IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2021.DOI

Accelerating Edit-Distance Sequence Alignment on GPU using the Wavefront Algorithm

QUIM AGUADO-PUIG[1], SANTIAGO MARCO-SOLA[1,2], JUAN CARLOS MOURE[1], DAVID CASTELLS-RUFAS[1], LLUC ALVAREZ[2,3], ANTONIO ESPINOSA[1] and MIQUEL MORETO[2,3]

[1]Departament d'Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona, Barcelona, 08193, Spain.
[2]Computer Sciences Department, Barcelona Supercomputing Center, Barcelona, 08034, Spain.
[3]Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, 08034, Spain.

Corresponding author: Quim Aguado-Puig (e-mail: quim.aguado@uab.cat)

ABSTRACT Sequence alignment remains a fundamental problem with practical applications ranging from pattern recognition to computational biology. Traditional algorithms based on dynamic programming are hard to parallelize, require significant amounts of memory, and fail to scale for large inputs. This work presents eWFA-GPU, a GPU (graphics processing unit)-accelerated tool to compute the exact edit-distance sequence alignment based on the wavefront alignment algorithm (WFA). This approach exploits the similarities between the input sequences to accelerate the alignment process while requiring less memory than other algorithms. Our implementation takes full advantage of the massive parallel capabilities of modern GPUs to accelerate the alignment process. In addition, we propose a succinct representation of the alignment data that successfully reduces the overall amount of memory required, allowing the exploitation of the fast shared memory of a GPU. Our results show that our GPU implementation outperforms by 3-9× the baseline edit-distance WFA implementation running on a 20 core machine. As a result, eWFA-GPU is up to 265 times faster than state-of-the-art CPU implementation, and up to 56 times faster than state-of-the-art GPU implementations.

INDEX TERMS Approximate String Matching, Compute Unified Device Architecture (CUDA), Edit-Distance, Graphics Processing Unit (GPU), Levenshtein distance, Pairwise Sequence Alignment, Wavefront Alignment Algorithm (WFA).

I. INTRODUCTION

SEQUENCE comparison constitutes a fundamental problem for many practical applications in numerous fields such as pattern matching [1], information retrieval [2], network security [3], and computational biology [4], to name a few. In general, assessing the similarity (or dissimilarity) between two sequences is an essential building-block within multiple applications for data mining [5], spell correction [6, 7], speech recognition [8], signature matching [9], image analysis [10], and more [11, 12, 13, 14, 15].

In the past decade, sequence alignment has acquired a special relevance in computational biology and bioinformatics. In particular, it is a critical component for methods like read mapping [16, 17, 18], de-novo genome assembly [19, 20], variant detection [21, 22], multiple sequence alignment [23], and many others [24, 25]. Due to the unprecedented data-production rates of modern DNA sequencing machines, the need for fast and accurate algorithms for sequence analysis has become paramount. In the past years, computation has become a growing fraction of genomics cost as sequence data production has increased drastically and its costs have been significantly reduced [26]. Moreover, with ever-increasing sequence lengths, third-generation sequencing technologies pose an additional challenge to these algorithms and their ability to scale [27].

The need to process large volumes of genomic data has motivated the mainstream adoption of high-performance computing (HPC) methods and resources. In turn, the demanding computational requirements have forced researchers to investigate solutions using more efficient hard-

ware accelerators such as GPUs. Compared to multi-core processors, modern GPUs provide both higher computation throughput and memory bandwidth. For that reason, GPUs have been widely adopted as effective accelerators for many scientific and commercial applications [28, 29, 30, 31].

Sequence alignment algorithms have been intensively studied for more than 40 years, applying multiple strategies (including dynamic programming [32, 33], automata [34, 35, 36], and bit-parallelism techniques [37, 38]). Nonetheless, these algorithms require quadratic time and memory on the length of the sequences. With increasing sequence length, using these classical algorithms becomes impractical or not even possible. As opposed to classical methods, our proposal is based on the wavefront alignment algorithm (WFA) [39]. This novel method exploits similarities between sequences to accelerate the computation of the optimal alignment. As a result, its time complexity $O(ne)$ depends on the sequence length n and the optimal edit-distance e (i.e., the error between the sequences).

This paper presents a GPU implementation of the WFA algorithm for the exact computation of the edit-distance alignment between DNA sequences on GPUs. We propose an algorithmic adaptation of the WFA algorithm to exploit the parallel computing capabilities of GPU architectures. Moreover, we introduce a compact piggyback-encoding of the intermediate wavefront data that allows computing each alignment using the GPU fast on-chip memories. Furthermore, we propose using a bit-parallel strategy within the WFA to accelerate DNA sequence comparisons on the GPU. As a result, we provide a high-performance implementation based on specialised alignment kernels for input sequences with different alignment errors. Also, we implement a batch processing based system that allows computing thousands of alignments in parallel, overlapping data transfers with computations. We characterise the performance of our implementation and present the different performance trade-offs of our solution. Ultimately, experimental results demonstrate that our implementation outperforms other state-of-the-art proposals.

The rest of the paper is structured as follows. Section II presents the definitions and methods on which our algorithm is based. Section III describes the proposed algorithmic adaptations and optimisation strategies of the GPU implementation. Then, Section IV shows experimental results, compares the performance of our method against other state-of-the-art implementations for both CPU and GPU, and studies the performance trade-offs of our GPU implementation. Next, Section V presents an overview of the most relevant sequence alignment methods presented in the literature focusing on GPU-based implementations. Finally, Section VI summarises the main results and contributions of this work.

II. BACKGROUND

A. EDIT-DISTANCE SEQUENCE ALIGNMENT

Also known as *Levenshtein* distance, edit-distance is a metric that measures the difference between two sequences. It is defined as the minimum number of edit operations (i.e., mismatch, insertion, and deletion) required to transform one sequence into the other. For instance, the edit-distance between the sequences $P = "GATTACA"$ and $T = "GAATA"$ is $e = 3$. That is to say, the minimum number of edit operations required to transform P into T is 3 (i.e., substitute the first T for an A, and remove the last two elements of the sequence P). Computing the edit-distance between two sequences is commonly solved using a dynamic programming (DP) approach [4, 33]. Given two sequences $P = [p_0, ..., p_n - 1]$ and $T = [t_0, ..., t_m - 1]$ (of length n and m, respectively), the edit-distance e between the two sequences can be computed using the recurrence presented in the Eq. 1 (being $e = M_{n,m}$). By means of storing all the intermediate $M_{i,j}$ values of the DP-matrix, we can trace back the edit operations that originated the minimum edit-distance (i.e., the sequence alignment). It follows that classical algorithms based on this DP approach exhibit quadratic time complexity and quadratic space complexity on the sequence length (i.e., $O(nm)$).

$$M_{i,j} = \begin{cases} \text{i} & \text{if } j = 0 \\ \text{j} & \text{if } i = 0 \\ \min \begin{cases} M_{i-1,j-1} + \delta(P_i, T_j) \\ M_{i,j-1} + 1 \\ M_{i-1,j} + 1 \end{cases} & \text{Otherwise} \end{cases} \tag{1a}$$

$$\delta(p_i, t_j) = \begin{cases} 0 & \text{if } p_i = t_j \\ 1 & \text{if } p_i \neq t_j \end{cases} \tag{1b}$$

These DP-based solutions have been extensively studied and used for many years and in different application contexts. However, they exhibit a series of computational shortcomings that limit their scalability and prevent the implementation of effective parallelization techniques. First, the quadratic memory requirements limit their practical application to compute the alignment of long sequences (i.e., thousands of characters). Second, the computational pattern shown in Eq. 1 depicts data dependencies that hinder straightforward usage of SIMD (vector) instructions, which could accelerate execution speed. Also, in its classical formulation, the algorithm explores unnecessary regions of the DP-matrix that do not contribute to the optimal solution and generate needless computations.

Over the past years, many variations and optimizations have been proposed to overcome these limitations. These solutions include techniques such as computing the DP-matrix antidiagonal-wise [40], banded approaches that only compute a portion of the DP-matrix [41], data-layout organizations that allow using SIMD instructions [42, 43, 44], bit-packed encodings [45, 37], and other heuristic methods [33, 46, 47]. Due to its importance and performance impact in many applications, multiple libraries have emerged implementing those algorithms. Among the most widely used, it is worth mentioning Edlib [48] and BGSA [49], fast CPU implementations of the Myers bit-vector algorithm (BPM) [37];

DAligner [50], an efficient implementation of the *O(ND)* algorithm [45]; and NVBio [51], a GPU accelerated library for sequence alignment.

B. THE WAVEFRONT ALIGNMENT ALGORITHM

Recently, in [39], the authors proposed a fast and exact pairwise alignment algorithm: the WFA algorithm. As opposed to other approaches, the WFA algorithm proposes an alternative encoding of the DP-matrix and an efficient algorithm to compute partial alignments of increasing distance. As a result, the WFA algorithm only needs to calculate a small number of DP-matrix cells to find the optimal alignment. This way, WFA exploits similarities between sequences to reduce the time complexity to $O(ne)$, being n the sequence length and e the optimal edit-distance, reducing the memory requirements to $O(e^2)$. In the following, we formally present the WFA algorithm to compute the edit-distance alignment.

For a given distance e, let a wavefront $\widetilde{W}_{e,k}$ be a vector of integer offsets that, for each diagonal k, encodes the diagonal offset from the leftmost column of the DP-matrix to the farthermost cell that has distance e. As opposed to DP methods that explicitly represent the distance of each cell in the DP-matrix, the WFA algorithm uses wavefront offsets $\widetilde{W}_{e,k}$ that encodes only the farthermost cell in the diagonal k that has distance e. Then, starting from $\widetilde{W}_{0,0} = 0$ (i.e., the upper-left corner of the DP-matrix), the WFA algorithm progressively computes wavefronts $\widetilde{W}_e$ of increasing distance until a wavefront reaches the bottom-right corner of the DP-matrix (i.e., the end position of the alignment). For that, the WFA algorithm repeatedly applies two operators: *extend()* and *computeNext()*.

Given an initial wavefront $\widetilde{W}_e$, the *extend()* operator increases each offset of the wavefront vector according to the number of contiguous matching characters between the sequences. This way, the WFA algorithm exploits the property that diagonals are monotonically increasing [52]. In particular, for a given cell $M_{i,j}$ of the DP-matrix, we know from Eq. 1 that $M_{i,j} = \min(M_{i-1,j-1} + \delta(P_i, T_j), M_{i,j-1} + 1, M_{i-1,j} + 1)$. If $P_i = T_j$, there is no better outcome than retaining the same cell value along the diagonal; that is, $M_{i,j} = M_{i-1,j-1}$. Moreover, note that the $M_{i,j-1}$ and $M_{i-1,j}$ values do not affect this computation and therefore it is not necessary to explicitly compute these cells. WFA exploits this operation, denoted *diagonal extension* (Algorithm 1), to find the farthest reaching (f.r.) offset on each diagonal for a given distance.

Once all the offsets of a wavefront have been diagonally extended, the algorithm checks whether any offset $\widetilde{W}_{e,k}$ reaches the bottom-right cell (m, n). If that is not the case, WFA proceeds to generate the next wavefront $\widetilde{W}_{e+1}$ using the *computeNext()* operator. For each diagonal k, *computeNext()* uses the previous offsets in $\widetilde{W}_e$ (i.e., $\widetilde{W}_{e,k-1}, \widetilde{W}_{e,k}, \widetilde{W}_{e,k+1}$) to compute the f.r. offset with distance $e + 1$ on diagonal k. Using Eq. 2, *computeNext()* finds the most advanced position with distance $e + 1$ considering a deletion, an insertion, or a mismatch from the f.r. offsets of the previous wavefront $\widetilde{W}_e$ (Algorithm 2).

$$\widetilde{W}_{e+1,k} = \max \begin{cases} \widetilde{W}_{e,k+1} & \text{(Deletion)} \\ \widetilde{W}_{e,k} + 1 & \text{(Mismatch)} \\ \widetilde{W}_{e,k-1} + 1 & \text{(Insertion)} \end{cases} \quad (2)$$

Algorithm 1: WFA *extend()* operator

Function *extend* $(P, T, \widetilde{W}_e)$ **:**
  for $k \leftarrow -e$ ***to*** e **do**
    `// Compute (v,h) position`
    $\mathrm{v} \leftarrow \widetilde{W}_{e,k} - k$
    $\mathrm{h} \leftarrow \widetilde{W}_{e,k}$
    `// Compute diagonal matches`
    while $P_v = T_h$ **do**
      $v \leftarrow v + 1$
      $h \leftarrow h + 1$
      $\widetilde{W}_{e,k} \leftarrow \widetilde{W}_{e,k} + 1$

Algorithm 2: WFA *computeNext()* operator

Function *computeNext* $(\widetilde{W}_e, \widetilde{W}_{e+1})$ **:**
  $k_{lo} \leftarrow -(e + 1)$
  $k_{hi} \leftarrow (e + 1)$
  for $k \leftarrow k_{lo}$ ***to*** k_{hi} **do**
    $\widetilde{W}_{e+1,k} \leftarrow \max\{\widetilde{W}_{e,k-1} + 1, \widetilde{W}_{e,k} + 1, \widetilde{W}_{e,k+1}\}$

The WFA algorithm (Algorithm 3) progressively computes wavefronts (containing f.r. offsets) of increasing distance applying the operators *extend()* and *computeNext()*. Once a $\widetilde{W}_{e,k}$ reaches the bottom-right cell (m, n), the algorithm concludes that e is the minimum alignment distance. Additionally, note that the sequence of operations that led to the offset $\widetilde{W}_{e,k}$ constitute the optimal alignment and can be recovered by tracing back the path from $\widetilde{W}_{e,k}$ to $\widetilde{W}_{0,0}$. To put it into perspective, Figure 1 shows a side-by-side comparison of the classical DP-based algorithm and the WFA to compute the edit-distance between $P = "GATTACA"$ and $T = "GAATA"$. Note how the WFA operations have a direct mapping on the DP-matrix.

Altogether, the WFA algorithm requires computing e wavefronts to compute an alignment of distance e. From the initial wavefront $\widetilde{W}_{0,0}$ of unitary length, each successive wavefront increases its length by two. It follows that the e-wavefront has length $1 + 2e$ and the total number of wavefront-offsets needed is $\sum_{n=0}^{e} 1 + 2n = (e + 1)^2$. Thus, that the overall memory complexity is $O(e^2)$. Moreover, note that the operator *computeNext()* runs in time proportional to the wavefront length. Then, for each diagonal, the total number of wavefront extensions performed by the *extend()* operator is bounded by the maximum number of diagonally

Algorithm 3: WFA edit-distance alignment

Function *WFA_align* $(P,T,\widetilde{W}_e,\widetilde{W}_{e+1})$ **:**

```
// Initial conditions
W̃_{0,0} ← 0
extend (W̃_{0,0})
// Compute wavefronts
e ← 0
while W̃_{e,m−n} ≠ m do
    computeNext (W̃_e, W̃_{e+1})
    e ← e + 1
    extend (P, T, W̃_e)
```

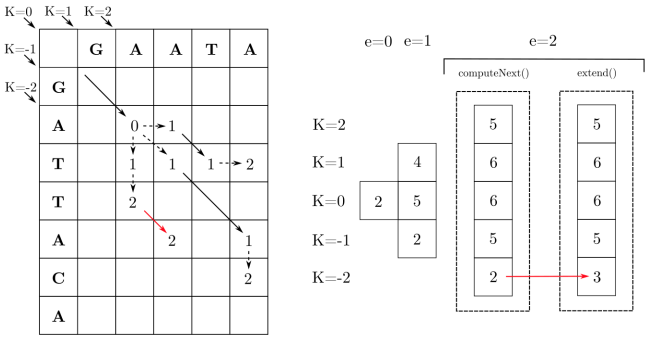


Figure 1. Edit-distance WFA depicted inside a DP table. Dotted lines represent the *computeNext* operator, while continuous lines represent the *extend* operator. The extend operator of the offset $\widetilde{W}_{2,-2}$ is shown as a red line. At $M_{2,4}$, as $T_2 = P_4$, the offset is extended. At $M_{3,5}$, $T_3 \neq P_5$, so the extend operator stops.

matching characters (i.e., $\max\{n, m\}$). Therefore, we conclude that the running time of the WFA algorithm is bounded in the worst case by $O(\max\{n, m\} \cdot e)$ or $O(ne)$ when the sequences have the same length.

Besides presenting a better time and memory complexity, the WFA algorithm presents additional advantages compared to classical DP-based alignment algorithms. Most notably, WFA presents a simple data-processing pattern that allows processing each wavefront offset independently and storing them consecutively in memory. As opposed to traditional DP-based algorithms, the WFA algorithm can be effectively vectorized using SIMD instructions. Furthermore, the WFA algorithm encodes offsets in the range of the sequence length instead of storing the actual distance or score, as DP-based algorithms do. Therefore, wavefront elements are bounded by the maximum sequence length and can be encoded using less memory. In turn, this succinct encoding allows enhancing SIMD performance further. In the present work, we aim to exploit these advantageous properties to implement an efficient parallel strategy on GPUs using a SIMT programming model.

C. GPU ARCHITECTURE AND CUDA PROGRAMMING MODEL

GPUs are massively parallel devices containing multiple throughput-oriented processing units called streaming multiprocessors (SMs). SMs execute hundreds of instructions in parallel by using deep pipelines and aggressive fine-grained multithreading. SMs share an L2 cache of a few MB and a global memory of several GB. Each SM is equipped with multiple SIMD cores capable of performing in-order execution of instructions. At the same time, each SM contains a register file (around 256KB) and a fast on-chip scratchpad memory that can be shared among threads (around 48KB per block of threads).

Since its release in 2006, CUDA has become the most popular programming model for general-purpose GPU computing. CUDA comes with a software environment that allows using a superset of C/C++, together with API calls, to program one or multiple GPU devices. The CUDA programming model provides a heterogeneous environment where the host code runs on the CPU, and the device code runs on a physically separate GPU. Both the host and device can maintain their own separate memory spaces; meanwhile, CUDA supports data transfer between host and device memory. The CUDA programming model defines a computation hierarchy formed by kernels, thread blocks, warps, and threads:

- **Kernel:** Minimum unit of work sent from the CPU to the GPU. In short, a kernel is a function executed in parallel on a GPU by a large number of different CUDA threads.
- **Thread block:** Group of threads that are executed by one SM and cannot migrate to other SMs (except during preemption or dynamic parallelism). Threads within a block can cooperate via synchronization primitives, using registers, or shared memory. Thread blocks are scheduled non-deterministically for independent MIMD execution into SMs.
- **Warp:** A thread block is divided into batches of 32 threads, called warps, which are the smallest scheduling unit.
- **Thread:** Minimum execution unit of programmed instructions in CUDA.

GPU applications must launch kernels composed of tens of thousands of threads to simultaneously achieve high-performance executions. To that end, between 32 and 64 warps from one or multiple thread blocks are dynamically scheduled for execution in the same SM. This mechanism, often known as H/W multithreading, is the primary latency-hiding strategy on GPUs. Furthermore, a GPU executes warps of parallel threads using a SIMT model (Single Instruction Multiple Threads), which allows each thread to access its registers, load and store from divergent addresses, and follow divergent control flow paths.

However, GPU executions can suffer from performance limitations due to several factors. In particular, when threads of a warp diverge due to conditional branches, only a subset

of the threads are active, which may reduce the overall performance. This situation is known as divergence, and it is an inherent performance limitation of SIMD architectures that must be addressed when designing the algorithm. Similarly, another critical performance limitation can arise from sparse memory accesses. When executing a SIMD load/store instruction, the memory addresses provided by all the threads in the same warp coalesce (i.e., combine) to generate one or multiple memory block access requests. GPU applications seek to coalesce data requests from global memory into a few memory blocks to achieve efficient transfers. Access to global memory is relatively slow compared to fast on-chip memory (i.e., shared memory and registers). For that reason, it is always preferred that all threads in a CUDA block exploit local memory whenever possible. However, the amount of shared memory and registers used by a CUDA block limits the number of concurrent CUDA blocks running on the same SM and may reduce the GPU occupancy (i.e., threads assigned per SM). Having a high occupancy is important to hide the latency of memory accesses and compute operations.

III. GPU IMPLEMENTATION OF THE WFA ALGORITHM

Nowadays, analysing large-scale workloads requires aligning millions of relatively large sequences to a given reference genome in a very short time. Previous research work has shown the capabilities of modern GPUs to accelerate HPC applications in general and alignment tools in particular. Specifically, parallel programming using CUDA can be very effective to accelerate string matching algorithms, as shown in many recent studies [47, 30, 53, 54, 55, 56]. This section presents our proposed method to accelerate edit-distance sequence alignment using the WFA algorithm on GPU. In the following, we present the main challenges to adapt the WFA algorithm to the CUDA programming model and the contributions and trade-offs of the proposed implementation.

Mainly, there are two strategies to parallelize computations on GPU devices: coarse and fine-grained. In the case of the WFA algorithm, a coarse-grained parallelization strategy devotes each CUDA thread to compute a single alignment, whereas, in a fine-grained strategy, multiple CUDA threads collaborate to align a single pair of sequences.

In a coarse-grained approach, each thread within the block requires its own pair of sequences and wavefront data structures to perform the alignment. Due to the limited size of the shared memory, using this approach forces storing data in global memory space, resulting in long-latency memory accesses. Moreover, a coarse-grained strategy is bound to generate divergence across threads' execution within a block as each alignment requires a different amount of computations. Ultimately, a coarse-grain approach faces significant performance limitations that can largely reduce the overall execution speed of the algorithm on a GPU.

In contrast, a fine-grained strategy computes each alignment using a thread block. This way, all threads within the block cooperatively work to compute one alignment problem.

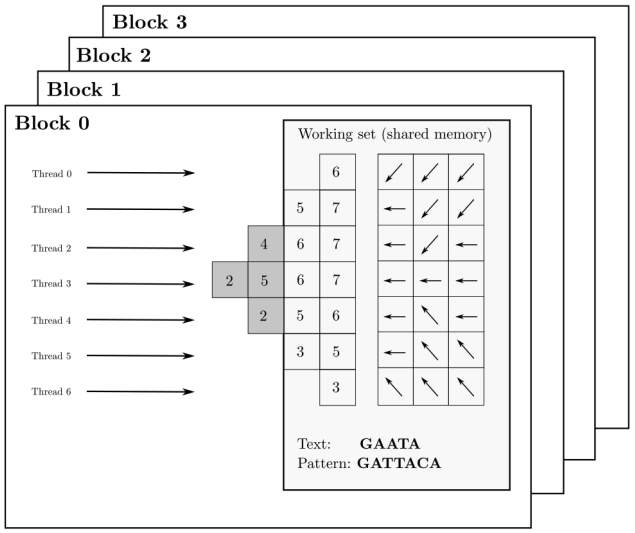


Figure 2. Mapping of CUDA resources into WFA work.

This approach heavily reduces the consumption of shared memory and registers, allowing the storage of the wavefront structures in shared memory for several thread blocks, which can operate concurrently in the same SM (increasing the occupancy). Furthermore, the computational pattern depicted by the WFA algorithm allows to efficiently map the computations across the threads of a block (Figure 2). We exploit the fact that computations on each diagonal are independent, allowing to compute every element in each wavefront $\widetilde{W}_e$ in parallel for both operations *extend()* and *computeNext()*. Our solution exploits this parallelism approach where each thread block computes a single alignment problem, and each thread within the block is assigned a different diagonal offset to compute. This way, we implement Algorithm 3 to be computed using a thread block. For each wavefront $\widetilde{W}_e$ (containing $2e+1$ diagonals), threads within the block extend independently each diagonal k offset (i.e., apply operator *extend()*); and then, compute the corresponding k offset of the next wavefront $\widetilde{W}_{e+1}$ (i.e., apply operator *computeNext()*).

Nevertheless, this approach faces some performance challenges of its own. Concerning the memory utilisation, wavefronts naturally become larger as the alignment error e considered grows during the alignment computation (i.e., $|\widetilde{W}_e| = 1 + 2e$). It follows that the overall number of wavefront elements required to align a pair of sequences with alignment error e is given by $\sum_{n=0}^{e} 1 + 2n = (e+1)^2$. Note that all the wavefronts need to be stored to retrieve the edit operations that originated the minimum edit-distance alignment. Consequently, the memory requirements grow quadratically with the alignment error, posing a scalability limitation when storing the data on shared memory. To palliate this limitation and exploit the benefits of using the fast shared memory, we propose a succinct encoding scheme where the wavefronts store partial backtraces as the alignment is computed (Section III-A).

Depending on the alignment error between the input sequences, some alignments may require more shared memory than others. Requesting memory for the worst-case alignments will limit the number of concurrent thread blocks running on an SM and, ultimately, the performance of the whole execution. For that reason, we implement three different kernel specialisations, each one supporting a different maximum alignment error. This way, our implementation can optimise the resource usage for each scenario and achieve higher performance for cases where the alignment error is bounded (Section III-B).

Moreover, the computation performed by the *extend()* operator can be largely irregular as it depends on the number of matching characters on each diagonal. To minimise thread divergence, we use a packed sequence encoding that allows performing bit-parallel sequence comparisons (i.e., block-wise comparisons), reducing the chances of divergence, and saving memory at the same time (see Section III-C).

Additionally, modern GPUs allow simultaneous data transfers and kernel execution to exploit parallelism further. In this way, the system minimises the impact of data offloading from the host and overlaps transference with computation on the device. Our solution implements an alignment batch system that allows multiple alignment problems to be solved in parallel while performing data transfers HtoD and DtoH (see Section III-D).

A. PIGGYBACKED ALIGNMENT OPERATIONS

As stated before, the WFA algorithm requires storing all the intermediate wavefront vectors $\widetilde{W}_e$ to be able to trace back the optimum alignment. As a result, the memory consumption of the algorithm grows quadratically with the alignment error, posing a severe constraint on the shared memory scalability. Here, we propose a succinct encoding of the wavefronts based on storing partial backtraces as the alignment is computed.

For an alignment of distance e, the WFA backtrace algorithm computes the optimum alignment path from (n, m) to $(0, 0)$, traversing all the wavefront vectors from $\widetilde{W}_e$ to $\widetilde{W}_0$. In particular, each step of the backtrace checks the adjacent offsets (e.g., from $\widetilde{W}_{e,k}$ to $\widetilde{W}_{e-1,k+1}$, $\widetilde{W}_{e-1,k}$, or $\widetilde{W}_{e-1,k-1}$) for the one that originated the minimum cost alignment according to Eq. 2. In essence, each iteration in the backtrace process computes a step in the alignment path. To avoid storing explicitly all the wavefront offsets, we propose to explicitly compute each backtrace step (i.e., $\leftarrow, \nwarrow, \swarrow$) and store it together with the previous steps in a backtrace vector. In this way, our implementation piggybacks the partial backtraces $\widetilde{B}_{e,k}$ from every offset $\widetilde{W}_{e,k}$ to the beginning of the alignment $\widetilde{W}_{0,0}$. As a result, our solution only needs to store two wavefronts (i.e., $\widetilde{W}_e$ and $\widetilde{W}_{e+1}$) and their partial backtraces $\widetilde{B}_e$ and $\widetilde{B}_{e+1}$ for each step of the algorithm.

Figure 3 illustrates our proposal aligning the sequences $T = "GAATA"$ and $P = "GATTACA"$. The example shows that the alignment process ends at $\widetilde{W}_{3,-2}$ (i.e., the minimum edit-distance between P and T is $e = 3$). The alignment path from $\widetilde{W}_{3,-2}$ to $\widetilde{W}_{0,0}$ is explicitly stored in the backtrace vector at $\widetilde{B}_{e,k} = " \leftarrow\nwarrow\nwarrow "$.

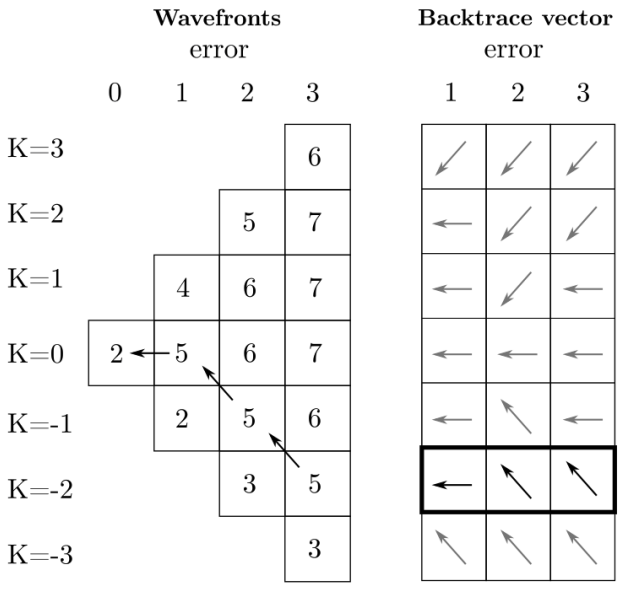


Figure 3. Wavefront data layout for aligning the sequences $T = "GAATA"$ and $P = "GATTACA"$.

However, the backtrace vector does not contain the full alignment path but just the edit operations (i.e., mismatches, insertions, and deletions) within the alignment. To recover the full alignment path, we need to recover the matches between backtrace steps. To that purpose, we use the WFA's *extend()* operator to compute stretches of matches between successive backtrace steps. This strategy is shown in Algorithm 4. Note that this algorithm only has to operate a single time over the backtrace vector of the optimum alignment, and its time complexity is proportional to the alignment path.

In practice, each backtrace step can be efficiently computed within the *computeNext()* operation and encoded using two bits (i.e., 32 backtrace steps for each 64-bit word). For that, we piggyback each offset in Eq. 2 with its corresponding backtrace step on its two less significant bits. After the maximum calculation, the resulting backtrace step is appended to the backtrace vector at the end.

The succinct encoding of the backtrace steps leads to a significant reduction in memory consumption. Using 32-bits offsets, the straightforward implementation of the WFA algorithm requires $(e + 1)^2 \times 4$ bytes to align a pair of sequences of error e. Using the proposed scheme, we reduce the required memory structures to the last two computed wavefronts and their corresponding backtrace vectors (i.e., $4e \times (4 + 2e/8)$ bytes). For any sufficiently large e, this represents up to a 4x reduction in memory usage. In practice, for moderately large e values, all the backtrace vectors can be fitted in shared memory. Furthermore, to enable coalesced memory accesses and avoid bank conflicts, we implement a *struct-of-arrays* approach, separating the wavefront offsets

Algorithm 4: Algorithm to retrieve the alignment from the backtrace vector

Function *retrieveAlignment* (P,T,$\widetilde{W}_e$,$\widetilde{W}_{e+1}$) **:**
- $offset \leftarrow 0$
- $k \leftarrow 0$
- $A \leftarrow \emptyset$
- **for** $i \leftarrow 0$ ***to*** e **do**
  - $m \leftarrow$ extend(P,T,k,offset)
  - $A \leftarrow A$+'M', $\overset{(m)}{\cdots}$,'M'
  - $op \leftarrow \widetilde{B}_k[i]$
  - **switch** op **do**
    - **case** $\nwarrow$ **do** // Deletion
      - $k \leftarrow k-1$
      - $A \leftarrow A$+'D'
    - **case** $\leftarrow$ **do** // Mismatch
      - $offset \leftarrow offset+1$
      - $A \leftarrow A$+'X'
    - **case** $\swarrow$ **do** // Insertion
      - $offset \leftarrow offset+1$
      - $k \leftarrow k+1$
      - $A \leftarrow A$+'I'
- $m \leftarrow$ extend(P,T,k,offset)
- $A \leftarrow A$+'M', $\overset{(m)}{\cdots}$,'M'

from the backtrace vectors. As a result, subsequent backtrace vectors are stored contiguously, enabling fast accesses when all threads in a warp access the backtrace vectors.

B. KERNEL SPECIALISATION

Even though the introduction of the backtrace vectors (Section III-A) reduces the memory requirements, shared memory usage is a major performance limitation when scaling to larger alignment errors (see Section IV-C). In practice, our implementation uses bit-vectors to store the backtrace vectors. For instance, using 64-bit words, we could store up to 32 edit operations (i.e., each edit operation encoded using 2 bits). As the maximum alignment error increases, this approach requires longer bit-vectors. In turn, large bit-vectors put additional pressure on the share memory usage and hinder performance. Therefore, it is important to bound the maximum alignment error for each batch of sequences and use the most suitable configuration that minimises the memory used by the backtrace vectors.

On that account, we implemented three different kernels, each one supporting a different maximum alignment error: 32, 64, and 128 errors. For convenience, we call these kernels E32, E64, and E128, respectively. Each kernel requires storing 64-bits, 128-bit, and 256-bits words per diagonal of the wavefront and therefore require more shared memory as the alignment error supported increases. The execution of these kernels display different performance tradeoffs discussed in Section IV-C.

It is important to note that the length of the backtrace vector imposes a limit on the maximum alignment error but not on the maximum sequence supported. For instance, the E128 implementation could be used to align sequences of 1000 nucleotides up to a 12.8% error rate or 10K long sequences up to a 1.28% error rate. For moderately long sequences (i.e., between 100 and 1000 nucleotides), our implementation supports alignments up to more than a 10% error rate. Nevertheless, it is possible to extend this approach to higher error rates, using longer bit-vectors, at the cost of using more memory and potential performance slowdowns (see Section IV-C).

C. BIT-PARALLEL PACKED SEQUENCE COMPARISON

As opposed to the *computeNext()* operation, the *extend()* operation can require performing a different amount of computations per diagonal. Specifically, the inner loop of Algorithm 1 iterates as many times as the total characters that match along each diagonal. Thus, threads within a block executing this operation are bound to diverge, which can diminish the overall performance.

To mitigate this problem, we use a packed sequence encoding that allows performing bit-parallel sequence comparisons; that is, comparing blocks of characters, anticipating comparisons, and reducing the variability between diagonals. Taking advantage of the reduced DNA alphabet (i.e., nucleotides A, C, G, and T), we propose to use a 2bits-packed encoding scheme to increase the number of nucleotides compared per block (i.e., 16 nucleotides per 32 bits word). Furthermore, packing and reducing the size of the input sequences reduces the memory requirements on the shared memory and, in turn, allows fitting more CUDA blocks in the same SM.

Nonetheless, this approach introduces the need of packing the input sequences beforehand. Sequence packing can be performed on the host CPU, or it can be offloaded to the GPU. Although packing sequences on CPU would help to reduce the amount of data that has to be transferred to the GPU, packing computations and memory transfers can be overlapped with the alignment kernels (see Section III-D). Not to mention that current high-speed transfer technologies, such as NVLink, allow even faster transfers from the host to the device. For instance, using a Nvidia V100, the offloading of raw sequences and packing on the GPU turns out to be faster than packing the sequences on the CPU and transferring the packed sequences.

Furthermore, sequence packing turns out to be a straightforward operation. Due to the ASCII representation of the DNA letters (i.e., A=1000**00**1, C=1000**01**1, G=1000**11**1, T=1010**10**0), it only requires to extract the bits on position 1 and 2 (unique bits in every DNA letter encoded in the ASCII). This encoding has been extensively used in multiple bioinformatics and genomics applications for packing DNA sequence databases and genome references. However, our implementation does not assume the preprocessing of

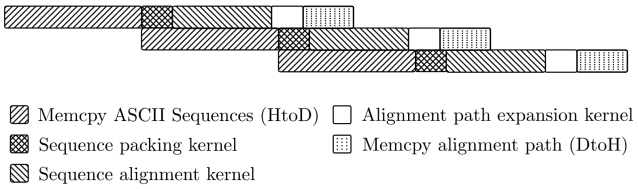


Figure 4. Compute kernels of multiple batches are overlapped with data transfers (DthH and HtoD).

the input sequences and allows using ASCII-encoded DNA sequences, packing its content on the GPU.

Altogether, this approach accelerates the computations performed within the *extend()* kernel, decreasing the execution divergence between threads, and reducing the number of instructions executed as well as the overall shared memory used. Compared to the vanilla implementation, our experiments show that this strategy accelerates the kernel execution time from 1.6× to 1.9× and reduces the number of executed instructions by a factor of 1.7× to 2.1×. Most importantly, it reduces between 1.2× and 1.7× the number of predicated-off threads in a warp (i.e., inactive threads when divergent branches occur and threads take separated paths).

D. BATCH EXECUTION. OVERLAPPING KERNEL COMPUTATION WITH DATA TRANSFERS

At the system level, memory transfers from host to device take a significant percentage of the total execution time since all the sequences have to be stored in the device to perform the alignment. Hiding transfer latencies with computation is key to avoid performance slowdowns due to the offloading of computation to the GPU. The CUDA programming model allows the creation of various streams to overlap computing kernels with memory transfers. All operations within a CUDA stream are synchronous; however, they can operate asynchronously between other running streams. As a result, launching independent kernels and memory transfers to different CUDA streams can effectively overlap computation with memory transfers.

To effectively implement this strategy, we created batches of sequences to be transferred and aligned in parallel. This way, compute kernels of a given batch can be overlapped with computations and memory transfers from other batches. This concept is illustrated in Figure 4.

IV. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of the eWFA-GPU. We compare our implementation against state-of-the-art libraries and tools for pairwise sequence alignment. Then, we present a detailed study of the performance, scalability, and limitations of our implementation, showing a comprehensive profiling of the kernel executions on GPU. Afterwards, we evaluate the performance effect of parameter tuning our implementation and conclude presenting an evaluation on other GPU devices.

A. EXPERIMENTAL SETUP

We performed the experimental evaluation of our solution on an IBM Power9 processor (20 cores with 4 threads per core), equipped with an NVIDIA V100 GPU with 16GB of HBM2 memory connected through NVLink. We used synthetic datasets consisting of 10 million sequence pairs of lengths 150, 300, and 1000 nucleotides, and error rates of 2%, 5%, and 10%. For comparison, we selected representative and widely-used libraries and tools from the state-of-the-art. We focused on those CPU and GPU implementations that stand out in terms of performance or implement the latest algorithmic approaches.

For the CPU evaluation, we selected Edlib [48]; eWFA, an optimised CPU version of the WFA [39] adapted to the edit-distance; BPM, a highly optimised version of the BitParallel Myers algorithm [37]; and the $O(ND)$ algorithm [45] used at the core of the Linux diff-tool. All CPU executions were performed using 80 threads.

From the multiple GPU implementations available, we have selected those that could be deployed, executed without faults, and had a competitive execution time. In particular, we evaluated two methods from the widely-used NVBio [51] framework, the WmCudaTile algorithm from xbitpar [57], and the highly optimised GASAL2 [58]. Note that NVBio implementation only computes the alignment distance, not producing the complete alignment. Also, note that GASAL2 implements the gap-affine distance and, consequently, requires more computation than edit-distance. Notwithstanding, its inclusion in the benchmark is interesting for comparison purposes. We tuned GASAL2's gap-affine parameters to this end, so the library computes the edit-distance alignment.

B. PERFORMANCE EVALUATION

In order to present a comprehensive evaluation of the different methods' performance, Table 1 shows the alignment time taken by each implementation for aligning 10 million sequences of different lengths and error rates. We report total execution time, including transfer times (i.e., host to device and back) for the GPU executions. All CPU implementations were executed using 80 threads. Overall, results show that eWFA-GPU executes 2.9-265× faster than the CPU-based methods and 8-56× faster than other GPU implementations.

Compared to established CPU alignment algorithms, eWFA-GPU performs 24-102× faster than the BPM algorithm and 19-100× faster than the O(ND) implementation. Similarly, we obtain speedups of 31-265× compared to Edlib. Compared to the CPU implementation of the eWFA, our GPU implementation delivers 3-9× times more performance. In particular, the speedups obtained by eWFA-GPU increase with higher alignment error rates as the wavefronts increase in size and more wavefront computation can be done in parallel (see Section IV-C).

Regarding the GPU implementations, eWFA-GPU outperforms the widely-used NVBio library, achieving speedups of 2.5-7.4× compared to NVBio's classical DP-based implementation and speedups of 4.5-7.2× compared to NVBio's

Table 1. Alignment time (in milliseconds) for an input of 10 million alignments using different alignment implementations on CPU and GPU. Note that CPU implementations were executed using 80 threads. The first half of the table presents alignment times of implementations that only compute the edit-distance (not the alignment). The second half shows alignment times of implementations that compute the edit-distance and the full alignment path. Best execution times are marked in bold.

			length=150			length=300			length=1000		
			e=2%	e=5%	e=10%	e=2%	e=5%	e=10%	e=2%	e=5%	e=10%
Distance only	GPU	NVBIO.DP	622	656	642	1013	1020	1050	2534	2506	2446
		NVBio.BPM	608	630	658	856	833	894	2474	n/a	n/a
		eWFA-GPU	**89**	**92**	**100**	**143**	**140**	**197**	**399**	**487**	**994**
Full Alignment	CPU	Edlib	8928	9001	9057	16853	16338	16591	120248	50988	59799
		O(ND)	7872	7798	7444	14456	14350	13873	21217	38553	36110
		BPM	6773	7204	6985	12484	12340	12526	46062	47043	45610
		eWFA	577	758	1110	672	1150	2090	1310	3900	11500
	GPU	GASAL2	1206	1228	1239	4365	4366	4394	n/a	n/a	n/a
		wmCudaTile	734	1074	1473	2142	2836	5038	13706	30571	108447
		eWFA-GPU	**91**	**95**	**116**	**144**	**160**	**252**	**453**	**689**	**1928**

BPM. Compared to wmCudaTile, eWFA-GPU achieves up to 12× speedup for short sequences (i.e., 150 nucleotides) and up to 56× speedup for longer sequences. Compared to GASAL2, eWFA-GPU is 10-30× faster. In general, eWFA-GPU execution time scales better with the sequence length, compared to the other GPU implementations. In particular, the performance of DP-based methods, like GASAL2, is strongly limited by the sequence length. Ultimately, aligning long sequences with GASAL2 becomes impractical (e.g., 1000 nucleotides or more). For a fair comparison, it is important to acknowledge that GASAL2 implements the gap-affine distance, which is more complex and costly than computing the edit-distance alignment.

Unsurprisingly, DP-based implementations (i.e., BPM, Edlib, NVBio, and GASAL2) are insensitive to the alignment error, performing the same amount of computations to align similar sequences as to align very divergent ones. As a result, the performance of classical DP-based algorithms is heavily constrained by the sequence length and not by the sequences homology. For that reason, some tools, like Edlib, implement heuristics that prune the DP computations at the expense of potentially missing the optimal alignment (note the reduction in the execution time when aligning sequences of 1000 nucleotides with e>=5%). In contrast, error-sensitive methods, like the eWFA-GPU, perform faster when aligning highly similar sequences, exploiting similarities between the sequences to accelerate the alignment process. These methods are only constrained by the nominal amount of differences between the sequences.

C. PROFILING, SCALABILITY, AND LIMITATIONS

Our solution relies on exploiting the fast on-chip memory of the GPU to improve the execution time. As explained in Section III, our implementation stores the algorithm's working set (i.e., sequences, offsets, and backtraces) in shared memory, enabling fast accesses at the expense of limiting the maximum amount of memory that each alignment can use. As the shared memory required by the algorithm grows quadratically with the alignment error, the memory consumed by the offsets and backtraces becomes the most limiting factor. In turn, increasing the shared memory consumed per each alignment limits the amount of thread blocks that can be executed concurrently on each SM. Therefore, the maximum alignment error supported strongly constrains the number of alignments that can be processed on each SM, thus limiting the performance and scalability of the solution to high error rates. Due to these limitations, our solution implements three specialised alignment kernels, each supporting a different maximum number of errors per alignment (i.e., 32, 64, and 128 errors; see Section III-B). In this section we show that selecting the proper kernel, adjusted the maximum expected alignment error, is crucial to obtain the best performance.

1) Overall system profiling

Having three different specialised kernels, the performance of the executions change depending on the alignment error between the sequences. Fig. 5 shows the application execution times aligning datasets with different error rates, broken down into transference time (i.e., HtoD and DtoH), kernels execution time, and total execution time. In the figure, each execution is represented using three columns: the first one showing the aggregated time of the memory copies between CPU and GPU, the second one showing the GPU kernels computation times, and the third one showing the overall execution time. Note how transference times are being effectively overlapped with kernel computations.

In particular, when aligning homologous sequences (e.g., 20 differences between the sequences) with the E32 kernel, we observe that data transfers become the main performance bottleneck. In this case, the kernels' computation can be effectively overlapped with transfers (disregarding initialisation times), resulting in the fastest execution times. As the number of differences increases, our implementation requires using kernels that support higher error rates. In these scenarios (E64 and E128), the kernel's computing time overtakes the transfer time and becomes the main bottleneck. Most notably, the alignment kernel time increases with higher error rates (specially, due to increments in the size of the backtrace

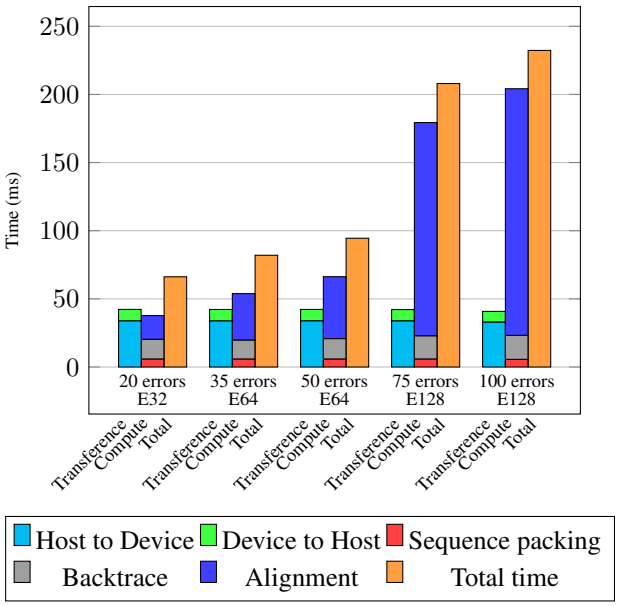


Figure 5. Application execution time broken down into transference time (i.e., host to device and device to host), kernel execution time, and total execution time. Each execution was performed using a dataset contain 1 million sequences of 1000 nucleotides with different error rates (i.e., 20, 35, 50, 75, and 100 nominal errors).

vectors from E64 to E128). As opposed, transfer, packing, and backtrace times remain constant across all executions for all datasets used (i.e., sequences of 1000 nucleotides).

2) Alignment kernel performance profiling

Due to its significance, we focus on the alignment kernel to characterise its performance and understand the GPU resource utilisation. Table 2 reports a summary of the most relevant performance metrics of the execution of the three alignment kernel specialisations.

Concerning memory utilisation, the alignment kernel only accesses global memory at the beginning of the execution to copy the input sequences into shared memory. Due to the limited usage of global memory, the effective throughput reached is very low and rapidly decreases as the compute time grows for executions using higher alignment error rates. For the rest of the execution, the alignment kernel only accesses the fast on-chip shared memory.

Table 2. Performance metrics of each specialised alignment kernel on the Nvidia V100 GPU. Executions were performed using datasets of 1M sequences of 1000 nucleotides. Each dataset contains sequences that align with an average error rate of 2%, 5%, and 10% (i.e., 20, 50, and 100 nominal differences). Each execution was performed using the minimum alignment error supporting kernel (i.e., E32, E64, E128)

Dataset:	1000nt 2% error	1000nt 5% error	1000nt 10% error
Alignment kernel executed	E32	E64	E128
Maximum error supported	32	64	128
Threads per block.	32	64	128
Shared memory per block (KiB)	2.14	5.74	19.08
Occupancy (active warps per SM)	31.86	31.87	19.94
Kernel time (ms)	17.27	45.19	190.36
SM busy (%)	87.74	82.50	61.96
Global memory throughput (GiB/s)	35.86	14.05	3.46
Executed warp instructions ($x10^9$)	6.31	15.47	48.82
Avg. active threads per warp	10.10	21.36	27.40
Warp cycles per issued instruction	9.08	9.66	8.04

Regarding computation on the GPU, in Table 2 we observe that all the alignment kernel specialisations are consistently between 60% and 87.74% of the maximum SM core instruction throughput (i.e., SM busy). Furthermore, a more detailed profiling reveals that none of the SM computing pipelines is fully saturated. In particular, the most used computing pipeline, the ALU pipeline, reaches a 87% utilisation on the E32 kernel, 80% on the E64 kernel, and 30% utilisation on the E128 kernel. Additionally, note that the warp stall time (i.e., warp cycles per issued instruction) remains similar across all executions.

These results reveal that the real limiting factor of these executions is not the lack of computing resources on the GPU but the lack of computing parallelism. When aligning up to higher alignment error rates, the wavefronts become larger; and thus, an SM can exploit more threads to perform parallel computations. Accordingly, Table 2 shows that the average active threads per warp increases from 10.1 to 27.4 (out of a maximum of 32 threads per warp) when executing kernels with higher alignment error support. In turn, this increase in parallelism is reflected on the total warp instructions executed. As the alignment error increases, we would expect an $O(e^2)$ increase in the number of warp instructions. However, we observe a much gentle growth alleviated by the utilisation of more threads per each warp.

Nevertheless, this increase in the number of active threads per warp does not immediately translates into higher SM utilisation (i.e., SM busy). Note that higher alignment error supporting kernels require more shared memory per block (Table 2). Therefore, the maximum number of active warps per SM is bounded by the total shared memory available and the shared memory required per block. Table 2 shows that the occupancy drops from 31.86 to 19.94 when aligning sequences up to 100 nominal differences using the E128 kernel. As a result, the SM busy and the computing pipelines usage is reduced from 87.74% to 61.96%. Ultimately, as the profiling results show, the performance of the alignment kernel execution attends to a trade-off between the shared memory required by each thread block and the maximum active threads per warp that can be exploited to perform the alignment computations.

3) Alignment kernel selection

In order to maximise performance, it is crucial to select the alignment kernel that minimises the shared memory consumption while being capable of aligning up to the maximum error required by the input dataset. Table 3 presents the performance results from using the three different kernel specialisation to align the same dataset. First, we can observe

how gradually each alignment kernel requires more shared memory (from 1.65KiB up to 18.61KiB per thread block), reducing the occupancy (from 31.50 down to 4.88), and ultimately leading to longer kernel execution times (i.e., an slowdown of 11× from E32 to E128). When using the same dataset, all three executions compute the same alignments and process wavefronts of the same length. Consequently, the effective parallelism attained is the same for all the kernels (i.e., average active threads per warp) and the executed warp instructions remains constant for all the executions (ignoring overheads associated to operating with longer backtrace vectors). Hence, the maximum amount of parallel computations depends on the maximum alignment error, not on the alignment kernel specialisation. Considering that the three kernels are capable of supporting the maximum alignment error of the dataset, selecting an oversized kernel can lead to a slowdown up to 3.8×.

In conclusion, utilising the best fitted kernel (in terms of maximum alignment error supported and shared memory consumed) is key for performance. Specially, for alignments with a small alignment error where the parallelism is rather limited and only a few threads per block can effectively compute useful work in parallel. Balancing the number of alignments per SM and the maximum number of active threads per block is crucial for an efficient exploitation of the GPU computing resources.

D. PARAMETER TUNING

Most often, GPU-based implementations depict specific parameters that can strongly impact performance and have to be configured with caution. For the eWFA-GPU, the number of threads per block (and, therefore, the number of threads per alignment) determines the maximum work that can be done in parallel computing an alignment. If there are more threads than wavefront elements, some threads never perform useful work, and GPU resources are not efficiently used. Conversely, if a wavefront is larger than the number of threads in a block, the implementation requires multiple iterations; hence, losing parallelism.

Table 3. Performance metrics of each specialised alignment kernel on the Nvidia V100 GPU. All executions were performed using 32 threads per block, aligning a dataset of 1M sequences of 150 nucleotides with an average error rate of 5% (i.e., average of 7.5 nominal differences). Each execution was performed using a different alignment kernel; that is, E32, E64, and E128.

Dataset: 150 nucleotides (5% alignment error)	Kernel E32	Kernel E64	Kernel E128
Maximum error supported	32	64	128
Shared memory per block (KiB)	1.65	5.27	18.61
Occupancy (active warps per SM)	31.50	17.63	4.88
Kernel time (ms)	3.93	4.91	14.96
SM busy (%)	92.95	79.37	28.43
Global memory throughput (GiB/s)	37.15	33.00	12.43
Executed warp instructions (x10^9)	1.51	1.61	1.76
Avg. active threads per warp	10.48	11.34	13.03
Warp cycles per issued instruction	8.46	5.54	4.28

Table 4. Alignment time (in milliseconds) of 10 million alignments using a different number of threads per block. All the datasets used for this comparison have a 10% error rate.

Nucleotides	Threads per block			
	32	64	128	256
150	**116.2**	121.3	149.5	228.9
300	**252.0**	255.8	327.9	436.6
1000	2795.2	2091.0	**1928.1**	2092.4

Table 4 lists the performance trade-offs using a different number of threads per block. For short sequences and small error rates, using small blocks (e.g., one warp) reduces the number of idle threads per block. In the case of short and medium sequences (i.e., 150-300 nucleotides), using 32 and 64 threads per block gives very similar performance results. However, using more threads per block leads to a performance drop caused by idle threads consuming GPU resources. Similarly, for aligning long sequences (i.e., 1000 nucleotides), the best performance is achieved by using 128 threads per block. Note that using fewer threads per block leads to an underutilization of GPU resources and, using more threads, to a waste of GPU resources by idle threads.

E. EVALUATION ON OTHER DEVICES

To offer a thorough analysis of the performance of the proposed solution, we also evaluated our implementation using two other GPU models: an Nvidia GeForce RTX 2080 Ti and an Nvidia GeForce RTX 3080. The computing capabilities of each device used are listed in Table 5.

The results of the execution on other GPU devices are shown in Table 6. On the GeForce RTX 2080 Ti, our implementation is bounded by the bandwidth between the CPU and the GPU. The device is connected through PCI Express, achieving a bandwidth of 13GB/s on average. For instance, a batch of 10 million sequences of 1000 nucleotides represents 21GB of input data. Transferring all this data to the GPU using the available peak bandwidth of 13GiB/s would take 1615 milliseconds. That is about 87% of the total execution

Table 5. Properties of devices used for evaluation.

	V100	RTX 2080 Ti	RTX 3080
Compute capability	7.0	7.5	8.6
Clock frequency (MHz)	877	1605	1710
SMs	80	68	68
Cores	5120	4352	8704
Maximum warps per SM	64	32	48
Register space per SM (KiB)	256	256	256
Shared memory per SM (KiB)	96	64	100
Global memory size (GiB)	16	11	10
L2 cache size (KiB)	6144	5632	5120
Host to Device bandwidth (GB/s)	67.1	13.2	12.3
Device to Host bandwidth (GB/s)	65.8	13.2	13.1

Table 6. Alignment time (in milliseconds) of 10 million alignments using eWFA-GPU on different devices.

Average nucleotides	Error	V100	RTX 2080 Ti	RTX 3080
150	2%	91	299	333
	5%	95	298	332
	10%	116	301	335
300	2%	144	555	585
	5%	160	563	585
	10%	252	565	590
1000	2%	453	1864	1921
	5%	689	1878	1989
	10%	1928	2900	2103

time. Even with the proposed strategy to overlap computation with transfers, the overall execution time is bounded by data transfers to the device.

In the case of the RTX 3080, most execution times are similar to the RTX 2080 results, as they have similar PCI Express bandwidth. Overall, computation kernels are faster than memory transfers and can be effectively overlapped. However, when aligning 1000 nucleotides long sequences with 10% of error, computation kernels take more time than memory transfers, mainly due to the intensive usage of shared memory. As shown in Table 5, the RTX 3080 has more shared memory available per SM than other devices, allowing it to have more alignments per SM, and therefore, achieving better performance than the RTX 2080.

V. RELATED WORK

Over the years, many efforts have been invested in finding new algorithms and more efficient implementations to compute pairwise edit-distance alignments. In [59], Navarro provides a comprehensive review of the most relevant algorithms and a performance evaluation for different datasets and configurations. Most alignment algorithms can be classified into four categories: DP-based, automaton, filters, and bit-parallel algorithms. In practice, bit-parallel algorithms outperform the rest approaches. Most notably, these include the BPM [37], the O(ND) [45], and the Wu-Manber (WM) [35] algorithms.

Based on the most successful algorithmic approaches, many high-performance CPU libraries have been presented. Some of them have become extensively used due to their efficiency or versatility, most notably, Edlib [48], BGSA [49], and SeqAn [60]. Edlib is an efficient CPU implementation of the BPM algorithm used within many Bioinformatics tools. BGSA is also a very efficient implementation of the BPM algorithm, optimised to exploit vectorization on multi-core and many-core CPUs. SeqAn is a sequence analysis library that implements a hybrid algorithm that combines the memory-efficient Hirschberg's algorithm [60] with the BPM algorithm.

Additionally, there have been many efforts to adapt and optimise these algorithms on GPU devices. Most relevant proposals are based on DP, computing cells antidiagonal-wise in parallel [61, 62, 63, 64, 65]. Meanwhile, some research efforts have been focused on producing efficient CUDA implementation of the classical Needleman-Wunsch [66] algorithm; other proposals have focused on novel organisations of the DP-matrix to exploit efficiently the GPU resources [67]. In particular, in [68] and [69], the authors propose an algorithm to reduce memory operations when computing the DP-matrix, by using *warp-shuffle* instructions of current Nvidia GPU architectures.

Many other GPU-based methods have opted for accelerating bit-parallel algorithms. In [57], the authors propose using warp-shuffle operations to simulate a 1024-bit machine word, allowing to perform approximate string matching on long patterns. Also, in [70], the authors exploit the Crochemore algorithm based on Suffix automaton for bit-parallel alignment. Like [71], other proposals revisit the Shift-Or and Wu-Manber algorithms, implementing them as inclusive-scan operations to allow multiple parallel computations. Similarly, in [30] the authors propose a thread-cooperative version of the BPM algorithm, achieving very high performance results in a Nvidia GTX 680 GPU.

Furthermore, there has been many proposal to optimise sequence alignment on field programmable gate array devices (FPGA)[72, 73, 74, 75]. Most notable FPGA implementations exploit bit-parallel techniques and custom processing designs to accelerate the computation of multiple alignments in parallel.

Comparing the performance of multiple methods implemented on different hardware platforms can be a challenging task. For the purpose of making meaningful comparisons, it is common to compare the peak number of Giga Cells Updated Per Second (GCUPS) achieved by each implementation. GCUPS is an established metric used to measure the performance of alignment algorithms regardless of the target devices and other implementation specifics. It represents the number of cells from the DP-matrix computed per second by each implementation. GCUPS can be computed using Eq. 3 for an alignment of two sequences of length n and m, taking s seconds. This way, Table 7 compares peak GCUPS reported by the most relevant implementations. Note that the eWFA-GPU algorithm doesn't require computing the full DP-matrix to obtain the optimal alignment. Even so, for a fair comparison, we report the total number of CUPS required to compute to obtain the same alignment as our implementation. Overall, our solution obtains between 8-1790× more GCUPs than other GPU implementations. Notwithstanding the inherent inaccuracies of this comparison method, it is significant that eWFA-GPU produces 2 orders of magnitude more GCUPS than the most efficient methods found in the literature.

$$GCUPS = \frac{nm}{s} \times 10^{-9} \tag{3}$$

VI. CONCLUSION

This paper presents eWFA-GPU, a GPU-accelerated algorithm based on the WFA algorithm to compute the edit-

Table 7. Peak GCUPs of different edit-distance alignment tools as reported on their work.

Device	Paper	Year	Device Model	GCUPs
GPU	Ours	2022	Tesla V100	22075
	[71]	2016	GeForce GTX TITAN X	2800
	[30]	2014	Geforce GTX 680	2300
	[51]	2014	Tesla K40c	1000
	[76]	2013	Geforce GTX 480	470
	[77]	2013	Geforce GTX 480	470
	[57]	2016	Tesla V100	420
	[58]	2019	Tesla V100	206
	[68]	2015	GeForce GTX 980	65
	[78]	2016	GeForce GTX 960	50
	[70]	2015	GeForce GTX 580	28
	[79]	2020	GeForce GTX TITAN Black	14
	[55]	2018	Tesla K40c	14
CPU	[48]	2017	Intel i7-4710HQ	388
	[80]	2016	Intel Xeon E5-2670	136
	[60]	2008	3.2 GHz Intel Xeon	2
FPGA	[73]	2019	Kintex KCU1500	161
Others	[49]	2018	Intel Xeon Phi-7210	1895

distance. Our implementation provides exact edit-distance alignment (i.e., not heuristic), outperforming other state-of-the-art methods. Also, we present the piggybacked backtrace strategy, a novel optimisation technique that dramatically reduces the amount of memory needed for aligning sequences. Not only this technique requires storing only two wavefronts (fitting in the fast shared GPU memories), it also makes the alignment generation faster. Additionally, we implemented a high-performance sequence packing kernel that allows block-wise comparisons between sequences. This accelerated operation significantly improves one of the most time-consuming operations of the WFA (the *extend* operator). Moreover, our implementation is fully asynchronous and overlaps compute kernels and memory transfers to accelerate the algorithm execution, hiding memory transfer latencies with computation.

We compared our eWFA-GPU implementation against other CPU alignment libraries. Results obtained on the Nvidia V100 GPU demonstrate speedups up to 265× compared to Edlib, and up to 9.2× compared with the CPU version of the WFA algorithm. Also, we obtain speedups up to 101.7× compared to the BPM, and up to 100.4× compared to the O(ND) CPU implementation. Additionally, we compared our implementation against GPU aligners: wmCudaTile from XBitPar, GASAL2, and NVBio. We achieve a speedup up to 56.2× compared to wmCudaTile, up to 30.3× compared to GASAL2, and up to 7.4× compared with NVBio. Beware that GASAL2 is capable of computing gap-affine distance (hence it performs more work).

All in all, our implementation represents an efficient solution for applications that require fast computation of exact edit-distance alignment of large DNA sequence datasets. eWFA-GPU is MIT-licence open-source, and its code is publicly available at https://github.com/quim0/eWFA-GPU.

VII. FUNDING

This research was supported by the European Unions's Horizon 2020 Framework Programme under the DeepHealth project [825111], by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of total cost eligible under the DRAC project [001-P-001723]. It was also supported by the Ministerio de Ciencia e Innovacion MCIN AEI/10.13039/501100011033 under contracts PID2020-113614RB-C21 and TIN2015-65316-P and by the Generalitat de Catalunya GenCat-DIUiE(GRR) (contracts 2017-SGR-313, 2017-SGR-1328 and 2017- SGR-1414). M.M. was partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104.

References

[1] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. "Consequences of faster alignment of sequences". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2014, pp. 39–51.

[2] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. ACM press New York, 1999.

[3] Sandeep Kumar and Eugene H Spafford. "A pattern matching model for misuse intrusion detection". In: (1994).

[4] Dan Gusfield. "Algorithms on stings, trees, and sequences: Computer science and computational biology". In: *Acm Sigact News* 28.4 (1997), pp. 41–60.

[5] Gautam Das et al. "Episode matching". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1997, pp. 12–27.

[6] Tommi A Pirinen and Krister Lindén. "State-of-the-art in weighted finite-state spell-checking". In: *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer. 2014, pp. 519–532.

[7] Rafael C Gonzalez and Michael G Thomason. "Syntactic pattern recognition: an introduction". In: (1978).

[8] Jasha Droppo and Alex Acero. "Context dependent phonetic string edit distance for automatic speech recognition". In: *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2010, pp. 4358–4361.

[9] Zhongwen Ying and Thomas G Robertazzi. "Signature searching in a networked collection of files". In: *IEEE Transactions on Parallel and Distributed Systems* 25.5 (2013), pp. 1339–1348.

[10] Tomasz Luczak and Wojciech Szpankowski. "A suboptimal lossy data compression based on approximate pattern matching". In: *IEEE transactions on Information Theory* 43.5 (1997), pp. 1439–1451.

[11] Zhan Su et al. "Plagiarism detection using the Levenshtein distance and Smith-Waterman algorithm". In: *2008 3rd International Conference on Innovative Computing Information and Control*. IEEE. 2008, pp. 569–569.

[12] Daniel Lopresti and Andrew Tomkins. "On the searchability of electronic ink". In: *Proceedings of the 4th International Workshop on Frontiers in Handwriting Recognition*. Citeseer. 1994, pp. 156–165.

[13] David Sankoff. "Time warps, string edits, and macromolecules". In: *The Theory and Practice of Sequence Comparison, Reading* (1983).

[14] Jens Heine et al. "Algorithm for driver intention detection with fuzzy logic and edit distance". In: *2015 IEEE 18th international conference on intelligent transportation systems*. IEEE. 2015, pp. 1022–1027.

[15] Stefan Jakšić et al. "Quantitative monitoring of STL with edit distance". In: *Formal methods in system design* 53.1 (2018), pp. 83–112.

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

[17] Santiago Marco-Sola et al. "The GEM mapper: fast, accurate and versatile alignment by filtration". In: *Nature methods* 9.12 (2012), pp. 1185–1188.

[18] Angelika Merkel et al. "gemBS: high throughput processing for DNA methylation data from bisulfite sequencing". In: *Bioinformatics* 35.5 (2019), pp. 737–742.

[19] Jared T Simpson et al. "ABySS: a parallel assembler for short read sequence data". In: *Genome research* 19.6 (2009), pp. 1117–1123.

[20] Sergey Koren et al. "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation". In: *Genome research* 27.5 (2017), pp. 722–736.

[21] Aaron McKenna et al. "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data". In: *Genome research* 20.9 (2010), pp. 1297–1303.

[22] Bernardo Rodrıguez-Martın et al. "ChimPipe: accurate detection of fusion genes and transcription-induced chimeras from RNA-seq data". In: *BMC genomics* 18.1 (2017), pp. 1–17.

[23] Cédric Notredame, Desmond G Higgins, and Jaap Heringa. "T-Coffee: A novel method for fast and accurate multiple sequence alignment". In: *Journal of molecular biology* 302.1 (2000), pp. 205–217.

[24] Richard Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[25] Neil C Jones, Pavel A Pevzner, and Pavel Pevzner. *An introduction to bioinformatics algorithms*. MIT press, 2004.

[26] Zachary D Stephens et al. "Big data: astronomical or genomical?" In: *PLoS biology* 13.7 (2015), e1002195.

[27] Lauren M Petersen et al. "Third-generation sequencing in the clinical laboratory: exploring the advantages and challenges of nanopore sequencing". In: *Journal of Clinical Microbiology* 58.1 (2019), e01315–19.

[28] Wen-Mei W Hwu. *GPU computing gems emerald edition*. Morgan Kaufmann Publishers Inc., 2011.

[29] John D Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[30] Alejandro Chacón et al. "Thread-cooperative, bit-parallel computation of levenshtein distance on GPU". In: *Proceedings of the 28th ACM international conference on Supercomputing*. 2014, pp. 103–112. DOI: 10.1145/2597652.2597677.

[31] Cheng-Hung Lin et al. "Perfect hashing based parallel algorithms for multiple string matching on graphic processing units". In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), pp. 2639–2650.

[32] Peter H Sellers. "The theory and computation of evolutionary distances: pattern recognition". In: *Journal of algorithms* 1.4 (1980), pp. 359–373.

[33] Esko Ukkonen. "Finding approximate patterns in strings". In: *Journal of algorithms* 6.1 (1985), pp. 132–137.

[34] Ricardo A Baeza-Yates. "Text-Retrieval: Theory and Practice." In: *IFIP Congress (1)*. Vol. 12. Citeseer. 1992, pp. 465–476.

[35] Sun Wu and Udi Manber. "Fast text searching: allowing errors". In: *Communications of the ACM* 35.10 (1992), pp. 83–91.

[36] Gonzalo Navarro12. "A partial deterministic automaton for approximate string matching". In: (1997).

[37] Gene Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 395–415.

[38] Ricardo Baeza-Yates. *Efficient text searching*. University of Waterloo, 1989.

[39] Santiago Marco-Sola et al. "Fast gap-affine pairwise alignment using the wavefront algorithm". In: *Bioinformatics* 37.4 (2021), pp. 456–463.

[40] Alden H Wright. "Approximate string matching using withinword parallelism". In: *Software: Practice and Experience* 24.4 (1994), pp. 337–362.

[41] Hajime Suzuki and Masahiro Kasahara. "Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming". In: *BioRxiv* (2017), p. 130633.

[42] Torbjørn Rognes and Erling Seeberg. "Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors". In: *Bioinformatics* 16.8 (2000), pp. 699–706.

[43] Michael Farrar. "Striped Smith–Waterman speeds database searches six times over other SIMD implementations". In: *Bioinformatics* 23.2 (2007), pp. 156–161.

[44] Andrzej Wozniak. "Using video-oriented instructions to speed up sequence comparison". In: *Bioinformatics* 13.2 (1997), pp. 145–150.

[45] Eugene W Myers. "An O(ND) difference algorithm and its variations". In: *Algorithmica* 1.1-4 (1986), pp. 251–266.

[46] Mengyao Zhao et al. "SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications". In: *PloS one* 8.12 (2013).

[47] Alberto Zeni et al. "Logan: High-performance gpu-based x-drop long-read alignment". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 462–471.

[48] Martin Šošić and Mile Šikić. "Edlib: a C/C++ library for fast, exact sequence alignment using edit distance". In: *Bioinformatics* 33.9 (2017), pp. 1394–1395.

[49] Jikai Zhang et al. "BGSA: A bit-parallel global sequence alignment toolkit for multi-core and many-core architectures". In: *Bioinformatics* 35.13 (2019), pp. 2306–2308.

[50] Gene Myers. "Efficient local alignment discovery amongst noisy long reads". In: *International Workshop on Algorithms in Bioinformatics*. Springer. 2014, pp. 52–67.

[51] Subtil N Pantaleoni J. *NVBIO*. https://nvlabs.github.io/nvbio. Accessed: 2021-09-15. 2015.

[52] Esko Ukkonen. "Algorithms for approximate string matching". In: *Information and control* 64.1-3 (1985), pp. 100–118.

[53] Alejandro Chacón et al. "Boosting the FM-index on the GPU: Effective techniques to mitigate random memory access". In: *IEEE/ACM transactions on computational biology and bioinformatics* 12.5 (2014), pp. 1048–1059.

[54] Alejandro Chacón et al. "FM-index on GPU: A cooperative scheme to reduce memory footprint". In: *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2014, pp. 1–9.

[55] Keh Kok Yong and Hong Hoe Ong. "Accelerating Bit-Parallel Approximate Matching On GPU Platforms For Small Patterns". In: *2018 Fourth International Conference on Advances in Computing, Communication & Automation (ICACCA)*. IEEE. 2018, pp. 1–5.

[56] Khaled Balhaf et al. "Accelerating Levenshtein and Damerau edit distance algorithms using GPU with unified memory". In: *2017 8th international conference on information and communication systems (ICICS)*. IEEE. 2017, pp. 7–11.

[57] Tuan Tu Tran, Yongchao Liu, and Bertil Schmidt. "Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi". In: *Parallel Computing* 54 (2016), pp. 128–138.

[58] Nauman Ahmed et al. "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data". In: *BMC bioinformatics* 20.1 (2019), pp. 1–20.

[59] Gonzalo Navarro. "A guided tour to approximate string matching". In: *ACM computing surveys (CSUR)* 33.1 (2001), pp. 31–88.

[60] Andreas Döring et al. "SeqAn an efficient, generic C++ library for sequence analysis". In: *BMC bioinformatics* 9.1 (2008), pp. 1–9.

[61] Amine Dhraief, Raik Issaoui, and Abdelfettah Belghith. "Parallel computing the longest common subsequence (LCS) on GPUs: efficiency and language suitability". In: *The 1st International Conference on Advanced Communications and Computation (INFOCOMP)*. 2011.

[62] Ayumu Tomiyama and Reiji Suda. "Automatic parameter optimization for edit distance algorithm on GPU". In: *International Conference on High Performance Computing for Computational Science*. Springer. 2012, pp. 420–434.

[63] Khaled Balhaf et al. "Using gpus to speed-up levenshtein edit distance computation". In: *2016 7th International Conference on Information and Communication Systems (ICICS)*. IEEE. 2016, pp. 80–84.

[64] Zuqing Li, Aakashdeep Goyal, and Haklin Kimm. "Parallel longest common sequence algorithm on multicore systems using openacc, openmp and openmpi". In: *2017 IEEE 11th international symposium on embedded multicore/many-core systems-on-chip (MCSoC)*. IEEE. 2017, pp. 158–165.

[65] Kailash W Kalare et al. "Parallelization of Global Sequence Alignment on Graphics Processing Unit". In: *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*. IEEE. 2020, pp. 1–5.

[66] Da Li and Michela Becchi. "Multiple pairwise sequence alignments with the needleman-wunsch algorithm on GPU". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 1471–1472.

[67] Reza Farivar et al. "An algorithm for fast edit distance computation on GPUs". In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–9.

[68] Lucas SN Nunes et al. "A fast approximate string matching algorithm on GPU". In: *2015 Third international symposium on computing and networking (CANDAR)*. IEEE. 2015, pp. 188–192.

[69] ThienLuan Ho, Seung-Rohk Oh, and HyunJin Kim. "A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations". In: *PloS one* 12.10 (2017), e0186251.

[70] Katsuya Kawanami and Noriyuki Fujimoto. "A GPU Implementation of a Bit-parallel Algorithm for Computing the Longest Common Subsequence". In: *Information and Media Technologies* 10.1 (2015), pp. 8–16.

[71] Yasuaki Mitani, Fumihiko Ino, and Kenichi Hagihara. "Parallelizing exact and approximate string matching via inclusive scan on a GPU". In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (2016), pp. 1989–2002.

[72] David Castells-Rufas et al. "OpenCL-based FPGA Accelerator for Semi-Global Approximate String Matching Using Diagonal Bit-Vectors". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 174–178.

[73] Liangwei Cai et al. "A Design of FPGA Acceleration System for Myers bit-vector based on OpenCL". In: *2019 International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*. IEEE. 2019, pp. 305–312.

[74] Jörn Hoffmann, Dirk Zeckzer, and Martin Bogdan. "Using FPGAs to accelerate Myers bit-vector algorithm". In: *XIV Mediterranean Conference on Medical and Biological Engineering and Computing 2016*. Springer. 2016, pp. 535–541.

[75] Abbas Haghi et al. "An FPGA Accelerator of the Wavefront Algorithm for Genomics Pairwise Alignment". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 151–159.

[76] Adnan Ozsoy, Arun Chauhan, and Martin Swany. "Achieving teracups on longest common subsequence problem using GPGPUs". In: *2013 International Conference on Parallel and Distributed Systems*. IEEE. 2013, pp. 69–77.

[77] Adnan Ozsoy, Arun Chauhan, and Martin Swany. "Towards Tera-Scale Performance for Longest Common Subsequence Using Graphics Processor". In: *IEEE Supercomputing (SC)* (2013).

[78] Lucas SN Nunes et al. "A memory-access-efficient implementation of the approximate string matching algorithm on GPU". In: *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE. 2016, pp. 483–489.

[79] Muhammad Umair Sadiq et al. "NvPD: novel parallel edit distance algorithm, correctness, and performance evaluation". In: *Cluster Computing* 23.2 (2020), pp. 879–894.

[80] Jeff Daily. "Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments". In: *BMC bioinformatics* 17.1 (2016), pp. 1–11.

QUIM AGUADO-PUIG received the BSc degree in computer science in 2019 from the Universitat Autònoma de Barcelona (UAB). He is an MSc student at the Universitat Politècnica de Catalunya (UPC) in the innovation and research in informatics programme. He works as a research engineer in the project Designing RISC-V-based Accelerators for next-generation Computers (DRAC) in collaboration with the Barcelona Supercomputing Center (BSC). His research interests include high-performance computing, massively parallel architectures, and GPU programming; with applications to genomics, computational biology, and sequence alignment.

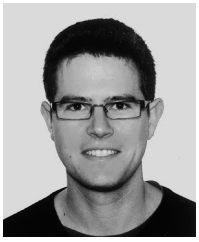
SANTIAGO MARCO-SOLA received the MSc and PhD degrees on computer science from the Universitat Politècnica de Catalunya (UPC) in 2017. During his PhD, he worked in the Algorithm Development and Bioinformatics group at the Spanish National Centre for Genome Analysis (CNAG). Currently, he is a postdoctoral researcher at the Barcelona Supercomputing Center (BSC) and lecturer at the Universitat Autònoma de Barcelona (UAB). He participates in the project Designing RISC-V-based Accelerators for next-generation Computers (DRAC). His research interests include high-performance computing, heterogeneous architectures, genomic data analysis, and machine learning algorithms in the context of bioinformatics and computational biology.

JUAN CARLOS MOURE is an associate professor in the Computer Architecture and Operating Systems Department at the Universitat Autònoma of Barcelona (UAB), where he teaches Computer Architecture, Performance Engineering and Parallel Programming. His current research interests include massive parallel architectures, programming, and algorithms, mainly focused on Computer Vision, Signal Processing and Bioinformatics applications. He is the author of more than 50 papers, and has participated in several European and Spanish projects related to high-performance computing.

DAVID CASTELLS-RUFAS received the B.S., M.S., and Ph.D. degrees in Computer Science from Universitat Autònoma de Barcelona (UAB), Spain in 1994, 2009, and 2016 respectively. From 2003 to 2018 he was working as research assistant and Post-doc researcher with CEPHIS/CAIAC research center of the UAB, where he also was adjunct professor. From 2020, he is with the Computer Architecture and Operating Systems Department of the same university. His research interests include high performance computing, reconfigurable systems, and embedded systems

LLUC ALVAREZ Lluc Alvarez received the BSc degree from the Universitat de les Illes Balears (UIB), Palma, Spain, in 2006, and the MSc and PhD degrees from the Universitat Politecnica de Catalunya (UPC), Barcelona, Spain, in 2009 and 2015, respectively. Currently he is a researcher with the Barcelona Supercomputing Center (BSC) and a lecturer with the Universitat Politecnica de Catalunya (UPC), and his main research interests include parallel architectures, memory systems, programming models for high-performance computing, and accelerators for bioinformatics applications.

ANTONIO ESPINOSA received the B.Sc. degree in computer science in 1994 and the Ph.D. degree in computer science in 2000. He is a associate professor in the Computer Architecture and Operating Systems Department at the Universitat Autònoma de Barcelona. During the last 10 years, he has participated in several European and national projects related to bioinformatics and high-performance computing, in collaboration with a number of biotechnology companies and research institutions.



MIQUEL MORETO is a Ramón y Cajal fellow at the Universitat Politècnica de Catalunya (UPC) and an associate researcher at the Barcelona Supercomputing Center (BSC). Prior to joining UPC, he was a Fulbright post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the Ph.D. degree from UPC in 2010. His research interests include high performance computer architectures and domain-specific accelerators.