WFA-GPU: Gap-affine pairwise alignment using GPUs

Quim Aguado-Puig [1,*], Santiago Marco-Sola [2,1,*], Juan Carlos Moure [1], Christos Matzoros [2], David Castells-Rufas [1], Antonio Espinosa [1], Miquel Moreto,[2,3]

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

* To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Advances in genomics and sequencing technologies demand faster and 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 WFA algorithm paves the way for more efficient alignment tools, improving time and memory complexity over previous methods. Notwithstanding, modern high performance computing (HPC) platforms rely on accelerator-based architectures that exploit parallel computing resources to improve over classical computing CPUs.

Results: This paper presents the WFA-GPU, a GPU (Graphics Processing Unit)-accelerated tool to compute exact gap-affine alignments based on the WFA algorithm. We present the algorithmic adaptations and performance optimizations that allow exploiting the massive parallel capabilities of modern GPU devices to accelerate the alignment computations. In particular, 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, and up to 12× when using heuristic methods on long and noisy sequences. Compared to other state-of-the-art tools and libraries, the WFA-GPU is up to 175× faster than other GPU implementations and up to four orders of magnitude faster than other CPU implementations.

Availability: WFA-GPU code and documentation is publicly available at https://github.com/quim0/WFA-GPU

Contact: quim.aguado.p@gmail.com

1 Introduction

Pairwise sequence alignment is a fundamental building block in many tools and libraries used in genomics and bioinformatics. In particular, it plays a critical role for methods like read mapping (Li, 2013; Marco-Sola *et al.*, 2012), de-novo genome assembly (Simpson *et al.*, 2009; Koren *et al.*, 2017), variant calling (McKenna *et al.*, 2010; Rodríguez-Martín *et al.*, 2017), and many others (Durbin *et al.*, 1998; Jones *et al.*, 2004).

Consequently, sequence alignment algorithms have been extensively studied over the last 40 years, introducing multiple strategies like dynamic programming (DP) (Sellers, 1980; Ukkonen, 1985), automatas (Baeza-Yates, 1992; Wu and Manber, 1992; Navarro, 1997), and bit-parallelism techniques (Myers, 1999; Baeza-Yates, 1989). Nonetheless, these algorithms are bound by the quadratic time and memory requirements on the sequence length. Thus, the use of classical alignment algorithms becomes impractical as the input sequences increase in length. Many variations and optimizations have been proposed over the years to overcome these limitations. These solutions include techniques such as banded approaches that only compute a portion of the DP-matrix (Suzuki and Kasahara, 2017), data-layout organizations that allow using SIMD instructions (Rognes and Seeberg, 2000; Farrar, 2007; Wozniak, 1997), bit-packed encodings (Myers, 1986, 1999), and other methods (Ukkonen, 1985; Zhao *et al.*, 2013). Nevertheless, all these proposals retrain the quadratic requirements and fail to scale for long sequences.

Recently, the Wavefront Alignment (WFA) algorithm (Marco-Sola *et al.*, 2021) was proposed. The WFA is able to compute the exact alignment between two sequences using gap-affine penalties. In essence, the WFA algorithm computes partial alignments of increasing score s

© The Author 2022.

until the optimal alignment is found. The WFA algorithm takes advantage of homologous regions between sequences to accelerate the alignment process. As a result, the WFA algorithm largely outperforms other state-of-the-art methods, requiring $O(ns)$ time and $O(s^2)$ memory (where n is the length of the sequence and s is the optimal alignment score).

Notwithstanding, the ever-increasing yields and lengths of sequencing systems challenges the scalability of current sequence alignment methods. In particular, modern sequencing technologies, like PacBio or Oxford Nanopore, can produce sequences more than 100× longer than those produced by classical Illumina sequencers at a fraction of the cost. The adoption of these technologies calls for the development of faster and more scalable alignment solutions (Petersen *et al.*, 2019). So imperative is the need to scale to larger volumes of genomic data, that the adoption of HPC solutions has become more and more frequent. In particular, GPUs have been widely adopted as hardware accelerators in many scientific applications (Hwu, 2011; Owens *et al.*, 2008; Chacón *et al.*, 2014; Lin *et al.*, 2017) as they provide higher computational throughput and memory bandwidth compared with traditional multi-core processors. In this context, alignment algorithms and tools need to address the efficient exploitation of these hardware accelerators to keep up with the pace of modern sequence data production.

This paper presents a GPU-accelerated implementation of the WFA algorithm for exact gap-affine pairwise sequence alignment. To that end, we describe the algorithmic adaptations performed on the WFA algorithm to exploit the massive parallel capabilities of modern GPU architectures. In particular, we propose an inter-sequence and, at the same time, intra-sequence alignment strategy to compute multiple WFA alignments in parallel in the GPU. Moreover, we propose a succinct backtrace encoding to reduce the overall memory consumption of the original WFA algorithm. Also, we present a supplementary extension of the WFA algorithm to incorporate the efficient adaptive band heuristic to further improve its performance on GPU, achieving nearly the same accuracy as the original exact WFA algorithm. We characterize the performance of our implementation and present the different performance trade-offs of our solution. As a result, we demonstrate that our implementation outperforms other GPU and CPU state-of-the-art libraries and tools for sequence alignment.

The rest of the paper is structured as follows. Section 2 presents the definitions and methods of our proposal. Section 3 shows the experimental results, comparing the performance of our method against other state-of-the-art implementations on both CPU and GPU systems. Finally, Section 4 presents a discussion on the methods presented and summarizes the contributions and impact of this work.

2 Methods

2.1 Wavefront Pairwise Alignment

Let the query $q = q_0 q_1 ... q_{n-1}$ and the text $t = t_0 t_1 ... t_{m-1}$ be strings of length n and m, respectively. Similarly, let $q_{i...j}$ denote a substring of q from the i-th to the j-th character (both included). Also, let $\{x, o, e\}$ be the set of gap-affine penalties, where x is the mismatch cost and the gap cost is expressed as the linear function $g(l) = o + l \cdot e$ (where l is the length of the gap).

In essence, the WFA algorithm computes partial alignments of increasing score until an alignment with score s reaches coordinate (n, m) (i.e., the end of the DP matrix). This way, the algorithm is able to compute the optimal score s and retrieve the optimal alignment by tracing back the alignment operations (i.e., $\{M, X, I, D\}$ for match, mismatch, insertion, and deletion) that led to the solution.

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. In (Marco-Sola *et al.*, 2021), the authors prove that the farthest reaching. points of $\widetilde{W}_s$ can be computed using the wavefronts with score $s - o - e$, $s - e$, and $s - x$, using Eq. 1 (where *LCP*(v,w) is the longest common prefix between two string v and w). Note that the wavefront component $\widetilde{X}_{s,k}$ can be inferred using $\widetilde{M}_{s,k}$, $\widetilde{I}_{s,k}$, and $\widetilde{D}_{s,k}$, and we can avoid storing it.

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

Starting with $\widetilde{X}_{0,0} = 0$, the WFA algorithm progressively computes wavefronts of increasing score. For a given score s, it first increases each wavefront offset $\widetilde{X}_{s,k}$ according to the number of matching characters along the diagonal k (i.e., computing $LCP(q_{\widetilde{X}_{s,k}-k...n-1}, t_{\widetilde{X}_{s,k}...m-1})$ for every diagonal k). Then, the algorithm computes the next wavefronts with score $s + 1$ using Eq. 1 and the previously computed wavefronts. This process iterates until a wavefront $\widetilde{M}$ reaches the bottom-right cell (n, m) of the DP matrix (i.e., $\widetilde{M}_{s,m-n} = m$). Hence, the optimal alignment score is s and the alignment operations can be retrieved by tracing back the wavefronts that originated the farthest reaching point $\widetilde{M}_{s,m-n}$.

In the worst case, the WFA algorithm requires computing s wavefronts of increasing length ($\sum_{l=0}^{s} 1 + 2l$) and compute the *LCP()* for each wavefront offset. Nevertheless, each diagonal offset cannot be incremented more than the length of the sequence. Thus, the WFA algorithm requires $O(ns + s^2)$ time ($O(ns)$ if the sequences are homologous as $s << n$) and $O(s^2)$ memory in the worst case.

2.2 Graphics Processing Units

GPUs have rapidly emerged as successful hardware accelerators in the HPC community for scientific applications. GPUs are massively parallel devices containing multiple throughput-oriented processing units called Stream Multiprocessors (SMs). Each SM can host over a thousand concurrent threads, and each clock cycle, using aggressive fine-grained multithreading techniques, can start executing hundreds of instructions on multiple SIMD cores (a.k.a. CUDA cores). A CPU-GPU co-design involves determining a computationally intensive function or region of a program (i.e., computation kernel) to be offloaded to the GPU. To exploit GPU parallel computing capabilities, applications must launch tens of thousands of threads, grouped in thread blocks, across all available SMs. Threads within a block can cooperate (e.g., via synchronization primitives, registers, or shared memory) to solve a given computational kernel. To maximize performance, GPUs rely on exploiting high degrees of parallelism, much higher than those of regular CPU multi-cores.

Each SM is equipped with fast on-chip memories with a relatively small capacity. They are distributed among compiler-managed register memory (up to 256KB), processor-managed L1 cache, and programmer-managed shared memory (up to 100KB for the total sum of L1 and shared memory). The SMs get their data from an off-chip RAM memory of several GBs capable of delivering bandwidths of over 600 GB/s. An L2 cache holds

Algorithm 1: 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()

end

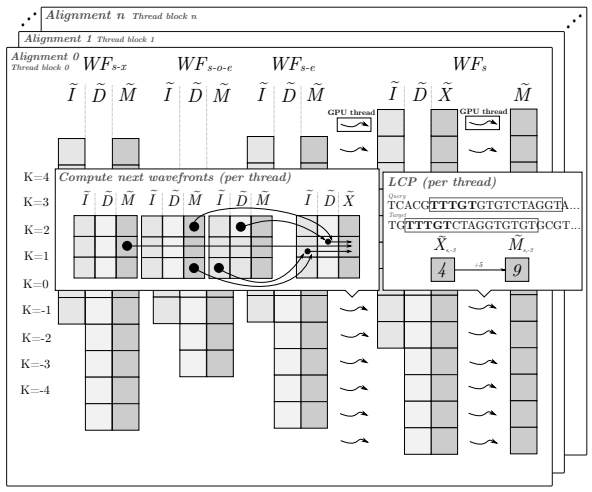

Fig. 1: Detail of the parallel computation of a given wavefront s using multiple GPU threads. At the same time, other alignments are computed by different thread blocks on the GPU. This figure shows how each wavefront diagonal can be computed independently using a single GPU thread.

several MBs of the RAM memory and delivers 3-5 times higher bandwidth. The design of the GPU-accelerated program determines the amount of data required per concurrent thread, and the mapping of these data to registers, shared memory, or global memory. There is a critical compromise between the number of concurrent threads hosted by each SM and the amount of data that the threads can store in fast memories (registers, L1, or shared memory). Allocating too many threads per SM increases the number of data accesses to slow memory, and performance may suffer. Reducing the number of threads per SM can improve memory performance, but at the price of reducing parallelism and the ability to hide long execution and memory latencies, potentially decreasing performance.

2.3 GPU Parallel Wavefront Alignment

The WFA algorithm depicts simple computational patterns when computing Eq. 1. 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 1 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. 1 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. 1 (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.

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 2.4 and 2.5, 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 2.6, we present a strategy to accelerate the *LCP()* computation and, in Section 2.7, we explore the trade-offs of using an adaptive band strategy to reduce the volume of computation performed on the GPU. Finally, in Section 2.8, 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.

2.4 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 2.8.

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.

2.5 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.

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 performed in Algorithm 1 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. 1, 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. 2 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.

2.6 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.

2.7 Adaptive Band Wavefront Alignment

Motivated by the necessity to scale and provide faster alignment solutions, in many situations the exact computation of the alignment is unnecessary. In those cases, heuristic strategies allow reducing the number of computations required at the expense of a potentially small loss of accuracy.

Notably, one of the most successful heuristics proposed to date is the adaptive band strategy in the spirit of (Suzuki and Kasahara, 2017). This heuristic differs from the classical static band heuristics by allowing the band to move up or down between diagonals. As a result, the band focuses on the most promising alignment paths based on the content of the DP matrix.

In the context of the WFA algorithm, the adaptive band strategy allows restricting the computation to a band of diagonal offsets, discarding unpromising alignment paths unlikely to lead to the optimal solution. If the adaptive band heuristic is enabled, the WFA-GPU implementation samples the wavefront offsets at the quarters and moves the band up or down by one diagonal if the upper or lower diagonals are closer to the alignment completion. The effective size of the band can be configured to best adapt to the requirements of the experiment. For convenience, we also implement an automatic mode where the threads do at most two iterations on a wavefront. In practice, for sufficiently large band sizes (e.g., about 10% of the sequence length), this strategy yields nearly the same accuracy results as the exact WFA, performing faster in many cases.

2.8 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.

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.

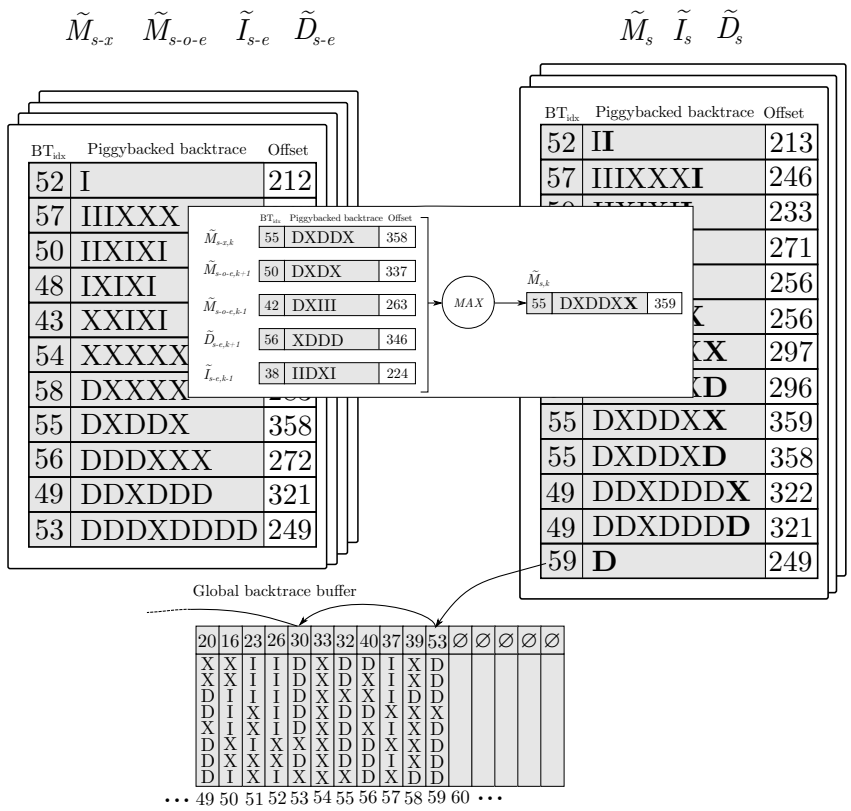

Fig. 2: Illustration of the piggybacked backtrace strategy and data-layout organization. From left to right of the figure, we show the source wavefronts ($\widetilde{M}_{s-x}$, $\widetilde{M}_{s-o-e}$, $\widetilde{I}_{s-e}$, and $\widetilde{D}_{s-e}$) and how they are combined to generated wavefronts 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. 1 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).

Furthermore, the asynchronous implementation of WFA-GPU allows employing idle CPU time to rescue alignments returned by the GPU. As explained in Section 2.3, 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.

3 Results

We evaluated the performance of the WFA-GPU, together with other CPU and GPU tools for sequence alignment representing the current state-of-the-art. In Section 3.2 we present an evaluation on simulated datasets and, in Section 3.3, we present the experimental results obtained using real datasets.

3.1 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 1). The target sequences are retrieved from mapping the source sequences against GRCh38 using Minimap2 (Li, 2018) 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 (Ahmed *et al.*, 2019), and two NVIDIA libraries (NVBio (Pantaleoni J, 2015) and CudaAligner (cud, 2022) from Clara Parabricks Genomeworks). Unfortunately, we were unable to include the recently presented GPU aligners ADEPT (Awan *et al.*, 2020) and Logan (Zeni *et al.*, 2020) due to execution errors or inadequacy to perform basic pairwise alignment, respectively. As for the CPU tools, we selected the most widely-used and efficient libraries available to date. That is, Seqan (Döring *et al.*, 2008), Parasail (Daily, 2016), Edlib (Šošić and Šikić, 2017), and KSW2 (Suzuki and Kasahara, 2018). Naturally, we also include the original WFA implementation (WFA2-lib (wfa, 2022)) into

Table 1. 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

Table 2. 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 that 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	64	87.2	64	83.8	65	81.7	271	1.2	272	0.6	274	0.5	error	n/a	error	n/a	error	n/a
GPU	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
GPU	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
GPU	WFA-GPU	11	100	18	100	27	100	6	100	23	100	78	100	4	100	21	100	71	100
GPU	WFA-GPU(band)	11	100	18	100	26	100	7	100	27	100	79	100	4	100	18	100	40	100
CPU	Seqan	2028	100	2028	100	2190	100	9066	100	9288	100	9774	100	9912	100	10098	100	10440	100
CPU	Parasail(strip)	8436	100	8358	100	8238	100	32274	100	32178	100	31842	100	33408	100	33306	100	32760	100
CPU	Parasail(scan)	1197	100	1183	100	1197	100	3882	100	3902	100	3911	100	4869	100	4857	100	4825	100
CPU	Parasail(diag)	1908	100	1914	100	1916	100	7482	100	7488	100	7488	100	7686	100	7692	100	7698	100
CPU	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
CPU	KSW2	298	100	306	100	317	100	876	100	882	100	882	100	2028	100	2034	100	2040	100
CPU	WFA	20	100	52	100	77	100	34	100	70	100	199	100	15	100	86	100	277	100

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.

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.

3.2 Evaluation on Simulated Data

Table 2 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 outperfoms 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, and between 2.4 and 6 times faster than GASAL2 with higher accuracy.

For medium length sequences (i.e., ∼1Kbps), other GPU implementations either fail due to execution errors, like NVBio, or obtain a recall lower than 10%. Compared to CPU implementations, WFA-GPU executes 2.5-5.8× faster than the original WFA and up to 5500× faster than other libraries. In both cases, 150bps and 1Kbps long sequences, the adaptive band execution of WFA-GPU does not provide any significant improvement. In those cases, there is little margin to improve the original WFA-GPU algorithm.

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., GASAL2 and NVBio) or give incorrect results (i.e., CudaAligner). 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. In this case, the adaptive band strategy can achieve an additional 1.8× speedup, retaining a 100% recall.

3.3 Evaluation on Real Genomic Data

Table 3 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 1).

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). In contrast, WFA-GPU is 7.7-175× faster than the other GPU libraries, achieving perfect recall (100%). 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. Our implementation outperforms by four orders of magnitude other CPU tools and libraries. Again, the only GPU implementation able to finish is CudaAligner, even though it obtains a significantly low recall (less than 50%).

Considering the Nanopore dataset, having large sequences with high error rate, WFA-GPU is 4 times faster than the CPU implementation of WFA. 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.

Aligning high-quality datasets (like Illumina, Moleculo, and even PacBio datasets) the banded strategy doesn't yield any performance advantage, obtaining similar execution times compared to the non-banded version of WFA-GPU. On the other hand, for long and noisy datasets (containing sequences with high error rate, like the Nanopore dataset) the adaptive band strategy obtains significant speedups (up to 3×) compared to the regular WFA-GPU version, giving an overall speedup of 12× compared to the CPU version of WFA.

In general, other GPU implementations either fail to scale with increasing sequence lengths, drop enormously in recall, or simply fail to execute. Although CudaAligner delivers good performance results, it is bounded to produce edit-distance alignments that fail to capture

Table 3. 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	1027	98.6	2956	94.1	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(band)	9	100	22	100	16	100	80	99.7	145	99.7	1472	98.9
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

the biological insights that gap-affine models do. While traditional DP-based algorithms fail to scale with sequence length, WFA-based implementations demonstrate to scale with increasing error rates and lengths. In practice, DP-based implementations require impractical execution times. As opposed, our WFA-GPU implementation delivers good performance results, even when aligning long and noisy sequences.

4 Discussion

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. We proposed algorithmic adaptations and optimizations of the WFA to effectively parallelize the alignment task, exploiting the high performance capabilities of modern GPU cards.

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 175× 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. Using the adaptative band strategy to align long and noisy sequences, our method reaches a maximum speedup of 12× compared to the original WFA CPU attaining a 98.9% accuracy. To the best of our knowledge, WFA-GPU is the only GPU-based pairwise aligner capable of producing exact gap-affine alignments for long-sequencing datasets, like PacBio HiFi or Oxford Nanopore, in practical time using commodity GPU devices.

With the advent of improved sequencing technologies and more sophisticated genome studies, WFA-GPU offers an accurate, fast, and scalable sequence alignment solution that effectively exploits the massive computing capabilities of modern GPU devices. Therefore, we expect the WFA-GPU to become a valuable and practical addition to the toolkit of bioinformatics tools that support efficient research in future genome-scale analysis.

Funding

This research was supported by the European Unions's Horizon 2020 Framework Programme under the DeepHealth project [825111], by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of total cost eligible under the DRAC project [001-P-001723]. It was also supported by the 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). M.M. was partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104. S.M. was supported by Juan de la Cierva fellowship grant IJC2020-045916-I funded by MCIN/AEI/ 10.13039/501100011033 and by "European Union NextGenerationEU/PRTR".

References

(2022). Cudaaligner. `https://github.com/clara-parabricks/GenomeWorks`. Accessed: 2022-04-06.

(2022). Wfa2 library. `https://github.com/smarco/WFA2-lib`. Accessed: 2022-04-09.

Ahmed, N., Lévy, J., Ren, S., Mushtaq, H., Bertels, K., and Al-Ars, Z. (2019). Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data. *BMC Bioinformatics*, **20**(1), 520.

Awan, M. G., Deslippe, J., Buluc, A., Selvitopi, O., Hofmeyr, S., Oliker, L., and Yelick, K. (2020). Adept: a domain independent sequence alignment strategy for gpu architectures. *BMC bioinformatics*, **21**(1), 1–29.

Baeza-Yates, R. (1989). *Efficient text searching*. University of Waterloo.

Baeza-Yates, R. A. (1992). Text-retrieval: Theory and practice. In *IFIP Congress (1)*, volume 12, pages 465–476. Citeseer.

Chacón, A., Marco-Sola, S., Espinosa, A., Ribeca, P., and Moure, J. C. (2014). Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112.

Daily, J. (2016). Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, **17**(1), 1–11.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). Seqan an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, **9**(1), 1–9.

Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. (1998). *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press.

Farrar, M. (2007). Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, **23**(2), 156–161.

Hwu, W.-M. W. (2011). *GPU computing gems emerald edition*. Morgan Kaufmann Publishers Inc.

Jones, N. C., Pevzner, P. A., and Pevzner, P. (2004). *An introduction to bioinformatics algorithms*. MIT press.

Koren, S., Walenz, B. P., Berlin, K., Miller, J. R., Bergman, N. H., and Phillippy, A. M. (2017). Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, **27**(5), 722–736.

Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*.

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

Lin, C.-H., Li, J.-C., Liu, C.-H., and Chang, S.-C. (2017). Perfect hashing based parallel algorithms for multiple string matching on graphic processing units. *IEEE Transactions on Parallel and Distributed Systems*, **28**(9), 2639–2650.

Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. (2012). The gem mapper: fast, accurate and versatile alignment by filtration. *Nature Methods*, **9**(12), 1185–1188.

Marco-Sola, S., Moure, J. C., Moreto, M., and Espinosa, A. (2021). Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, **37**(4), 456–463.

McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., *et al.* (2010). The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, **20**(9), 1297–1303.

Myers, E. W. (1986). An o(nd) difference algorithm and its variations. *Algorithmica*, **1**(1-4), 251–266.

Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, **46**(3), 395–415.

Navarro, G. (1997). A partial deterministic automaton for approximate string matching. *Department of Computer Science, University of Chile*.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, **96**(5), 879–899.

Pantaleoni J, S. N. (2015). Nvbio. `https://nvlabs.github.io/nvbio`. Accessed: 2021-09-15.

Petersen, L. M., Martin, I. W., Moschetti, W. E., Kershaw, C. M., and Tsongalis, G. J. (2019). Third-generation sequencing in the clinical laboratory: exploring the advantages and challenges of nanopore sequencing. *Journal of Clinical Microbiology*, **58**(1), e01315–19.

Rodríguez-Martín, B., Palumbo, E., Marco-Sola, S., Griebel, T., Ribeca, P., Alonso, G., Rastrojo, A., Aguado, B., Guigó, R., and Djebali, S. (2017). Chimpipe: accurate detection of fusion genes and transcription-induced chimeras from rna-seq data. *BMC genomics*, **18**(1), 1–17.

Rognes, T. and Seeberg, E. (2000). Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, **16**(8), 699–706.

Sellers, P. H. (1980). The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, **1**(4), 359–373.

Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., and Birol, I. (2009). Abyss: a parallel assembler for short read sequence data. *Genome Research*, **19**(6), 1117–1123.

Šošić, M. and Šikić, M. (2017). Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, **33**(9), 1394–1395.

Suzuki, H. and Kasahara, M. (2017). Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming. *BioRxiv*, page 130633.

Suzuki, H. and Kasahara, M. (2018). Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, **19**(1), 33–47.

Ukkonen, E. (1985). Finding approximate patterns in strings. *Journal of Algorithms*, **6**(1), 132–137.

Wozniak, A. (1997). Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, **13**(2), 145–150.

Wu, S. and Manber, U. (1992). Fast text searching: allowing errors. *Communications of the ACM*, **35**(10), 83–91.

Zeni, A., Guidi, G., Ellis, M., Ding, N., Santambrogio, M. D., Hofmeyr, S., Buluç, A., Oliker, L., and Yelick, K. (2020). Logan: High-performance gpu-based x-drop long-read alignment. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 462–471. IEEE.

Zhao, M., Lee, W.-P., Garrison, E. P., and Marth, G. T. (2013). Ssw library: an simd smith-waterman c/c++ library for use in genomic applications. *PloS one*, **8**(12).