WFAsic: A High-Performance ASIC Accelerator for DNA Sequence Alignment on a RISC-V SoC

Abbas Haghi
abbas.haghi@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Lluc Alvarez
lluc.alvarez@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Jordi Fornt
jordi.fornt@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Juan Miguel de Haro Ruiz
juan.deharoruiz@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Roger Figueras
roger.figueras@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Max Doblas
max.doblas@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

Santiago Marco-Sola
santiago.marco@bsc.es
Barcelona Supercomputing Center
Universitat Autònoma de Barcelona
Barcelona, Spain

Miquel Moreto
miquel.moreto@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain

ABSTRACT

The ever-increasing yields in genome sequence data production pose a computational challenge to current genome sequence analysis tools, jeopardizing the future of personalized medicine. Leveraging hardware accelerators (GPUs, FPGAs, and ASICs) to accelerate computationally-intensive algorithms like sequence alignment has become paramount. Recently, the wavefront alignment algorithm was introduced, significantly reducing the execution time to perform sequence alignment. This paper presents the first-ever ASIC accelerator of the WFA integrated into a RISC-V system-on-chip. Our designed chip greatly accelerates sequence alignment, delivering up to 1076× better performance over the CPU implementation of the WFA running on the RISC-V core of the chip.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; **Hardware-software codesign**; **Application specific integrated circuits**; • **Computer systems organization** → **System on a chip**; • **Applied computing** → *Computational genomics.*

KEYWORDS

Genomics, WFA, Sequence Alignment, Co-design, ASIC, RISC-V

ACM Reference Format:
Abbas Haghi, Lluc Alvarez, Jordi Fornt, Juan Miguel de Haro Ruiz, Roger Figueras, Max Doblas, Santiago Marco-Sola, and Miquel Moreto. 2023. WFAsic: A High-Performance ASIC Accelerator for DNA Sequence Alignment on a RISC-V SoC. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605651

1 INTRODUCTION

Genome analysis has become the cornerstone of many advances in biology and medicine, such as personalized cancer treatment or early genetic disorder diagnosis and treatment. DNA sequencing technologies sample the DNA in contiguous genome *sequences* (also called *reads*) of varying lengths, from tens or hundreds of base pairs (bp) in Next-Generation Sequencing (NGS) technologies up to thousands of base pairs in third-generation sequencing technologies.

The first step in most DNA sequence analysis pipelines is to determine the location of each of the sequenced reads in the reference genome. This problem is known as *read mapping*. One of the main and computationally-intensive steps of read mapping algorithms, *pairwise read alignment*, is to compare and align two reads to identify regions of similarity or difference. Modern read mappers such as BLAST [2], GEM [16, 17], BWA-MEM [11] and Minimap2 [12], use variants of the Smith-Waterman (SW) algorithm [18] as their pairwise read alignment step. All these variants are based on dynamic programming (DP) and require $O(n^2)$ execution time and memory, proportional to the sequence length n. Hence, by increasing the sequence length, the computational and memory requirements of the SW algorithm increase drastically.

To overcome these limitations, the breakthrough WaveFront Alignment (WFA) algorithm has been proposed [15]. The WFA algorithm runs in $O(n \cdot s)$ time, proportional to the sequence length n and the error score s between sequences. To do so, the WFA uses

a novel approach that only computes a reduced number of the DP-matrix cells to find the optimal alignment. With this approach, since the error score is typically much smaller than the sequence length, the WFA algorithm is significantly faster than SW algorithms and scales much better with increasing sequence lengths.

Recently, the FPGA accelerator of the WFA algorithm is proposed [9]. This accelerator performs well for short DNA reads up to 300 bases, sequenced by next-generation sequencing technologies. However, it does not scale to support long reads, sequenced by third-generation sequencing technologies. Third-generation sequencing technologies are expected to dominate the sequencing market in the future. They provide reads of a few thousand bases, make DNA assembly easier, faster and more accurate. In addition, despite the fact that FPGAs are available in a variety of form factors, in high-performance systems they are often attached to high-end machines which are non-portable, consume excessive amounts of energy, and require frequent maintenance.

In this paper, we present WFAsic, the first ASIC accelerator for exact pairwise alignment of long reads based on the WFA algorithm. WFAsic supports DNA reads with lengths up to 10K bases, sequenced by third-generation sequencing technologies. When read length increases, the error rate also increases. Hence, we configure our WFAsic to support error rates of up to 10% of 10K-base reads. In other words, WFAsic is able to align pairs of sequences with as many as 1K differences. The biggest design of the WFA-FPGA supports maximum 16 differences.

In addition, we integrate our WFAsic accelerator in a Linux-capable RISC-V System-on-Chip (SoC). The WFAsic accelerator is configured using a standard Linux driver and API. It runs as an independent process in parallel to other CPU processes. Integrating the WFAsic accelerator with the CPU in the same SoC provides great benefits to genomics applications. It eliminates the need for external accelerators and their costly communication.

Our WFAsic accelerator fits in an area as small as 1.6mm^2, consumes an energy as low as 312mW, and reaches a frequency of 1.1GHz, after synthesis and Place and Route (PnR) in GlobalFoundries 22nm technology. Along with the RISC-V CPU core, it fits in a chip of a size of approximately 3mm^2, which is easily portable and could be supplied with batteries or other portable power supplies.

The integrated WFAsic accelerator provides performance improvements of up to 1076× compared to the CPU implementation of the WFA running on the RISC-V core of the chip.

This paper is organized as follows: Section 2 introduces the background on read alignment, the SW algorithm, and the WFA algorithm. Then, Sections 3 and 4, respectively, present the SoC architecture and the WFAsic accelerator. Section 5 evaluates our proposal, and Section 6 compares it with the related work. Finally, Section 7 remarks the main conclusions of this work.

2 BACKGROUND

2.1 Pairwise Read Alignment

Read mapping includes two main steps. First, the *Seeding* step filters the possible locations of the query sequences in the reference genome; then, the *seed extension* step performs the pairwise read alignment of the query sequences to the candidate locations of the reference genome. Pairwise read alignment identifies similarities

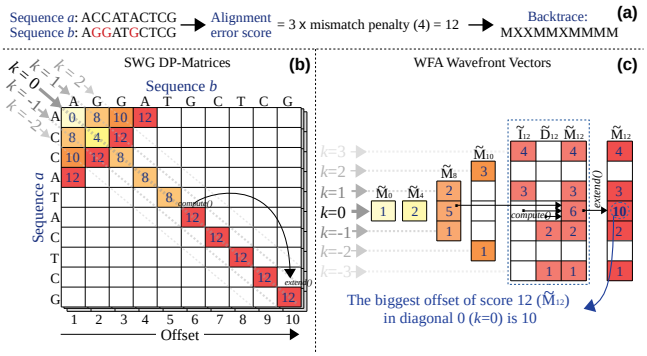

Figure 1: (a) A simple example of the concept of aligning 2 sequences, calculating error score, and performing backtrace. (b) The alignment of sequences by calculating SWG DP-matrices, highlighting only the cells that are computed by the WFA. (c) The alignment of sequences by calculating necessary wavefront vectors by the WFA. In this example penalties are $(x, o, e) = (4, 6, 2)$.

between the elements of a pair of sequences, revealing mutations, insertions or deletions of bases between them. The result of the pairwise read alignment is an alignment error score, where the more similar the sequences are, the lower the score is. Figure 1 (a) shows the concept of aligning sequences a_i and b_j, calculating error score, and performing backtrace. The alignment obtains the error score, while backtrace shows the differences and similarities between sequences, i.e., 'M' for matches, 'X' for mismatches/substitutions, 'I' for insertions and 'D' for deletions.

2.2 Smith-Waterman Algorithm

The SW algorithm is the most widely used approach for performing pairwise read alignment, which uses DP techniques. In this method, we need to calculate a DP-matrix, in which one of dimensions corresponds to sequence a (or query) and the other dimension corresponds to sequence b (or reference) as shown in Figure 1 (b). SW calculates each cell of the matrix using Equation 1. In this equation each gap (insertion or deletion) between sequences is penalized by g, where each mismatch is penalized by x.

$$H(i,j) = min \begin{cases} H(i-1,j-1) + \begin{cases} 0 & \text{if } a_i = b_j \\ x & \text{Otherwise} \end{cases} \\ H(i-1,j) + g \\ H(i,j-1) + g \end{cases} \quad (1)$$

To do the backtrace, when the value of each cell is calculated, the direction (top, left or diagonal) from which the value is achieved is stored in another matrix. Using this matrix, we trace back the directions from the last to the first cell. A diagonal direction determines a match or mismatch, while a horizontal/vertical direction determines an insertion/deletion.

The SW algorithm implements the *gap-linear* scoring model. The gap-linear scoring model does not differentiate the penalty of a gap-opening (first gap) from a gap-extension (continuous gap), that is, it penalizes a gap proportional to its length. However, in biological analysis, the penalty of a gap may not increase linearly with its length, and it is preferred that a gap-opening is penalized more

than a gap-extension following the first gap. To this end, the Smith-Waterman-Gotoh (SWG) algorithm [8] implements the *gap-affine* scoring model. This scoring model penalizes the gap-opening more than the gap-extension. So, it is highly preferred by biologists and is typically found in production genome analysis pipelines, even at the cost of a more complex implementation than the simpler gap-linear scoring model.

In the gap-affine scoring model there are three DP-matrices to be solved, M, I and D, which track the scores of alignments ending with a match/mismatch, an insertion, and a deletion, respectively. Equation 2 indicates how SWG calculates each matrix.

$$
\begin{aligned}
M(i,j) &= min \begin{cases} M(i-1,j-1) + \begin{cases} 0 & \text{if } a_i = b_j \\ x & \text{Otherwise} \end{cases} \\ D(i,j) \\ I(i,j) \end{cases} \\
I(i,j) &= min \begin{cases} M(i,j-1) + o + e \\ I(i,j-1) + e \end{cases} \\
D(i,j) &= min \begin{cases} M(i-1,j) + o + e \\ D(i-1,j) + e \end{cases}
\end{aligned}
\tag{2}
$$

where x, o and e are the penalties of mismatch, gap-opening and gap-extension, respectively. Please note that a first gap (opening) is penalized for both gap-opening and gap-extension, while a continuous gap (extension) is only penalized for gap-extension. In the gap-affine method, as the values of the cells of each matrix are calculated based on the values of the cells of the other matrices, the backtrace also has to potentially traverse up to three matrices. However, the concept remains the same as when using the gap-linear scoring model.

2.3 Wavefront Alignment Algorithm

The WFA algorithm is an exact gap-affine-based pairwise read alignment algorithm with identical results to the SWG algorithm. However, the WFA computes only a minimal number of cells of the DP-matrix to find the optimal alignment. This is done by proposing a different way of encoding the DP-matrix, as shown in Equation 3.

$$
\begin{aligned}
\widetilde{M}_{s,k} &= \max \begin{Bmatrix} \widetilde{M}_{s-x,k} + 1 & \text{(Substitution)} \\ \widetilde{I}_{s,k} & \text{(Insertion)} \\ \widetilde{D}_{s,k} & \text{(Deletion)} \end{Bmatrix} \\
\widetilde{I}_{s,k} &= \max \begin{Bmatrix} \widetilde{M}_{s-o-e,k-1} & \text{(Open Insertion)} \\ \widetilde{I}_{s-e,k-1} & \text{(Extend Insertion)} \end{Bmatrix} + 1 \\
\widetilde{D}_{s,k} &= \max \begin{Bmatrix} \widetilde{M}_{s-o-e,k+1} & \text{(Open Deletion)} \\ \widetilde{D}_{s-e,k+1} & \text{(Extend Deletion)} \end{Bmatrix}
\end{aligned}
\tag{3}
$$

with initial condition $\widetilde{M}_{0,0} = 0$

In Equation 3, x, o, e are penalties, s is the alignment error score, and k is the diagonal offset. The WFA algorithm computes three wavefront vectors $\widetilde{M}_{s,k}$, $\widetilde{D}_{s,k}$ and $\widetilde{I}_{s,k}$ for each score, tracking alignments that end with a match/mismatch, a deletion or an insertion, respectively. The vectors length increases as the score increases. Unlike the SWG, which computes all the cells of the DP-matrix and runs in $O(n^2)$ time, the WFA encodes the diagonal cells, progressively as the score increases, from the left-most column to the farthermost cell that has score s. So, it runs in $O(n \cdot s)$ time.

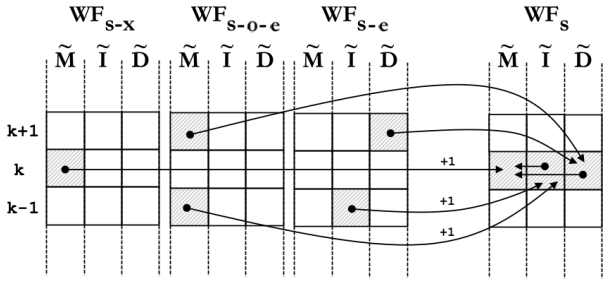

Figure 2: Dependencies between previous wavefronts to compute one element of the new wavefront [15].

Figure 1 (c) illustrates the alignment computation of sequences a and b with the WFA algorithm using wavefront vectors. For the sake of simplicity, the sequences of the example only include mismatches. However, insertions and deletions are also possible. Figure 1 (b) compares the number of cells that SWG calculates versus that of cells that WFA calculates. SWG computes all the cells in the matrix of Figure 1 (b), while WFA only computes the colored cells. WFA starts from score 0 and calculates only the cells which could have a score 0. Then it increases the score and calculates all possible cells with the new score. This process repeats until the end of alignment is reached. The positions of the cells with a specific score are kept in the wavefront vector of that score. For example, for score 8, the vector $\widetilde{M}_8$ holds the offsets 2, 5 and 1 for the diagonals k=1, k=0 and k=-1. This represents that, in the diagonal 1 (k=1) of the matrix, the cell with offset 2 has a score of 8. Similarly, the cells with score 8 in diagonals -1 and 0 are at offsets 1 and 5, respectively. Note that, in the diagonal 0, the offsets 3, 4 and 5 have cells with score 8, and the WFA only stores the biggest offset of a score in a specific diagonal.

The WFA algorithm has two main operators to perform the alignment: $extend()$ and $compute()$. First, $extend()$ compares the sequences for each diagonal cell from starting positions i and j in the DP-matrix until a mismatch is found and outputs the number of contiguous matching characters. The starting positions of the sequences, for each cell, are calculated regarding the offset (value) of the cell and its k index, according to Equation 4

$$
\begin{cases} i = offset - k & 0 \geq i \leq |seq_a| \\ j = offset & 0 \geq j \leq |seq_b| \end{cases}
\tag{4}
$$

In the example of Figure 1 (b,c), for score 12 and diagonal 0 ($\widetilde{M}_{12,0}$), both starting positions, i for sequence a and j for sequence b are 6, which result in four matching bases of 'CTCG' in the sequences.

After extending all the cells of the wavefront vectors, the $compute()$ operator computes the offsets of the next wavefront vectors based on Equation 3. Regarding Equation 3 computing wavefronts of a new score, WF_s, only depend on previously calculated wavefronts of scores $s-o-e$, $s-e$ and $s-x$, WF_{s-o-e}, WF_{s-e}, and WF_{s-x}, respectively. Figure 2 shows the dependencies between previously calculated wavefronts to compute one element of the new wavefront vectors of $\widetilde{M}$, $\widetilde{I}$ and $\widetilde{D}$.

The WFA iteratively performs $extend()$ and $compute()$ until a wavefront, with score s, reaches the end of both sequences. So, the final alignment score is s. After that, the $backtrace()$ operator is performed to obtain the differences between the sequences. This operator traces all the cells back from the cell ($n = |seq_a|$, $m = |seq_b|$) that gave the optimal alignment score to cell (0, 0) or the initial wavefront $\widetilde{M}_{0,0} = 0$. This is done by looking at the values that Equation 3 has generated for each cell towards the final alignment score.

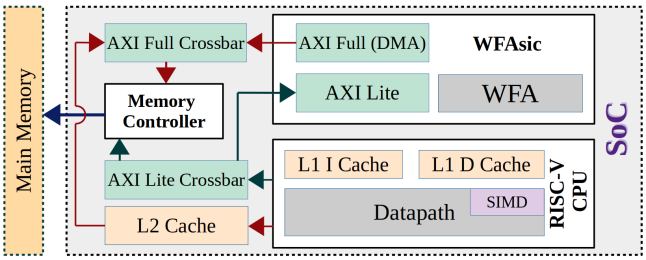

Figure 3: SoC architecture including the RISC-V CPU, the WFAsic accelerator and their connections.

3 SYSTEM-ON-CHIP ARCHITECTURE

The architecture of the SoC, including the WFAsic accelerator, the CPU and the intra-chip connections, is illustrated in Figure 3. The CPU communicates with the WFAsic accelerator through the AXI-Lite bus. The WFAsic accelerator includes a set of memory-mapped registers, and the CPU writes into these registers the configuration of the accelerator. The configuration includes the backtrace functionality (enabled or disabled), the maximum sequence length of the input set, and the DMA configurations, which consist of the address and the size of the input set in the main memory and the address where results should be written to the main memory. The WFAsic accelerator also has two registers, *Start* and *Idle*, that communicate with the CPU through the AXI-Lite bus. The CPU triggers the start of the accelerator by writing to the Start register, and it checks the completion of the computation in the accelerator by polling the Idle register. A dedicated interrupt could also be enabled to signal the job completion to the CPU.

The WFAsic accelerator has direct access to the off-chip main memory through the memory controller via the AXI-Full bus. In contrast, the CPU can access the main memory in two different ways: (1) via the AXI-Lite bus and the memory controller, and (2) via the AXI-Full bus, the L2 cache, and the memory controller.

The processor used in the SoC is Sargantana [19], a 64-bit in-order Linux-capable RISC-V CPU that implements the RV64G ISA. It uses a Single Instruction Multiple Data (SIMD) unit to accelerate domain-specific applications and supports the vector instructions defined in the vector extension RVV 0.7.1. The CPU has a 7-stage pipeline that implements register renaming, out-of-order write-back, and a non-blocking memory pipeline. It has two first level caches: an instruction cache of 16KB, and a non-blocking data cache of 32KB. The system also has a 512KB L2 cache outside the CPU. Sargantana fits in an area of 1.37mm^2 and reaches a frequency of 1.26GHz.

3.1 Co-design Scheme

Computing the alignment of DNA sequences using WFAsic is done in a co-designed manner, as shown in Figure 4. First, the CPU parses the input data and stores them in the main memory. Then, the WFAsic accelerator reads the sequences, and computes the alignments. The alignment is composed of two sequential steps, first calculating the alignment score, by iteratively performing extend and compute operations, and then performing the backtrace. The computation of alignment is done in WFAsic, while backtrace is performed in the CPU. This is because storing backtrace data of long reads requires a huge amount of on-chip memory, which is not available. Therefore, the backtrace data are generated in WFAsic, and as generated, are sent to the main memory. The alignment score is sent to the main memory at the end of each alignment. When WFAsic finishes the alignment of all inputs, the CPU checks the alignment results and performs the backtrace.

4 WFASIC ACCELERATOR

The WFAsic accelerator proposes a design for an SoC implementation with the support of long reads of 10K bases, targeting the alignment of sequences generated with third-generation sequencing technologies. WFAsic is able

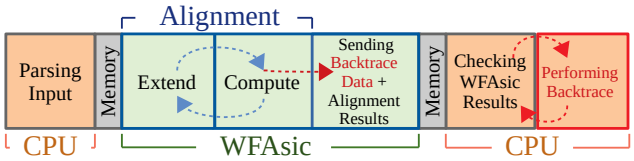

Figure 4: Steps in the WFAsic co-designed accelerator.

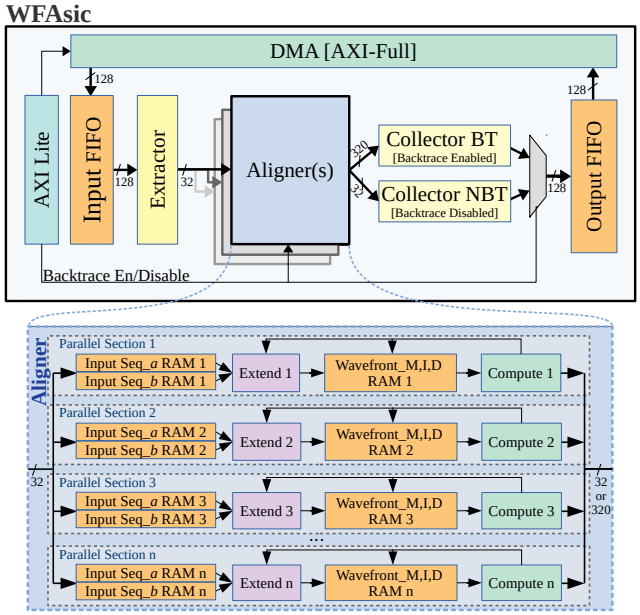

Figure 5: Scheme of the WFAsic accelerator architecture and its Aligner.

to align pairs of sequences with error scores up to 8K between them. For WFAsic to be able to align a pair of sequences accurately, the number of mismatches (num_x), gap-openings (num_o) and gap-extensions (num_e) between sequences should satisfy Equation 5.

$$8000 \geq num_x \times 4 + num_o \times (6+2) + num_e \times 2 \quad (5)$$

where 4, 6 and 2 are typical penalties of mismatch, gap-opening and gap-extension, respectively. Assuming worst case scenario in which all differences between sequences are gap-openings, WFAsic can detect up to 1K differences between each pair of sequences. We use the same penalties of x = 4, o = 6, e = 2 in the examples of this section.

4.1 WFAsic Structure

Figure 5 shows a diagram of the WFAsic accelerator. Depending on the available area and resources, the accelerator can include multiple *Aligner* modules to align sequences in parallel. The *DMA*, reads data from memory and stores them in the *Input FIFO*. The data width of the AXI-Full in this SoC is 16 bytes. Hence the width of input and output FIFOs is 16 bytes. The *Extractor* module extracts the input DNA reads and distributes them among different Aligners. The results of the Aligners are collected by the *Collector* module(s), which are written to the main memory through the *Output FIFO* and the DMA.

Transferring huge amount of backtrace data, i.e., 10MB for each pair of 10K bases reads with 10% error rate, may limit the performance of WFAsic. Hence, in order to be able to evaluate the accelerator design without being limited by memory-accelerator bandwidth, we add an option to enable and disable the backtrace functionality. Disabling backtrace prevents us

from transferring a huge amount of data from the accelerator to the memory. Therefore, if the backtrace is disabled, the alignment scores are only computed. Otherwise, the backtrace data are also generated.

4.2 Extractor Module

The Extractor module monitors the activity of the Aligner modules and, when one of them becomes idle, it starts extracting data of a pair of sequences from Input FIFO and passing it to the idle Aligner. For a pair of sequences, the Extractor module reads 16 bytes of input data at each clock cycle, decodes them, compacts them and writes them to the *Input_Seq* RAMs of the idle Aligner, as soon as it reads the data. This data includes, alignment ID which is unique for each pair of sequences, the length of sequence *a*, the length of sequence *b*, sequence *a* bases (characters), and sequence *b* bases. All these data types, on main memory, are stored in memory sections of 16 bytes. Sequence bases include multiple 16-byte memory sections.

The Extractor module defines a configurable maximum read length, and it can process reads with any length as long as they do not surpass the maximum length (10K bases in the current WFAsic implementation). To do so, the CPU defines a *MAX_READ_LEN* for the input set and sends it to the accelerator via the AXI-Lite bus. The MAX_READ_LEN must be divisible by the data width of the AXI-Full (16 bytes). For example, if the longest sequence in the input set has a length of 9010 bases, the MAX_READ_LEN is set to 9024 bases and the extra 14 bases are filled by dummy bases in the CPU. Dummy base padding is applied to all the sequences of the input set, and the Extractor module ignores the dummy bases when it reads them. Dummy bases are detectable from the lengths of the sequences.

The two sequences (*a* and *b*) are stored in separate Input_Seq RAMs, since parallel accesses to the two sequences are required during the processing of the alignment (see Figure 5 (bottom)). In addition, as will be explained in the next section, the sequences are replicated multiple times. Each sequence is stored in its Input_Seq RAMs using the following format: alignment ID (four bytes), sequence length (four bytes), sequence bases (MAX_READ_LEN bytes). When reading sequence bases, the Extractor module maps each base of one byte to two bits, so the blocks of 16 bases fit in four bytes, instead of 16 bytes. Hence, the width of the Input_Seq RAMs is four bytes, and the depth is at least 627 words (10K of max supporting read length / 16 bases in each RAM row + 2 4-byte words of ID and length). Alignment ID is stored in address 0, length in address 1, and sequence bases from address 2 onward.

In this phase, the Extractor module is also in charge of detecting two types of unsupported reads: those with a length longer than MAX_READ_LEN, and those including 'N' (unknown) bases. If an unsupported read is detected, the Extractor module signals the corresponding Aligner to ignore the sequences already stored in the Input_Seq RAMs of that Aligner. Then the Aligner does not process the alignment and sets the *Success* flag of the alignment to zero. This flag is sent to the CPU along with the results to determine if the alignment of a pair of sequences has succeeded. The alignment results also include an alignment ID, which determines each result belongs to which pair of sequences.

4.3 Aligner Module

The Aligner module is the main module that performs the sequence alignment. It contains two main sub-modules, *Extend* and *Compute*. The number of these sub-modules is configurable. Each set of Extend and Compute sub-modules processes one cell of the wavefront vectors at a time. Since multiple of these sub-modules work in parallel they need parallel access to both input sequences and wavefront data. So, each Aligner replicates input sequences in multiple Input_Seq RAMs, one per each set of sub-modules. However, it distributes the wavefront data (vectors) among the Wavefront RAMs. The Extend and Compute sub-modules are pipelined and internally parallelized. As shown in Figure 5 (bottom), one set of Extend and Compute sub-modules along with their dedicated RAMs constitute one *parallel section*.

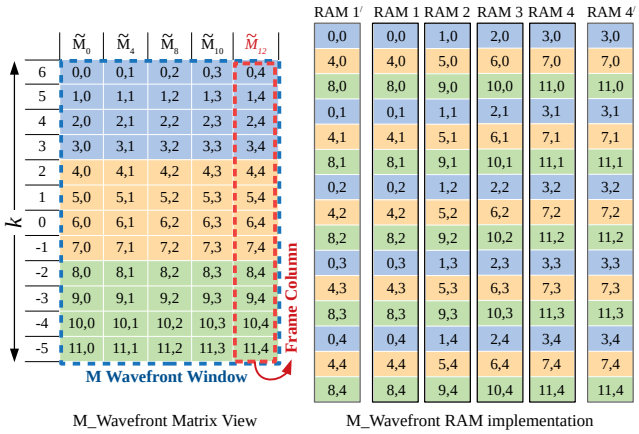

Figure 6: Mapping $\widetilde{M}$ wavefront vectors into RAMs.

4.3.1 Wavefront Vectors Implementation. WFAsic stores the necessary wavefront vectors, described in Section 2, in Wavefront RAMs. Regarding Figure 1 (c), only for some scores wavefront vectors are generated, i.e., 0, 4, 8, 10, 12, 14, and so on. In addition, regarding Equation 3 and Figure 2, for calculating the wavefront vector of a new score, only 4, 1 and 1 previous wavefront vectors of $\widetilde{M}$, $\widetilde{I}$ and $\widetilde{D}$ are respectively required. Hence, in the hardware, we only keep those necessary wavefront vectors.

The length of each wavefront vector increases with the score. However, in the hardware, dynamic memory allocation is impossible. So, we allocate the same length to all wavefront vectors and limit it to a design parameter called k_{max} (k as in Equation 3). As the score increases, the wavefront vectors expand the length from both sides to include the data of more diagonals or ks (see Figures 1 (b,c)). Therefore by limiting the vectors length to a threshold k, we limit the maximum supporting score of our design. This score is calculated by Equation 6. If the error score between the pairs of sequences passes this score, the alignment in the WFAsic remains incomplete and is terminated. Then the corresponding Aligner sets the Success flag of that alignment to zero.

$$Score_{max} = k_{max} \times 2 + 4 \tag{6}$$

Figure 6 (left) unifies in a matrix, the necessary $\widetilde{M}$ wavefront vectors for calculating the new wavefront vector (of $\widetilde{M}_{12}$ in the example). We call this matrix the $\widetilde{M}$ *wavefront matrix*, and the wavefront vector that is being calculated the *frame column*. Each column of the wavefront matrix represents a wavefront vector. Although the length of the matrix columns is $2{\times}k_{max}$, the actual length of each vector is smaller. Hence, some cells of each column are invalid. The corresponding score of a column identifies the valid cells of that column. For example, for score 8, only cells k = -1 to k = 1 are valid (see Figures 1 (c)). Depending on the score, the design only processes the valid cells of each column. Columns are initialized by negative values. Since invalid cells are never processed, remain negative.

After calculating the frame column, instead of moving all data, we just move the frame column to the right where the data of that column is not needed any more. If the frame column is on the right-most column, we move it to column 0. In the example of Figure 6 (left), after calculating the cells of the frame column which is tagged as $\widetilde{M}_{12}$, we move the frame column to column 0 and tag it as $\widetilde{M}_{14}$ (instead of $\widetilde{M}_0$ which is not needed anymore).

In order to increase parallelism, we process multiple cells in parallel. Multiple parallel sections process the same colored cells of the frame column in Figure 6 (left) at the same time, so parallel writes to those cells are needed. Moreover, computing frame column cells requires parallel data readings from previous columns cells (regarding Equation 3). Hence, we distribute the wavefront window among multiple RAMs to have parallel accesses

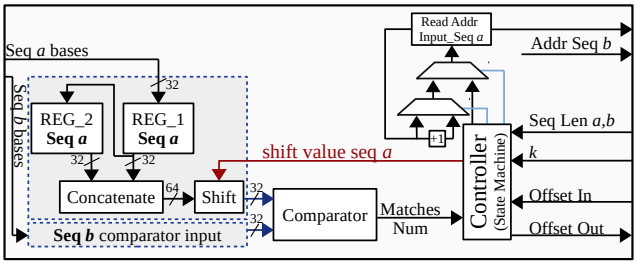

Figure 7: Structure of the Extend sub-module.

to the required cells. Therefore, each parallel section can access its own Wavefront RAM, without blocking other parallel sections RAM accesses.

Figure 6 (right) shows how $\widetilde{M}$ wavefront window is distributed among multiple Wavefront RAMs. For clarity, each cell of the window contains its coordinates, showing where they are stored in the Wavefront RAMs.

The $compute()$ operation described in Section 2, to compute the wavefront vector of a new score, requires three accesses to the data of $\widetilde{M}$ wavefront window and one access to each of $\widetilde{I}$ and $\widetilde{D}$ wavefront windows (see Figure 2). According to the penalties we use in our design, and the example of Figure 6 (left) to compute the wavefront vector $\widetilde{M}_{12}$ which is in the column 4 of the $\widetilde{M}$ wavefront matrix, we need one access for reading data from column 2 ($\widetilde{M}_8$) and two accesses for reading data from column 0 ($\widetilde{M}_0$). Hence, we distribute wavefront data among Wavefront RAMs in a way that enables parallel accesses to the cells of each column that have the same color, as shown in Figure 6 (right). In this example, four RAMs are required for each wavefront window ($\widetilde{M}$, $\widetilde{I}$, $\widetilde{D}$) to compute four cells of the same color in parallel. In addition, only for the $\widetilde{M}$ wavefront window we duplicate the first and the last RAMs (RAM $1'$ and RAM $4'$). This is because, for calculating the orange-colored cells of the frame column (cells (4:7,4)) in parallel, we require parallel readings from cells (3:8,0), however, cells (3,0) and (7,0) are both stored in RAM 4, and cells (4,0) and (8,0) are stored in RAM 1. So, duplicating RAM 1 and RAM 4 enables parallel readings from all needed cells of column 0. Moreover, to compute the cells (4:7,4) of the frame column, parallel accesses to the cells (4:7,2) are also required. We do not replicate RAMs to access data in column 0 and column 2 of the $\widetilde{M}$ wavefront window in parallel. Instead, we access the $\widetilde{M}$ Wavefront RAMs in two sequential accesses (one for column 0 and one for column 2).

We use the same data distribution method to access the $\widetilde{I}$ and $\widetilde{D}$ wavefronts windows, without replicating RAMs as only one access to these wavefront windows is required for computing the frame column. The accesses to the $\widetilde{I}$ and $\widetilde{D}$ wavefront windows happen in parallel with the accesses to the $\widetilde{M}$ wavefront window.

4.3.2 Extend Sub-module. The Extend sub-module compares the bases of the two sequences, starting from a position (base) in sequence a which may be different from the starting position of sequence b. The architecture of Extend sub-module is shown in Figure 7.

The Extend sub-module receives the offset (value) of a cell of the frame column, its k position and a start signal. From these inputs, the Extend sub-module calculates the starting positions in sequence a and sequence b, and calculates the addresses of Input_Seq RAMs a and b in which the bases of the starting positions are stored.

As the width of the Input_Seq RAMs is 32 bits (16×2 bits per base), and to increase the speed of the design, the comparison is done in blocks of 16 bases. In the WFA algorithm, the starting bases of the sequences could be at any position from 0 up to the length of the sequence. Hence, it is likely that the starting positions of the sequences are not at the boundaries of the blocks of 16 bases. So, two blocks of 16 bases of each sequence are required to start the comparison.

The Extend sub-module, at each clock cycle, sends read requests to each of its Input_Seq RAMs (RAMs a and b) starting from the address which holds the starting position, and increasing the address by one. The received blocks of sequences a and b are stored in 32-bit registers of REG_1 of sequence a and REG_1 of sequence b, respectively. At each clock cycle, the value of these registers is shifted in two other registers, REG_2 of sequence a and REG_2 of sequence b, and their values are overwritten by the new values from Input_Seq RAMs. When both registers of the sequences have valid bases, the value of both registers of each sequence are concatenated in 64 bits and shifted to the starting position of each sequence. This means the starting base, positions in the left-most position of the 32-bit comparator input and the unneeded bases are truncated.

The design is pipelined in such a way that the comparator compares 16 bases of the sequences at each clock cycle, after five initial cycles. The Extend sub-module compares sequences until a mismatch is found, or one of the sequences reaches the end. Then it returns the new value for the cell, which is written in the corresponding $\widetilde{M}$ Wavefront RAM.

The Extend sub-modules also require information about the lengths of the sequences. The lengths of the sequences are read from the address 1 of any of Input_Seq RAMs of sequences a and b at the beginning of the alignment by the Aligner and are provided to each Extend sub-module.

4.3.3 Compute Sub-module. The Compute sub-module computes the values of the frame column as expressed in Equation 3. The computed offsets of the $\widetilde{M}$ wavefront are buffered to be extended by the Extend sub-modules. However, the computed offsets of the $\widetilde{I}$ and $\widetilde{D}$ wavefronts are written in the $\widetilde{I}$ and $\widetilde{D}$ Wavefront RAMs.

If the backtrace is enabled, this sub-module also tracks the origin of each computed cell. As shown in Equation 3 and Figure 2, the origin of a cell in the $\widetilde{I}$, $\widetilde{D}$, and $\widetilde{M}$ wavefront matrices can come from 2, 2 and 5 positions, respectively, so we need 1, 1 and 3 bits to store them. At the end of the compute step, the origin of each computed cell is concatenated into five bits. Then, the origins of all the cells computed in parallel, i.e., parallel sections, are concatenated and provided to the *Collector BT* module. In the design of WFAsic, the number of parallel sections is 64. So, the Compute sub-module generates backtrace data in blocks of 320 bits (5×64).

4.4 Collector Module

We design two Collector modules with different input widths to handle backtrace functionality. Collector BT is activated when backtrace is enabled and *Collector NBT* is activated when it is disabled.

If backtrace is enabled, the Aligner provides backtrace data in blocks of 40 bytes (320 bits), as explained in the previous section. However, the output data width is 16 bytes. Hence, each output data of the Aligner should be divided by 16 bytes and sent in multiple memory transactions.

One block of backtrace data fits in three 16-byte memory transactions. However, when we divide it to be sent in multiple transactions, each part of the divided data requires attached information to be identifiable later in the CPU. Therefore, in each transaction, we combine 10 bytes of the backtrace data with six bytes of information in one block of 16 bytes, and send each backtrace data in four memory transactions. The attached information includes a counter of the block (three bytes), the *Last* flag (one bit) and the alignment ID (23 bits).

When backtrace is enabled, the last data that the Aligner provides to the Controller BT is the alignment score. This last data, in the CPU is detected from the Last flag. The alignment score is sent to the Controller BT in 40 bytes, but only five bytes of it are useful. These five bytes include the Success flag in one byte, the k that the alignment reaches in two bytes, and the alignment score in two bytes. This information is sent to the memory in one memory transaction.

If the backtrace is disabled, the Aligner only provides the alignment score to the Controller NBT in four bytes. These four bytes include the Success flag in one bit, the alignment score in 15 bits, and the alignment ID in two bytes. Controller NBT merges the alignment results of four alignments

and sends them to the memory in one transaction. Controller NBT does not attach any extra information to this data as the Aligner adds all the necessary information in four bytes, as explained. This way, the design is less limited by the accelerator-memory bandwidth.

4.5 Backtrace in CPU

If the backtrace is enabled, the CPU computes it when the alignment in the accelerator is finished. The backtrace operation starts from the backtrace data of the last cell which determines the alignment score. The backtrace data are divided and stored in groups of 10 bytes with information data of six bytes between them. So the CPU code should correctly handle the gaps between backtrace data.

The CPU code decodes the origin of the cell, and based on the five bits that encode the origin in the backtrace data, it determines if there is a mismatch, insertion or deletion. Then, it reads the backtrace data of the origin and iteratively repeats the process until it reaches the first address of the backtrace data. This process identifies all the differences between two sequences, however, the position of each difference is unknown. In particular, it does not identify how many matches there are between each difference. To find the exact position of each difference, the CPU traverses the two sequences and inserts all the necessary matches between the differences.

As mentioned earlier, there might be multiple Aligners in the design. Then the backtrace data of each alignment is not consecutively written in the memory. The backtrace data of all alignments are distributed among the memory based on how the Controller BT schedules them. In this case, to do the backtrace, first, the data of each alignment should be identified based on the attached information and moved and written consecutively in another part of the memory specific to that alignment ID. Then the backtrace could be performed on that data.

Since memory bandwidth might be limited, the process of separating data on a single-core CPU would be time-consuming. However, if there is only one Aligner in the accelerator, the data separation is unnecessary, as the backtrace data of different alignments are written consecutively in the memory. The only important matter is to determine the data boundaries of each alignment. We implement a method that identifies these boundaries and performs the backtrace of each alignment. The CPU code of the co-designed accelerator includes both single-Aligner and multi-Aligner backtrace computation methods.

4.6 ASIC Memory Implementation

To test the functionality of our design, first, we implement our design on the FPGA and use Vivado IP-cores to implement input and output FIFOs, and Input_Seq and Wavefront RAMs. However, in the ASIC design, these memories are implemented using GlobalFoundries memory macros.

In the FPGA prototype, we implement input and output FIFOs as *show ahead* FIFOs, in which the last unread data is available at the output port of the FIFO and is cleared by triggering the read request signal of the FIFO. These two FIFOs are the biggest memories in our design, with a width of 16 bytes and a depth of 256 words. To implement these FIFOs in ASIC, we have used high-performance dual port register files. To do this, we create a wrapper for these memories, which handles the internal pointers and read/write procedures to mimic the functionality of a show ahead FIFO for other modules (DMA, Extractor, and Collector modules). Hence, the interactions of the modules with the input/output memories remain the same as in the FPGA prototype.

Depending on the design configuration, each Aligner could have multiple sets of Extend and Compute sub-modules, called parallel sections. The number of parallel sections determines the number of Input_Seq and Wavefront RAMs of the Aligner. These RAMs in the design of the FPGA prototype are implemented as dual port RAMs. One port for writing and another independent port for reading. The number of each type of these RAMs, i.e., Input_Seq *a,b* and, Wavefront_M,I,D, for a design of long reads, is more than 16, and 64 in our case. Hence, there are a lot of dual port RAMs in our design that should be replaced by memory macros. The large number of RAMs can reduce the frequency of the ASIC due to complicated routing and their maximum frequency limitations. To reach higher frequencies for our WFAsic accelerator, we choose the memory macros that can achieve the highest frequency, which are high performance single port memory macros. In addition, to decrease the number of RAMs, we merge the data of $\widetilde{I}$ and $\widetilde{D}$ wavefronts in Wavefront_I/D RAMs. In this case, again, to avoid changing the interaction protocols between RAMs and other modules of the FPGA prototype, we design a wrapper that handles pointers and read/write procedures of a single port memory, but from the perspective of other modules, it looks like a dual port RAM. Also, we ensure that read and write requests to a RAM are not triggered simultaneously in the ASIC design.

5 EVALUATION

This work is part of a bigger project with actual chip production. We configure our accelerator according to the area budget dedicated to the WFAsic. The characteristics of the WFAsic implementation are summarized below.

- WFAsic aligns DNA reads of lengths up to 10K bases with error rates of up to 10%.
- WFAsic contains one Aligner module which aligns one pair of sequences at a time.
- The Aligner module is able to process 64 cells of wavefront vectors in parallel. In other words, it includes 64 parallel sections.
- The backtrace functionality could be enabled or disabled.
- An interrupt could be programmed to be generated at the end of the accelerator job.
- The data width of the WFAsic DMA is set to 16 bytes.

5.1 WFA Verification

For verification, we follow the below steps:

- We test the functionality of the WFAsic design in the SoC with the CPU core using an FPGA prototype.
- We use the conformal Logic Equivalence Checker (LEC) tool from Cadence to ensure the WFAsic post-synthesis and post-PnR netlists are equivalent to the WFAsic RTL design.
- We perform post-synthesis and post-PnR Gate Level Simulations (GLS) using Xcelium from Cadence.

We test our WFAsic design on the FPGA prototype with six different input sets of different characteristics, as shown in Table 1. We also test the functionality of the backtrace by enabling and disabling it for all input sets. The CPU code includes a self-checking mechanism for alignment scores, which reports the number of failed alignments and the expected values. In addition, to check that the WFAsic does not cause the CPU to hang in case of receiving broken data, we intentionally send data in different unexpected formats to the WFAsic. In these tests, we did not observe any CPU freeze. Moreover, although the WFAsic is configured with one Aligner and 64 parallel sections, we test the WFAsic with other configurations and with more Aligners, as the FPGA has more available resources.

To both increase the speed of GLS and thoroughly test the WFAsic accelerator in the whole SoC, we replace the WFAsic design with its post-synthesis and post-PnR netlists, while keeping the rest of the SoC as the RTL design. In GLS tests, we use a less number of inputs compared to the FPGA prototype tests. In all combinations, we check the backtrace functionality.

5.2 ASIC Synthesis and Place and Route

We synthesize the SoC using the Genus tool from Cadence with the GlobalFoundries 22nm Fully-Depleted Silicon-On-Insulator technology (GF22FDX). We use Synopsys Standard Cells libraries in GF22FDX technology for 8-track platform. The place and route is performed using the Innovus tool from Cadence using eight metal layers. We use the reports of the synthesis and

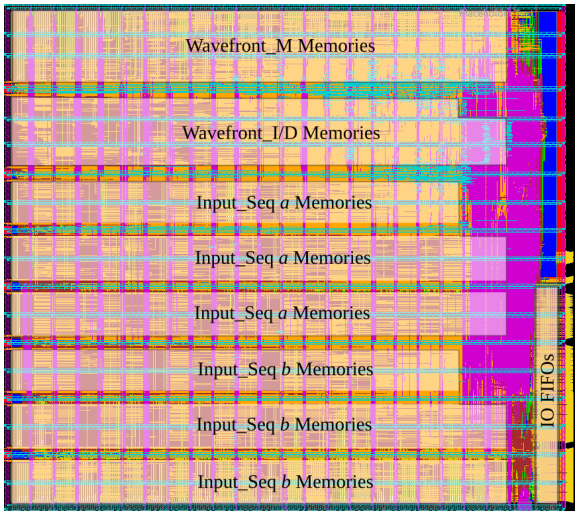

Figure 8: Accelerator layout. The size is 1330um×1200um with all the connectivity on the right side.

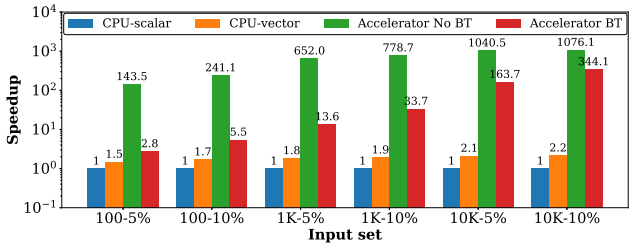

Figure 9: Speedup with respect to the CPU-scalar code.

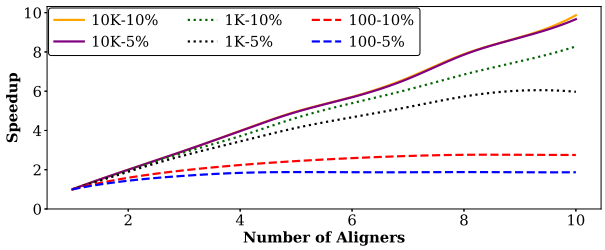

Figure 10: Scalability with increasing number of Aligners.

the PnR tools to measure area and frequency. Power estimation is performed with Cadence Voltus on the post-PnR netlist with annotated activations from the GLS simulations.

In the post-synthesis netlist, the WFAsic accelerator reaches a frequency of 1.5GHz and requires an area of 1.107mm^2 (1.057mm^2 is cell area and 0.05mm^2 is net area). In the post-PnR netlist, the WFAsic accelerator reaches a frequency of 1.1GHz in typical corner with 0.8V supply and at 85^{o}C, and it has a power consumption of 312mW. Figure 8 shows the layout of the WFAsic accelerator in the GF22FDX technology. The WFAsic accelerator occupies an area of 1.6mm^2 and uses 0.48MB of memory. The memories are implemented as register file memory macros. There are 260 memory macros that occupy 85% of the area (see Figure 8).

5.3 FPGA Prototype Performance Results

We have emulated the whole SoC using FPGA prototyping on an Alveo U280 FPGA board connected to a server with AMD Ryzen 9 5900X CPUs (12 cores/24 threads) and 4×32GB 3200MHz DDR4s. Alveo is built on the AMD (Xilinx) 16nm UltraScale+ architecture and offers 8GB of HBM2 with 460GB/s of bandwidth, and includes PCI Express 4.0 support. The FPGA device runs on 50MHz and has 2607K FFs, 1304K LUTs, 9024 DSPs, 2016 BRAMs and 960 URAMs.

We compare the performance of the WFAsic accelerator with a publicly available C implementation of the WFA [14] executed on the RISC-V CPU of the SoC. The performance of the WFAsic on the FPGA prototype is measured in clock cycles, regardless of the FPGA frequency. We use a standard Linux driver and API to configure the WFAsic accelerator.

We evaluate the WFAsic accelerator with six input sets (see Table 1). Although the accelerator is designed for long sequences, we evaluate its performance for short (100bp), medium (1Kbp) and long (10Kbp) sequences with error rates of 5% and 10%. We generate synthetic input sets with random mismatches, insertions and deletions, using the same methodology as in [13, 15]. For the synthetic inputs, the sequence errors follow a uniform and random distribution. However, it is important to note that the WFAsic performance is proportional to the error rate between the input sequences and not to the error distribution across the sequences. Hence, the overall performance depends on the nominal sequencing errors of the input sequences rather than on the specific distribution of errors within the sequences.

Figure 9 shows the speedup of the WFAsic accelerator with and without calculating the backtrace with respect to the execution on the CPU scalar code. The figure also compares the CPU vector code with the scalar code. Our accelerator achieves speedups over the CPU scalar code of 143× to 1076× without performing the backtrace, and of 2.8× to 344× when performing the backtrace. The next paragraphs explain the reasons of the different speedups obtained for the different read lengths and after enabling and disabling the computation of backtrace.

Figure 10 shows the scalability of the WFAsic accelerator with different numbers of Aligners over a design with only one Aligner. The available resources in the FPGA prototype are larger than in the final chip, so we can fit multiple Aligners and evaluate the scalability of the WFAsic accelerator on the FPGA. To this end, first we disable the backtrace to avoid memory bandwidth limitations. Results show that, for input sets with long sequences, the design scales perfectly. In particular, for the input sets of 10K-10% and 10K-5%, the accelerator with 10 Aligners provides speedups of 9.87× and 9.67× over the accelerator with one Aligner, respectively. This represents speedups of 10621× and 10062× over the WFA-CPU scalar code, respectively. The speedup is saturated with less Aligners for inputs with short sequences and smaller error rates. This is because the design is bound to the accelerator memory bandwidth for these inputs. This justification confirms the lower speedups of inputs with short lengths in Figure 9.

Next we explain the accelerator memory bandwidth restrictions, especially for short sequences, when having more than one Aligner in the accelerator. Table 1 shows, for each input, how many clock cycles are required to read a pair of sequences from main memory and to perform the alignment of the pair of sequences. Note that, first, the pairs of sequences are stored in the RAMs of the Aligners, and then the Aligners compute the alignments in parallel. Using Equation 7, the maximum efficient number of Aligners for each input set is calculated and shown in the last column of Table 1. For example, the design of 100-5% does not scale further than four Aligners because reading four pairs of sequences (4×75=300) takes more time than computing the four alignments in parallel (214 alignment cycles + 75 reading cycles = 289). Increasing the accelerator-memory bandwidth would reduce the time for reading the sequences and, thus, improve the scalability of the designs for short reads.

$$MaxAligners = Roundup(\frac{Alignment\_cycles}{Reading\_cycles}) + 1 \qquad (7)$$

When the backtrace is enabled, first the accelerator performs the alignment and then the CPU performs the backtrace. The backtrace time on the CPU dominates the total execution time, as it is much higher than the accelerator alignment time. This situation does not take place in the

Table 1: Maximum number of Aligners for each input based on the execution cycles of reading and aligning reads.

Input		Alignment Cycles	Reading Cycles	Max Efficient Aligners
Length	Error Rate (%)			
100	5	214	75	4
100	10	327	75	6
1K	5	2541	376	8
1K	10	8461	376	24
10K	5	278083	3420	83
10K	10	937630	3420	276

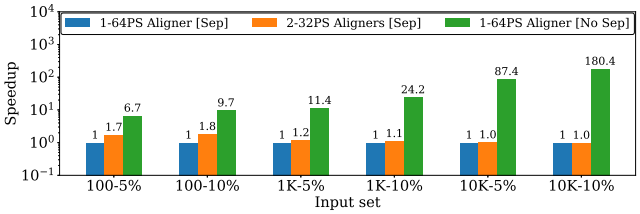

Figure 11: Performance comparison between WFAsic configurations of one Aligner of 64 parallel sections with data separation (1-64PS Aligner [Sep]), one Aligner of 64 parallel sections without data separation (1-64PS Aligner [No Sep]), and two Aligners of 32 parallel sections with data separation (2-32PS Aligner [Sep])).

CPU executions of the WFA because the computation of the alignments is much slower. More importantly, the backtrace computation on the CPU is bound to the CPU-memory bandwidth, which quickly becomes saturated. For these reasons, the speedups achieved by the WFAsic accelerator when the backtrace is disabled are higher than when the backtrace is enabled.

5.4 Design Configurations Analysis

As mentioned earlier, due to the limited area budget of the WFAsic, we are only able to fit one Aligner with 64 parallel sections in the design. However, it is also possible to reduce the Aligner size by reducing the number of parallel sections and fitting two smaller Aligners with 32 parallel sections in the chip. We have chosen the best configuration by comparing the performance of different configurations, WFAsic with one Aligner of 64 parallel sections versus WFAsic with two Aligners of 32 parallel sections. Note that as explained in Section 4.5, if there is one Aligner in the design, the time-consuming step of separating data of different alignments is not needed. The performance results shown in Figure 11 also compare both backtrace methods for the design with one Aligner and 64 parallel sections.

Figure 11 shows that by eliminating data separation step in 1-64PS Aligner design, it outperforms other configurations for all inputs, especially longer inputs. Comparing two other configurations which perform data separation, the design with two Aligners and 32 parallel sections performs better for shorter reads. However, for longer reads, the performance is same. In principle, doubling the number of parallel sections doubles the execution time. While in the case of short reads, this is not true. This is because, for short reads, the wavefront matrix is very small and most of the parallel sections are idle. So for short reads, increasing the number of parallel sections does not improve performance, while increasing the number of Aligners does.

For our final WFAsic implementation, we select the configuration of one Aligner with 64 parallel sections, in which the backtrace method does not separate data of different alignments. Apart from the significant performance improvement we get with this configuration, below are listed other reasons justifying why the design with one Aligner and 64 parallel sections, regardless of the backtrace method, is better than the one with two Aligners and 32 parallel sections.

- One Aligner with 32 parallel sections is only 1.5× smaller than one Aligner with 64 parallel sections. So using two Aligners with 32 parallel sections requires more area than one Aligner with 64 parallel sections.
- A design with one Aligner is simpler in terms of distributing inputs to, and collecting outputs from, the Aligner.
- Our target in this project is third-generation sequencing technologies that provide longer reads where both accelerator designs perform equally fast when performing data separation.

Table 2: GCUPS and area comparison of different platforms/methods aligning reads of 10Kbp.

Platform/Design	GCUPS	Area (mm^2)	GCUPS per mm^2
GACT[1]-ASIC [Heuristic]	2129	85.6	25
WFA-CPU on AMD EPYC[2] [1 thread]	7.5	1008	0.0074
WFA-CPU on AMD EPYC[2] [64 threads]	98	1008	0.0972
WFA-GPU [NVIDIA GeForce 3080]	476	628	0.76
WFAsic [With Backtrace]	61	1.6	38
WFAsic [Without Backtrace]	390	1.6	244

[1] The GACT module performs pairwise read alignment, which is the focus of our work, implemented in the state-of-the-art Darwin genomics co-processor.
[2] The AMD EPYC processor contains 8 Core Complex Dies (CCDs) and a central I/O Die (IOD) [10]. The die size of each CDC is 74mm^2 and that of IOD is 416mm^2.

5.5 Performance Comparison

Cell Updates Per Second (CUPS) is a well-known performance metric of SW algorithms that describes the number of cells of the DP matrix that are computed per second. This metric is used to compare SW algorithms and accelerators independently of their implementation details. Table 2 compares the GCUPS (Giga CUPS), the area, and the GCUPS per mm^2 of the WFAsic accelerator with other methods/platforms when aligning reads of 10Kbp. All GCUPS in Table 2 include backtrace computation time unless stated otherwise.

Darwin is the state-of-the-art accelerator that uses a heuristic method that does not process the whole DP-matrix, but some tiles of it. Hence, the CUPS achieved by the GACT module of Darwin is calculated based on the peak performance reports of the tiles computations (20.8M tiles/sec) and the tile size (320×320) in the original paper [20]. The AMD EPYC is a high-end server-class processor with 64 cores and 16 DIMMs of 64GiB. The table shows the GCUPS obtained when running the CPU implementation of the WFA algorithm on the AMD EPYC processor with 1 and 64 threads. The CPU execution of WFA on 64 cores is strongly limited by memory accesses as 10K-long sequence alignment requires a large memory footprint. For this reason, the number of GCUPS does not scale linearly from 1 to 64 threads on this processor. WFA-GPU paper [1] does not report GCUPS. However, it provides a supplementary material from which we calculate GCUPS for inputs with similar characteristics as WFAsic inputs. The GPU used in this work is a NVIDIA GeForce 3080 with a die size of 628mm^2. The GCUPS of the WFAsic accelerator on the ASIC is estimated by scaling the cycle counts measured on the FPGA prototype to the ASIC frequency. To highlight the impact of ASIC-memory bandwidth on the performance, we present WFAsic GCUPS both with and without performing backtrace.

The last column of Table 2 shows GCUPS per mm^2, in which we can see that the WFAsic accelerator (with and without backtrace) outperforms GACT, WFA-CPU on AMD EPYC processor and WFA-GPU. Note that GACT achieves the highest total GCUPS, but with a much larger area budget than WFAsic. The single-threaded WFA-CPU on AMD EPYC processor is the slowest of all, both in terms of absolute GCUPs and of GCUPS per mm^2.

Although WFA-FPGA is designed for short reads, we compare its GCUPS with the WFAsic GCUPS. The comparison is excluded form Table 2 because WFA-FPGA does not support read lengths of 10Kbp. The WFA-FPGA reports 1252 peak GCUPS and it integrates at least 40 parallel Aligners, that is, 31.3 GCUPS per Aligner, while WFAsic offers 61 GCUPS per Aligner.

Note that the WFA-based designs (WFA-CPU, WFA-GPU, WFA-FPGA and WFAsic) avoid the full computation of the DP-matrix, but as this algorithm is an exact method, we compute the CUPS considering the equivalent number of DP cells that the SWG algorithm would need to compute the optimal alignment.

6 RELATED WORK

In recent years, many ASIC accelerators have been proposed to improve the performance of read mappers [4–7, 13, 20, 21]. Among them [5, 6, 20] accelerate both seeding (filtering) and seed extension (pairwise alignment) steps, while [4, 7, 13, 21] only focus on seed extension step.

Darwin [20] and GenAx [6] seed extension accelerator modules are, respectively, called GACT and SillaX, which use an approximate string matching method and calculate gap-affine distance. Darwin supports long reads and has been evaluated with reads up to 10Kbp. However, SillaX has been evaluated with short reads of 100bp, but it reaches a high frequency of 2GHz. Although SeGram [5] only calculates approximate edit distance, it is the first accelerator for sequence-to-graph mapping. SeGram is both evaluated for short and long reads.

SeedEx [7] and ABSW [13] both accelerate seed extension step using an approximate method and calculate gap-affine distance. SeedEx is designed for short reads, while ABSW has been evaluated for long reads up to 10Kbp. GenASM [4] accelerates the Bitap [3] algorithm for approximate string matching based on calculating edit distance. GenASM implements a modified version of Bitap that supports long reads as well as short reads. Mao-Jan [21] proposes the only optimal seed extension accelerator based on gap-affine scoring in our literature. However, the design obtains a relatively low frequency and has been only evaluated for short reads.

Unlike WFAsic, many of these methods incorporate heuristics that can compromise the accuracy of the results. Compared to other accelertors, WFAsic is the only exact long-read aligner that implements the gap-affine scoring while being compatible with the backtrace algorithm. Moreover, WFAsic design is able to reach a frequency of 1.1GHz in GF22nm technology.

7 CONCLUSIONS

This paper presents the first WFA ASIC accelerator integrated in a RISC-V processor SoC. The accelerator is designed for long reads and evaluated for reads up to 10K bases. WFAsic provides optimal results based on gap-affine scoring model. The design is well parallelized by efficiently storing and distributing necessary data into multiple RAMs, so that 64 cells of the wavefront vectors are calculated in parallel. WFAsic is able to perform the alignment independently and in parallel with other CPU processes as it includes a DMA which has direct access to the main memory through the AXI-Full bus. Results show that the accelerator reaches speedups of up to 1076× and 344× when backtrace is disabled and enabled, respectively, over the WFA-CPU running on the RISC-V of the chip. In addition, the accelerator perfectly scales by increasing the number of the Aligners in the accelerator, if the accelerator-memory bandwidth is not saturated. The post-layout of the WFAsic, in GF22nm technology, reaches a frequency of 1.1GHz, fits in an area of 1 .6mm^2 and consumes a power of 312mW.

ACKNOWLEDGMENT

This work has been partially supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contracts PID2019-107255GB-C21, PID 2020-113614RB-C21 and TED2021-132634A-I00), by the Generalitat de Catalunya (contract 2021-SGR-00763 and 2021-SGR-00574), by the European Union within the framework of the ERDF of Catalonia 2014-2020 under the DRAC project [001-P-001723], by the European NextGenerationEU/PRTR, by the Lenovo-BSC Contract-Framework (2022), and by the IBM/BSC Deep Learning Center initiative. M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship No. RYC-2016-21104.

REFERENCES

[1] Quim Aguado-Puig, Santiago Marco-Sola, Juan Carlos Moure, Christos Matzoros, David Castells-Rufas, Antonio Espinosa, and Miquel Moreto. 2022. WFA-GPU: Gap-affine pairwise alignment using GPUs. *bioRxiv* (2022).

[2] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410.

[3] Ricardo Baeza-Yates and Gaston H Gonnet. 1992. A new approach to text searching. *Commun. ACM* 35, 10 (1992), 74–82.

[4] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, et al. 2020. GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. IEEE, 951–966.

[5] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, et al. 2022. SeGraM: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping. *arXiv preprint arXiv:2205.05883* (2022).

[6] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 69–82.

[7] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2020. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. IEEE, 937–950.

[8] Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162, 3 (1982), 705–708.

[9] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. 2021. An FPGA Accelerator of the Wavefront Algorithm for Genomics Pairwise Alignment. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 151–159.

[10] Anre Kashyap. 2020. High performance computing: Tuning guide for AMD EPYC 7002 series processors.

[11] Heng Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997* (2013).

[12] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

[13] Yi-Lun Liao, Yu-Cheng Li, Nae-Chyun Chen, and Yi-Chang Lu. 2018. Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–9.

[14] Santiago Marco-Sola. [n. d.]. WFA CPU implementation. https://github.com/smarco/WFA

[15] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. 2021. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics* 37, 4 (2021), 456–463.

[16] Santiago Marco-Sola and Paolo Ribeca. 2015. Efficient Alignment of Illumina-Like High-Throughput Sequencing Reads with the GEnomic Multi-tool (GEM) Mapper. *Current Protocols in Bioinformatics* 50, 1 (2015), 11–13.

[17] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. 2012. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature methods* 9, 12 (2012), 1185.

[18] Temple F Smith and Michael S Waterman. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[19] Víctor Soria-Pardos, Max Doblas, Guillem López-Paradís, Gerard Candón, Narcís Rodas, Xavier Carril, Pau Fontova-Musté, Neiel Leyva, Santiago Marco-Sola, and Miquel Moretó. 2022. Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI. In *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 254–261.

[20] Yatish Turakhia, Gill Bejerano, and William J Dally. 2018. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. *ACM SIGPLAN Notices* 53, 2 (2018), 199–213.

[21] Jing-Ping Wu, Yi-Chien Lin, Ying-Wei Wu, Shih-Wei Hsieh, Ching-Hsuan Tai, and Yi-Chang Lu. 2021. A Memory-Efficient Accelerator for DNA Sequence Alignment with Two-Piece Affine Gap Tracebacks. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.