.C. Cheung, A.S. Fong, T. Chan, Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform, Microelectron. J. 44 (8) (2013) 670–682, http://dx.doi.org/10.1016/j.mejo.2013.05.004.

[32] Z.-p. Zhao, Z.-m. Yin, Q.-p. Wang, X.-z. Xu, H.-F. Jiang, An improved algorithm of Levenshtein Distance and its application in data processing: An improved algorithm of Levenshtein Distance and its application in data processing, J. Comput. Appl. 29 (2009) 424–426, http://dx.doi.org/10.3724/SP.J.1087.2009.00424.

[33] S. Rani, J. Singh, Enhancing Levenshtein's edit distance algorithm for evaluating document similarity, in: R. Sharma, A. Mantri, S. Dua (Eds.), Computing, Analytics and Networks, Springer Singapore, Singapore, 2018, pp. 72–80.

Alberto Marchisio received his B.Sc. and M.Sc. degrees in Electronic Engineering from Politecnico di Torino, Turin, Italy, in October 2015 and April 2018, respectively. Currently, he is Ph.D. Student at Computer Architecture and Robust Energy-Efficient Technologies (CARE-Tech.) lab, Institute of Computer Engineering, Technische Universität Wien (TU Wien), Vienna, Austria, under the supervision of Prof. Dr. Muhammad Shafique. His main research interests include hardware and software optimizations for machine learning, brain-inspired computing, VLSI architecture design, emerging computing technologies, robust design, and approximate computing for energy efficiency. He (co-)authored 20+ papers in prestigious international conferences and journals. He received the honorable mention at the Italian National Finals of Maths Olympic Games in 2012, and the Richard Newton Young Fellow Award in 2019.

Federico Teodonio received his B.Sc. degree in Electronic Engineering from "La Sapienza", University of Rome, Italy, in September 2017. He received his M.Sc. degree in Electronic Engineering, from "La Sapienza", University of Rome, Italy, in January 2021 after a 6 months thesis at the Technische Universität Wien (TU Wien), Vienna, Austria, under the supervision of Prof. Dr. Antonello Rizzi and Prof. Dr. Muhammad Shafique. Currently he is working in the Automotive industry for Maserati designing the Battery Electric Vehicles ECUs logic requirements.

Antonello Rizzi received the Ph.D. degree in information and communication engineering from the University of Rome "La Sapienza". In September 2000, he joined the Information and Communication Department, as an Assistant Professor. Since July 2010, he has been with the Department of Information Engineering, Electronics and Telecommunications (DIET), University of Rome "La Sapienza," where he currently serves as an Associate Professor. Since 2008, he has been the Scientific Coordinator and the Research and Development Technical Director of the Intelligent Systems Laboratory, Research Center for Sustainable Mobility of Lazio region, Italy. He is also working on smart grids and microgrids modeling and control, intelligent systems for sustainable mobility, battery management systems, granular computing, data mining and knowledge discovery, computational biology, machine learning in non-metric spaces, graph and sequence matching, agent-based clustering, and parallel and distributed computing. He has (co)authored more than 190 international journal/conference papers and book chapters. His major research interests include computational intelligence and pattern recognition, including supervised and unsupervised machine learning techniques, neural networks, fuzzy systems, and evolutionary algorithms. His research interest includes the design of automatic modeling systems, focusing on classification, clustering, function approximation, and prediction problems.

Muhammad Shafique received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2011. Afterward, he established and led a highly recognized research group at KIT for several years as well as conducted impactful collaborative R&D activities across the globe. In Oct.2016, he joined the Institute of Computer Engineering at the Faculty of Informatics, Technische Universität Wien (TU Wien), Vienna, Austria as a Full Professor of Computer Architecture and Robust, Energy-Efficient Technologies. Since Sep.2020, he is with the Division of Engineering, New York University Abu Dhabi (NYU-AD), United Arab Emirates, and is a Global Network faculty at the NYU Tandon School of Engineering, USA. He is the director of the eBrain research lab, and is also a Co-PI/Investigator in multiple NYUAD Centers, including Center of Artificial Intelligence and Robotics (CAIR), Center of Cyber Security (CCS), Center for InTeractIng urban nEtworkS (CITIES), and Center for Quantum and Topological Systems (CQTS). His research interests are in AI & machine learning hardware and system-level design, brain-inspired computing, autonomous systems, wearable healthcare, energy-efficient systems, robust computing, hardware security, emerging technologies, FPGAs, MPSoCs, and embedded systems. His research has a special focus on cross-layer analysis, modeling, design, and optimization of computing and memory systems. The researched technologies and tools are deployed in application use cases from Internet-of-Things (IoT), smart Cyber-Physical Systems (CPS), and ICT for Development (ICT4D) domains. Dr. Shafique has given several Keynotes, Invited Talks, and Tutorials, as well as organized many special sessions at premier venues. He has served as the PC Chair, General Chair, Track Chair, and PC member for several prestigious IEEE/ACM conferences. Dr. Shafique holds one U.S. patent has (co-)authored 6 Books, 15+ Book Chapters, 300+ papers in premier journals and conferences, and 50+ archive articles. He received the 2015 ACM/SIGDA Outstanding New Faculty Award, the AI 2000 Chip Technology Most Influential Scholar Award in 2020 and 2022, the ASPIRE AARE Research Excellence Award in 2021, six gold medals, and several best paper awards and nominations at prestigious conferences. He is a senior member of the IEEE and IEEE Signal Processing Society (SPS), and a member of the ACM, SIGARCH, SIGDA, SIGBED, and HIPEAC.