Parallel Computing 116 (2023) 103019

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

GPU acceleration of Levenshtein distance computation between long strings

David Castells-Rufas [*]

Department of Microelectronics & Electronic Systems, Universitat Autònoma de Barcelona, Cerdanyola del Vallés, 08193, Spain
Barcelona Supercomputing Center, Barcelona, 08034, Spain

ARTICLE INFO

Keywords:
Levenshtein distance
Edit distance
WFA algorithm
GPU
Parallel processing

ABSTRACT

Computing edit distance for very long strings has been hampered by quadratic time complexity with respect to string length. The WFA algorithm reduces the time complexity to a quadratic factor with respect to the edit distance between the strings. This work presents a GPU implementation of the WFA algorithm and a new optimization that can halve the elements to be computed, providing additional performance gains. The implementation allows to address the computation of the edit distance between strings having hundreds of millions of characters. The performance of the algorithm depends on the similarity between the strings. For strings longer than million characters, the performance is the best ever reported, which is above TCUPS for strings with similarities greater than 70% and above one hundred TCUPS for 99.9% similarity.

1. Introduction

The Levenshtein distance [1] measures the number of edit operations (insertion, deletion, and substitution) required to convert a string P into a string T. Levenshtein distance is also known as edit distance (ED). It is used in several computing domains, including text analysis [2], music analysis [3], and bioinformatics [4]. Computer Science uses the concepts of time complexity and space complexity [5] to be able to compare algorithms. These concepts refer to the relation between the problem size (n) and the execution time and the required memory, respectively. Wagner–Fischer (WF) [6] proposed a dynamic programming (DP) algorithm to compute ED with an $O(m \times n)$ time and space complexity, where $m = |P|$ and $n = |T|$ (i.e. the lengths of the strings). The space complexity is given by the need to store the DP table to be able to back-trace the alignment sequence between the two strings. If only the distance score is required, time complexity remains the same but space complexity can be reduced to $O(m)$.

The Wavefront Alignment Algorithm (WFA) is a DP technique that reduces the time and space complexity to $O\left(d^2\right)$, where d is the edit distance between both strings. WFA was originally proposed to compute Smith-Waterman (SW) gap-affine global alignment [7] but it can be adapted to compute other distance or similarity scores based on computing a DP table. In [8], it was adapted to compute Levenshtein distance for limited strings lengths up to a thousand characters. The problem of using DP algorithms on very long strings has historically been hindered by the difficulty of managing the high memory demand and excessive computing time. This paper proposes several optimizations to compute the Levenshtein distance between very long strings using the WFA algorithm with $O(d)$ space complexity on Graphical Processing Units (GPUs) for fast execution times and limited memory use.

This paper is organized as follows. In Section 2 we analyze previous work. In Section 3 we analyze the fundamentals of the WFA algorithm and its adaptation to the computation of Levenshtein distance. The adaptation to GPU architectures is presented in Section 4, where we follow an iterative improvement process to get the maximum performance. Finally, we analyze the results and contrast them with other works from the state of the art in Section 5 before concluding.

2. Previous work

The classic WF algorithm is based on building a DP table D using Eq. (1).

$$D_{i,j} = \begin{cases} i & \text{, if } j = 0 \\ j & \text{, if } i = 0 \\ \min \begin{cases} D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \\ D_{i-1,j-1} + [P[i-1] \neq T[j-1]] \end{cases} & \text{, otherwise} \end{cases} \tag{1}$$

The ED between P and T is determined by the cell $D_{m,n}$ from the table. This algorithm can be considered a brute-force approach where all possible edit operations are considered but it uses memoization to avoid repeating the same computing for several results. Eq. (1) can be found in the literature in different ways. In our case, we use the Iverson

[*] Correspondence to: Department of Microelectronics & Electronic Systems, Universitat Autònoma de Barcelona, Cerdanyola del Vallés, 08193, Spain.
E-mail address: david.castells@uab.cat.

https://doi.org/10.1016/j.parco.2023.103019
Received 4 October 2022; Received in revised form 16 March 2023; Accepted 30 March 2023
Available online 3 April 2023

bracket notation for the extra value when characters do not match. In Iverson bracket notation [9], if the expression inside the brackets is true the value is considered 1, and zero otherwise.

The ED is not the only interesting result that can be obtained with the WF algorithm. Many applications recover the sequence of operations required to transform P into T by analyzing the elements of the D table. This sequence can be represented as an alignment string, where M is used to denote a *match*, and I,D, and X are used to denote *insert, delete*, and *substitution* operations respectively. There can be multiple valid alignment strings, as different edit operations can result into the same ED. The following example illustrates one alignment string obtained from two short strings,

```
P:        -ACCATGGACTG-
           ||| || |||
T:        CACC-TG-ACTTA
alignment: IMMMDMMDMMMXI
distance:  5
```

The backtrace method to obtain the alignment string is based on recovering the active branches of the conditional statements of Eq. (1). When dealing with very long strings, the cost of storing the table is prohibitive, and piece-based strategies (such as the one proposed by Hirschberg [10]) are required to be able to obtain an alignment string. This work focuses on obtaining the distance between the input strings. Obtaining the alignment string is left for future works.

There have been many attempts to improve the performance of matrix-based DP algorithms. The DP matrix offers a very regular layout that seems adequate for a parallel implementation at a first glance. However, the conditional branches appearing in Eq. (1) imply an irregular data dependency among the cells, and makes the contribution of many of the computed cells irrelevant to the final result. Hence, the methods to speed up the algorithm are focused on two aspects: (1) speeding up the process computation of the cells. (2) avoiding computing some of the cells of the DP matrix.

In the first group, parallel processing is usually applied. The existing data dependency requires to use of an anti-diagonal approach which is hard to parallelize since multiple threads receive an irregular workload. On the other hand, other approaches target the speedup of the single-threaded execution. The BPM Myers algorithm [11] is a remarkable example. It uses bit-vectors and lookup tables to reduce the computational load by a factor related to the word length of the computing architecture. In the second group, Ukkonen [12] proposed a banded approach, which was later combined with Myers' bit vectors by Hyyrö [13].

These two different strategies make comparing the performance of different works tricky. We know that DP algorithms' time complexity is $O(m \times n)$. Hence, many works report the number of computed cells from the DP table in cell updates per second (CUPS), which are defined as Eq. (2).

$$\text{CUPS} = \frac{m \times n}{t} \tag{2}$$

However, when using algorithms from the second group, the CUPS unit could be misleading as they do not require computing all cells but a subset of them. Nevertheless, we think that CUPS are still valid as a throughput measure as it maintains an inverse relation (i.e. $1/x$) with execution time.

The Edlib library [4] is a popular implementation of the BPM algorithm, which exploits the intrinsic parallelism of the add instruction in carry propagation to compute several cells from the table in a single instruction execution. With the end of Denard's scaling regime, computing platforms have been forced to embrace parallel architectures. The BGSA [14] is a remarkable extension to benefit from the vectorization and multi-threading opportunities available on modern CPUs to get performance above 100 GCUPS.

On the other hand, GPUs offer a massively parallel execution platform. DP algorithms have been adapted to GPUs in search of scalability. In [15], they reached a significant performance speed (above 1 TCUPS) using the Myers BPM algorithm on strings up to 1k characters. Longer sequences (8k and 22.5k) were targeted by Balhalf in [16,17] using an anti-diagonal parallel approach with modest throughput results under 1 GCUPS. Tiling [18] is a general strategy to increase data locality in GPU workloads. It is used in [19,20], and [21] to address longer strings of up to 2 M with a peak performance of 50 GCUPS. FPGAs also offer a good implementation platform for applications with a regular data layout. Our recent work [22] achieves more than 5 TCUPS for short genomics strings (<300 characters). Computer clusters have not been generally used to speed up this problem. Sadiq and Yousaf [23] is one of the few works trying to parallelize ED for clusters with modest results.

The low research attention given to ED between long strings contrast with the enormous interest of other related DP algorithms such as SW [24] and their variants. The reason for this is its superlative impact in bioinformatics, especially genomics, which always demands faster methods to compare long DNA strings. The advances in this algorithm are making possible genome-wide string comparisons. The human genome length is approximately 3 G. Many parallel strategies have been studied to compare at that scale [25].

Some of the most relevant results on accelerating SW for long strings with GPUs have been achieved by the several versions of the CUDAlign library [26–29]. Their goal is to recover the full alignment. Their approach is based on sampling some of the rows during DP table computation to later refine the alignment in a piece-wise approach using massive parallelism. A large part of the execution time is devoted to the first phase of the algorithm, which is the building of the DP table. Its complexity is comparable to computing the ED. Thus, we will use datasets from SW works to compare the execution time of our ED solution.

3. WFA algorithm

The wavefront algorithm (WFA) [7] belongs to the second group of algorithms, which avoid to compute cells to speed-up the computation of the DP table. WFA only computes the cells from the diagonals of the DP table where the distance is incremented. As the algorithm is based on diagonals, it is convenient to define a coordinate system that refers to the elements of the matrix D in terms of diagonals and displacements within the diagonals.

Several coordinate systems could be used to identify the cells of the matrix as a displacement within the diagonals. In Fig. 1 we show the four coordinate systems we foresee: offset, radius, row, and column. We can describe the D table using any of these coordinate systems. In the original Marco-Sola's paper [7] a column coordinate system is used. In the rest of this paper we will use a radius coordinate system, which might be more intuitively understood by the reader. Using a different coordinate system has an impact on the definition of the equations used by the algorithm, which we will introduce below. However, they can be easily adapted to all coordinate systems.

The transformation between matrix (Cartesian) D and pyramid (radius) $\hat{D}$ coordinate systems is described with Eqs. (3) and (4), which also use the Iverson bracket notation. An example D table with its corresponding $\hat{D}$ pyramid are shown in Figs. 2(a) 2(b) respectively.

$$D_{i,j} = \hat{D}_{j-i,[i>j]j+[i\leq j]i} \tag{3}$$

$$\hat{D}_{k,r} = D_{[k<0](-k)+r,[k\geq 0](k)+r} \tag{4}$$

We define the wavefront pyramid $\widehat{W}$ as the pyramid that contains in $\widehat{W}_{k,r}$ the farthest displacement in the diagonal k of the original

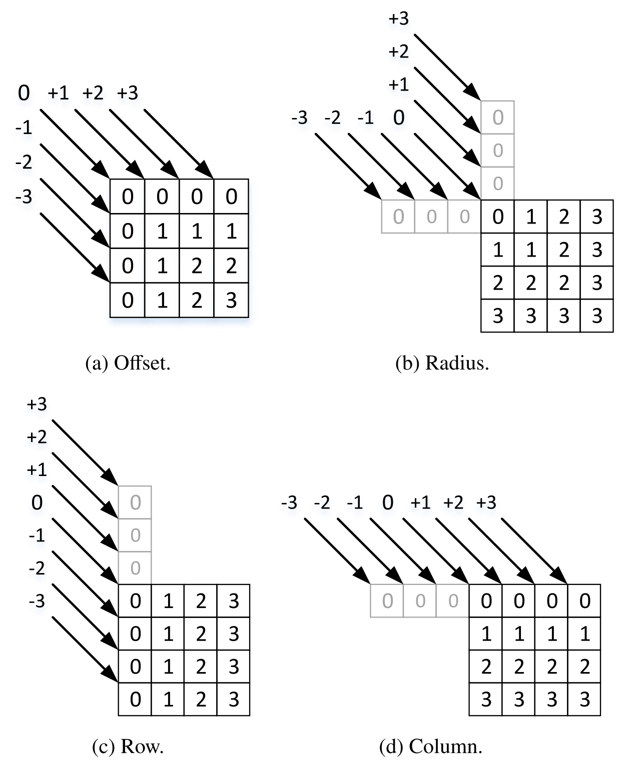

(a) Offset. (b) Radius. (c) Row. (d) Column.

Fig. 1. Possible diagonal coordinate systems using an index to identify the diagonal and another index to identify the displacement within the diagonal. In the figures, the arrows define the diagonal coordinates, and the values on the boxes refer to the displacement value within the diagonal.

$\hat{D}$ pyramid having ED r. $\widehat{W}$ can be formally derived from $\hat{D}$ using Eq. (5).

$$\widehat{W}_{k,r} = \max_{\forall q} q \mid \hat{D}_{k,q} = r \tag{5}$$

The great advantage of the WFA algorithm is that it finds an alternative way of building the $\widehat{W}$ pyramid without computing $\hat{D}$. Instead, it uses two additional pyramids $\hat{C}$ and $\hat{E}$ as described in Eq. (6). These pyramids are described below.

$$\widehat{W}_{k,r} = \hat{C}_{k,r} + \hat{E}_{k,\hat{C}_{k,r}} \tag{6}$$

The *compute* pyramid $\hat{C}$ is built by selecting the maximum values from the depending cells of the $\widehat{W}$ pyramid as described in Eq. (7).

$$C_{k,r} = \begin{cases} 0 & \text{, if } k = 0 \wedge r = 0 \\ \max \begin{cases} W_{k+1,r-1} + [k < 0] \\ W_{k,r-1} + 1 \\ W_{k-1,r-1} + [k > 0] \end{cases} & \text{, otherwise} \end{cases} \tag{7}$$

The *extend* pyramid $\hat{E}$ is the pyramidal form of the *extend* table E as defined in Eq. (8), which contains the length of the longest common prefixes of P and T starting at positions i and j respectively.

$$E_{i,j} = LCP(P_i, T_j) \tag{8}$$

In practice, only a small subset of values from $\hat{E}$ are usually needed as the radius of the accessed elements depend on the values of the previously computed elements of $\hat{C}$ pyramid as shown in Eq. (6). Moreover, the data dependency allows to avoid storing the pyramids $\hat{C}$, $\hat{E}$, which are computed as needed. Fig. 2 depicts an example of a D matrix, its pyramidal form $\hat{D}$, and the wavefront pyramid $\widehat{W}$.

3.1. Algorithm optimizations derived from data dependency

In the WF algorithm, the solution to the ED problem is found in $D_{m,n}$. However, in WFA, we can only ensure that the solution is in the diagonal $k_{sol} = n-m$, and that its radius is $r_{sol} \leq max(m,n)$. From Eq. (7), it is obvious that $\hat{C}_{k_{sol},r}$ only depends on the neighbors of the cell in previous radius (i.e. $\hat{C}_{k_{sol}-1,r-1}$, $\hat{C}_{k_{sol},r-1}$, and $\hat{C}_{k_{sol}+1,r-1}$). These cells, in turn, will depend on their neighbors from the previous radius, and so forth.

Therefore, in the worst case, the only required cells to compute the $\hat{C}$ pyramid, and hence, the $\widehat{W}$ pyramid, will be the cells inscribed inside the diamond shaped parallelogram described by Eq. (9).

$$\begin{gathered} r \in [0, r_{sol_0}] \\ k \in [min(r, k_{sol_0} - (r_{sol_0} - r)), min(r, k_{sol} + (r_{sol_0} - r))] \end{gathered} \tag{9}$$

Unkkonnen and later Myers [11] and Hyyrö [13] exploited the fact that, in ED, the increments on the diagonals can only be 0 or +1. In the worst case (having no match between P and T), the values of the solution diagonal in both $\hat{D}$ and $\widehat{W}$ will increase from 0 to r_{sol} in +1 increments. In case of a single match in the diagonal, all increments will be +1 except a single value being zero, and then, the resulting ED will be $r_{sol} - 1$. After this observation, we can generally state that after computing a value of the solution diagonal, $f = \widehat{W}_{k_{sol},r}$, the new worst case radius must be $r_{sol_r} \leq r_{sol_0} - (f - r)$.

We can use the new upper bound to dynamically define the range of cells to compute from the $\widehat{W}$ pyramid as the algorithm progresses along the pyramid columns. This approach, that we name dynamic diamond, reduces even more the number of the required cells to compute.

In the example illustrated in Fig. 2, 36 cells from $\widehat{W}$ are computed to get the final result using the classic WFA algorithm illustrated by Fig. c. With the Dynamic Diamond approach illustrated by Fig. d, we require to compute 24 cells, which represents a factor 1.5 × of savings. Actually, the savings of this approach will be between 1× and 2×, being higher as the distance between P and T is higher.

If the alignment path is not required, we can reduce the storage to just two columns from the $\widehat{W}$ pyramid, as the whole pyramid is only required to perform the backtrace. In this case, when computing a column, only the information from the previous column as can be observed by the data dependency pattern. We will have to dimension the column as the worst case, which corresponds to the final column of the pyramid containing $m + n + 1$ elements.

4. GPU implementation

GPUs offer a massively parallel execution platform based on a large number of processing elements (PEs) and a multi-ported memory hierarchies with high memory bandwidth. One difference of GPU architectures with respect to general purpose chip multiprocessors (CMP) is that, at a given time, a large number (if not all) PEs execute the same program (called kernel), which is typically small in size. By doing this, instruction fetch can be shared among a large number of PEs in a SIMD fashion. To maximize the efficiency of instruction execution among the PEs, control flow divergence is minimized with several techniques, such as predicated conditionals, stack memory avoidance, low latency context switching, and large number of in-flight threads and memory access latency hiding techniques.

Several programming frameworks can be used to program GPUs. NVIDIA favors CUDA and OpenACC but it also supports OpenCL, which offers a wider platform support for other accelerator technologies like FPGAs. Our implementation is mainly based on OpenCL, although we also code some variants in CUDA because of the richer available profiling tools. Before going further into the implementation, we should introduce some GPU programming concepts.

The GPU has a global memory external to the chip. This is usually a multi-ported big memory implemented in several banks of some of the

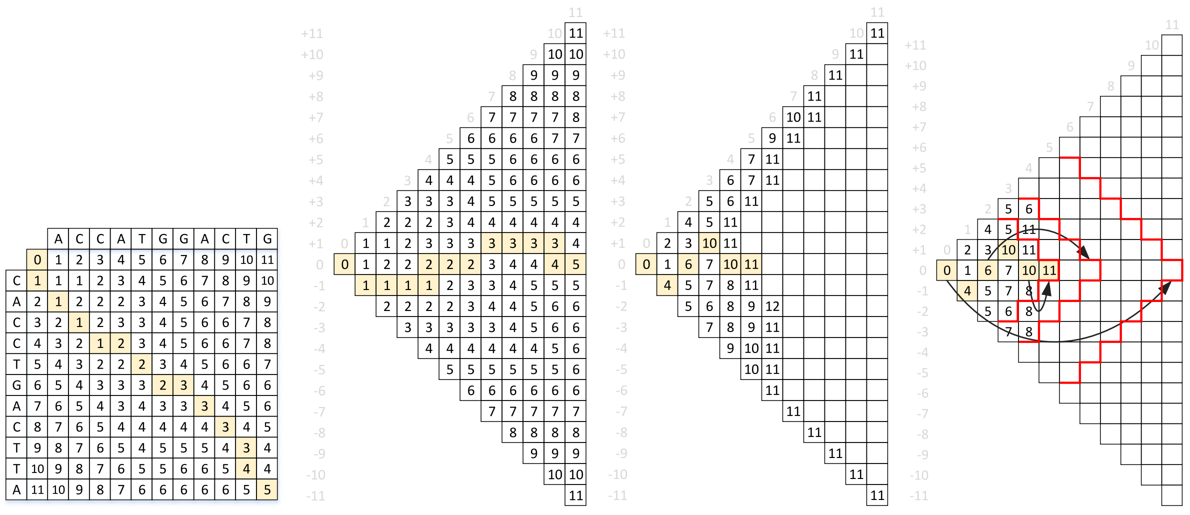


Fig. 2. Visual example of how the classic DP approach is transformed into the WFA approach and the dynamic diamond optimization presented in this paper. (a) DP 12 × 12 table D to compute the distance between the example strings P = "CACCTGACTTA" and T = "ACCATGGACTG". (b) Pyramidal form $\widehat{D}$ of the DP table in radius coordinates. (c) Pyramidal form of the WFA algorithm. (d) Dynamic-diamond optimization presented in 4.3 to reduce the number of computed cells. The arrows represent the farthest pyramid column that the algorithm considers. As the algorithm computes more columns from left to right, the furthest pyramid column is reduced and less cells must be computed thanks to the diamond shaped data dependency.

Table 1
Equivalence among different terminology in OpenCL and CUDA.

Concept	OpenCL	CUDA
Processor	Compute unit	Streaming multiprocessor
Workload unit	Workitem	Thread
Group of workload units executed in the same processor	Workgroup	Thread block
Batch of workload units computed in a kernel invocation	NDRange	Grid

latest DRAM technologies. "Single Instruction Multiple Data" (SIMD) processing elements (PEs) are grouped into computing units (CU). NVIDIA calls them Streaming Multi-processors (SM). PEs in the same CU can be synchronized by means of specific synchronization primitives and they can share some on-chip memory. When invoking kernels in a GPU device, the host specifies the number of workitems required to execute the workload. A workitem is a unit of work associated to an invocation of the kernel with a unique identifier that can be a 1D, 2D, or 3D index. Workgroups are groups of workitems, and they can be used to ensure that a range of workitems are executed in the same CU. A scheduler distributes the workitems among the available PEs. Once a workitem starts executing in a PE it can be context-switched, but it generally will not be retired from the CU until it has completed its execution. Contrary to what happens in a normal preemptive multitasking system, there might be workitems (threads) not starting until there are available PEs for execution. This behavior prevents from having global synchronization primitives among all workitems participating in a kernel invocation. Global synchronization can only occur by host kernel invocations, as the host can ensure that the whole batch of workitems launched are completed and cache memories are flushed. The CUDA framework from NVIDIA uses a different terminology, but the equivalence is shown in Table 1.

As stated in the previous section, WFA space complexity can be reduced by just storing two columns of the pyramid: the currently being computed and the previous one. An easy GPU parallelization strategy is to devote one workitem to every cell of the pyramid column currently computed. As depicted in Fig. 3, each workitem computing a cell must access three cells values (b, c, d) from the previous column to obtain the $\widehat{C}_{d,r}$ value and must access one or several values from the P and T strings to finally assign the a value to the $\widehat{W}_{d,r}$ pyramid.

The probability to have a run of u consecutive matches in the substrings from P and T during the extension phase can be approximated

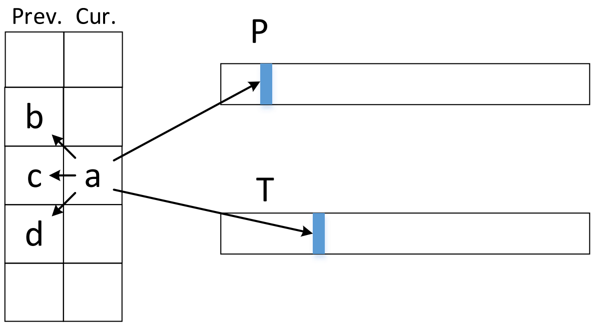


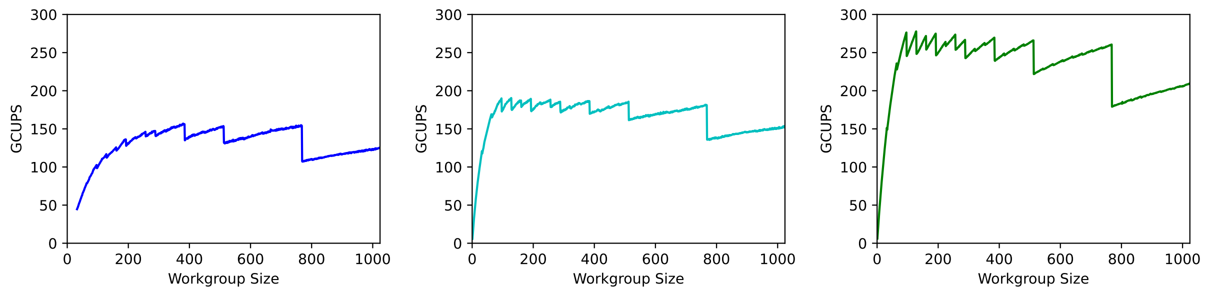
Fig. 3. Data dependency to compute a cell (a) from the $\widehat{W}$ pyramid. For each cell, three cells from the previous column are accessed (b,c,d), as expressed in Eq. (7). One or more characters from the P and T strings are also accessed to compute the LCP (corresponding to Eq. (8)).

by Eq. (10), where Σ is the alphabet of the strings being compared.

$$P(\text{run } u) \approx \left(\frac{1}{|\Sigma|}\right)^u \tag{10}$$

Thus, in practice, the majority of accesses will require very few iterations during the extension phase, which will result in a low divergence between workitems. Moreover, we expect a large number of cache hits as workitems access nearby positions.

When working with short strings, the workload is relatively small compared to the large number of processing elements (PEs) available on a GPU, resulting in low GPU occupancy. To address this, it is common practice to group a large number of short string pairs and process them in parallel, as described in [8]. However, when dealing with large strings, even with a pyramidal data structure, the amount of work



(a) Baseline. (b) Dynamic column size. (c) Minimization of reads.

Fig. 4. Performance achieved in NVIDIA GeForce RTX 3090 with different kernel invocation strategies as a function of the workgroup size. GCUPS are computed as defined in Eq. (2). (a) Baseline. (b) Adapting the number of workitems as the algorithm progresses along the columns has a beneficial impact in performance. (c) However, the biggest additional benefit is achieved when we reduce the frequency at which the host reads the memory positions that indicate the processing is complete.

required to compute a single column can far exceed the available PEs after the initial iterations.

Several strategies can be used trying to maximize the throughput of the system. In the following subsections we will iteratively analyze some implementation alternatives and measure their impact on the system performance. To demonstrate these approaches, we will use two files from the National Center for Biotechnology Information (NCBI), with each file being over 3 million characters long, corresponding to accessions BA000035.2 and BX927147.1. The ED between these files exceeds 1.5 million, providing an opportunity to achieve significant performance improvements. We will report performance in GCUPS.

4.1. Kernel invocation strategies

Memory requirements are some of the fundamental aspects that influence the performance of an implementation. In the worst case scenario of the strings being completely different (i.e. not a single match), the size of the required pyramid is the same than the DP table, i.e. n^2 for same-length strings. Therefore, the supported maximum length of same-length strings is determined by the expression $n_{max} < \sqrt{M_{GPU}/w}$, where M_{GPU} is the global memory size of the GPU and w is the size of the data type used to store the pyramid values (both measured in bytes). In our case, we use an NVIDIA RTX 3090 with $M_{RTX3090} = 24$ GB and a $w = 4$ B, so $n_{max-RTX3090} < 77$ k. Therefore, using the standard WFA algorithm, the pyramid for longer strings would not fit into the GPU memory. For longer strings, we have to use the two column computation optimization presented at the end of Section 3.1. In the general case, we are not restricted to same-length strings. We allocate space in global memory to store two columns of length $m+n+1$, which corresponds to the maximum possible number of rows of the pyramid. Note that we generally use a fraction of the cells of the two columns, since the full column will be completely used in the right-most column of the pyramid. Although dynamically allocating memory required by each column as the algorithm progresses may be considered as a space-saving measure, it is counterproductive due to the requirement for the host to coordinate the memory allocation process, which is known to be slow. With the fixed size strategy, the number of cells required for same-length strings is $2(2n+1)$. So, in this case the maximum supported string length would be determined by the expression $n_{max} < M_{GPU}/4w$. Hence, $n_{max-RTX3090} < 1.5$ G. 32 bit numbers are sufficient to store the pyramid values as ED cannot be bigger than the string length. Ignoring the amount of memory of the GPU, our implementation does not support strings longer than 2^{32}. Longer strings would require a piece-wise strategy and the use a bigger integer types.

4.1.1. Baseline

In a first baseline version, we implement a kernel in which each workitem computes only one cell from the column. Every kernel invocation computes a column from the pyramid. The algorithm has a dependency between columns, so we must wait until a whole column is computed to start the next one. We use kernel invocation as a global synchronization method. For every column, we invoke the kernel with an NDRange to compute all the possible cells of the column, which correspond to the worst case ($m + n + 1$ elements). Each workitem determines whether the cell has to be executed or not. In the later case, it finishes early.

Although there is no local memory sharing or fine-grain synchronization among workitems, the workgroup size has an influence in the execution time, since the GPU benefits from having multiple workitems executing the same instruction in a SIMD fashion and workitems are scheduled in multiples of the SIMD lanes of the GPU. Fig. 4(a) shows the achieved performance as a function of the workgroup size. The workgroup size is known to influence performance and exhibit a zig-zag pattern [30]. The fluctuations are explained by changes in the amount of used hardware and occupancy of the SIMD units. The figure considers the total execution time of the application, not just a single kernel invocation. The maximum performance (156 GCUPS) is achieved with a workgroup size of 380 workitems, and corresponds to an execution time of 65.8 s. In this strategy, the host invokes the kernel as many times as the number of required columns from the $\widehat{W}$ pyramid.

We analyze how the kernel execution varies as the algorithm progresses in column radius. The blue line in Fig. 5 depicts the execution time of kernel invocations. Although invocations are launched with the worst case number of workitems (the rows from the pyramid column), early detection of useless computation in the OpenCL kernel code will make many threads to exit in few instructions. The x axis of the figure represents the radius of the column in the kernel invocation. For low values, the execution time remains constant and many workitems exit by checking early exit conditionals. At a certain threshold the number of useful workitems are involved in slightly more complex code that increases the overall execution time. As the cells invocations reach the end point many workitems from distant diagonals will already reach early exit conditions making a decrease on the overall execution time of those kernel invocations. This approach obviously has an overhead, since it launches more workitems than are really necessary.

4.1.2. Dynamic number of workitems

In a second implementation we invoke the kernel with a dynamic number of workitems in the NDRange. In the first invocation it just contains 1 cell. In the second invocation 3, and the n column, $2n + 1$. In the previous approach it seems a waste of resources to have most of the $m + n + 1$ workitems working in useless computations. With this change, the test-case execution time is reduced from 65.85 to 54.24 s, which represents 190 GCUPS and a speedup factor of almost 1.21× with respect to the baseline.

We could expect that the execution time of the baseline is proportional to $d \times (m + n + 1)$, where d is the ED between P and T. We also could expect that the dynamic workitems improvement execution time is proportional to $(d + 1)^2$. Using this estimations we compute the expected speedup factor from our files, which is 4.2 ×. But, in reality,

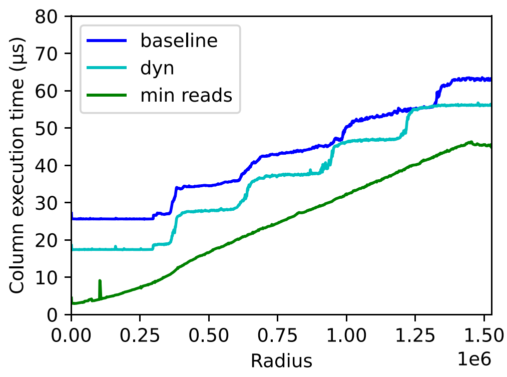
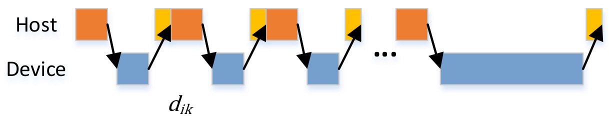
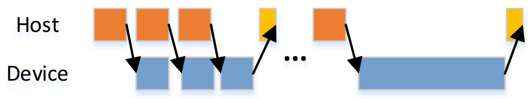


Fig. 5. Kernel execution time to compute $\widehat{W}$'s columns on the motivating example as a function of column number (radius) for 3 different kernel invocation strategies. Baseline and dynamic strategies suffer a constant overhead caused by the additional time used doing unnecessary memory transfers to check for the final condition. In addition, the baseline shows a greater execution time than the dynamic version because it always creates more workitems than the other.



(a) Basic invocation method.



(b) "min reads" optimization.

Fig. 6. Conceptual time-line of the interaction between host and OpenCL kernel depending on the number of final condition checks. Blue areas depict kernel execution in the GPU. Red areas depict the host code manages kernel invocation, while orange areas depict the host code that determines the final condition of the algorithm. Arrows depict memory transfers in both directions (host-to-device, and device-to-host). (a) Baseline, where final condition is checked after every kernel invocation. (b) Reducing the number of final condition checks reduces the number of memory transactions and eliminates the delay between successive kernel executions d_{ik}.

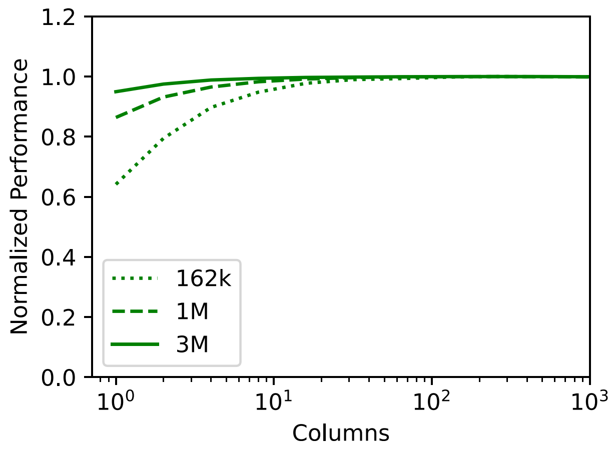


Fig. 7. Effect of reducing the number of read transactions on performance for various string lengths. Reducing the number of read transactions per kernel invocation below a factor 1/100 does not provides any significant additional benefit for all string lengths.

we observe that execution time is not exactly linear with the number of workitems as radius progresses (see cyan line in Fig. 5). What happens is that read operations dominate over kernel execution slowing the overall execution.

However, we can see that, in the worst execution time regime, for the same number of active cells the baseline kernel still takes more time to execute. This is because it always launches the worst case distance number of workitems, while the dynamic always launches less.

4.1.3. *Minimizing reads*

To check if a solution is found, the host must check if the furthest reaching radius computed by the kernel on the solution diagonal already reached the final point. This requires to perform an OpenCL read transaction to transfer memory from the device to the host. As depicted in Fig. 6(a), the transaction time can be of the same order, or even longer, than the kernel execution time, becoming a bottleneck. In addition, the memory transactions introduce a delay d_{ik} that prevents the streaming execution of successive kernel invocations.

To reduce the impact of read transactions, we can enqueue several kernel invocations before reading the furthest reaching value in the solution diagonal. This strategy increases the number of executed invocations by a constant value but this becomes irrelevant when dealing with very long strings. Fig. 6(b) illustrates how the execution time is reduced by grouping kernel invocations before a read operation.

When using this strategy in the example, the execution time is reduced from 55.53 to 37.18 s, which is a 1.49× factor with respect to the execution time of the previous optimization. The performance is increased to 277 GCUPS with a workgroup size of 128 workitems. This has an impact, since the GPU is active more time without waiting for the host to interpret the data. The green line in Fig. 5 shows how now the execution time of the kernel basically depends on the number of elements of each column.

We analyze how the performance is affected by the reduction of read operations with several string lengths (see Fig. 7). After the analysis we decide that 1 read transaction every 100 kernel invocations is a good tradeoff to get a good performance on various string lengths. Higher values do not provide additional gains.

4.2. *Strategies to reduce access to global memory*

From the programmer perspective, previous designs access the pyramid $\widehat{W}$ with read and write transactions to global memory. The GPU's cache hierarchy effectively prevents to pay the price of long latencies to global memory, as many accesses result in cache hits. However, the cache hierarchy acts in a transparent way without an specific knowledge of the algorithm. In the following subsections we analyze some strategies to try to increase data locality and, as a result, achieve better performance.

4.2.1. *Pre-fetching to shared memory*

As illustrated in Fig. 3, the processing of a pyramid cell requires to access three cells from the previous pyramid column. Hence, each computed cell (which is stored in global memory) is accessed 3 times when computing the following column.

Pre-fetching is an optimization strategy [31] that involves copying necessary data from a slower, distant memory to a faster, closer memory. When multiple accesses are made to the same data, the overhead of the initial copy is offset by the faster subsequent memory accesses on the faster memory. In our case, we try to reduce the memory bandwidth to global memory by using an strategy based on pre-fetching data from global memory to shared memory. Fig. 8 depicts this strategy. Each workitem works with a row of the pyramid and only fetches data from the cell in the previous column. But for each cell, the algorithm needs to access 2 additional neighboring cells. The benefit of pre-fetching is that these additional accesses are done on shared memory rather than global memory. A corner case happens in the workitems on the boundaries of the workgroup. They have to do an extra access to global memory since one of the neighboring cells is not computed in the workgroup.

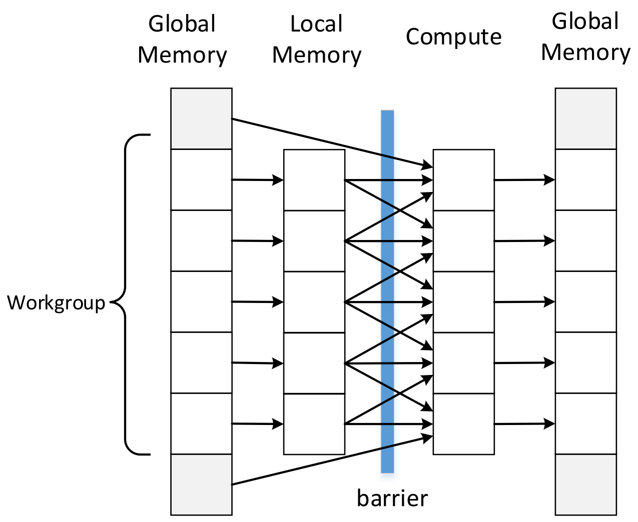


Fig. 8. Strategy to fetch elements from global memory to shared memory before computing pyramid cells. Workitems in a workgroup can reuse the data fetched by their sibling workitems after being sure that everyone completed the fetch with a synchronization barrier (illustrated by a blue vertical line). Workitems at the boundaries of the workgroup require an extra access to global memory.

This approach reduces the memory accesses to global memory by three but requires to synchronize workitems in the workgroup. After fetching their related cell from global memory, all workitems execute a `barrier` instruction (`syncthreads` in CUDA) to ensure that neighboring workitems in the workgroup already fetched their data. Workitems at the boundary of the workgroup must access global memory, since one of the neighboring cells is part of another workgroup.

Fig. 10(a) shows the that this strategy is counterproductive. The expected benefit from reducing accesses to global memory does not occur because of two reasons. First, because in the original version, cells that are accessed from multiple workitems are already present in the cache hierarchy of the GPU, so there is no significant difference between the global memory bandwidth used by both versions. Secondly, because the shared memory version introduces a synchronization primitive that delays the execution of some workitems.

4.2.2. Tiling

The previous shared memory approach suffers from too much synchronization. A way to reduce the synchronization needs is to increase the amount of work that a workitem is processing. Tiling is used in most GPU solutions addressing the SW DP algorithm [25]. The goal of tiling is to improve data locality.

A tiled approach is designed by identifying the data dependencies of the application and extracting the blocks that, after some initial data fetching, can be computed without additional accesses to the distant data store. In our case we can divide the pyramid in diamond shaped tiles that, after fetching data from the tile boundary, can be computed independently. We define tiles by a tile length t parameter, which corresponds to the length of the side of the parallelogram. The number of cells in the tile is $2t^2$. Tile length can set arbitrarily, while it might be limited by the required local memory. Increasing tile length increases the work done by a workitem and minimizes accesses to global memory. It also results in launching less workitems, less kernel invocations, and an increase the computation/communication factor, which is usually beneficial to increase throughput.

When using tiling, compared with the non-tiled approach, the host must do less invocations and each invocation must use less workitems. Each kernel invocation computes a vertical tile band instead of the column computed in previous approaches. In the computation of a tile

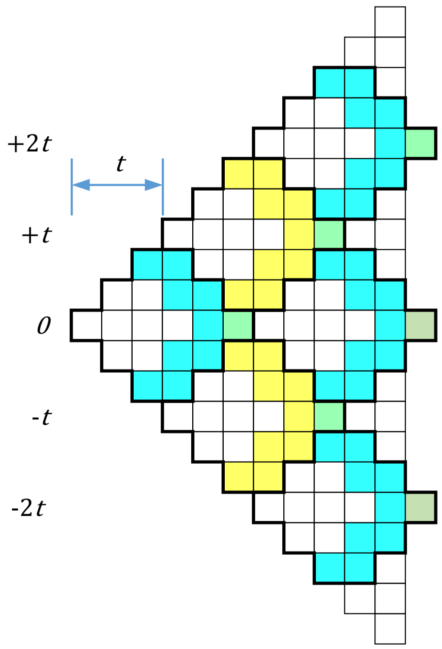


Fig. 9. Example of a tiling approach with $t = 3$ where a workitem computes all cells in a tile. Colored cells depict the cells that must be shared with the workitems in the next vertical band.

band, the number of addressed columns is $2t$, which is higher than the 2 columns addressed in previous approaches. At a first sight this could impose a significantly higher memory demand. However, we notice that only two columns following a zig-zagged pattern are effectively accessed. In Fig. 9, we highlight the ziz-zag data dependencies from previous tiles with blue and yellow colors. The kernel prefetchs all the depending cells from the previous tile band boundary, computes the tile using the local storage, and stores the boundary cells back into global memory.

In GPUs, the obvious local storage is shared memory. The shared memory is not very big (128 kB per SM in RTX 3090), but is faster than global memory because it is located on-chip. As multiple threads will be executed in the SM (the workgroup), a code using the shared memory should divide its use among threads of the workgroup. This means that we must allocate $2t^2 \times wgs$ bytes for the tiles, where wgs is the workgroup size, and then use the workitem local index to index the tile we have to use.

In addition to the use of shared memory, we also investigate two additional options for the local storage: global memory and registers. The use of global memory can be interpreted as contradictory with our goal, but we want to analyze if the memory access pattern inside the tile has a beneficial effect in the cache hierarchy fetching policies. The use of the register file as a memory store can be somewhat challenging.

The register file of a GPU is a big memory (compared with classic processors register files) used to store registers associated with threads. Every thread has its own set of registers, which allow a swap-free context switch between threads. In addition, the number of registers of a thread is not defined by the architecture (as in most processors like RISC-V having 32) but dynamically defined when kernel code is compiled. As a result, the register file is divided by the number of registers required by a thread to know the number of simultaneous on-flight threads in a SM. In RTX 3090 every block has 65 536 registers. The use of registers to store the local tile memory is more intricate as it requires to describe all tile cells as variables in the source code and implementing additional addressing functions to write and read them.

The results of these analyses are shown in Figs. 10(b), 10(c), 10(d). The x axis of the figures describe the number of workitems in the

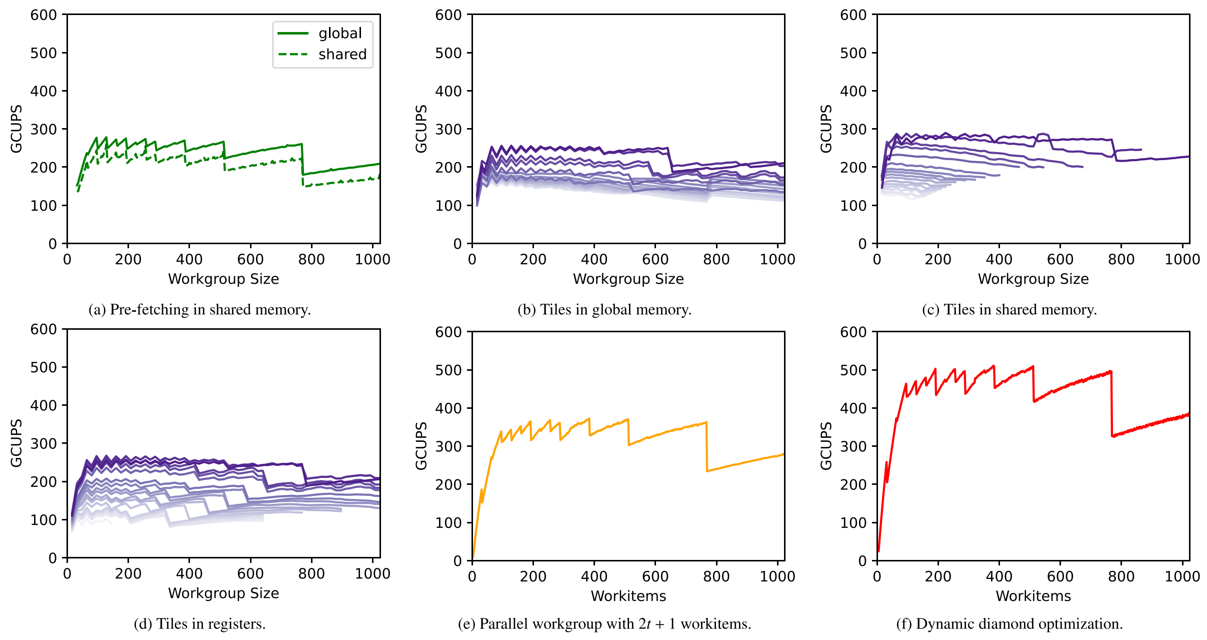


Fig. 10. Performance achieved in NVIDIA GeForce RTX 3090 with different methods that try to reduce access to global memory as a function of the workgroup size. GCUPS are computed as defined in Eq. (2). (a) Pre-fetching in shared memory does not provide a significant with respect to the previous approach due to synchronization overheads. Tiling using one workitem per tile does not provide significant gains as well. Diagrams (b), (c), (d) show multiple tests using different tile lengths (t). Performance for tile lengths greater than 3 decreases due to thread divergence and shared memory is the best option for this tiling approach. (e) Having multiple workitems collaborating in a tiled approach provides significant gains. (f) A substantial gain is obtained with the dynamic diamond algorithmic optimization.

workgroup. This has an impact on GPU occupancy as higher workgroup sizes require more memory. Multiple tile lengths are tested, starting from a tile length of 2 increasing tile lengths are drawn with lighter color. It can be observed than generally bigger tile lengths perform worse.

The use of global memory for local storage does not provide any benefit over previous approaches. The achieved best performance is 256 GCUPS with a tile length of 2 and 128 workitems per workgroup. A profiling analysis reveals that memory accesses are dispersed in memory reducing the chance to coalesce memory operations and increasing the chance to have threads waiting for long memory operations. As a result, the number of non-available threads in the GPU scheduler is increased.

When using registers for the local store, we use conditionals to implement the equivalent to address decoding. Due to the diamond shape of the tile, some diagonals are more frequently accessed than others, so it is convenient that they are checked before less frequent ones. A convenient conditional ordering provides a 10 GCUPS gain with respect to a naive approach. However the best achieved performance with the register-based local store is 267.2 GCUPS with a tile length of 3 and a workgroup size of 128 workitems.

Although not being very significant, the shared memory local store provides an improvement with respect to previous designs. The best achieved performance is 289.6 GCUPS with a tile length of 2 and a workgroup size of 224 workitems.

4.2.3. *Parallel workgroup*

In previous tiling strategies, a whole tile was computed by every workitem. Another approach is studied, where several workitems to collaborate to compute a tile. The principle of this approach is that each workitem within the workgroup is responsible to fetch the memory and compute the cells of a single diagonal. Fig. 11 depicts the strategy. In a first step, all workitems fetch the two dependent positions from global memory and store them in a local store using shared memory. Notice that this requires $2t+1$ workitems (and not $2t-1$) because the cells in the extremes of the tile access cells from upper and lower diagonals. Then, a loop of $2t$ iterations is required to compute the tile values using the local memory. In these iterations, some workitems will do useless work since they do not have anything to compute. Finally, the local memory is transferred to global memory. The execution of the steps is synchronized with the workitems collaborating in the computation done by the workgroup by using a barrier.

The ‘performance of this design is influenced by the workgroup size, which is determined to the tile size $w = 2t + 1$. This approach provides a significant gain with respect to previous designs. The maximum achieved performance is 372 GCUPS, when using 381 workitems per workgroup, which corresponds to tile length of 190.

4.3. *Dynamic diamond*

We left the dynamic diamond approach for the final step, as the code to control the elements of the pyramid the must be processed has a higher complexity. When applying this approach, we basically reduce the number of tiles that are computed. This reduction factor is not constant and depends on the distance between both strings. In this case, the maximum achieved performance is 516 GCUPS, when using the configuration used in the last iteration (381 workitems per workgroup, which corresponds to tile length of 190).

4.4. *Summary*

After an iterative improvement process, we have achieved a 3.4× speed-up factor with respect to the initial WFA design. The main elements that make it possible to achieve this speedup factor are: an appropriate kernel invocation strategy that minimizes memory transactions between the host and the device, the use of tiling to increase data locality within the kernel, and leveraging the diamond-shaped data dependency to reduce the amount of data to compute.

Fig. 12 illustrates the improvements of each step by reporting the maximum achieved performance (in GCUPS) of every method when computing the distance between the strings of the illustrating example.

The presented implementation is open-source, and its code is publicly available at https://github.com/davidcastells/WavefrontVariants.

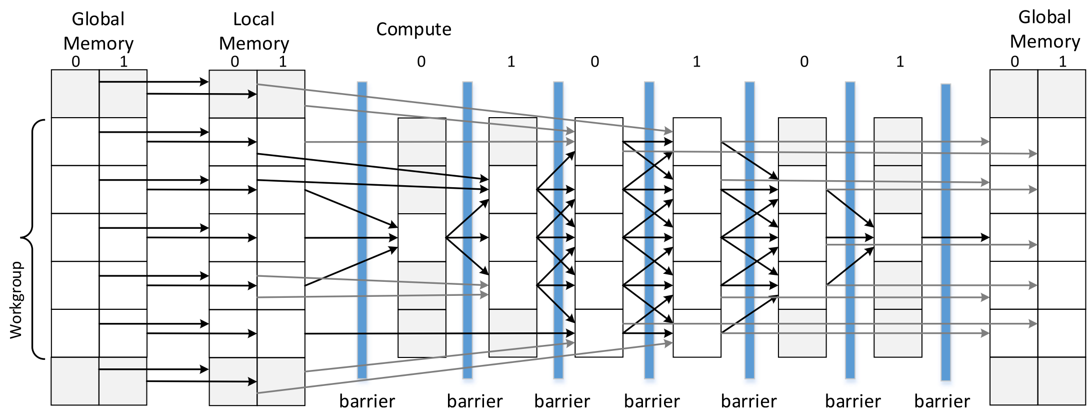


Fig. 11. Parallel computation of a tile of $t = 3$. In a first step values are pre-fetched from global memory to shared memory. The computation of tile columns is synchronized with barrier primitives (depicted in blue).

Table 2
Execution time of this work on a NVIDIA GeForce RTX 3090 compared with the results from Sadiq and Yousaf [23] (2020) in a cluster running an hybrid MPI OpenMP application.

Exp.	String A	Size	String B	Size	Distance	Similarity	Reference		This work		
							Execution time (s)	GCUPS	Execution time (s)	GCUPS	Speed-up factor
1	gbgss201	156,931	gbpln104	79,314	91,590	41%	42.44	0.29	0.12	100.22	353.7×
2	gbgss201	156,931	gbhtg11	606,452	450,982	25%	83.44	1.14	0.86	109.92	97.0×
3	gbhtg11	606,452	gbgss116	1,517,819	969,770	36%	606.71	1.51	4.81	191.11	126.1×
4	gbuna1	308,453	gbinv32	3,424,429	3,116,517	5%	553.04	1.90	6.04	111.83	91.5×

Table 3
Execution time of this work on a NVIDIA GeForce RTX 3090 compared with the first-phase of MASA-Cudalign on the same platform.

Exp.	String A	Size	String B	Size	Distance	Similarity	Reference		This work		
							Execution time (s)	GCUPS	Execution time (s)	GCUPS	Speed-up factor
1	NC_000898.1	162,114	NC_007605.1	171,823	92,254	46%	0.05	557.10	0.13	202.50	0.4×
2	NC_003064.2	542,868	NC_000914.1	536,165	271,907	49%	0.55	529.21	0.79	366.20	0.7×
3	CP000051.1	1,044,459	AE002160.2	1,072,950	549,784	48%	2.00	560.33	2.73	409.05	0.7×
4	BA000035.2	3,147,090	BX927147.1	3,282,708	1,528,200	53%	20.00	516.55	20.81	496.39	1.0×
5	AE016879.1	5,227,293	AE017225.1	5,228,663	2,595	99%	36	759.22	0.34	78,734.09	103.7×
6	NC_005027.1	7,145,576	NC_003997.3	5,227,293	3,700,051	48%	78.00	478.87	103.38	361.28	0.8×
7	NT_033779.4	23,011,544	NT_037436.3	24,543,557	12,565,054	48%	1,200.00	470.65	1,310.61	430.93	0.9×
8	BA000046.3	32,799,110	NC_000021.7	46,944,323	15,700,576	66%	2,258.00	681.90	2,159.39	713.03	1.0×
chr1	NC_000001.10	249,000,000	NC_006468.3	228,000,000	50,882,030	79%	92,162.00	616.00	32,060.21	1,775.17	2.8×
chr19	NC_000019.9	59,128,983	NC_006486.3	63,644,993	15,090,152	76%	6,557.00	573.93	2,779.20	1,354.08	2.3×
chr20	NC_000020.10	63,025,520	NC_006487.3	61,729,293	7,609,553	87%	5,933.00	655.74	889.55	4 373.53	6.6×
chr21	NC_000021.8	48,129,895	NC_006488.2	46,489,110	4,000,655	91%	3,290.00	680.10	167.85	13,330.27	19.8×
chr22	NC_000022.10	51,304,566	NC_006489.3	49,737,984	6,818,507	86%	3,827.00	666.78	681.13	3,746.37	5.6×
23M	NT_033779.4	23,011,544	NT_037436.3	24,543,557	12,565,054	48%	1,196.00	472.23	1,309.54	431.28	0.9×
10M	NC_017186.1	10,236,779	NC_014318.1	10,236,715	374	99.9%	136.00	770.52	0.65	161,216.91	209.2×
5M	AE016879.1	5,227,293	AE017225.1	5,228,663	2,595	99%	36.00	759.22	0.33	82,823.50	109.0×

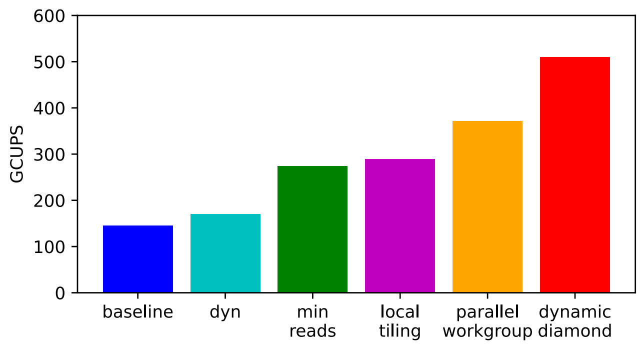


Fig. 12. Maximum performance achieved by the analyzed approaches on the motivating example.

5. Results

In this section, we evaluate our best design and compare it with the state of the art. Since very few works are addressing ED on very long strings we compare our results with the only work fulfilling this requirement and with the first step of the MASA-Cudalign [29] which has a complexity similar to ED.

To base our experiments, we use the test files used by [23] and the CUDAlign project publications [26,28] which cover enough differences in terms of string similarity and length. In the case of [23] the files are not exactly the same reported in the original paper as they were the result of concatenating multiple genomic sequences combined in a `.seq` file, and it has been evolving over time. We have repeated the process with the same sequences files and similar string lengths. The results are similar but not exactly the same reported in the paper. The execution platform consist of a Intel® Xeon® W-2155 CPU with 10 cores running at 3.30 GHz, and a NVIDIA™ GPU GeForce RTX 3090, with 82 SMs running at 1.6 GHz. All GPU measurements are using a single GPU.

Table 4
Maximum length and performance of works in the literature computing the ED between two strings.

Work	Method/Platform	Max reported length	Max performance (GCUPS)
[22] (2021)	Banded-BPM/FPGA	300	5076.00
[8] (2022)	WFA/GPU	1000	(90% sim.) 10,060.30 (98% sim.) 25,062.60
[15] (2014)	BPM/GPU	1000	2300.00
[32] (2018)	BPM/CPU	?	1620.00
[17] (2017)	ED/GPU	22,500	0.88
[20] (2015)	ED/GPU	128,000	20.48
[23] (2020)	ED/CPU	3,424,429	1.90
This work	WFA/GPU	249,000,000	(41% sim.) 100.22 (79% sim.) 1775.17 (99.9% sim.) 161.216.91

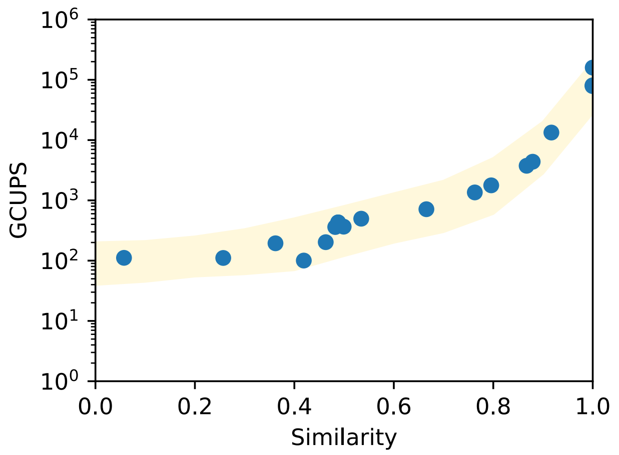


Fig. 13. Our implementation performance (in GCUPS) depends on the similarity of the strings being compared.

The Table 2 compares the results of the proposed method in execution platform with Sadiq and Yousaf's results reported in the paper [23]. The input strings have a rather low similarity below 50%, but Sadiq and Yousaf's top performance is below 2 GCUPS. Our achieved speed-up factor (using a single GPU) is always above two orders of magnitude with respect to Sadiq and Yousaf's work (which is using a computing cluster), achieving up to a 353× factor for the higher similarity file.

For the files used in [26,28] we compare with a compilation in our execution platform of the latest version of MASA-Cudalign software available at https://github.com/edanssandes/MASA-CUDAlign. The original papers were reporting a performance between 20 and 30 GCUPS using a single GPU. But, as shown in Table 3, our tests reveal that the performance in latest GPUs goes up to the region of 600 GCUPS. In this case, our implementation performs slightly worse when similarity is low, as our performance for similarities lower than 70% falls in the range of 200 to 700 GCUPS. However, for higher similarity strings we easily jump above the TCUPS range reaching a top value of 161 TCUPS for almost identical files. This dependence of performance on similarity is depicted in Fig. 13, which collects all the results from the table. String length also has some impact on performance. For the same similarity value, longer sequences will generally provide higher throughput.

We compare our obtained performance with previous works on the literature specifically addressing ED. Table 4 shows the results sorted by the maximum reported length. For short strings (below thousands of characters) a high performance in the order of TCUPS is generally achieved by processing multiple strings concurrently. In this case, the problem becomes embarrassingly parallel and the performance is limited by the communication bandwidth to access input data.

6. Conclusions

In this paper, we have presented a GPU implementation of the WFA algorithm adapted to the computation of the ED problem for very long strings. To the best of our knowledge this is the first work considering the problem of computing the ED for strings longer than one hundred million characters. Our baseline GPU implementation is already orders of magnitude faster than previously reported alternatives specifically targeting ED. In addition, we present a tiled approach together with the dynamic diamond optimization that contribute to increase the performance by a factor of 3.4× with respect to our baseline implementation.

For low similarity strings MASA-Cudalign results suggest that its adaptation to the ED problem would provide a competitive performance. However, for long strings with similarities above 70%, to the best of our knowledge, our results are the best ever reported. In this work we have focused on obtaining the ED value. In future works, we will focus on obtaining the ED alignment.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2023.103019. David Castells-Rufas reports financial support was provided by European Regional Development Fund.

Data availability

The data used is available from public repositories (data sources have been already described in the paper)

Acknowledgments

This research was supported by the European Union Regional Development Fund (ERDF) within the framework of the ERDF Operational Program of Catalonia 2014–2020 with a grant of 50% of the total cost eligible under the Designing RISC-V based Accelerators for next generation computers project (DRAC) [001-P-001723], in part by the Catalan Government under grant 2017-SGR-1624, and in part by the Spanish Ministry of Science, Innovation and Universities under grant RTI2018-095209-B-C22.

References

[1] V.I. Levenshtein, et al., Binary codes capable of correcting deletions, insertions, and reversals, Sov. Phys. Dokl. 10 (8) (1966) 707–710.
[2] J. Wang, Y. Dong, Measurement of text similarity: a survey, Information 11 (9) (2020) 421, http://dx.doi.org/10.3390/info11090421.

[3] F. Foscarin, F. Jacquemard, R. Fournier-S'niehotta, A diff procedure for music score files, in: 6th International Conference on Digital Libraries for Musicology, 2019, pp. 58–64, http://dx.doi.org/10.1145/3358664.3358671.

[4] M. Šošić, M. Šikić, Edlib: a C/C++ library for fast, exact sequence alignment using edit distance, Bioinformatics 33 (9) (2017) 1394–1395, http://dx.doi.org/10.1093/bioinformatics/btw753.

[5] S. Arora, B. Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.

[6] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173, http://dx.doi.org/10.1145/321796.321811.

[7] S. Marco-Sola, J.C. Moure, M. Moreto, A. Espinosa, Fast gap-affine pairwise alignment using the wavefront algorithm, Bioinformatics 37 (4) (2021) 456–463, http://dx.doi.org/10.1093/bioinformatics/btaa777.

[8] Q. Aguado-Puig, S. Marco-Sola, J.C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, M. Moreto, Accelerating edit-distance sequence alignment on GPU using the wavefront algorithm, IEEE Access 10 (2022) 63782–63796, http://dx.doi.org/10.1109/ACCESS.2022.3182714.

[9] K.E. Iverson, A programming language, in: Proceedings of the May 1-3, 1962, Spring Joint Computer Conference, 1962, pp. 345–351.

[10] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343, http://dx.doi.org/10.1145/360825.360861.

[11] G. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, J. ACM 46 (3) (1999) 395–415, http://dx.doi.org/10.1145/316542.316550.

[12] E. Ukkonen, Algorithms for approximate string matching, Inf. Control 64 (1–3) (1985) 100–118, http://dx.doi.org/10.1016/S0019-9958(85)80046-2.

[13] H. Hyyrö, A bit-vector algorithm for computing levenshtein and damerau edit distances, Nordic J. Comput. 10 (1) (2003) 29–39.

[14] J. Zhang, H. Lan, Y. Chan, Y. Shang, B. Schmidt, W. Liu, BGSA: a bit-parallel global sequence alignment toolkit for multi-core and many-core architectures, Bioinformatics 35 (13) (2019) 2306–2308, http://dx.doi.org/10.1093/bioinformatics/bty930.

[15] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, J.C. 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, http://dx.doi.org/10.1145/2597652.2597677.

[16] K. Balhaf, M.A. Shehab, T. Wala'a, M. Al-Ayyoub, M. Al-Saleh, Y. 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, http://dx.doi.org/10.1109/IACS.2016.7476090.

[17] K. Balhaf, M.A. Alsmirat, M. Al-Ayyoub, Y. Jararweh, M.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, http://dx.doi.org/10.1109/IACS.2017.7921937.

[18] Y. Li, L. Schwiebert, Memory-optimized wavefront parallelism on GPUs, Int. J. Parallel Program. 48 (6) (2020) 1008–1031, http://dx.doi.org/10.1007/s10766-020-00658-y.

[19] A. Tomiyama, R. 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, http://dx.doi.org/10.1007/978-3-642-38718-0_38.

[20] M. Kruliš, D. Bednárek, M. Brabec, Improving parallel processing of matrix-based similarity measures on modern gpus, in: International Conference on Similarity Search and Applications, Springer, 2015, pp. 283–294, http://dx.doi.org/10.1007/978-3-319-25087-8_27.

[21] D. Bednárek, M. Brabec, M. Kruliš, Improving matrix-based dynamic programming on massively parallel accelerators, Inf. Syst. 64 (2017) 175–193, http://dx.doi.org/10.1016/j.is.2016.06.001.

[22] D. Castells-Rufas, S. Marco-Sola, Q. Aguado-Puig, A. Espinosa-Morales, J.C. Moure, L. Alvarez, M. 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, http://dx.doi.org/10.1109/FPL53798.2021.00036.

[23] M.U. Sadiq, M.M. Yousaf, Distributed algorithm for parallel edit distance computation, Comput. Inform. 39 (4) (2020) 757–779, http://dx.doi.org/10.31577/cai_2020_4_757.

[24] T.F. Smith, M.S. Waterman, et al., Identification of common molecular subsequences, J. Mol. Biol. 147 (1) (1981) 195–197, http://dx.doi.org/10.1016/0022-2836(81)90087-5.

[25] Z. Xia, Y. Cui, A. Zhang, T. Tang, L. Peng, C. Huang, C. Yang, X. Liao, A review of parallel implementations for the Smith–Waterman algorithm, Interdiscip. Sci.: Comput. Life Sci. (2021) 1–14, http://dx.doi.org/10.1007/s12539-021-00473-0.

[26] E.F.O. Sandes, A.C.M. de Melo, CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010, pp. 137–146, http://dx.doi.org/10.1145/1693453.1693473.

[27] E.F.d.O. Sandes, A.C.M. de Melo, Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU, IEEE Trans. Parallel Distrib. Syst. 24 (5) (2012) 1009–1021, http://dx.doi.org/10.1109/TPDS.2012.194.

[28] F.d.O. Edans, G. Miranda, A.C. de Melo, X. Martorell, E. Ayguadé, Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2014, pp. 160–169, http://dx.doi.org/10.1109/CCGrid.2014.18.

[29] E.F. de Oliveira Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, A.C.M. Melo, CUDAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2838–2850, http://dx.doi.org/10.1109/TPDS.2016.2515597.

[30] T. Allen, R. Ge, Characterizing power and performance of gpu memory access, in: 2016 4th International Workshop on Energy Efficient Supercomputing (E2SC), IEEE, 2016, pp. 46–53, http://dx.doi.org/10.1109/E2SC.2016.012.

[31] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, H.E. Bal, Optimization techniques for GPU programming, ACM Comput. Surv. (2022) http://dx.doi.org/10.1145/3570638.

[32] Y. Chan, K. Xu, H. Lan, B. Schmidt, S. Peng, W. Liu, Myphi: efficient levenshtein distance computation on xeon phi based architectures, Curr. Bioinform. 13 (5) (2018) 479–486, http://dx.doi.org/10.2174/1574893612666171122150933.