BiWFA: gap-affine pairwise sequence alignment using GPUs

Simone Bevilacqua

June 29, 2024

POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Abstract

Pairwise sequence alignment is a key problem in computational biology and bioinformatics. Recent advances in sequencing technologies set sequence analysis algorithms as fundamental tools for genomics. Classical pairwise alignment algorithms based on dynamic programming are strongly limited by quadratic requirements in time and memory. To overcome this issue, *Marco-Sola et al.* [10] recently proposed the Bidirectional Wavefront Alignment algorithm (BiWFA) that manages to perform exact gap-affine alignment with a reduced space complexity of O(s), where s is the optimal alignment score. However, all these solutions present computational limitations when performing multi-sequence alignments since memory operations become the bottleneck. To overcome this issue, we present a GPU-based implementation of the BiWFA algorithm that exploits thread-level parallelism to increase the performance of the software version achieving speedups up to 31.73x.

1 Introduction

Pairwise sequence alignment is a fundamental bioinformatics technique used to identify regions of similarity between two biological sequences (DNA, RNA, or proteins). In genomics, pairwise sequence alignment plays a key role in several applications. It is employed in the identification and annotation of genes, where it helps in comparing newly sequenced genes with known sequences to predict their functions. It is also used in evolutionary studies to trace the lineage of genes and organisms, revealing how species have diverged over time. Furthermore, pairwise alignment aids in detecting mutations and variants, which is essential for understanding genetic diseases. Most of these applications are based on alignment algorithms like the Needleman-Wunsch algorithm (NW) [11] for global alignment and the Smith-Waterman algorithm (SW) [14] for local alignment. These algorithms use scoring matrices to quantify sequence similarities and gaps. Some extended versions of these algorithms add some more detailed properties, such as the Smith-Waterman-Gotoh algorithm (SWG) [5], which considers a slightly different and more precise concept of the gap. All these algorithms take a quadratic time and space complexity and this makes them particularly expensive when aligning rather long sequences. This limitation led the research effort in this field towards better-performing sequence alignment algorithms. The first improvements were reached just by focusing on the algorithm process. The WFA algorithm proposed by *Marco-Sola et al.* [9] is an extension of SWG that achieves better performances by avoiding computing the whole matrices, still producing

correct results. The asymptotic bounds of time and space complexity in WFA are $O(ns)$ and $O(s^2)$, respectively, where n is the sequence length and s is the optimal alignment score. The authors recently introduced a further optimization of WFA, called BiWFA, which achieves better performances obtaining a lower space complexity.

When searching for improvements, researchers have also tried to exploit heterogeneous architectures to parallelize the execution of these algorithms. Nowadays, modern genomics applications are always required to process an increasing amount of data, so hardware accelerators play a key role in going beyond the limitations of general-purpose architectures. The two main solutions in the State of the Art are GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays). Despite the heterogeneity of these architectures, adapting to algorithms like SW and NW does not bring significant improvements due to their intrinsic space and time complexities. For these reasons, researchers decided to focus both on the improvement of the algorithms themselves, trying to exploit some optimizations, but also on the heterogeneity of the architecture to achieve better performances. We propose an implementation of the BiWFA algorithm running on GPU architectures to try exploiting thread parallelism inside the same alignment to improve the execution time of the algorithm.

2 Background

2.1 WFA

The WaveFront Alignment (WFA) algorithm proposed by *Marco-Sola et al.* is a pairwise sequence alignment algorithm that extends the NW algorithm employing gap-affine scoring to assign different penalty scores for opening new gaps and for extending the already opened ones.

Let the pattern p and the text t be strings of length n and m respectively. Let also (x, o, e) be the gap-affine penalties: a mismatch costs x, the opening of a new gap costs o and the extension of a new one costs e, so that, given a gap of length l, its costs is $o + l * e$. We assume that $x > 0$, $o > 0$, $e > 0$, and also that the score between two matching characters is 0. The main idea of WFA is to compute partial optimal alignments of increasing scores until one of them reaches the bottom right coordinate of the DP matrix (n, m). The score of such alignment will determine the overall optimal alignment score. The algorithm uses three matrices to store information about the already processed characters: M, I, and D. A cell $M(i, j)$ of matrix M keeps the information about the best score for the alignment of $p(0, i)$ and $t(0, j)$, that

is the first i characters of the pattern compared against the first j characters of the text. Cells $D(i,j)$ and $I(i,j)$ store the best alignment score when the alignment between the two substrings ends with a gap in the sequence p or t, respectively. Given a position $v \in [0,n]$ in the pattern and a position $h \in [0,m]$ in the text the equations to update the three matrices are the following:

$$\begin{cases} I_{v,h} = min\{\ M_{v,h} + o + e,\ I_{v,h-1} + e\ \} \\ D_{v,h} = min\{\ M_{v-1,h} + o + e,\ D_{v-1,h} + e\ \} \\ M_{v,h} = min\{\ I_{v,h},\ D_{v,h},\ M_{v-1,h-1} + s(q_{v-1}, t_{h-1})\ \} \end{cases} \tag{1}$$

M, I, and D are matrices since the pattern is placed horizontally and the text that is compared against it is placed vertically to form, indeed, a grid. The alignment of the two sequences starts from the top left corner of the grid and each movement corresponds to a specific edit operation: a diagonal shift represents the equivalence between pattern and text characters, a horizontal move corresponds to the deletion from the pattern, while a vertical movement stands for an insertion in the text. By iterating this process and the recursive equations, we can fill the three matrices. The score contained in the bottom right cell will represent the optimal alignment score. Unlike other algorithms like NW, SW, and SWG that compute the whole matrices, WFA starts from the top left cell and moves towards the opposite corner in diagonal waves, avoiding computing all the cells in the matrix. A key concept in WFA is the one of Furthest Reaching Point (FRP), which, given a diagonal k and a score s, represents the cell in the DP matrix with score s, which is the most far off the beginning of diagonal k. To represent FRPs, for each of the three matrices M, I and D, WFA defines the offsets $M_{s,k}$, $I_{s,k}$, $D_{s,k}$ as the lengths of the diagonals that start from the top left cell and end in the corresponding FRP. From this definition we can also introduce the concept of wavefront: given a score s, the corresponding wavefront is the collection of the offsets having score s (M_s, I_s, D_s). In light of these new definitions, we can redefine the recursive equations as:

$$\begin{cases} I_{s,k} = 1 + max\{\ M_{s-o-e,k-1},\ I_{s-e,k-1}\ \} \\ D_{s,k} = max\{\ M_{s-o-e,k+1},\ D_{s-e,k+1}\ \} \\ M_{s,k} = max\{\ M_{s-x,k} + 1,\ I_{s,k},\ D_{s,k}\}\ \} \end{cases} \tag{2}$$

At this point, the computed points are extended following the matching characters along the diagonal. The series of wavefronts that store an FRP that reaches the end of the sequences having the smallest score s, is the optimal alignment score.

The WFA algorithm is divided into three main stages: *Initialization*, *Extend*, and *Next Wavefront*. The first phase sets all the wavefronts to a null offset and the scores to a null score. In the *Extend* stage, given the current computed score s, each offset $M_{s,k}$ is computed by comparing the characters of the pattern and the text shifted by k positions. If any of the calculated FRPs reaches the end of the sequences, then the algorithm stops. Otherwise, WFA proceeds with the *Next Wavefront* phase by starting the computation of the next wavefront with an increased score s. The algorithm computes the cells within the wavefronts of score s for each diagonal k, by following the recursive equations. These last two phases are repeated until one wavefront contains an FRP that reaches the end of the sequences. WFA algorithm has $O(s^2)$ spatial complexity and $O(ns)$ time complexity, which suggests that the more similar the sequences, the faster the execution time. For this reason, if we define the error rate as the ratio between the number of mismatching bases between two aligned sequences and their length, it can empirically be found that, with error rates higher than 30%, the score dominates the sequence length making the algorithms that compute the whole DP-matrix more efficient.

2.2 BiWFA

The main idea behind Bidirectional WaveFront Alignment (BiWFA) [10] algorithm is to perform WFA simultaneously in both directions on the sequences: from the start to the end (forward) and vice versa (reverse). The meeting point of the two executions is a breakpoint that stands around the middle of the sequences, dividing the optimal score roughly in half. Given the fact that BiWFA runs two parallel executions of WFA from the two ends of the sequences, we need to highlight the distinction between them by calling the forward and the reverse executions $\overrightarrow{W}_s = (\overrightarrow{M}_s, \overrightarrow{I}_s, \overrightarrow{D}_s)$ and $\overleftarrow{W}_s = (\overleftarrow{M}_s, \overleftarrow{I}_s, \overleftarrow{D}_s)$, respectively. The algorithm then proceeds by alternately computing forward and reverse alignments (i.e. $\overrightarrow{W}_1$, $\overleftarrow{W}_1$, $\overrightarrow{W}_2$, $\overleftarrow{W}_2$,$\overrightarrow{W}_3$, ...). This process iterates until the offsets overlap to compute the position of a breakpoint in the optimal alignment. There are a few technical details involved in the breakpoint search. First, it's not possible to split an alignment into an equally scoring prefix and suffix meaning that the optimal score is not always the sum of the two scores. This is because of the gap-affine score computation: if the two alignments meet in a gap, both the forward and the reverse iterations have already taken into account the gap-opening penalty. Another issue is that offsets of the two directions may not precisely meet, since the two iterations might pass close to each other without actually

meeting. However, it's possible to still detect when they meet, by checking if they are on the same antidiagonal and not just focusing on the exact cell.

It can be proven that the algorithm does not need to store all the computed wavefronts since, at a given position, we just need to access the previous w wavefronts, where $w = max\{x,\ o + e\}$. This means that both for the forward and the reverse iteration, we need to keep a buffer storing only the w previous computed wavefronts. In this way, the overall space complexity is $O(s)$, while the time complexity is $O((m + n)s)$.

3 State of the art

In the State of the Art, many tools use WFA as their primary aligner algorithm. AncestralClust [13] is a clustering method designed for batching divergent sequences which uses the WFA algorithm to pairwise align a set of reads before building the phylogenetic tree. Accel-Align [15] is a short-read mapper and aligner that exploits WFA to perform alignments to lower the execution time, but also to provide the CIGAR which is a string reporting a set of the alignment characteristics, like the number of matches, mismatches, deletions and insertions. All these software, when the sequence length exceeds a fixed threshold, switch to Minimap2's ksw library [8] to limit memory usage. In fact, as previously mentioned, WFA performance is limited by memory requirements when aligning long and noisy sequences.

Haghi et al. [6] proposed an FPGA implementation of WFA that has been tested against the state-of-the-art software solutions, reporting an 8.8x speed-up with one Xilinx Virtex UltraScale Plus XCUVU37P-2E FPGA and up to 13.5x using two of them. This solution is also more energy-efficient, being able to reduce the energy-to-solution by up to 9.7x and 14.6x, using one and two FPGAs, respectively. However, this design only supports a fixed read length and they used a 2-bit encoding of the sequences, limiting the possibility of processing sequences of any length and with a reduced alphabet (usually A, C, G, T). *Branchini et al.* [3] recently proposed an FPGA design without these limitations which also has a significant performance improvement, reaching up to 2876 Giga Cell Updates Per Second (GCUPS) with one Xilinx Alveo U250 FPGA board.

Regarding WFA implementations that run on GPUs, *Quim Aguado-Puig et al.* [1] introduced a CPU-GPU co-design of the WFA algorithm to perform inter-sequence and intra-sequence parallel sequence alignment. They managed to outperform the original multi-threaded WFA implementation by up to 4.3x and up to 18.2x when using heuristic methods on long and noisy sequences. *Gerometta et al.* [4] also proposed a GPU-based imple-

mentation that aims to accelerate the WFA algorithm by adopting a new alignment methodology that exploits homologous regions between the target sequences. With this solution, they achieved speedups of up to 14.81x with respect to the best hardware-accelerated solutions.

Due to the recentness of the algorithm, BiWFA does not have many hardware implementations yet. A first solution was proposed in May 2024 by *Alejandro Alonso-Marín et al.* [2] called BIMSA (Bidirectional In-Memory Sequence Alignment), which does not involve any heterogeneous architectures, but tries to accelerate this memory-bounded algorithm. To achieve that, they exploited Processing-In-Memory (PIM) which is an architectural paradigm that places computation mechanisms in or near where the data is stored (inside the memory or in the memory controllers) so that data movement between the computation units and memory is reduced or eliminated. BIMSA achieves a speedup of up to 22.24x compared to other PIM-enabled implementations of sequence alignment algorithms and up to 5.84x with respect to the CPU implementation of BiWFA.

Currently, we have a CPU implementation of BiWFA that is limited by two main factors: the first is memory latency which quickly becomes the bottleneck of the algorithm due to the high amount of memory operations made on the data structures during the computation of the final score. The second one is given by the fact that CPUs can suffer from overheads due to context switching when dealing with multiple threads, which can reduce performance for highly parallel tasks.

The other state-of-the-art implementation is based on PIM which tries to mitigate the first of the two issues stated before. PIM-based solutions, however, aren't actually real hardware-based implementations, so there is a lot of work to do in order to adapt classical architectures to PIM techniques. In the State of the Art, several different implementations try to solve many issues related to this design. Deploying PIM techniques requires, for instance, revising the mapping and address translation process of conventional memory systems, so *Olgun et al.* [12] proposed a custom supervisor software to handle it. To avoid this overhead, instead, *Lee et al.* [7] proposed reserving memory space dedicated to PIM operations during the booting process. Another issue was mitigated by proposing a modified version of the RISC-V GNU compiler, adding new instructions to manage cache blocks to keep coherence. These are just a couple of examples to show that PIM-based solutions are usually optimized for specific types of applications which limits both their versatility and their programmability. Finally, considering executing a large number of alignments on this design would mean integrating complex hardware components into memory modules, which can be costly in terms of development and resources.

For these reasons, we try to overcome these issues by proposing a GPU-based implementation that leverages GPUs, which are well suited for handling large-scale parallel tasks.

4 Implementation

The main point of this implementation is to exploit the different parallel operations of the BiWFA algorithm, maximizing, at the same time, the available resources on the GPUs. Before taking care of parallelism, we first created a lighter version of BiWFA by extracting only the key features we were interested in. BiWFA is a parametric software that can be run with many different configurations and optimizations. Some examples are the CIGAR, a final brief report of the alignment characteristics, and the backtrace, which contains the information to report the final alignment highlighting the positions of insertions, deletions, matches, and mismatches between the aligned sequences. Thanks to these simplifications, which, of course, still ensured the algorithm's correctness, we were able to drop some complex data structures, reducing the algorithm to its actual core functionalities. We then started focusing on parallelizing the execution of a single BiWFA instance, analyzing a single execution's steps. By doing this, we achieved intra-task parallelism, and we were able to schedule multiple GPU threads to make the execution of the $Initialization$ and $NextWavefront$ stages faster. Then we also tried to gain performance when running multiple sequence alignments, by exploiting inter-parallelism and running one GPU block per alignment.

The implementation is divided into two main parts: a preparation phase CPU-side and then the actual algorithm phase GPU-side. The first one takes care of reading the algorithm configuration parameters as well as the sequences to align. As stated before, the algorithm only needs to store a small number of wavefronts equal to $w = max\{x,\ o + e\}$, so thanks to this property, we just need to allocate w wavefronts for each matrix M, I and D and we need to do this twice, once for the forward iteration and once for the reverse one. A key point is also related to the choice of the parameter K, which represents the maximum length of each wavefront: when mismatching characters are found between pattern and text, the algorithm proceeds to increase the number of computed diagonals. The value of K sets a limit to the maximum number of diagonals computed during the execution of the algorithm. This choice is very important since if K is too small, the algorithm can't correctly compute the final optimal score, while if it's too high, we're wasting memory. This value depends on the sequence length and, most importantly, on the error rate which is related to the number of mismatching

characters that the algorithm will detect during the execution. CPU-side we sequentially allocate the patterns all together and also the texts to align. Given that the reverse iteration starts from the end of the sequences, we make a copy of the patterns and the texts and we reverse them. The same process is applied to allocate the wavefronts for the alignments so that we set six wavefronts per alignment: three for each matrix and we set them twice for the two iterations of the algorithm.

At this point, the second part of the algorithm is run and launched with as many blocks as the number of alignments, each composed of a fixed number of threads (32). The declared wavefronts, along with other flags used during the computation, are then set to a default value. Then, each block, based on its block ID, retrieves its pattern and text sequences from the ones declared CPU-side so that all the data structures are ready for computation. Now the core of BiWFA gets executed: two stages keep alternating one after the other until the forward and the reverse iterations overlap.

Algorithm 1 BiWFA algorithm

```
1 input: pattern p, text t
2 output: optimal alignment score
3
4 W→_{i,j}, W←_{i,j} = null;
5 W→_{0,0}, W←_{0,0} = 0;
6 k = len(p) − len(t);
7 max_antidiag = len(p) + len(t) − 1;
8 max→_{ak}, max←_{ak} = 0;
9 s→, s← = 0;
10
11 EXTEND(p, t, W→_s, max→_{ak});
12 EXTEND(p, t, W←_s, max←_{ak});
13 while true do
14      if (max→_{ak} + max←_{ak} >= max_antidiag) break;
15      NEXT_WAVEFRONT(p, t, s→, W→_s);
16      EXTEND(p, t, W→_s, max→_aks);
17
18      if (max→_{ak} + max←_{ak} >= max_antidiag) break;
19      NEXT_WAVEFRONT(p, t, s←, W←_s);
20      EXTEND(p, t, W←_s, max←_aks);
21
22 OVERLAP(p, t, s→, W→_s, s←, W←_s)
```

The first one is called *Extend* and basically compares patterns and text to look for matching characters. When this function is called, it takes the current wavefront with score s and loops through all diagonals k. An important thing to note here is that each wavefront is preallocated with all its available diagonals, but they're not all actually used: at first, a range of diagonals centered in the main diagonal ($k = 0$) is fixed and then, as the algorithm explores longer parts of the sequences, this range gets extended when new mismatching characters are found.

For this reason, the *Extend* function only considers the diagonals inside the current range. For each of them, it compares the pattern against the text sequence shifted by k characters. This comparison starts from the i-th character, both for the pattern and for the text, where i is the value of the offset related to the current diagonal of the wavefront with score s.

This phase is followed by the $Next\ Wavefront$ stage, which is in charge of computing the wavefront with the next score. The first operation is the extension of the diagonals range, delimited by the values lo and hi, which grows as the score increases. Then follows a sort of initialization phase where the three wavefronts of score s are set up: the offsets are reset to the default value in case we're using a wavefront we've already computed and the other flags and values are updated based on the computed values. Now the key operation starts: the offsets of the wavefront having score s are computed for the three matrices. This operation simply applies Equation (2) to update the wavefront values for matrix M, I, and D. The $Next\ Wavefront$ stage ends by updating the values related to each wavefront: lo and hi are the boundaries of the diagonals range, $null$ is a boolean value that indicates whether the wavefront is empty or not, while $wf\_elements\_init\_max$ and $wf\_elements\_init\_min$ keep track of the highest and lowest diagonal ever computed. When an overlap is found, the forward and reverse iterations stop. At this point, the last phase of the algorithm follows, the so-called *Overlap* stage. The algorithm loops through all the computed scores and checks whether the overlap happened in the M, I, or D matrix. Then, based on the type of overlap, the final optimal alignment score is computed.

After realizing a correct implementation of the algorithm, the next step was focused on exploiting the heterogeneous architecture of the GPU to improve the performance of the software.

Intra-task parallelization is the parallelization of a single alignment instance. The first stage of BiWFA is the $Initialization$ one, which handles the correct initialization of the data structures. In our case, we have three sets of wavefronts, one for each matrix (M, I, and D): each of them is rep-

resented by an array of sets of offsets. Each set contains K offsets and each wavefront is composed of w sets of offsets. These values have to be set, by default, to a null offset value. This operation has to be repeated twice: once for the forward iteration and once for the reverse one. Given that we don't have any relation or dependency between the offset values, we exploited the 32 available threads inside the block to independently perform this operation. Each thread updates the value of one cell, both for the three forward wavefronts and the reverse ones. In the $Initialization$ phase, we also initialize the values related to each wavefront that we previously mentioned (i.e., $null$, lo, hi), and that are useful to quickly access information about a wavefront's current state. This initialization is done in the same way as described above. The other stage we focused on is the $Next\ Wavefront$ one, where the wavefront with an increased score is computed. In this phase, the algorithm iterates on the diagonals and computes the elements within the s wavefront. Analyzing the execution of the loop that updates the three matrices, it's easy to see that a given iteration depends neither on the previous nor on the following iterations. For this reason, we exploited thread parallelism to execute the loop iterations all at the same time. Each one of the threads in the same block is assigned a diagonal k and is in charge of computing and updating the new offset values for diagonal k for the three matrices.

Inter-task parallelization consists of optimizing the execution of the BiWFA algorithm on multiple sequence alignments at the same time. To exploit this type of parallelism, we mapped one sequence alignment per GPU block, each of which is composed of 32 threads. By combining these two techniques, we schedule 32 threads that handle one alignment and we schedule multiple blocks of this type to run many of them simultaneously.

Lastly, we tried to exploit the GPU's shared memory to store the wavefronts which are the most frequently accessed data structures. In this version of our implementation, the first thing done by each block is to statically declare the wavefronts used for the computation of the final score in the GPU's shared memory. This choice comes from the need to reduce the access time to the memory since this is an operation performed a huge amount of times during the algorithm execution, but also because the data structures are shared among all the threads in the same GPU block that all refer to the same alignment. In this way, we can also increase data locality making the access to the wavefronts quicker.

Table 1: Execution times and GCUPS values of the GPU- and CPU-based implementations

Sequence length	Error rate	K	Execution time [s]	GCUPS	CPU-based implementation execution time [s]	CPU-based implementation GCUPS	Speedup
128	1	16	0.012	69.974	0.361	2.205	31.73x
512	1	16	0.024	529.976	0.589	22.244	23.83x
1024	1	32	0.053	985.777	1.069	49.024	20.11x
128	5	32	0.027	30.294	0.706	1.167	25.96x
512	5	64	0.114	114.927	2.998	4.373	26.28x
1024	5	128	0.339	154.431	8.793	5.962	25.90x
128	10	64	0.053	15.402	1.284	0.637	24.18x
512	10	128	0.321	40.803	8.894	1.473	23.70x
1024	10	256	1.238	42.323	29.101	1.801	23.50x

5 Results

This section covers the performance analysis of the previously described implementation. We evaluated and compared its performance against state-of-the-art solutions. The results were collected using an AMD Instinct MI210 Accelerator with HIP version 5.7.1 and ROCm version 1.1 and a dual-socket AMD EPYC 7V13 with 64 threads per socket. By exploiting the tool provided in the WFA2-lib, we generated many datasets, each of them having different sequence lengths and error rates. For each experiment, we measured the execution time only considering the kernel time and we reported the average execution time of 30 executions. We also computed the value of GCUPS (Giga Cell Updates Per Second) which is a measure of how many DP cells are updated per second during the algorithm execution. This value is computed using the following equation:

$$GCUPS = \frac{len(p) * len(t) * num\_alignments}{execution\_time * 10^9} \tag{3}$$

Even though the WFA and BiWFA algorithms do not require computing the whole DP matrices, we calculated the value of GCUPS as if all the matrices are updated, as done by *Haghi et al.*. [6] and *Branchini et al.*. [3]. We then used it to calculate the final speedup given by our implementation against the other ones, since it's a metric independent of the device and the underlying architecture of the different implementations. We carried out two different experiments, both aligning 50K couple of sequences. In Table 1, we considered datasets having sequence lengths of 128, 512, and 1024 bases and error rates of 1%, 5%, 10%. The value of K was tuned based on the combination of sequence length and error rate since it depends on how many diagonals the algorithm has to explore which is, in turn, proportional to the number

Table 2: Execution times and GCUPS values of the GPU-based implementation with and without shared memory usage

Sequence length	Error rate	K	Execution time [s]	GCUPS	GPU with shared memory execution time [s]	GPU with shared memory GCUPS	Speedup
128	1	16	0.012	69.974	0.012	70.534	0.99x
512	1	16	0.024	529.976	0.029	439.063	1.21x
1024	1	32	0.053	985.777	0.094	552.121	1.79x
128	5	32	0.027	30.294	0.042	19.184	1.58x
512	5	64	0.114	114.927	0.387	33.840	3.40x
1024	5	128	0.339	154.431	3.221	16.273	9.49x
128	10	64	0.053	15.402	0.163	5.002	3.08x
512	10	128	0.321	40.803	2.893	4.530	9.01x
1024	10	256	1.238	42.323	14.420	3.635	11.64x

of mismatches between pattern and text. The nine experiments were run both on the GPU implementation of BiWFA and on the lighter version of the WFA2-lib mentioned before where we extracted the main functionalities from those proposed in WFA2-lib, discarding many additional ones like backtrace or some heuristics applied to the algorithm. We also reduced to the bare minimum the number of data structures used to compute the final optimal alignment score to reduce memory usage. This lighter implementation faithfully reproduces what WFA2-lib performs. This CPU-based implementation was run with a total of 128 threads, given that the architecture has 64 threads per socket. The speedup of our implementation is computed as the ratio between the GPU-GCUPS and the CPU-GCUPS. Also, our implementation was run with as many blocks as the number of alignments, each block composed of 32 threads.

From the reported results and from Figure 1, we can see that our implementation brings a distinct improvement over the CPU-based solution. More specifically, the highest reached speedup amounts to 31.73x for the shortest sequence length, while the worst is related to longer alignments only achieving 20.11x. These results are in line with the issues of the CPU-based solutions mentioned before and with some design choices of our implementation. When aligning shorter sequences the GPU-based solution performs at its best since it can exploit the parallelism between different blocks. This is why the best speedups are achieved for a sequence length of 128 bases.

When considering higher error rates (5%, 10%) the speedup obtained with different sequence lengths levels off. Overall, we can state that these results are also in line with the intrinsic characteristics of WFA and BiWFA algorithms which perform at their best when run on short sequences with low error rates since they manage to significantly reduce the memory used by following the main diagonal of the DP-matrix. In fact, even a

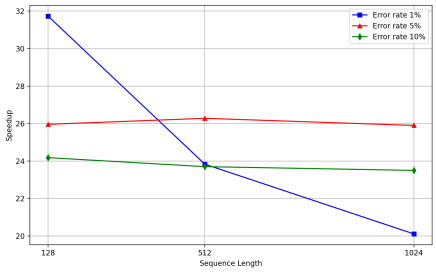

Figure 1: Speedups grouped by error rate

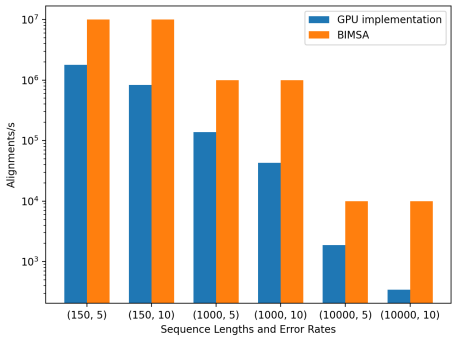

Figure 2: Alignments/s grouped by sequence length and error rate

hardware-accelerated solution like ours keeps being limited by the length of the sequences to align and can only bring quite limited speedups.

We repeated this set of tests to compare the performance of our implementation to the one exploiting shared memory. The results in Table 2 show that the advantage given by the data proximity fades as we align longer sequences that require higher values of K. We exploited shared memory to reduce the wavefronts access time, so this version of our implementation should get better performances when aligning sequences with low optimal scores (high scores in absolute value). In order to get to low optimal scores, we need to work with high error rates, which leads to having higher values of K. So, this version would outperform the base one when frequently accessing the computed wavefronts, but this situation is only possible for high values of K which means that the space in shared memory occupied by the GPU blocks grows, reducing the maximum number of blocks that can be executed at the same time. For these reasons, inter-parallelism is reduced affecting the overall performance of this version of our implementation.

The other set of tests was run to compare our implementation against the BIMSA design. The comparison is based on the number of alignments performed per second, as *Alejandro Alonso-Marín et al.* reported in [2]. The experiments test the performance of our implementation with longer sequence lengths than before, going from 150 to 10K bases. As expected, BiWFA does not scale well when aligning long sequences with high error rates. We can see in Table 3 that a sequence length of 10K bases with an error rate of 10% achieves the lowest result in terms of alignments per second, also reporting a quite high execution time compared to the other experiments. The best results are achieved on short sequences reaching up to 1.8M alignments per second. At the same time, as the length increases, the number of alignments

Table 3: Alignments/s of the GPU-based implementation and the design

Sequence length	**Error rate**	**Execution time** [s]	**Alignments/s**	**Alignments/s** [2]
150	5	0.028	1785714	10000000
150	10	0.060	833333	10000000
1000	5	0.360	138889	1000000
1000	10	1.169	42772	1000000
10000	5	26.727	1871	10000
10000	10	146.080	342	10000

per second drops, reaching the minimum value of 342 when aligning 10K bases long sequences with an error rate of 10%.

As we can see in Figure 2, our implementation achieves slightly worse results, and it is outperformed by the BIMSA implementation by one order of magnitude across all sequence lengths. The worse results are achieved when aligning longer sequences (i.e. 10K bases). These results are in line with the current state of our implementation since there are still some improvements to do in order to gain in performance. For example, the *Extend* function does not properly exploit thread parallelism when comparing the sequences, character by character. Also, in the $NextWavefront$ function, we could let the two ends proceed on their own until they reach roughly the middle of the sequences where the overlap is more likely to be found to then proceed with the *Overlap* phase. Given that the BIMSA implementation does not have that much room for improvement, we can say that our implementation, when improved with the changes just described, will easily catch up with the performance of BIMSA and also outperform it.

6 Conclusions and Future Work

This work presents a GPU-optimized design of the Bidirectional WaveFront Alignment algorithm. The experimental results highlight that our implementation achieves speedups ranging from 20.11x to 31.73x against the software version. This design, though, gets outperformed by the current state-of-the-art PIM-based solution, even though the performance gap between the two could be easily filled by implementing some improvements to our tool. Overall, our solution outperforms the CPU-based implementation of BiWFA by exploiting thread-level parallelism and limiting the memory access issue that characterizes the software version.

In light of the results of this analysis, our implementation can be upgraded by adding further improvements to some key phases of the algorithm, like the *Extend* function, but also the management of the forward and reverse

wavefronts.

References

[1] Quim Aguado-Puig et al. “WFA-GPU: gap-affine pairwise read-alignment using GPUs”. In: *Bioinformatics* 39.12 (Nov. 2023), btad701.

[2] Alejandro Alonso-Marín et al. “BIMSA: Accelerating Long Sequence Alignment Using Processing-In-Memory”. In: *bioRxiv* (2024).

[3] B. Branchini et al. “Surfing the wavefront of genome alignment”. In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2022, pp. 1754–1758.

[4] G. Gerometta, A. Zeni, and M. D. Santambrogio. “TSUNAMI: A GPU Implementation of the WFA Algorithm”. In: (Oct. 2023), pp. 150–161.

[5] Osamu Gotoh. “An improved algorithm for matching biological sequences”. In: *Journal of Molecular Biology* 162.3 (1982), pp. 705–708.

[6] A. Haghi et al. “An FPGA accelerator of the wavefront algorithm for genomics pairwise alignment”. In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 151–159.

[7] S. Lee et al. “Hardware architecture and software stack for PIM based on commercial DRAM technology : Industrial product”. In: (2021), pp. 43–56.

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

[9] Santiago Marco-Sola et al. “Fast gap-affine pairwise alignment using the wavefront algorithm”. In: *Bioinformatics* 37.4 (Sept. 2020), pp. 456–463.

[10] Santiago Marco-Sola et al. “Optimal gap-affine alignment in O(s) space”. In: *Bioinformatics* 39.2 (Feb. 2023), btad074.

[11] 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.

[12] A. Olgun et al. “PiDRAM: A holistic end-to-end FPGA-based framework for processing-in-DRAM”. In: (2021).

[13] L. Pipes and R. Nielsen. “Ancestralclust: clustering of divergent nucleotide sequences by ancestral sequence reconstruction using phylogenetic trees”. In: *Bioinformatics* 38.3 (2022), pp. 663–670.

[14] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197.

[15] Y. Yan, N. Chaturvedi, and R. Appuswamy. “Accel-align: a fast sequence mapper and aligner based on the seed–embed–extend method”. In: *BMC Bioinformatics* 22.1 (2021), pp. 1–20.