Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona (UPC-FIB)

Master in Innovation and Research in Informatics: High Performance Computing (MIRI-HPC)

Accelerating pairwise sequence alignment on GPUs using the Wavefront Algorithm

Author:
Quim Aguado Puig

Supervisors:
Santiago Marco-Sola
Miquel Moretó Planas

October 2022

Abstract

Motivation:
Advances in genomics and sequencing technologies demand faster and more scalable analysis methods that can process longer sequences with higher accuracy. However, classical pairwise alignment methods, based on dynamic programming (DP), impose impractical computational requirements to align long and noisy sequences like those produced by PacBio, and Nanopore technologies. The recently proposed Wavefront Alignment (WFA) algorithm paves the way for more efficient alignment tools, improving time and memory complexity over previous methods. Notwithstanding the advantages of the WFA algorithm, modern high performance computing (HPC) platforms rely on accelerator-based architectures that exploit parallel computing resources to improve over classical computing CPUs. Hence, a GPU-enabled implementation of the WFA could exploit the hardware resources of modern GPUs and further accelerate sequence alignment in current genome analysis pipelines.
Results:
This thesis presents two GPU-accelerated implementations based on the WFA for fast pairwise DNA sequence alignment: eWFA-GPU and WFA-GPU. Our first proposal, eWFA-GPU, computes the exact edit-distance alignment between two short sequences (up to a few thousand bases), taking full advantage of the massive parallel capabilities of modern GPUs. We propose a succinct representation of the alignment data that successfully reduces the overall amount of memory required, allowing the exploitation of the fast on-chip memory of a GPU. Our results show that eWFA-GPU outperforms by 3-9× the edit-distance WFA implementation running on a 20 core machine. Compared to other state-of-the-art tools computing the edit-distance, eWFA-GPU is up to 265× faster than CPU tools and up to 56 times faster than other GPU-enabled implementations. Our second contribution, the WFA-GPU tool, extends the work of eWFA-GPU to compute the exact gap-affine distance (i.e., a more general alignment problem) between arbitrary long sequences. In this work, we propose a CPU-GPU co-design capable of performing inter and intra-sequence parallel alignment of multiple sequences, combining a succinct WFA-data representation with an efficient GPU implementation. As a result, we demonstrate that our implementation outperforms the original WFA implementation between 1.5-7.7× times when computing the alignment path, and between 2.6-16× when computing only the alignment score. Moreover, compared to other state-of-the-art tools, the WFA-GPU is up to 26.7× faster than other GPU implementations and up to four orders of magnitude faster than other CPU implementations.
Availability:
The source code of both implementations is freely available under the MIT license at `https://github.com/quim0/eWFA-GPU` and `https://github.com/quim0/WFA-GPU`
Contact: quim.aguado@estudiantat.upc.edu

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Sequence alignment remains a fundamental problem in bioinformatics and computational biology. It is a critical component for methods like read mapping [1, 2, 3], denovo genome assembly [4, 5], variant detection [6, 7], multiple sequence alignment [8], and many others [9, 10]. 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 [11]. Moreover, with ever-increasing sequence lengths, third-generation sequencing technologies pose an additional challenge to these algorithms and their ability to scale [12].

1.2 Goals and contributions

This thesis presents several contributions in the field of pairwise alignment. We adapt the WFA algorithm to efficiently exploit GPU architectures, presenting two solutions: A fast, specialized, edit-distance aligner for short sequences, and a more general gap-affine aligner capable of working with very large sequences.

We first present the eWFA-GPU tool. eWFA-GPU is an implementation of the WFA algorithm to exploit the computing power of modern GPUs adapted for the edit-distance. This implementation has specialized kernels for relatively short sequences (i.e. up to a few thousands nucleotides) with bounded errors (i.e. up to 128 errors), with an algorithmic adaptation to reduce the memory consumption of the WFA, being able to fit the algorithm working set on the GPU fast *shared memory*. Thanks to the bounded error of the specialized kernels, this implementation can achieve massive computing throughput.

Another implementation, WFA-GPU, is presented. This implements the WFA for gap-affine distance function on GPUs. It is also a more general implementation, by not having bounded sequences length or alignment error, so it can align noisy alignments of very long sequences. An extended version of the previously mentioned memory reduction method is used. It also includes an efficient CPU-GPU co-design to fully exploit heterogeneous systems, and a dynamic work scheduler that dispatches alignments to a fixed number of CUDA blocks, to achieve lower memory consumption and better data locality.

We characterize both implementations and compare them with other state-of-the-art pairwise alignment tools.

1.3 Thesis organization

This thesis is organized as follows: Chapter 2 gives background about genomics, sequencing, pairwise alignment, and GPUs. Chapter 3 explains in depth eWFA-GPU, the edit-distance solution. It explains in depth the adaptation of the WFA for the edit-distance, the design decisions for the GPU implementation, discusses the obtained results, and reviews the solution performance bottlenecks. Chapter 4 explain the gap-affine aligner, comparing it with widely used state-of-the-art tools on simulated and real genomics datasets. Finally, Chapter 5 gives some conclusions, list the publications produced during this work, and discusses future steps.

2 Background

2.1 Genomes and sequencing

A genome is the complete set of genetic information of an organism, it consists of sequences of DNA, which are a chain of nucleotides. There are four nucleotides: Adenosine, Guanine, Cytosine, and Thymidine, usually referred to as A, G, C, and T.

DNA is packed in chromosomes, which contain all or some genetic information of an organism. For example, each human cell has 23 pairs of chromosomes. DNA chains in chromosomes may contain different genes. A gene is a region linked with a specific function and is considered the basic unit of inheritance, containing basic physical and biological traits. Most genes encode a sequence of amino acids, which can eventually form a protein.

Nowadays, there is no sequencing technology capable of producing a complete genome from end to end. Current technological approaches are based on slicing the DNA molecule into smaller pieces that can be read by modern sequencing machines. Each DNA chunk is called a *read*. Currently, the most widely-spread sequencing technologies are the following:

- **Short reads**: Illumina is the main company producing short-read sequencing machines. Illumina machines deliver high-throughput and high-quality short reads (i.e., circa 100-200bp long). This technology is also called *second generation sequencing*.
- **Long reads**: Also called *third generation technologies*, the main ones being the ones from ONT and PacBio.
  - **Ultra-long reads**: Oxford Nanopore Techonologies (ONT) produces very long reads by passing the DNA molecule through a nano-scale pore and measuring the changes in the electrical field around the pore. This technology produces very long reads (up to millions of nucleotides per read) and has very high throughput, but the generated data is noisy (between 2% and 10% of error introduced by the sequencing machine).
  - **Accurate long reads**: Sequencing machines from the company Pacific Biosciences (PacBio) are able to produce HiFi reads (i.e., High Fidelity). HiFi sequences are usually relatively long (of the order of tens of thousands of nucleotides) and depict a high accuracy. Nevertheless, this technology has a lower throughput than other technologies, as the high accuracy is achieved by sequencing DNA fragments multiple times and computing the consensus of all iterations.

The sequencing process starts with the acquisition of genomic data, which can come from a real organism, a real dataset, or a simulated dataset.

Sequencing machines produce different raw data depending on the technology used, this data must be converted to a digital DNA alphabet for further analysis. This process is called basecalling, and is the first computationally intensive step in the genomics pipeline. Raw data can be in different forms depending on the sequencing technology used, for example, it is electric current for Nanopore, or images for Illumina or PacBio.

When sequences are in digital DNA format, a quality control process decides which nucleotides should be edited or removed to compensate for error during library preparation or the sequencing process. This step is also different for each sequencing technology, as they have different error properties (e.g., Illumina has degraded quality on the end of the reads). As a result, reads with the highest quality that each technology can offer are obtained.

With the final reads available, the next step is read mapping: locating subsequences of a reference genome that are similar to the sequencing read. As individuals do not have exactly the same genome as the reference one (each genome is unique, with different traits and mutations), and sequencing machines may have introduced some errors in the read, we can not expect exact matches. Some edits must be allowed, this can be done using the edit-distance (also known as *Levenshtein* distance) to speed up the computational problem, or using the gap-affine distance to get a biologically more relevant result. Although some research has proposed that using concave distance functions may be more appropriate [13], it is not used in practice due to the increased computational complexity and lack of consense [14].

Read mapping consists of four steps: indexing, seeding, pre-alignment filtering, and sequence alignment. The first three steps are used to reduce the search space of the problem, by removing alignments that are very unlikely to be relevant. This way, much fewer data must be processed by the computationally expensive sequence alignment step.

The final analysis step is called variant calling, which finds and analyses differences between an individual and the reference genome, a good variant calling process must be able to distinguish genetic variation from sequencing errors.

Each sequencing technology has different computational bottlenecks, but in all of them, *read mapping* is one of the slowest steps of the pipeline, having between 2.4 and 341 times less throughput than the sequencing step [15]. In this work, we pursue accelerating the read mapping step.

2.2 Exact pairwise alignment

2.2.1 Historical context

Saul Needleman and Christian Wunsch initially proposed the dynamic-programming (DP) algorithm to find similarities between amino acid sequences of proteins [16]. They use a similarity matrix that determines the matching score, and mismatching penalty of each combination of elements in the sequences alphabet. They also use a gap penalty, which is added to insertions and deletions. The algorithm proposed has $O(n^2m)$ time

complexity, where n and m are the lengths of the sequences being aligned.

Several algorithms based on the Needleman-Wunsch were proposed [17, 18, 19], reducing the time and space complexity of the algorithm to $O(nm)$, and adapting it to the edit-distance (introduced by Levenshtein on 1966 [20]). These algorithms with quadratic complexity are usually called Needleman-Wunsch (NW) too, even though the initial formulation had cubic time complexity.

In 1975, Hirschberg introduces a divide-and-conquer algorithm to compute the Longest Common Subsequence (LCS) derived from the NW algorithm [21]. It maintains the same $O(nm)$ time complexity, but showing linear space complexity (from $O(nm)$ to $O(min(n, m))$). Future contributions have been inspired by this method to achieve linear space in other distance functions or algorithm types [22, 23].

An algorithm to align two sequences using the gap-affine distance function in $O(nm)$ time and space was introduced by Gotoh [24] by improving the initial proposal by Waterman, Smith, and Beyer [25]. This algorithm uses three matrices and is usually referred to as Smith-Waterman-Gotoh (SWG) algorithm.

Another type of alignment algorithm was introduced by Ukkonen [26] and Myers [27], based on diagonal transitions. These methods exploit similarities between sequences to speed up the alignment process, obtaining an $O(ns)$ time complexity, where s is the optimal alignment score. They only formulate these kinds of algorithms for edit-distance and the LCS problem.

In 1988, Myers and Miller use Hirschberg's ideas to propose an algorithm that computes the gap-affine distance with linear memory space [22] while maintaining the quadratic time complexity on the length of the sequences.

Recently, in 2021, Marco-Sola presented the Wavefront Alignment algorithm (WFA) [28]. This proposal generalizes the diagonal transition idea to the gap-affine distance function, obtaining a $O(ns)$ time algorithm in $O(s^2)$ space complexity. Additionally, its formulation makes it very suitable to exploit parallel or SIMD architectures. Furthermore, in 2022, the same authors combine ideas from the WFA and Hirschberg algorithm to decrease the memory complexity to $O(s)$ while retaining the same $O(ns)$ time complexity [23]. This improved algorithm is known as the Bidirectional Wavefront Algorithm (BiWFA).

2.2.2 Distance functions

A distance function is a metric used to model the differences between two sequences. They define which transformations are valid (e.g., mismatch, insertion, deletion, transposition) and which costs are associated with each. Examples of commonly used distance functions are linear and gap functions.

A widely used linear distance function is the edit-distance [20]. Described in detail in Section 2.2.4, this distance function accepts three transformations: mismatch, insertion, and deletion. Each of those has a penalty of 1, while the match has a penalty of 0. A variation of edit-distance is the Damerau–Levenshtein distance [29], which adds the transposition of two adjacent characters as a valid operation.

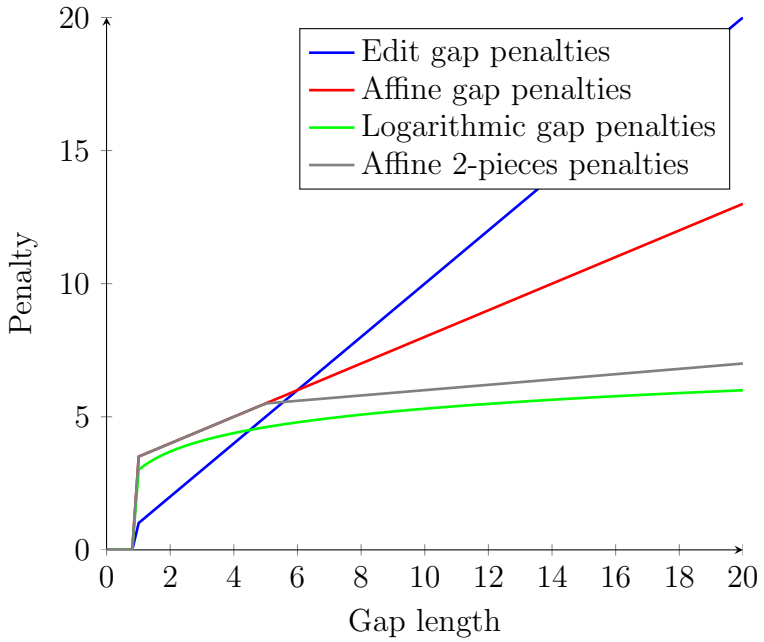


Figure 1: Penalties evolution with gap length on different distance functions.

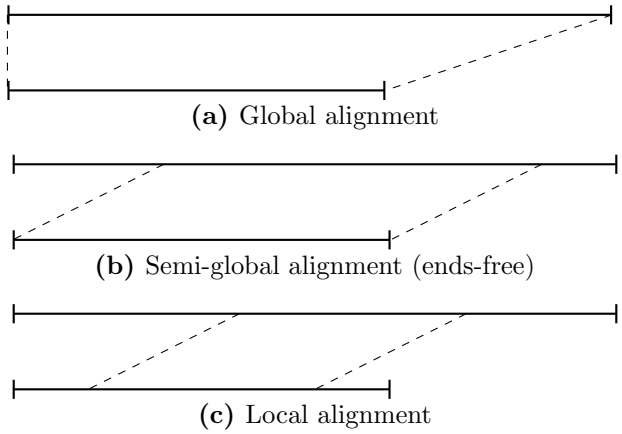
The simplest gap-based distance function is the gap-linear function, which gives a fixed value to each operation. This is similar to the edit-distance, but insertions, deletions, and mismatches have independent costs that are different from 1.

A more biologically relevant distance function is gap-affine (explained in detail in Section 2.2.5). In this case, starting an insertion or deletion have an initially associated penalty (opening a gap), and extending an opened gap has a different penalty (extending a gap). This way, longer contiguous gaps are preferred over smaller scattered gaps.

Another proposal for biological applications is the gap-log (convex) distance. In this case, like gap-affine, there is a gap-open cost when opening an insertion or a deletion, but the extension of an opened gap is now the logarithm of a fixed value. This way, a penalty of a gap is $O+\ell \times log(E)$, where O is the gap-open penalty, E is the gap-extend penalty, and ℓ is the gap length.

General convex distance functions are significantly expensive to compute. For that, 2-pieces gap-affine was proposed [30]. It is presented as a lightweight compromise between gap-affine and gap-log distances. It proposes having two different opening and extension penalties. Using the right combination (i.e., one piece with a smaller opening and bigger extension, and the other piece with a bigger opening and smaller extension), 2-pieces gap-affine can produce a good approximation to convex distance functions.

In Figure 1, the penalty increase produced by a gap depending on its length is graphically shown. As explained before, 2-pieces gap-affine and convex distance functions have similar behaviors.



(a) Global alignment

(b) Semi-global alignment (ends-free)

(c) Local alignment

Figure 2: Global, semi-global and local alignment. The dotted lines indicate the sections in the sequences taken into account to compute penalties.

2.2.3 Alignment modes

Different aligning modes can be used, an alignment mode determines which parts of the sequences are taken into account when computing the score. The main ones are the following:

- *Global alignment*: The goal is to align the sequences in their entire length; that is, end-to-end.
- *Semi-global alignment*: Also called *glocal alignment* or *end-free alignment*. The goal is to allow for gaps at the beginning and at the end of one sequence, without introducing any penalty. It allows finding overlaps between sequences.
- *Local alignment*: The goal is to align two local sections of sequences minimizing the penalty. The final alignment may not align the ends of both sequences, but a small fragment with high similarity.

Figure 2 illustrates these concepts.

2.2.4 Edit distance

The edit-distance (also known as *Levenshtein* distance) is a metric used to measure similarity between two sequences using insertion, deletion, and mismatch transformations. Each transformation has a penalty of 1, while matching characters have a penalty of 0. Usually, the minimum edit-distance is solved by using dynamic-programming (DP) methods [16][31][32].

Let P and Q be two sequences of length n and m, the edit-distance e between the can be computed using Eq. 1 to fill the matrix M that contains $n \times m$ cells. The minimum edit-distance is located at the cell $M_{n,m}$.

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

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

It is possible to trace back the operations that led to the edit-distance e. Starting at $M_{n,m}$, we *move* to the cell that generated that penalty, repeating the operation until we arrive at $M_{0,0}$

2.2.5 Gap-affine distance

Gap-affine is a more biologically relevant distance function because it can benefit long gaps over multiple small gaps. It transforms one sequence into the other using the same operations as the edit-distance (i.e. mismatch, insertions, and deletions), but the cost of opening a gap may not be the same as the cost of extending it. The penalties are defined as: mismatch (x), gap-open (o), and gap-extend (e). This way, the cost of a gap (contiguous chain of insertions or deletions) of length ℓ has a penalty score of $o + e \times \ell$. Compared with the edit-distance, opening a gap is more costly than extending an existing gap, this behavior can be graphically seen in Figure 1.

2.3 The Smith-Waterman-Gotoh Algorithm

The Smith-Waterman-Gotoh (SWG) is the classical algorithm to compute the gap-affine distance between two sequences using dynamic programming. The algorithm has three dynamic-programming matrices to compute the scores produced by different operations: M for the mismatches, I for the gaps generated by insertions, and D for the gaps generated by deletions. These matrices are filled according to the recurrence shown in Eq. 2.

$$
\begin{aligned}
I_{i,j} &= \min \begin{cases} M_{i,j-1} + o + e & \text{Open insertion} \\ I_{i,j-1} + e & \text{Extend insertion} \end{cases} \\
D_{i,j} &= \min \begin{cases} M_{i-1,j} + o + e & \text{Open deletion} \\ D_{i-1,j} + e & \text{Extend deletion} \end{cases} \\
M_{i,j} &= \min \begin{cases} I_{i,j} & \text{Deletion} \\ D_{i,j} & \text{Insertion} \\ M_{i-1,j-1} + \delta(P_i, Q_j) & \text{Match/Mismatch} \end{cases} \\
\delta(P_i, Q_j) &= \begin{cases} 0 & \text{if } P_i = Q_j \\ \text{x} & \text{if } P_i \neq Q_j \end{cases}
\end{aligned}
\tag{2}
$$

The SWG algorithm has an interesting property that is key for the Wavefront Alignment algorithm (WFA) (Section 2.4). If the match score is set to 0, matching characters along diagonals do not increase the score, as illustrated in Fig. 3. This is because diagonal values are monotonically increasing, as observed by [26, 27, 33]. This way, if a match is found, the minimum possible score the cell $M_{i,j}$ can have is $M_{i-1,j-1}$. If $P_i = Q_j$, then $\delta(P_i, Q_j)$ always return 0, therefore, $M_{i,j}$ will always be $M_{i-1,j-1}$ in that case (Eq. 2).

2.4 The Wavefront Alignment Algorithm

The Wavefront Alignment algorithm (WFA) is a diagonal transition algorithm initially formulated for global gap-affine distance [28]. The same algorithm can be applied to other distance functions, and to the semi-global alignment mode. It can use any positive integer value for the mismatch, gap-open, and gap-extend penalties, while the match value is fixed to 0.

As diagonals in the DP matrix increase their score monotonically, the algorithm only needs to store the position of the f.r. (furthest reaching) points in each diagonal with a certain score. The algorithm defines the group containing the diagonal indexes of the f.r. points with a certain score s as a Wavefront ($\widetilde{W}_s$). The goal is to obtain the minimum distance s such as any point in $\widetilde{W}_s$ reaches (n, m).

Each wavefront is represented as a vector of offsets centered at the diagonal 0 ($k = 0$). Also note that, depending on the combinations of penalties, some scores may never exist in the DP matrix, therefore, the wavefronts that represent the f.r. points at that distance do not exist either.

Let $\widetilde{M}_s$, $\widetilde{X}_s$, $\widetilde{I}_s$, and $\widetilde{D}_s$ be the wavefront components that describe partial alignments of score s that end with a match, mismatch, insertion, and deletion, respectively. In general, we denote $\widetilde{W} = \{\widetilde{M}, \widetilde{X}, \widetilde{I}, \widetilde{D}\}$ as the set of wavefront components. We define $\widetilde{W}_{s,k}$ as the farthest reaching point of score s on diagonal k. That is, $\widetilde{W}_{s,k}$ denotes the coordinate $(h, v) = (\widetilde{W}_{s,k}, \widetilde{W}_{s,k} - k)$ in the DP matrix that is farthest in the diagonal k

with score s. Thus, a wavefront $\widetilde{W}_{s,k}$ is a vector containing the farthest reaching points with score s on each diagonal k.

The algorithm iterates over all distances, starting from zero up to the optimal solution $(0 \dots s)$ using the recurrences of Eq. 3 (where $LCP(v,w)$ is the longest common prefix between two strings v and w).

As shown in Algorithm 1, the original authors and this work implements Eq. 3 by using two main operators:

- ***Next* operator**: Generates the wavefront $\widetilde{X}_s$.
- ***Extend* operator**: Generates the wavefront $\widetilde{M}_s$ by extending the diagonals until a mismatch is found (i.e. computing the LCP of the diagonal starting at the point indicated by $\widetilde{X}_{s,k}$). The implementation of this step is shown in Algorithm 2

$$
\begin{aligned}
\widetilde{I}_{s,k} &= \max\{\widetilde{M}_{s-o-e,k-1} + 1, \widetilde{I}_{s-e,k-1} + 1\} \\
\widetilde{D}_{s,k} &= \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\} \\
\widetilde{X}_{s,k} &= \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\} \\
\widetilde{M}_{s,k} &= \widetilde{X}_{s,k} + LCP(q_{\widetilde{X}_{s,k}-k \dots n-1}, t_{\widetilde{X}_{s,k} \dots m-1})
\end{aligned}
\tag{3}
$$

Algorithm 1: WFA algorithm.

Input: q, t strings, $\{x, o, e\}$ gap-affine penalties

Function WFA_ALIGN($q, t, \{x, o, e\}$) **begin**

```
// Initial conditions
```
$\widetilde{M}_{0,0} \leftarrow LCP(q_{0 \dots n-1}, t_{0 \dots m-1})$

$s \leftarrow 0$

while $\widetilde{M}_{s,m-n} \neq m$ **do**

$s \leftarrow s + 1$

for k ***in*** *diagonals*($\widetilde{W}_s$) **do**

```
// Compute wavefronts with score s (next operator)
```
$\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k-1} + 1, \widetilde{I}_{s-e,k-1} + 1\}$

$\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\}$

$\widetilde{X}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$

```
// Compute LCP() using the extend operator
```
$\widetilde{M}_{s,k} \leftarrow$ WFA_EXTEND($q, t, \widetilde{X}_{s,k}$)

		G	A	A	T	A
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	0	1	2	3
T	3	2	1	1	1	2
T	4					
A	5					

Figure 3: Alignment score is not increased until a mismatch is found.

Algorithm 2: WFA *extend* operator

Function `WFA_EXTEND`$(q, t, \widetilde{W}_{s,k})$**:**

```
// Compute (v,h) position
```
$\mathrm{v} \leftarrow \widetilde{W}_{s,k} - k$

$\mathrm{h} \leftarrow \widetilde{W}_{s,k}$

```
// Compute diagonal matches
```
while $q_v = t_h$ **do**

$v \leftarrow v + 1$

$h \leftarrow h + 1$

$\widetilde{W}_{s,k} \leftarrow \widetilde{W}_{s,k} + 1$

2.5 Graphical Processing Units

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 scratch-pad 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 sepa-

rate 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.

3 Edit-distance WFA GPU alignment

This section 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 specialized 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 characterize 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.

3.1 WFA for the edit-distance

Even though it is originally formulated for the gap-affine distance function, the Wavefront Alignment algorithm can be simplified to compute the edit-distance. Only one wavefront per distance e is needed ($\widetilde{W}_e$) as the score of opening and extending a gap is the same. The recurrences for the edit-distance WFA are shown on Eq. 4. As the original WFA, there is an adapted *next* operator (Algorithm 4), and *extend/LCP* operator (Algorithm 3). Algorithm 5 presents how the two operators work to compute the alignment.

Algorithm 3: Edit-distance 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 4: Edit-distance 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**
  - $$\widetilde{W}_{e+1,k} \leftarrow \max \begin{cases} \widetilde{W}_{e,k-1} + 1 \\ \widetilde{W}_{e,k} + 1 \\ \widetilde{W}_{e,k+1} \end{cases}$$

Algorithm 5: WFA edit-distance alignment

Function *WFA_align*$(q,t,\widetilde{W})$**:**

```
// Initial conditions
```
$\widetilde{W}_{0,0} \leftarrow 0$

extend $(q,t,\widetilde{W}_0)$

```
// Compute wavefronts
```
$e \leftarrow 0$

while $\widetilde{W}_{e,m-n} \neq m$ **do**

  computeNext $(\widetilde{W}_e,\widetilde{W}_{e+1})$

  $e \leftarrow e+1$

  extend $(q,t,\widetilde{W}_e)$

$$\delta = \max \left\{ \begin{matrix} \widetilde{W}_{e-1,k+1} & \text{(Deletion)} \\ \widetilde{W}_{e-1,k}+1 & \text{(Mismatch)} \\ \widetilde{W}_{e-1,k-1}+1 & \text{(Insertion)} \end{matrix} \right\}$$

$$\widetilde{W}_{e,k} = \delta + LCP(q_{\delta-k\ldots n-1}, t_{\delta\ldots m-1}) \tag{4}$$

3.2 GPU implementation

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 [34, 35, 36, 37, 38, 39]. 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.

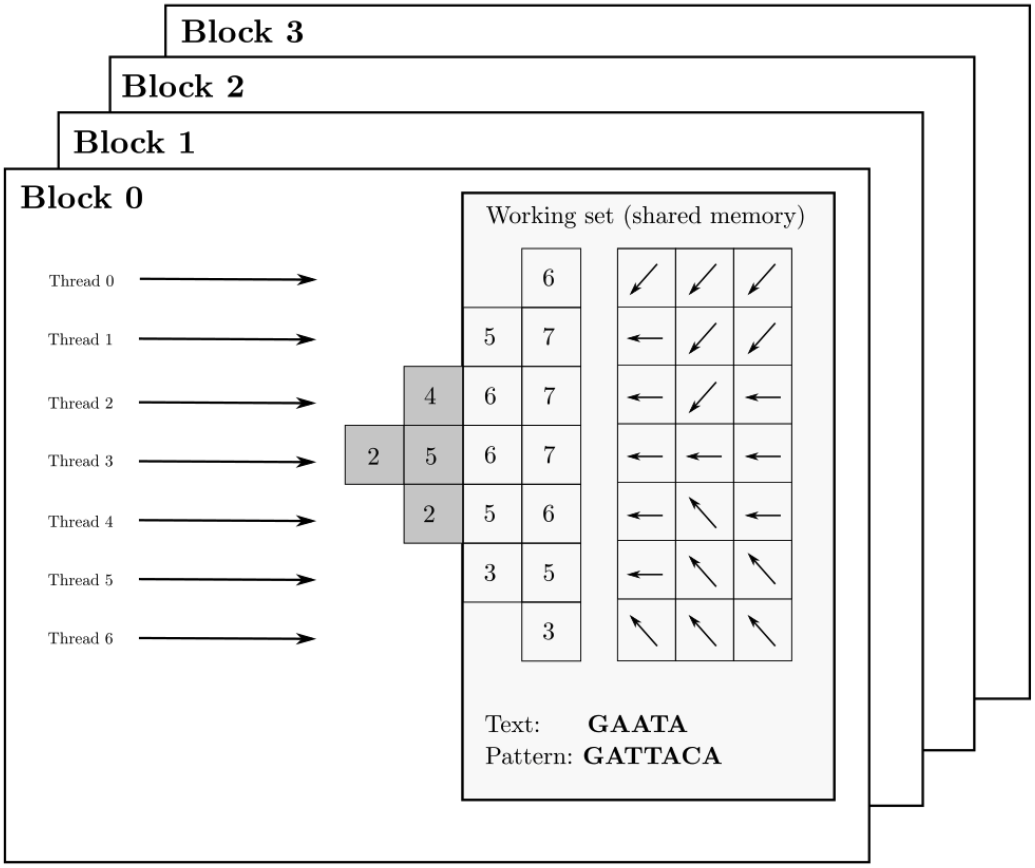


Figure 4: Mapping of CUDA resources into WFA work.

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. 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 4). 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 5 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. Concern-

ing 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 3.2.1).

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 3.2.3).

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 3.2.2).

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 3.2.4).

3.2.1 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. 4. 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 5 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 $\widetilde{B}_{e,k} =$ " $\leftarrow\nwarrow\nwarrow$ ".

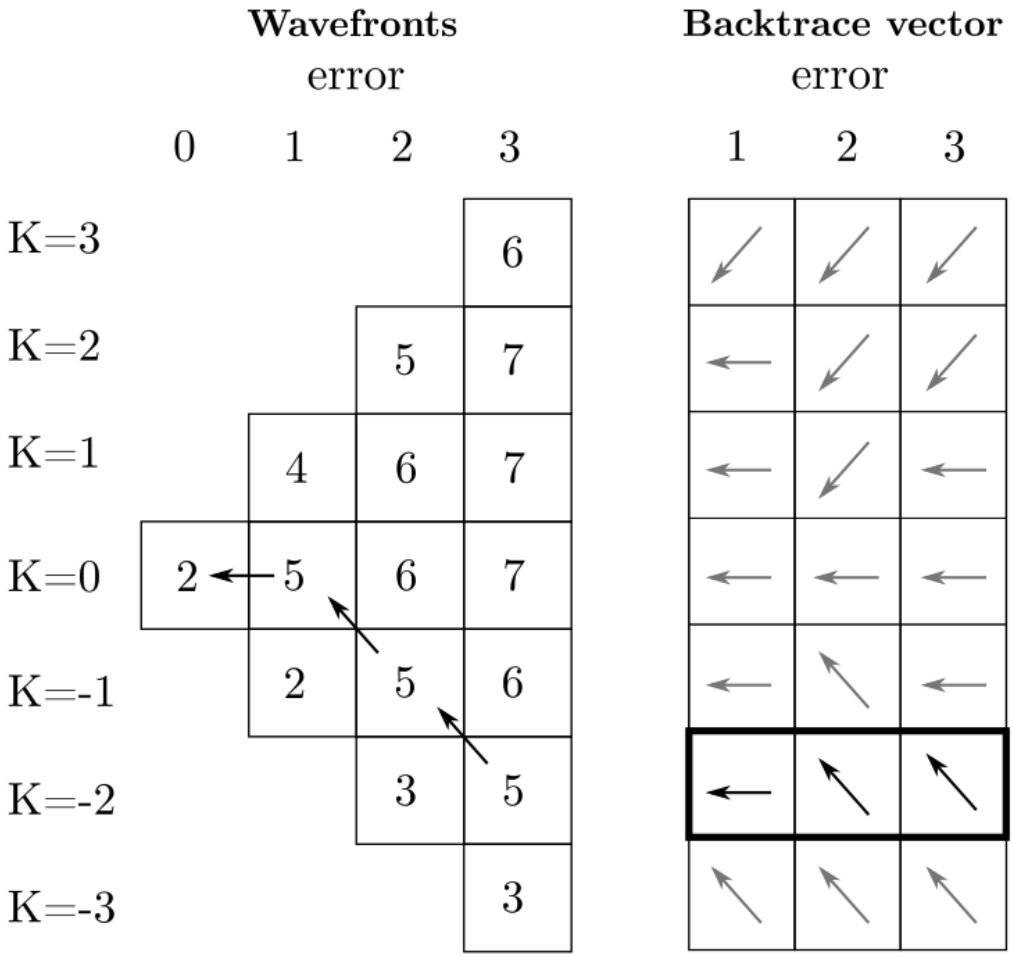


Figure 5: Wavefront data layout for aligning the sequences $T =$ "$GAATA$" and $P =$ "$GATTACA$".

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, the WFA's *extend()* operator is used to compute stretches of matches between successive backtrace steps. This strategy is shown in Algorithm 6. 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.

Algorithm 6: Algorithm to retrieve the alignment from the backtrace vector

```
Function retrieveAlignment(P,T,W̃_e,W̃_{e+1}):
    offset ← 0
    k ← 0
    A ← ∅
    for i ← 0 to e do
        m ← extend(P,T,k,offset)
        A ← A+'M', (m)⋯,'M'
        op ← B̃_k[i]
        switch op do
            case ↖ do // Deletion
                k ← k − 1
                A ← A+'D'
            case ← do // Mismatch
                offset ← offset + 1
                A ← A+'X'
            case ↙ do // Insertion
                offset ← offset + 1
                k ← k + 1
                A ← A+'I'
    m ← extend(P,T,k,offset)
    A ← A+'M', (m)⋯,'M'
```

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, each offset in Eq. 4 is piggybacked 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 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.

3.2.2 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 3 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, a packed sequence encoding that allows performing bit-parallel sequence comparisons is used; 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 3.2.4). 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 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).

3.2.3 Kernel specialisation

Even though the introduction of the backtrace vectors (Section 3.2.1) reduces the memory requirements, shared memory usage is a major performance limitation when scaling to larger alignment errors (see Section 3.4). 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 shared 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, three different kernels are implemented, 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 3.4. 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 3.4).

3.2.4 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

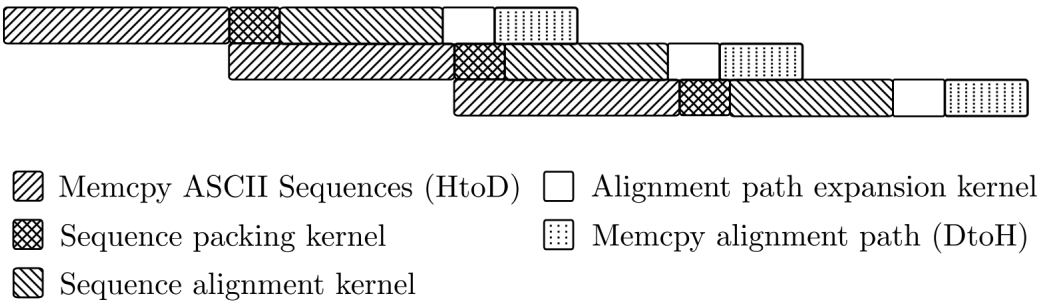


Figure 6: Compute kernels of multiple batches are overlapped with data transfers (DthH and HtoD).

is illustrated in Figure 6.

3.3 Experimental evaluation

3.3.1 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 [40]; eWFA, an optimised CPU version of the WFA [28] adapted to the edit-distance; BPM, a highly optimised version of the BitParallel Myers algorithm [41]; and the $O(ND)$ algorithm [27] used at the core of the Linux diff-tool. All CPU executions were performed using 80 threads, as the best performance (execution time) is obtained with this number of 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 [42] framework, the WmCudaTile algorithm from xbitpar [43], and the highly optimised GASAL2 [44]. 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.

3.3.2 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 3.4).

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 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.

3.3.3 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 2.

The results of the execution on other GPU devices are shown in Table 3. On the GeForce RTX 2080 Ti, our implementation is bounded by the bandwidth between

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 are 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**

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 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 2, 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.

3.4 Performance characterization

Our solution relies on exploiting the fast on-chip memory of the GPU to improve the execution time. As explained in Section 3.2, 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 3.2.3). In this section we show that selecting the proper kernel, adjusted the maximum expected alignment error, is crucial to obtain the best performance.

3.4.1 Overall system profiling

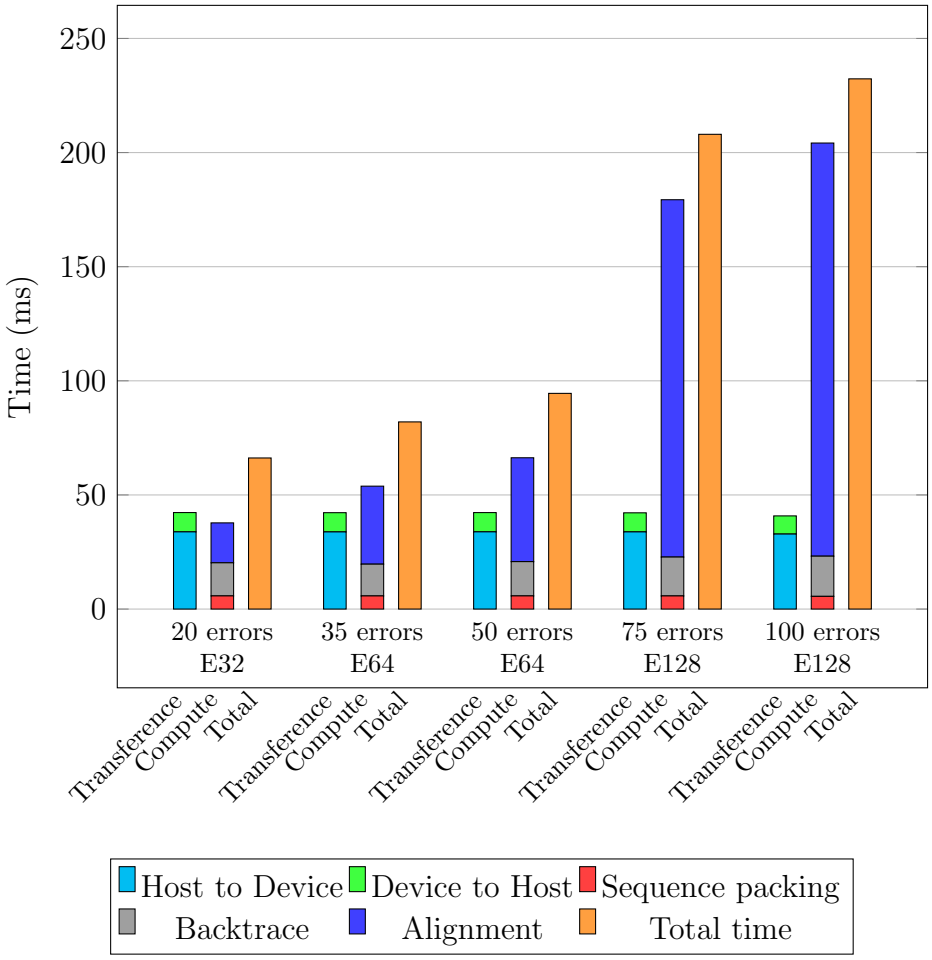


Figure 7: Application execution time broken down into transference time (i.e., host to device and device to host), kernel execution time, and total execution time. Each execution was performed using a dataset contain 1 million sequences of 1000 nucleotides with different error rates (i.e., 20, 35, 50, 75, and 100 nominal errors).

Having three different specialised kernels, the performance of the executions change depending on the alignment error between the sequences. Fig. 7 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 perfor-

mance 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 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).

3.4.2 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 4 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.

Regarding computation on the GPU, in Table 4 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 4 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 4). 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 4 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.

3.4.3 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 5 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.

3.5 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 [45], 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 [41], the O(ND) [27], and the Wu-Manber (WM) [46] 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 [40], BGSA [47], and SeqAn [48]. 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 [48] 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 [49, 50, 51, 52, 53]. Meanwhile, some research efforts have been focused on producing efficient CUDA implementation of the classical Needleman-Wunsch [54] algorithm; other proposals have focused on novel organisations of the DP-matrix to exploit efficiently the GPU resources [55]. In particular, in [56] and [57], 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 [43], 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 [58], the authors exploit the Crochemore algorithm based on Suffix automaton for bit-parallel alignment. Like [59], other proposals revisit the Shift-Or and Wu-Manber algorithms, implementing them as inclusive-scan operations to allow multiple parallel computations. Similarly, in [35] 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)[60, 61, 62, 63]. 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. 5 for an alignment of two sequences of length n and m, taking s seconds. This way, Table 6 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.

$$GCUPS = \frac{nm}{s} \times 10^{-9} \tag{5}$$

Table 6: 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
	[59]	2016	GeForce GTX TITAN X	2800
	[35]	2014	Geforce GTX 680	2300
	[42]	2014	Tesla K40c	1000
	[64]	2013	Geforce GTX 480	470
	[65]	2013	Geforce GTX 480	470
	[43]	2016	Tesla V100	420
	[44]	2019	Tesla V100	206
	[56]	2015	GeForce GTX 980	65
	[66]	2016	GeForce GTX 960	50
	[58]	2015	GeForce GTX 580	28
	[67]	2020	GeForce GTX TITAN Black	14
	[38]	2018	Tesla K40c	14
CPU	[40]	2017	Intel i7-4710HQ	388
	[68]	2016	Intel Xeon E5-2670	136
	[48]	2008	3.2 GHz Intel Xeon	2
FPGA	[61]	2019	Kintex KCU1500	161
Others	[47]	2018	Intel Xeon Phi-7210	1895

Table 2: 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 3: 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

Table 4: 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

Table 5: 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

4 Gap-affine WFA GPU algorithm

4.1 GPU parallel Wavefront Alignment

The WFA algorithm depicts simple computational patterns when computing Eq. 3. To compute any wavefront $\widetilde{W}_s$, we only require wavefronts $\widetilde{W}_{s-o-e}$, $\widetilde{W}_{s-e}$, and $\widetilde{W}_{s-x}$. Moreover, note that each diagonal offset $\widetilde{W}_{s,k}$ can be computed independently. Therefore, the WFA allows computing each wavefront diagonal in parallel, which makes the algorithm perfectly suited to the GPU execution model.

Algorithm 7: WFA-GPU parallel algorithm.

Input: q, t strings, $\{x, o, e\}$ gap-affine penalties

Function WFAGPU_ALIGN_KERNEL($q, t, \{x, o, e\}$) **begin**

```
// Initial conditions
```
$\widetilde{M}_{0,0} \leftarrow LCP(q_{0\ldots n-1}, t_{0\ldots m-1})$

$s \leftarrow 0$

while $\widetilde{M}_{s,m-n} \neq m$ **do**

$s \leftarrow s + 1$

parallel foreach thread k ***in*** *diagonals($\widetilde{W}_s$)* **do**

```
// Compute wavefronts with score s
```
$\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k-1} + 1, \widetilde{I}_{s-e,k-1} + 1\}$

$\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\}$

$\widetilde{X}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$

```
// Compute LCP()
```
$\widetilde{M}_{s,k} \leftarrow \widetilde{X}_{s,k} + LCP(q_{\widetilde{X}_{s,k}-k\ldots n-1}, t_{\widetilde{X}_{s,k}\ldots m-1})$

Synchronize_threads()

Algorithm 7 presents the high-level pseudocode of the WFA-GPU. Basically, our solution offloads the computation of multiple WFA alignments to the GPU. For each alignment, multiple threads in the same block cooperate to compute consecutive wavefronts until the optimal alignment is found. In particular, for every score s and diagonal k, each GPU thread in the block computes $\widetilde{W}_{s,k}$ independently. After every diagonal of wavefronts $\widetilde{W}_s$ is computed, GPU threads synchronize and proceed to compute the following wavefronts $\widetilde{W}_{s+1}$. This way, the WFA-GPU implements a combined inter-sequence and intra-sequence parallelization strategy.

Fig. 8 illustrates in detail the parallel computation of a given wavefront s using multiple GPU threads (i.e., intra-sequence parallelization). At the same time, other alignments are computed by different thread blocks on the GPU (i.e., inter-sequence parallelization). Each GPU thread undertakes the computation of a diagonal offset independently applying Eq. 3 (using the previously computed wavefronts and the *LCP()* function). Notably, as the algorithm progresses, wavefronts become increasingly larger and the potential parallelism of the problem grows. For large and noisy sequences, the problem becomes embarrassingly parallel and perfectly suited for massively parallel devices like modern GPUs.

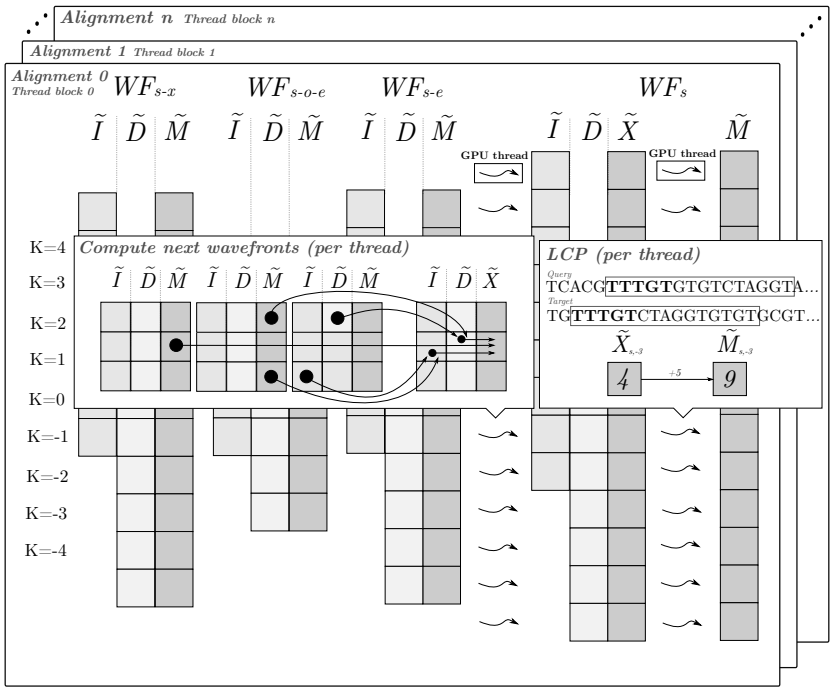


Figure 8: Detail of the parallel computation of a given wavefront s using multiple GPU threads. At the same time, other alignments are computed by different thread blocks on the GPU. This figure shows how each wavefront diagonal can be computed independently using a single GPU thread.

Nevertheless, porting the WFA algorithm to GPU architectures presents some performance challenges of its own. First, as the alignment error increases, the memory requirements grow quadratically, limiting the scalability of the method when aligning tens of thousands of sequences in parallel. In Sections 4.2 and 4.3, we propose solutions to perform a better memory management in the GPU and reduce the overall memory usage of the algorithm, respectively. Second, when aligning large and noisy sequences, the amount of computations to compute successive wavefronts becomes a limiting factor. In Section 4.4, we present a strategy to accelerate the *LCP()* computation. Finally, in Section 4.5, we present an overview of the CPU-GPU co-design implemented in WFA-GPU. Our co-design allows overlapping computations with data transfers (i.e., from CPU host to GPU device and vice versa) and performing computations in the CPU meanwhile the GPU device is busy.

4.2 Alignment scheduling and GPU memory management

A simple and naive implementation would spawn a thread block per each WFA alignment offloaded to the GPU. However, each WFA alignment kernel requires GPU memory to store all the intermediate wavefronts. It is not feasible to reserve GPU memory for every alignment in advance when processing tens of thousands of sequence alignments. However, it is possible to request an upper bound of the total WFA memory required for a number of alignments that can be processed in parallel in the GPU at

the same time.

Thus, our implementation creates a pool of outstanding WFA alignments and allocates memory for as many alignment blocks as can be processed simultaneously on the GPU. Then, an alignment scheduler assigns WFA alignments to thread blocks. Whenever a thread block finishes an alignment, it requests another from the alignment pool until all alignments offloaded to the GPU have been completed.

Nevertheless, having to allocate GPU memory beforehand forces to estimate the maximum memory required by each WFA alignment in advance. For that, our method establishes a configurable upper bound on the required memory based on a conservative estimation of the maximum error rate (i.e., 10% of sequence length by default on our tool). Nonetheless, some alignments may override initial estimations and require more memory. For those cases, the WFA-GPU implements a rescue mechanism that returns the alignment to the CPU to be computed using the original WFA algorithm. In practice, when aligning long and noisy sequences (like those produced by PacBio or Nanopore Technologies) the amount of rescued alignments is below 0.2%. Furthermore, the computation of the rescued alignments can be performed in the host CPU meanwhile the GPU is computing other alignments, as described in Section 4.5.

Although modern GPUs are equipped with large DRAM memories, accesses to global memory are relatively slow and can potentially reduce the performance of GPU applications. To take advantage of fast on-chip memories and minimize the latency of global memory accesses in the GPU, our implementation allocates the central diagonals of wavefronts in the shared memory. This way, the WFA-GPU benefits from fast on-chip memory accesses to the elements of the central diagonals.

4.3 Piggybacked backtrace operations

For an alignment with optimal score s, the WFA algorithm requires storing all the intermediate wavefronts up to $\widetilde{W}_s$ to be able to retrieve the alignment path (a.k.a. CIGAR) during the final backtrace step. However, alignments with a large nominal number of errors require non-negligible amount of memory. That is, an upper-bound of $3\sum_{i=0}^{s} 1+2i = 3(s+1)^2$ wavefronts offsets, consuming up to $12(s+1)^2$ Bytes per alignment. These memory requirements become impractical when aligning multiple noisy sequences in parallel, even for modern GPUs equipped with large amounts of global memory.

To reduce the memory consumption, our method piggybacks the backtrace operations (i.e., **X**, **I**, and **D**) to the wavefronts as they are being computed. Using only two bits, each backtrace operation is encoded in a bitmap stored for every diagonal of the wavefront. Therefore, for a given score s and diagonal k, our method stores a bitmap with the alignment operations required to reach $\widetilde{W}_{s,k}$. It follows that the bitmap associated to $\widetilde{M}_{s,m-n}$ contains the optimal alignment's backtrace.

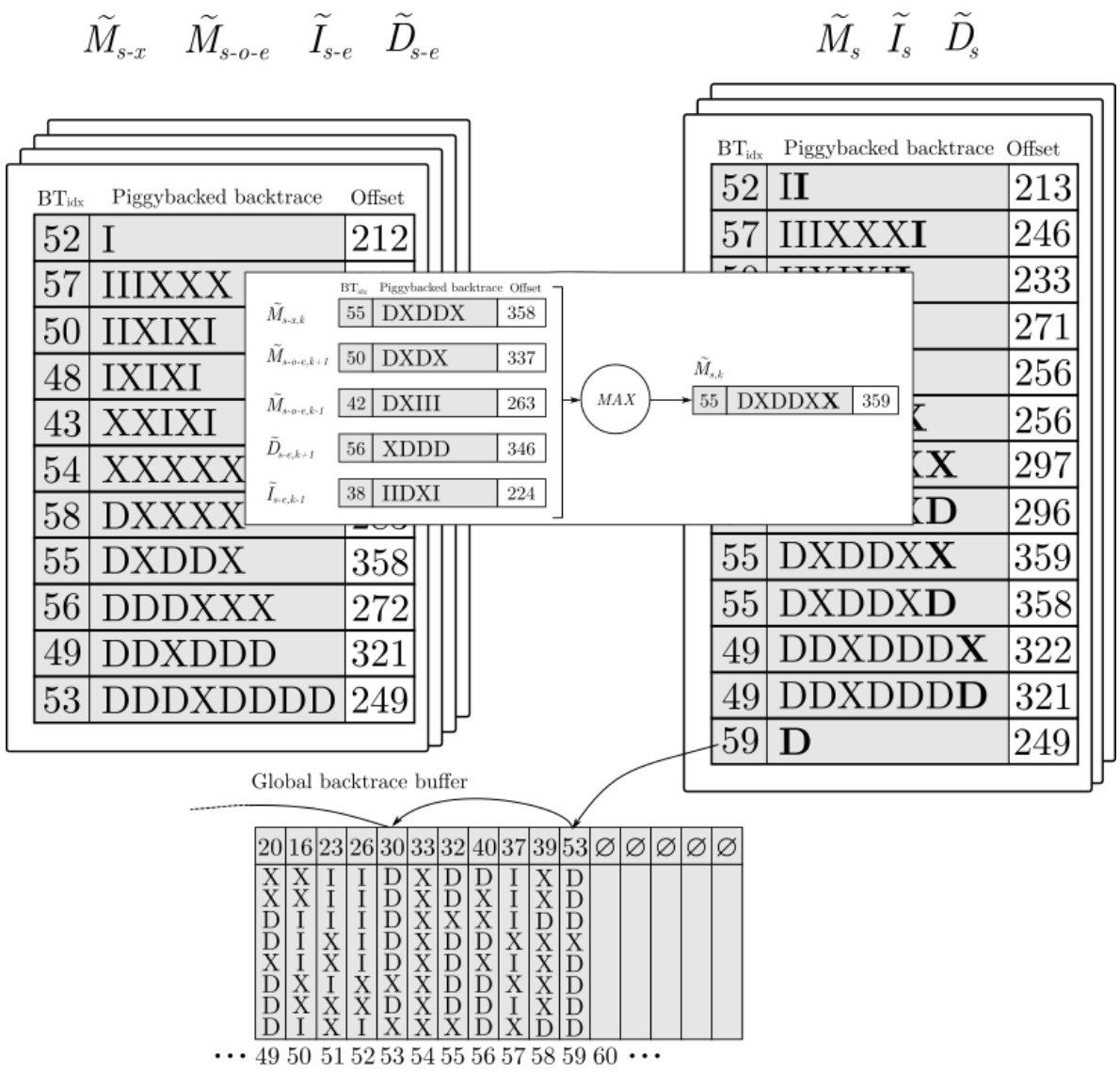


Figure 9: Illustration of the piggybacked backtrace strategy and data-layout organization. From left to right of the figure, we show the source wavefronts ($\widetilde{M}_{s-x}$, $\widetilde{M}_{s-o-e}$, $\widetilde{I}_{s-e}$, and $\widetilde{D}_{s-e}$) and how they are combined to generated wavefronts at score s. Each diagonal has a BT-block and an offset (shaded in grey). At the center, we detail the process of computing Eq. 3 for a single diagonal and the piggyback of the corresponding backtrace operation. At the bottom, the global BT-buffer is depicted where each slot represents a BT-block (displayed vertically for better readability of the figure).

In practice, our implementation uses 32-bit bitmap words to store backtrace operations (i.e., BT-block). Once a BT-block is full and cannot encode more backtrace operations, it is offloaded to a global backtrace buffer (i.e., BT-buffer). Each BT-block stores an index to the previous BT-block in the chain that allows retrieving the complete alignment backtrace associated with any $\widetilde{W}_{s,k}$ offset. A complete alignment backtrace is recovered by traversing the linked BT-blocks starting from the last one.

The computation of each backtrace operation is coupled with the computation per-

formed in Algorithm 7 to generate each diagonal offset. For that, the corresponding backtrace operation (i.e., **X**, **I**, and **D**) is piggybacked to each source wavefront offset ($\widetilde{M}_{s-o-e,k-1}$, $\widetilde{M}_{s-o-e,k+1}$, $\widetilde{I}_{s-e,k-1}$, $\widetilde{D}_{s-e,k+1}$, and $\widetilde{M}_{s-x,k}$) in Eq. 3, using the two less significant bits. Then, as a byproduct of the computation of the maximum offset, the corresponding backtrace operation is found piggybacked. In practice, this strategy turns out to be computationally lightweight. Fig. 9 illustrates the piggybacked backtrace strategy and its data organization.

Note that BT-blocks only contain edition operations (i.e. **X**, **I**, **D**) and not the matches in between. To retrieve the complete alignment CIGAR, the algorithm needs to compute any missing matches between backtrace operations. Nonetheless, this is a remarkably simple operation. Using the same *LCP()* function presented earlier, the algorithm computes matches until a mismatch is found. Then, it adds the following backtrace operation and proceeds again to compute the *LCP()*. This process halts when all the backtrace operations from the chain of BT-blocks have been processed.

Overall, the piggyback strategy effectively reduces the memory consumed by the wavefronts to 4 bits per entry (accounting for the BT-block indices). Compared to storing the raw wavefront offsets as the original WFA does (i.e., 4 Bytes per entry), this strategy represents an 8× reduction.

4.4 Bit-Parallel sequence comparison using packed DNA sequences

WFA's execution time is dominated by the computation of the *LCP()* function. A naive implementation would compare sequences character by character until a non-matching character is found. This approach not only executes a non-negligible amount of instructions per *LCP()* call but also creates divergence across threads computing the same alignment. That is, each GPU thread within a block performs a different number of comparisons depending on the characters being compared. Because GPU threads execute in groups of 32 threads in lock-step mode, divergent execution (i.e., variable-iterations loops) forces idle threads to wait until all threads have finished iterating.

To alleviate this problem, we propose a bit-packed encoding of DNA sequences using 2 bits per base. This encoding turns out to be remarkably simple, as the ASCII representation of each base has two unique bits on position 1 and 2 (i.e., A=1000**00**1, C=1000**01**1, G=1000**11**1, T=1010**10**0). Using this bit-packed representation, our implementation compares blocks of 16 bases at once using 32-bit operations. This strategy reduces execution divergence and, most importantly, the total number of instructions executed. In turn, this translates into faster execution times.

4.5 CPU-GPU co-design system

The WFA-GPU implementation presents a CPU-GPU co-design that allows the simultaneous execution of GPU computations overlapped with data transfers and CPU alignment rescue.

Table 7: Description of the real datasets used in the experimental evaluation. †Datasets obtained from NIST's Genome in a Bottle (GIAB) project can be found at `https://github.com/genome-in-a-bottle/giab_data_indexes`. §Datasets obtained from PrecisionFDA Truth Challenge V2 can be found at `https://precision.fda.gov/challenges/10`.

Dataset	Sample	No. pairs	Seq. Length (bps) min	avg	max
Illumina 150	HG002†	100M	134	148	162
Illumina 250	HG002†	100M	140	248	275
Moleculo	ERR1590085	100M	42	4352	14463
PacBio CSS	HG002†	10M	263	9561	18317
PacBio HiFi	HG002§	10M	201	12847	24640
Nanopore	ERR3279200 ERR3279201	10M	104	6249	21708

To maximize performance, our implementation offloads batches containing multiple alignments to the GPU. For that, input sequences from a batch have to be transferred to the device. To minimize GPU idle times, our implementation makes asynchronous kernel launches, allowing overlapping data transfers with GPU computations. That is, while the GPU is computing the alignments for a given batch, the sequences of the following batch are being copied to the device. As a result, latencies due to transfer times are effectively hidden and overlapped with useful GPU computations.

Furthermore, the asynchronous implementation of WFA-GPU allows employing idle CPU time to rescue alignments returned by the GPU. As explained in Section 4.1, a small percentage of alignments may not be aligned in the GPU due to exceeding memory requirements. For those few cases, the implementation overlaps the CPU WFA execution with GPU computations and data transfers.

4.6 Experimental evaluation

4.6.1 Experimental setup

For the experimental evaluation, we select simulated and real datasets. For the simulated datasets, we generate synthetic pairs of sequences of 150, 1000, and 10,000 bases aligning with an average edit-error of 2%, 5%, and 10% differences. For the evaluation using real datasets, we select publicly available datasets representative of current sequencing technologies (see Table 7). The target sequences are retrieved from mapping the source sequences against GRCh38 using Minimap2 [69] and default parameters.

To compare the performance of the WFA-GPU, we select other sequence alignment libraries and tools representative of the state-of-the-art on both CPU and GPU devices. For the GPU tools comparison, we select the library GASAL2 [44], ADEPT [70] and two NVIDIA libraries (NVBio [42] and CudaAligner [71] from Clara Parabricks

Genomeworks). Additionally, we implement a kernel in WFA-GPU that only computes the gap-affine distance, but not the optimal alignment path, this is referenced as WFA-GPU(distance) on Table 9. Unfortunately, we were unable to include the recently presented GPU aligners Logan [34] and GenASM [72] due to inadequacy to perform basic pairwise alignment and the unavailability of the source code, respectively. As for the CPU tools, we selected the most widely-used and efficient libraries available to date. That is, Seqan [48], Parasail [68], Edlib [40], and KSW2 [73]. Naturally, we also include the original WFA implementation (WFA2-lib [74]) into the comparison. Note that Edlib and CudaAligner can only compute the edit distance alignment (a much simpler problem compared to computing gap-affine alignments). Regardless, we include them in the comparison as an interesting point of reference. In an attempt to evaluate the recall of these tools using gap-affine scores, we re-scored the reported CIGAR using using gap-affine penalties and compared against the optimal score.

ADEPT computes the local alignment of two sequences, as we compute global alignment, it can not be compared with WFA-GPU in terms of accuracy. This is indicated as *not-comparable* (n/c) in Tables 8 and 9.

All the experiments are executed using a 10-core Intel Xeon-W2155 (3.3GHz) processor equipped with 126GB of memory and a NVIDIA GeForce 3080 with 10GB of memory. Moreover, all CPU executions are performed in parallel using the 10 physical cores available in the platform. All GPU execution times include CPU-GPU data transfer, alignment, backtrace, and CIGAR generation time.

4.6.2 Evaluation on simulated data

Table 8 shows time (in seconds) and recall (percentage of sequences for which the optimal alignment was correctly reported) for the alignment executions, using simulated datasets.

Considering the alignment of short sequences (i.e., ∼150bps), NVBio outperforms all other tools at the expense of a major loss in alignment accuracy as the alignment error increases. WFA-GPU is between 1.9 and 3 times faster than the best CPU time obtained. GASAL2 has a very good performance (as it is specialized for short sequences), outperforming our implementations when the error grows. The other GPU aligners, ADEPT and CudaAligner, are one order of magnitude slower than WFA-GPU and GASAL2.

For medium length sequences (i.e., ∼1Kbps), most other GPU implementations either fail due to execution errors, like NVBio and ADEPT, or obtain a recall lower than 10%. Only GASAL2 remains competitive when the error increases, but at a significantly low accuracy (52.4%). Compared to CPU implementations, WFA-GPU executes 3.0-5.8× faster than the original WFA and up to 5500× faster than other libraries.

Experiments aligning long simulated sequences (i.e., ∼10Kbps) turns out to be the most challenging for most GPU tools. All other GPU implementations either fail (i.e., ADEPT and NVBio), give incorrect results (i.e., CudaAligner), or have significantly

Table 8: Time (T, in seconds) and recall (R, as a percentage of exact alignments) for simulated datasets with different error rates (e). All CPU executions use 10 threads. †Implementations can only produce edit-distance alignments.

		100 Million alignments × 150bps						**10 Million alignments × 1Kbps**						**100K alignments × 10Kbps**					
		e=2%		**e=5%**		**e=10%**		**e=2%**		**e=5%**		**e=10%**		**e=2%**		**e=5%**		**e=10%**	
		T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)
GPU	GASAL2	13	98.3	14	95.8	14	93.0	44	84.6	44	64.7	45	52.4	3219	47.0	3224	36.8	3229	34.1
	ADEPT	161	n/c	161	n/c	161	n/c	error	n/a	error	n/a	error	n/a	error	n/a	error	n/a	error	n/a
	NVBio	6	58.2	6	6.1	6	0.0	error	n/a	error	n/a	error	n/a	error	n/a	error	n/a	error	n/a
	CudaAligner†	113	95.3	113	66.1	117	25.7	86	9.1	86	1.6	86	0.3	20	0.9	20	0.0	20	0.0
	WFA-GPU	11	100	18	100	27	100	6	100	23	100	78	100	4	100	21	100	71	100
	WFA-GPU(distance)	6	100	8	100	13	100	3	100	9	100	32	100	2	100	6	100	16	100
CPU	Seqan	2028	100	2028	100	2190	100	9066	100	9288	100	9774	100	9912	100	10098	100	10440	100
	Parasail(strip)	8436	100	8358	100	8238	100	32274	100	32178	100	31842	100	33408	100	33306	100	32760	100
	Parasail(scan)	1197	100	1183	100	1197	100	3882	100	3902	100	3911	100	4869	100	4857	100	4825	100
	Parasail(diag)	1908	100	1914	100	1916	100	7482	100	7488	100	7488	100	7686	100	7692	100	7698	100
	Edlib†	210	95.9	202	69.6	200	30.0	118	67.6	118	9.5	135	0.0	36	1.8	38	0.0	43	0.0
	KSW2	298	100	306	100	317	100	876	100	882	100	882	100	2028	100	2034	100	2040	100
	WFA	20	100	52	100	77	100	34	100	70	100	199	100	15	100	86	100	277	100

Table 9: Time (T, in seconds) and recall (R, as a percentage of exact alignments) for real datasets. All CPU executions use 10 threads. †Implementations that can only produce edit-distance alignments. Executions taking more than 48 hours are marked as timeout.

		Illumina 150 100M alignments		**Illumina 250 100M alignments**		**Moleculo 100M alignments**		**PacBio CSS 10M alignments**		**PacBio HiFi 10M alignments**		**Nanopore 10M alignments**	
		T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)
GPU	GASAL2	12	99.9	28	94.7	error	n/a	error	n/a	error	n/a	error	n/a
	ADEPT	158	n/c	308	n/c	error	n/a	error	n/a	error	n/a	error	n/a
	NVBio	4	49.2	6	27.2	error	n/a	error	n/a	error	n/a	error	n/a
	CudaAligner†	102	97.5	169	91.6	83	82.4	2036	42.6	3105	32.5	1350	0.0
	WFA-GPU	13	100	17	100	16	100	76	100	159	100	4395	100
	WFA-GPU(distance)	7	100	13	100	11	100	48	100	100	100	1080	100
CPU	Seqan	1810	100	5076	100	timeout	n/a	timeout	n/a	timeout	n/a	timeout	n/a
	Parasail(strip)	8424	99.9	21240	99.9	timeout	n/a	timeout	n/a	timeout	n/a	timeout	n/a
	Parasail(scan)	1160	99.9	2814	99.9	89040	99.8	timeout	n/a	timeout	n/a	timeout	n/a
	Parasail(diag)	1878	99.9	4912	99.9	157320	99.8	timeout	n/a	timeout	n/a	timeout	n/a
	Edlib†	155	97.5	246	91.8	817	84.4	2127	47.7	7998	35.7	22620	0.0
	KSW2	251	100	661	100	33120	100	163860	100	timeout	n/a	86580	100
	WFA	20	100	51	100	28	100	194	100	1230	100	17350	100

low recall (i.e. GASAL2 with less than 50% accuracy). WFA-GPU is the only GPU implementation that can scale to long sequences reporting the optimal alignment result. When compared to the original WFA, WFA-GPU executes 3.3-4.1× faster.

Executing WFA-GPU without computing the backtrace (only computing the distance) is between 1.8-4.4× faster than the baseline WFA-GPU. This gives us an speedup of up to 17× compared to the CPU WFA implementation.

4.6.3 Evaluation on real genomic data

Table 9 presents a performance evaluation of the WFA-GPU compared to other state-of-the-art libraries and tools when aligning real datasets (referred in Table 7). Figure 10 visually represents the speedups of WFA-GPU in comparison with the most relevant CPU and GPU tools.

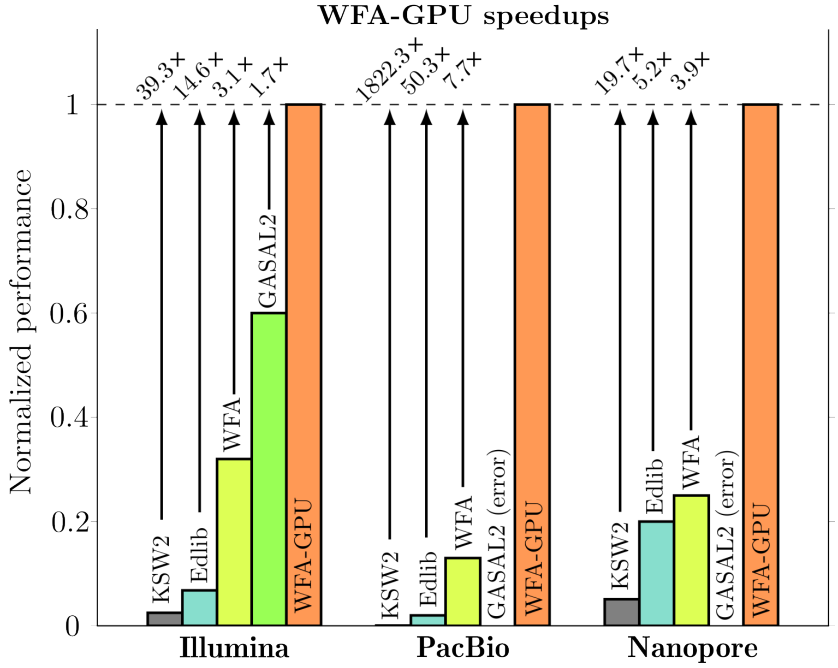


Figure 10: WFA-GPU compared with the most widely-used tools.

For the case of aligning high-quality short sequences, like those produced by Illumina sequencers, NVBio delivers the fastest results at the expense of scoring low in recall (only 49.2% and 27.2% of the alignments are correct). GASAL2 delivers similar performance than WFA-GPU when aligning the Illumina 150 dataset, and is 1.7 times slower when aligning Illumina 250 (that has slightly longer sequences). Compared to the original WFA, which achieves the best execution time among all CPU libraries, WFA-GPU is 1.5-3× faster. Compared to other CPU libraries, WFA-GPU obtains remarkable speedups (up to 1264× with respect to Parasail).

When aligning Moleculo sequences, WFA-GPU obtains an speedup of 1.8× compared to the CPU version of the WFA. On the GPU side, only CudaAligner and WFA-

GPU are able to complete successfully, being WFA-GPU 5.2× faster than CudaAligner.

Using PacBio sequences, WFA-GPU achieves a speedup of 2.5× (on PacBio CSS) and 7.7× (on PacBio HiFi) compared to the CPU version of the WFA. The speedup goes up to 12.2× if we don't compute the backtrace (distance-only version). Our implementation outperforms by up to four orders of magnitude other CPU tools and libraries. The only GPU implementation able to finish is CudaAligner, even though it obtains a significantly low recall (less than 50%), while being between 19.5-26.7× slower than our solution.

Considering the Nanopore dataset, having large sequences with high error rate, WFA-GPU is 4 times faster than the CPU implementation of WFA, and 16× faster when using the distance-only version of WFA-GPU. For this execution, CudaAligner is 3× faster, but it generates incorrect results (0% recall). Compared with other CPU libraries, our implementation is up to 19.7× faster. This speedup is not as high as the ones from previous datasets, because alignments with a high nominal error are the worst case scenario for the WFA algorithm.

5 Conclusions

Future advances in sequencing technologies and genomics present critical challenges to current bioinformatics methods and tools. This situation calls for improved methods and tools that can scale with the increasing production yields and sequence lengths. Modern HPC computing relies on GPUs as successful hardware accelerators for computing intensive applications in many areas of research. This work presents the first GPU-based tool for sequence alignment based on the efficient WFA algorithm. In this work, we presented two GPU implementations of the WFA algorithm for DNA pairwise alignment, discussing the algorithmic adaptations and design decisions adopted.

Our first contribution, eWFA-GPU, is an adaptation of the WFA algorithm for the edit-distance. It is designed to achieve maximum performance on relatively short reads with moderate errors. This is done by using the GPU on-chip fast memories, at the expense of having a limit on the maximum error supported by the application. To be able to store the WFA working set into the limited on-chip memories, we introduce the piggybacked backtrace strategy, a novel optimisation technique that significantly reduces the amount of memory needed for aligning sequences. Moreover, our implementation is fully asynchronous and overlaps compute kernels and memory transfers to accelerate the algorithm execution, hiding memory transfer latencies with computation.

To assess the benefits of our design and implementation, we compared eWFA-GPU with other state-of-the-art tools. The results obtained on the Nvidia V100 GPU show 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. Furthermore, 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.

The second contribution presented in this thesis is the WFA-GPU tool: a gap-affine GPU pairwise aligner based on the WFA algorithm. This version solves a more general alignment problem, supporting the alignment of very long sequences with high error rates. We demonstrate the benefits of WFA-GPU compared to other state-of-the-art CPU and GPU tools and libraries. Our WFA-GPU implementation performs up to 26.7× faster than other GPU tools, and up to four orders of magnitude faster than DP-based CPU libraries. When compared to the WFA CPU implementation (fastest CPU library to date) we obtain speedups between 1.5-7.7×, without any accuracy loss when computing the optimal alignment path. Without computing the alignment path (only computing the distance), WFA-GPU is 2.6-16× faster than the CPU implementation. To the best of our knowledge, WFA-GPU is the only GPU-based pairwise aligner capable of computing exact gap-affine alignments for long-sequencing datasets, like those produced by PacBio HiFi or Oxford Nanopore technologies, leveraging commodity GPU devices.

With the advent of improved sequencing technologies and more sophisticated genome studies, our tools offer an accurate, fast, and scalable sequence alignment solution that

effectively exploits the massive computing capabilities of modern GPU devices. Therefore, we expect eWFA-GPU and WFA-GPU to become a valuable and practical addition to the toolkit of bioinformatics tools that support efficient research in future genome-scale analysis.

5.1 Publications

As a result of the research work produced throughout this thesis, several scientific articles have been elaborated and published.

- **Quim Aguado-Puig**, Santiago Marco-Sola, Juan Carlos Moure, David Castells-Rufas, Lluc Alvarez, Antonio Espinosa, and Miquel Moreto. "Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm." IEEE access 10 (2022): 63782-63796. [75]
- **Quim Aguado-Puig**, Santiago Marco-Sola, Juan Carlos Moure, Christos Matzoros, David Castells-Rufas, Antonio Espinosa, and Miquel Moreto. "WFA-GPU: Gap-affine pairwise alignment using GPUs." bioRxiv (2022) [76]
- David Castells-Rufas, Santiago Marco-Sola, **Quim Aguado-Puig**, Antonio Espinosa-Morales, Juan Carlos Moure, Lluc Alvarez, and Miquel Moretó. "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), pp. 174-178. IEEE, 2021. [60]
- David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, **Quim Aguado**, and Antonio Espinosa. "FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS." IEEE Access 10 (2022): 22079-22100. [77]
- WFA-GPU: Accelerated gap-affine pairwise alignment, 21st European Conference on Computational Biology, 2022 (Poster).

5.2 Source code and datasets

All the materials generated from this thesis are open source and publicly available. The source code of both implementations is freely available under the MIT license. The eWFA-GPU edit-distance implementation (Section 3) is found on `https://github.com/quim0/eWFA-GPU`. The WFA-GPU gap-affine implementation (Section 4) is found on `https://github.com/quim0/WFA-GPU`. Datasets are freely available upon request to the author.

5.3 Future work

There are many interesting ideas to explore in the context of this work. First of all, the current WFA-GPU implementation can be extended with heuristics, multi-GPU

support, and a tile-based version to allow multiple GPU blocks to work cooperatively on the same alignment, reducing the memory pressure and allowing the exploitation of more GPU resources per alignment.

We are also working on implementing the Bidirectional WFA (BiWFA [23]) on GPU. This recent improvement over the WFA algorithm dramatically reduces the memory usage (from $O(s^2)$ to $O(s)$) and depicts the opportunity to perform more work in parallel thanks to its divide-and-conquer strategy.

Ultimately, we would like to integrate our library within existing mapping tools, accelerating genomics pipelines by using WFA-GPU instead of traditional dynamic-programming, CPU-based implementations.

5.4 Financial and technical support

This thesis 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 Spanish 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). Q. Aguado has been partially supported by PRE2021-101059 (founded by MCIN/AEI/10.13039/501100011033 and FSE+).

References

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

[2] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. "The GEM mapper: fast, accurate and versatile alignment by filtration". In: *Nature methods* 9.12 (2012), pp. 1185–1188.

[3] Angelika Merkel, Marcos Fernández-Callejo, Eloi Casals, Santiago Marco-Sola, Ronald Schuyler, Ivo G Gut, and Simon C Heath. "gemBS: high throughput processing for DNA methylation data from bisulfite sequencing". In: *Bioinformatics* 35.5 (2019), pp. 737–742.

[4] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. "ABySS: a parallel assembler for short read sequence data". In: *Genome research* 19.6 (2009), pp. 1117–1123.

[5] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation". In: *Genome research* 27.5 (2017), pp. 722–736.

[6] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, 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.

[7] Bernardo Rodrıguez-Martın, Emilio Palumbo, Santiago Marco-Sola, Thasso Griebel, Paolo Ribeca, Graciela Alonso, Alberto Rastrojo, Begoña Aguado, Roderic Guigó, and Sarah Djebali. "ChimPipe: accurate detection of fusion genes and transcription-induced chimeras from RNA-seq data". In: *BMC genomics* 18.1 (2017), pp. 1–17.

[8] 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.

[9] Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

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

[11] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. "Big data: astronomical or genomical?" In: *PLoS biology* 13.7 (2015), e1002195.

[12] Lauren M Petersen, Isabella W Martin, Wayne E Moschetti, Colleen M Kershaw, and Gregory J Tsongalis. "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.

[13] Xun Gu and Wen-Hsiung Li. "The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment". In: *Journal of molecular evolution* 40.4 (1995), pp. 464–473.

[14] Reed A Cartwright. "Logarithmic gap costs decrease alignment accuracy". In: *BMC bioinformatics* 7.1 (2006), pp. 1–12.

[15] Mohammed Alser, Joel Lindegger, Can Firtina, Nour Almadhoun, Haiyu Mao, Gagandeep Singh, Juan Gomez-Luna, and Onur Mutlu. "Going From Molecules to Genomic Variations to Scientific Discovery: Intelligent Algorithms and Architectures for Intelligent Genome Analysis". In: *arXiv preprint arXiv:2205.07957* (2022).

[16] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[17] David Sankoff. "Matching sequences under deletion/insertion constraints". In: *Proceedings of the National Academy of Sciences* 69.1 (1972), pp. 4–6.

[18] Peter H Sellers. "On the theory and computation of evolutionary distances". In: *SIAM Journal on Applied Mathematics* 26.4 (1974), pp. 787–793.

[19] Robert A Wagner and Michael J Fischer. "The string-to-string correction problem". In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173.

[20] Vladimir I Levenshtein et al. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[21] Daniel S. Hirschberg. "A linear space algorithm for computing maximal common subsequences". In: *Communications of the ACM* 18.6 (1975), pp. 341–343.

[22] Eugene W Myers and Webb Miller. "Optimal alignments in linear space". In: *Bioinformatics* 4.1 (1988), pp. 11–17.

[23] Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. "Optimal gap-affine alignment in O (s) space". In: *bioRxiv* (2022).

[24] Osamu Gotoh. "An improved algorithm for matching biological sequences". In: *Journal of molecular biology* 162.3 (1982), pp. 705–708.

[25] Michael S Waterman, Temple F Smith, and William A Beyer. "Some biological sequence metrics". In: *Advances in Mathematics* 20.3 (1976), pp. 367–387.

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

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

[28] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. "Fast gap-affine pairwise alignment using the wavefront algorithm". In: *Bioinformatics* 37.4 (2021), pp. 456–463.

[29] Fred J Damerau. "A technique for computer detection and correction of spelling errors". In: *Communications of the ACM* 7.3 (1964), pp. 171–176.

[30] Osamu Gotoh. "Optimal sequence alignment allowing for long gaps". In: *Bulletin of mathematical biology* 52.3 (1990), pp. 359–373.

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

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

[33] Gad M Landau and Uzi Vishkin. "Fast parallel and serial approximate string matching". In: *Journal of algorithms* 10.2 (1989), pp. 157–169.

[34] Alberto Zeni, Giulia Guidi, Marquita Ellis, Nan Ding, Marco D Santambrogio, Steven Hofmeyr, Aydın Buluç, Leonid Oliker, and Katherine Yelick. "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.

[35] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. "Thread-cooperative, bit-parallel computation of levenshtein distance on GPU". In: *Proceedings of the 28th ACM international conference on Supercomputing*. 2014, pp. 103–112.

[36] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. "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.

[37] Alejandro Chacón, Santiago Marco Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. "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.

[38] 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.

[39] Khaled Balhaf, Mohammad A Alsmirat, Mahmoud Al-Ayyoub, Yaser Jararweh, and Mohammed A Shehab. "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.

[40] 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.

[41] 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.

[42] Subtil N Pantaleoni J. *NVBIO*. `https://nvlabs.github.io/nvbio`. Accessed: 2022-07-12. 2015.

[43] 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.

[44] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data". In: *BMC bioinformatics* 20.1 (2019), pp. 1–20.

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

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

[47] Jikai Zhang, Haidong Lan, Yuandong Chan, Yuan Shang, Bertil Schmidt, and Weiguo Liu. "BGSA: A bit-parallel global sequence alignment toolkit for multi-core and many-core architectures". In: *Bioinformatics* 35.13 (2019), pp. 2306–2308.

[48] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. "SeqAn an efficient, generic C++ library for sequence analysis". In: *BMC bioinformatics* 9.1 (2008), pp. 1–9.

[49] 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.

[50] 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.

[51] Khaled Balhaf, Mohammed A Shehab, T Wala'a, Mahmoud Al-Ayyoub, Mohammed Al-Saleh, and Yaser Jararweh. "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.

[52] 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.

[53] Kailash W Kalare, Mohammad S Obaidat, Jitendra V Tembhurne, Chandrashekhar Meshram, and Kuei-Fang Hsiao. "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.

[54] 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.

[55] Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H Campbell. "An algorithm for fast edit distance computation on GPUs". In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–9.

[56] Lucas SN Nunes, Jacir L Bordim, Koji Nakano, and Yasuaki Ito. "A fast approximate string matching algorithm on GPU". In: *2015 Third international symposium on computing and networking (CANDAR)*. IEEE. 2015, pp. 188–192.

[57] 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.

[58] 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.

[59] 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.

[60] David Castells-Rufas, Santiago Marco-Sola, Quim Aguado-Puig, Antonio Espinosa-Morales, Juan Carlos Moure, Lluc Alvarez, and Miquel Moretó. "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.

[61] Liangwei Cai, Qi Wu, Tongsheng Tang, Zhi Zhou, and Yuan Xu. "A Design of FPGA Acceleration System for Myers bit-vector based on OpenCL". In: *2019 International Conference on Intelligent Informatics and Biomedical Sciences (ICI-IBMS)*. IEEE. 2019, pp. 305–312.

[62] 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.

[63] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. "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.

[64] 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.

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

[66] Lucas SN Nunes, Jacir Luiz Bordim, Koji Nakano, and Yasuaki Ito. "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.

[67] Muhammad Umair Sadiq, Muhammad Murtaza Yousaf, Laeeq Aslam, Muhammad Aleem, Shahzad Sarwar, and Syed Waqar Jaffry. "NvPD: novel parallel edit distance algorithm, correctness, and performance evaluation". In: *Cluster Computing* 23.2 (2020), pp. 879–894.

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

[69] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (2018), pp. 3094–3100.

[70] Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. "ADEPT: a domain independent sequence alignment strategy for gpu architectures". In: *BMC bioinformatics* 21.1 (2020), pp. 1–29.

[71] *CudaAligner*. `https://github.com/clara-parabricks/GenomeWorks`. Accessed: 2022-04-06. 2022.

[72] Joël Lindegger, Damla Senol Cali, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm". In: *arXiv preprint arXiv:2203.15561* (2022).

[73] Hajime Suzuki and Masahiro Kasahara. "Introducing difference recurrence relations for faster semi-global alignment of long sequences". In: *BMC bioinformatics* 19.1 (2018), pp. 33–47.

[74] *WFA2 Library*. `https://github.com/smarco/WFA2-lib`. Accessed: 2022-04-09. 2022.

[75] Quim Aguado-Puig, Santiago Marco-Sola, Juan Carlos Moure, David Castells-Rufas, Lluc Alvarez, Antonio Espinosa, and Miquel Moreto. "Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm". In: *IEEE access* 10 (2022), pp. 63782–63796.

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

[77] David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, Quim Aguado, and Antonio Espinosa. "FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS". In: *IEEE Access* 10 (2022), pp. 22079–22100.