Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment

Meven Mognol, Dominique Lavenier, Julien Legriel

Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment

Meven Mognol
meven.mognol@inria.fr
Univ. Rennes, IRISA, Inria & UPMEM
Rennes, France

Dominique Lavenier
dominique.lavenier@irisa.fr
Univ. Rennes, CNRS-IRISA, Inria
Rennes, France

Julien Legriel
jlegriel@upmem.com
UPMEM
Munich, Germany

ABSTRACT

Sequence alignment is an ubiquitous task in genomics analysis whose performance can be affected by the memory-wall on a processor-centric architecture. Processing-in-Memory (PiM) architectures provide a memory system with integrated computing capabilities to alleviate this bottleneck. In this paper, we present a long read optimized version of the Needleman and Wunsch (N&W) algorithm based on PiM devices developed by the UPMEM company. Such memories add 128 computing units to a standard 16GB DIMM. On a server equipped with 20 UPMEM PiMs, our N&W implementation outperforms standard alignment tools by an order of magnitude compared to a traditional multicore server. Code available at https://github.com/upmem/usecase_dpu_alignment.

CCS CONCEPTS

• **Theory of computation** → **Massively parallel algorithms**; *Pattern matching*; • **Hardware** → **Memory and dense storage**.

KEYWORDS

Processing-in-Memory, long read Alignment, parallelism

ACM Reference Format:
Meven Mognol, Dominique Lavenier, and Julien Legriel. 2024. Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

The alignment of protein or DNA sequences is a fundamental bioinformatics process. It finds applications in a wide variety of contexts, including genome assembly[4], error read correction[1][29], or the study of phylogeny between populations[12]. The advent of third-generation sequencers has transformed the field by generating long sequences (called reads) spanning from a thousand to millions of base pairs (bp)[21]. Despite the advantages of these technologies, such as addressing challenges related to structural variation and resolving repetitive regions, they come with a higher error rate compared to previous generations. Consequently, extensive effort is required to ensure the production of high-quality sequences from imprecise raw reads, to both assemble or polish the texts of the genomes[27]. Furthermore, alignment plays an essential role in constructing phylogenetic trees using conserved DNA or RNA sequences from different species[2]. For instance, for bacteria, the 16S ribosomal RNA is often employed to establish phylogenetic relationships. This involves conducting an all-against-all comparison between the bacteria studied, implying a quadratic number of comparisons in relation to the number of bacteria. Once again, this underscores the critical need for an efficient method for computing alignments.

This paper focuses on global alignment, a type of alignment that involves comparing entire sequences. Dynamic programming (DP) algorithms, known for providing optimal solutions[28], are frequently employed to address the approximate string matching problem, that is sequence alignment. It is worth noting that, being an algorithm with quadratic complexity, it is inherently time-consuming, particularly when confronted with long sequences. In addition, conducting multiple alignments in parallel exacerbates the situation by introducing a memory bottleneck[7]. Several dynamic programming (DP) algorithms exist, ranging from the classic Needleman and Wunsch (N&W)[20] to the Smith and Waterman algorithm[23], and more recently, the WFA algorithm[19]. Here, we implement a variation on the N&W algorithm specifically designed for comparing long DNA sequences.

In short, N&W consists in finding the minimal number of basic edit operations to go from one sequence to another. Edit operations can be substitution (one character is transformed into another one), insertion (one character is added) and deletion (one character is lost) as shown in figure 1.

```
G A G A T - T A C A
| |   | |   | |   |
G A C A T A T A - A
```

Figure 1: Two sequences aligned with, in order, one mismatch, one insertion and one deletion.

N&W is based on DP and it is optimal as it always returns the minimum number of edit operations needed to transform one sequence into the other. N&W computes a matrix and its complexity is $O(m \times n)$ in both time and space for sequences of respective length m and n.

The N&W algorithm's quadratic complexity is particularly penalizing when handling long sequences. Hence, multiple heuristics have been proposed to avoid computing all the values of the DP matrix while maintaining a good alignment accuracy [3][26][16]. Most of these heuristics improve the performance by exploiting the following property: when two sequences are similar, the optimal path from one to the other stays close to the diagonal of the matrix

HAL, 2024,

(the reciprocal is not true). Hence, when it is known in advance that sequences are close to each other, it is sufficient to calculate a band around the diagonal to find the optimal alignment.

This family of algorithms has been carefully optimized for the x86 architecture, as evidenced by various well-crafted implementations [5][9][25]. A particularly efficient and widely used implementation is found in the minimap2 software [16]. However, it is hard to scale these implementations to obtain high sequences alignment throughput: on the one hand, the semi-sequential nature of DP limits the parallelism for a single alignment; on the other hand, increasing the number of simultaneous alignments quickly leads to a bottleneck in data movement.

Processing-in-Memory (PiM) architectures propose to alleviate the memory bottleneck by bringing computing power closer to the memory. By the past, previous instances of architectures with a focus on sequence alignment have been proposed [14][11], but their evaluations were limited to simulations. Here, we used a production architecture developed by UPMEM [6]. Prior research has demonstrated promising performance of DP on PiM using the UPMEM platform, albeit limited to handling only small sequences [7]. In this work, we developed the first DP implementation designed specifically for the UPMEM PiM architecture, targeting long sequence alignment scenarios. We investigated the parallelization of the N&W algorithm and we compared our implementation with minimap2, one of the faster comparison sequence tool. To assess our approach, we conducted experiments on synthetic and real datasets. Using a PiM server with 20 DIMMs, we showed that a 9x acceleration can be obtained compared to a 2 Intel Xeon Silver 4215 server.

The rest of the paper is structured as follows. In Section 2, the UPMEM PiM server and its programming environment are presented. Section 3 describes the N&W algorithm, its affine version, and the static and dynamic banded optimizations, while Section 4 details the parallelization of N&W on the PiM server. Section 5 presents the experimentation results and finally we provide our conclusions in Section 6.

2 UPMEM PIM SYSTEM

2.1 Architecture

In a conventional computer, the central processing unit (CPU) and the memory are two separate components. The memory is packaged as dual in-line memory modules (DIMM), where each side of a module, called a memory rank, can be accessed independently. A rank consists of multiple memory chips, each chip containing several memory arrays (called banks).

The UPMEM company has developed a Processing-in-Memory device that integrates a processing unit next to each memory bank. This PiM device is a DIMM module with two ranks, each rank being composed of 8 chips with 8 memory banks of 64MB. Attached to each memory bank is a general-purpose data processing unit (DPU) with direct access to the bank through a DMA engine. A DPU has access only to the 64MB RAM of the bank to which it is attached (called the DPU MRAM). The memory of other DPUs, including those in the same rank or chip, cannot be accessed directly, i.e., the communication between DPUs happens via the host CPU. In summary, the UPMEM PiM DIMM is an 8GB DIMM module with two ranks of 64 DPUs, each DPU having access to 64MB of memory.

The PiM DIMMs are integrated into a x86 server with a host CPU and standard DIMM modules. The host CPU can access the DPUs MRAM directly through the DDR bus, which enables fast and direct data transfers between the CPU and the DPU MRAMs. Additionally, the CPU is responsible for the orchestration of the execution. It transfers data from the standard RAM to the PiM modules, uploads the program to the DPUs' instruction RAM, launches the execution, waits for the program termination, retrieves the results and so on. When DPUs are executing, they have exclusive access to the MRAM and the host cannot access the MRAM anymore. This exclusive access to the memory bank is necessary to ensure the integrity and coherence of computations within the PiM architecture, while respecting the DDR protocol. The granularity of access to DPUs is the rank (64 DPUs), meaning that operations are generally performed simultaneously on all DPUs of a rank (transfer, launch, etc) in order to maximize the system's performance.

Although it would be possible to load different programs into different DPUs, the typical usage is to load a common program and leverage data parallelism to speed up the execution of a memory-bounded algorithm. Inside each DPU, several hardware threads are also executing in parallel. These threads share the same instruction memory (and hence execute the same program), but each of them has its own context (registers, program counter) so that they can execute different functions within the same program, or execute independently the same function on different data. From the programming point of view, an executing hardware thread with its context is referred to as a tasklet.

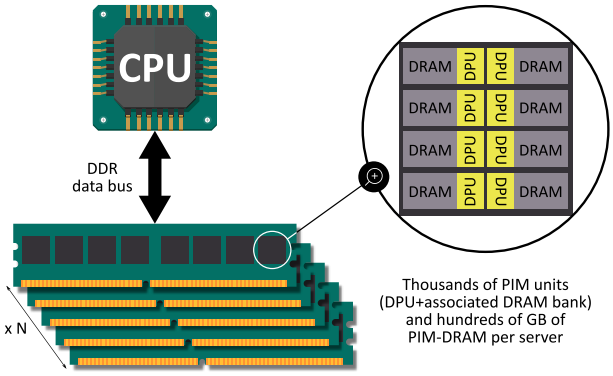

Figure 2: Server with PiM DIMMs.

The DPU instruction pipeline is shared by the tasklets through a simple round-robin mechanism. It is 14 cycles deep, with 11 consecutive steps of the pipeline imposing a blocking restriction on tasklet's reentry. In other words, a specific tasklet can only execute one instruction every 11 cycles, and a minimum of 11 tasklets must be running in order to attain the peak performance of 1 instruction per cycle. This fine-grained multi-threading execution of tasklets permits a relatively simple pipeline design by avoiding problems related to exceptions or branch prediction. This design choice, among others, is motivated by the challenge of implementing a PiM core inside a DRAM process (the DPU is on the same chip as the memory

bank). On the other hand, it requires the programmer to expose a good level of parallelism for efficient execution.

The DPU features a proprietary triadic instruction set architecture (ISA) particularly good for branching. It is enriched with a fused jump instruction mechanism, allowed by the pipeline's reentry restriction. This design allows cycle-free jumps before or after most instructions (e.g. jump when the result of an add is lower than zero), improving execution time through fast branching. Integer arithmetic is efficiently supported through hardware, but no floating point unit is present due to area and power efficiency considerations. The execution is quite predictable, as each instruction is taking the same number of cycles in the pipeline, with the exception of DMA load/store instructions which access the memory bank.

The working memory of the DPU is a 64KB scratchpad (called WRAM) directly accessible through load and store instructions. These instructions take the same number of cycles as the other instructions, and can be executed at a maximal rate of 1 instruction per cycle. In other words, from the tasklet perspective, a load or store to the WRAM is never a blocking operation and the tasklet's next instruction can be executed after 11 cycles. This is not the case for access to the MRAM (the memory bank), which is performed using a DMA engine transferring bytes from (or to) the MRAM to (or from) the WRAM at a rate of 2B/cycle. During the execution of the DMA transfer, a tasklet is waiting and cannot issue new instructions into the pipeline. In order to mask the latency of MRAM accesses, more than 11 tasklets (usually 16) are being executed in parallel. Data transfers between the DPU's MRAM and WRAM must be explicitly managed by the programmer and be preferably of large size (transfer size is between 8 and 2048 bytes) to maximize the performance.

This hardware architecture holds significant implications when developing parallel algorithms on it. Specifically, being able to expose massive parallelism within the application and split it into relatively independent tasks running across individual DPUs and tasklets is necessary for achieving optimal performance.

A typical PiM system from UPMEM (figure 2) is composed of a dual-socket x86 Intel server with 20 PIM DIMMs, totalizing 2560 DPUs. DPUs are running at a frequency between 350 and 450Mhz and offer a cumulative memory bandwidth of 2TB/s.

2.2 Programming environment

Programming an application for the UPMEM-PiM server requires the creation of two distinct programs, one for the CPU and one (at least) for the DPUs.

The role of the host program encompasses several responsibilities, including first allocating resources (ranks of DPUs), splitting the program data and distributing it into the DPUs' MRAM, and loading the program. After this initialization phase, a typical workflow involves broadcasting requests to each DPU and launching the execution, waiting for termination, getting the results from the DPUs and consolidating them before the next such iteration. The host program is developed using the x86 tool-chain and linking with UPMEM's SDK host library available in C, C++, Java or Python. The host library provides all functionalities necessary to the DPU program orchestration, as described above.

The DPU is programmed in C, with a subset of the C library being ported. A dedicated compiler based on LLVM is provided as well as simulation and debugging tools. The DPU library also provides functionalities helping with tasklet synchronization (e.g., mutex, barrier), MRAM accesses, performance counters, etc.

3 DYNAMIC PROGRAMMING ALGORITHM

This section recalls the dynamic programming (DP) algorithm for finding alignments between two sequences, and motivates the use of the adaptive band approach for PiM implementation.

3.1 Needleman & Wunsch algorithm

In the pairwise sequence alignment problem, we consider as input a pair of sequences $A = a_1, a_2, ..., a_i, ..., a_m$ and $B = b_1, b_2, ..., b_j, ..., b_n$ where a_i and b_j are chosen from a finite alphabet, e.g. A,T,G,C. The output is a sequence alignment and a score. The N&W algorithm computes the optimal score $H_{m,n}$ through the following recursion:

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + sub(a_i, b_j) \\ H_{i-1,j} - gap \\ H_{i,j-1} - gap \end{cases} \quad (1)$$

Where $i \in \{1; m\}$, $j \in \{1; n\}$, and $H_{i,j}$ is the score of the alignment between sub-sequences of A and B ending respectively at element i and j. The recursion ends when i or j is zero with a score obtained as follows:

$$\begin{aligned} H_{0,j} &= j \times -gap \\ H_{i,0} &= i \times -gap \end{aligned} \quad (2)$$

In equations 1, $sub(a_i, b_j)$ is the cost associated to a substitution (i.e., when replacing an element of the sequence with another), and gap is a constant value representing the cost for an insertion or deletion of an element. For DNA sequences, $sub(a_i, b_j)$ has a positive value if $a_i = b_j$ (called a match) and a negative value otherwise (called a mismatch or substitution). Equations 2 are derived from the fact that an alignment with an empty sequence has a score equal to the size of the non-empty sequence times the cost of deletion.

At the end of the recursion, the global alignment score between the entire sequences A and B is given by $H_{m,n}$. To obtain this score, the entire matrix H needs to be computed. The corresponding alignment (i.e., the set of substitutions, insertions and deletions on the optimal path) can be constructed using a backtracking procedure (a.k.a. traceback) starting at $H_{m,n}$ and following the path in the matrix that led to this score.

3.2 Affine gap extension

From a biological point of view, the insertion and deletion cost model used in equations 1 and 2 is actually unsatisfactory. The model is indeed too pessimistic for gaps, i.e., when successive insertions or deletions of one element occur in a row. In this case, the cost computed using a linear function based on the constant gap ends up too high. To solve this problem, Gotoh [10] proposed a cost model based on an affine function which separates two costs, the cost of opening a gap, and the cost of extending a gap. Since the cost of opening a gap is higher, this model indeed avoids over-penalizing the larger gaps. The recursion equations are however more complex as three matrices of size $m \times n$ need to be computed instead of one matrix for the original N&W algorithm.

$$D_{i,j} = \begin{cases} -\inf & \text{if } j = 0 \\ -gap_{open} - j \times gap_{ext} & \text{if } i = 0 \\ \max \begin{cases} D_{i,j-1} - gap_{ext} \\ H_{i,j-1} - gap_{open} - gap_{ext} \end{cases} \end{cases} \quad (3)$$

$$I_{i,j} = \begin{cases} -\inf & \text{if } i = 0 \\ -gap_{open} - i \times gap_{ext} & \text{if } j = 0 \\ \max \begin{cases} I_{i-1,j} - gap_{ext} \\ H_{i-1,j} - gap_{open} - gap_{ext} \end{cases} \end{cases} \quad (4)$$

$$H_{i,j} = \begin{cases} 0 & \text{if } i = 0, j = 0 \\ D_{0,j} & \text{if } i = 0 \\ I_{i,0} & \text{if } j = 0 \\ \max \begin{cases} H_{i-1,j-1} + sub(a_i, b_j) \\ D_{i,j} \\ I_{i,j} \end{cases} \end{cases} \quad (5)$$

Matrices D and I keep the values of the current most inexpensive gap, both for horizontal and vertical gap (deletion or insertion). Then it is used to choose between extending the current gap or opening a new one for the current position. The H matrix, the score matrix, takes into account those two matrices to update its score. The alignment can be constructed using a similar traceback procedure, but information from the three matrices is required.

3.3 Banded DP algorithm

As mentioned in introduction, the complexity of the dynamic programming algorithm described in 3.1 is quadratic. This follows from the fact that all of the $m \times n$ values of matrix H need to be computed. In practice, this means that the applicability of N&W is limited to the alignment of short sequences of few hundred bp. For long sequences, especially for DNA reads coming from the third generation of sequencing machines, whose length ranges from 1k bp to 100k bp, the execution time of N&W is becoming prohibitive.

On the other hand, when sequences to align are close to each other, which is often the case in genomics applications, the useful information for calculating the optimal score is located around the diagonal of the matrix. In other words, it is not necessary to compute the whole matrix to find the optimal alignment, but only the values around the diagonal. The banded DP algorithm therefore computes the values in a predefined band centered on the diagonal, and ignores other values. This strategy greatly reduces the amount of computation to be done as the algorithm's complexity becomes $O(w \times (m+n))$ with w the size of the band. Practically, the band's size can be set to a few hundreds of values, leading to a significant gain in execution time and memory space on long sequence alignment. The downside of the banded DP algorithm is that it is not guaranteed to find the optimal alignment as it does not evaluate all possibilities. Setting the size of the band appropriately is therefore important to ensure a good quality of results, and it requires knowledge on the typical amount of differences between the sequences to align (e.g., the maximum size of a gap).

Regarding the implementation of N&W on the UPMEM PiM architecture, using the banded DP strategy is essential. Each processing unit (DPU) has access to only one memory bank of 64MB which is too small to contain the 3 matrices involved in the algorithm with affine gap extension. On the other hand, storing a hundred values around the diagonal means storing 6 million of values for sequences of 10k bp, which fits into the memory available. Additionally, as explained in section 4, the values can be stored on 3 bits only.

3.4 Adaptive banded DP algorithm

In order to compute a correct score using the banded DP algorithm, the path leading to that score must stay within the band. The size of the band must be chosen according to the estimation of local maximum cumulative number of gaps and also depends of the difference between the lengths of the 2 sequences. The larger this estimation is, the bigger the band size must be. Estimating cumulative gaps is not easy, and often leads to an overestimation of the band size, and consequently an excess of calculations.

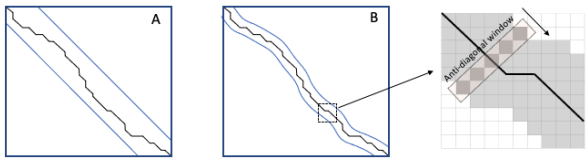

Figure 3: (A) Fixed band: the optimal path must be located within the band whose size depends of the number of gaps and the length difference between the two sequences. (B) Adaptive band: the position of the window is adjusted according to values computed in the anti-diagonal

The adaptive band proposed by [24] is an efficient heuristic to partially overcome these disadvantages. The computation of the band is performed on an anti-diagonal window as shown figure 3. At the beginning, the window is centered at position [0,0] of the matrix (top left corner). Depending on the values at the extremities of the window, it is shifted right or down to follow the most likely path. This heuristic significantly decreases the amount of calculation as the size of the band can be much smaller while maintaining equivalent accuracy. However, a new condition is introduced in the critical loop of the algorithm (inside the band computation), which leads to an increase in the branch miss-prediction rate on CPU, and consequently severely limits the performance benefit. In the case of UPMEM's PiM DPU, this heuristic is however a good choice. Since the DPU does not perform any speculative execution, the decrease in computation provided by the adaptive scheme directly translates into a proportional speedup of the program's execution time.

4 IMPLEMENTATION

As explained in section 2.2, implementing N&W on the PiM hardware requires developing a program running on the host CPU and a compute kernel running on the DPUs. The host program orchestrates the execution, and the DPU program performs the alignment by dynamic programming according to equations 3, 4, and 5. This section provides a short description of the most relevant implementation choices which enable to optimize the performance of the PiM implementation.

4.1 Host program

In a nutshell, the host program's main loop consists in the following four steps:

(1) Dispatch batches of pairs of sequences to the DPUs
(2) Launch the DPUs (send a boot command)
(3) Wait for the DPUs to finish their computations
(4) Collect and interpret the alignment results

There are two main aspects related to step 1 which affect the performance : the time taken to transfer the input sequences from the host to the DPUs and the load balancing across DPUs.

4.1.1 Data transfer optimization. On the PiM server used for our experiments, the measured peak bandwidth when transferring data from the standard memory to the PiM memory reaches around 60GB/s. The data transfer is performed by an API of UPMEM's SDK which uses multiple threads to handle the transfer to different PiM ranks in parallel. In practice, the transfer's throughput is affected by multiple characteristics such as the number of memory channels and the CPU L3 cache size.

But the time taken to transfer a large batch of input sequences to the PiM system can be significant in the total execution time, and as explained in section 2.1, the communication cannot be pipelined with the DPUs execution. By default, each character of the DNA sequence is encoded on one byte (ASCII character), as it comes from a human-readable text file on disk. In order to reduce the size of the transfer to the DPUs, we decided to first encode characters on 2 bits, which is possible since we use a restricted alphabet of 4 nucleotides A, C, T and G. This encoding is done on the fly while also distributing the data into different batches that are sent to different DPUs. After implementing this encoding, the transfer time is below 15% of the total execution (and represents a negligible fraction on large datasets), while the additional cost due to the encoding is minimal.

Note that sequencing is not perfect. In addition to gaps, we sometimes detect the presence of a nucleotide without being able to identify which specific one it is. The sequencer represents this uncertainty with the ambiguous base code "N". While a better compression scheme for handling these ambiguous bases could be devised, [17] suggests that converting N to any nucleotide in at least one sequence does not affect the alignment results. Additionally, other genomic tools, such as metaFlye [15], randomly substitute Ns with A, C, G, or T.

This implementation choice also affects the DPU program which needs to compare sequences where each nucleotide is encoded on 2 bits. It turns out that this choice is also beneficial for the DPU program's performance. On one hand, the DPU needs to perform smaller data transfers between the DRAM bank (MRAM) and the scratchpad (WRAM). On the other hand, extracting nucleotides stored on 2 bits can be made quite efficient using shift instructions on the DPU.

4.1.2 Load balancing. The distribution of DNA sequences among the DPUs plays an important role to get a good load balancing across the system. More precisely, it is of great importance to balance the workload between DPUs of the same rank (64 DPUs, see section 2.1), since the system imposes a barrier at the end of execution of all DPUs of the same rank. Hence, while it is possible to independently retrieve the results of different ranks of DPUs, the results collection of a rank cannot happen before all DPUs of the rank have completed their work. Minimizing the time interval between the fastest and the slowest DPU in a rank is thus desirable to ensure optimal performance. It is also necessary (but less critical) to balance the workload of each rank to avoid ranks being inactive or overloaded.

We rely on the time complexity of the banded DP algorithm with band size w (as defined in 3.3) to estimate the workload of the alignment of two sequences of size m and n :

$$Workload(m, n) = (m + n) \times w \tag{6}$$

The workload of a DPU is simply the cumulative workload of all alignments assigned to it. Input sequences are read in groups from the disk, the exact number read at once being a parameter of the program. Then, pairs of reads to compare are distributed equally in N batches with N being the number of ranks and sent to a FIFO queue. When a rank terminates its execution, the next batch of the queue is assigned to it. The process of reading the input sequences and filling the queue is happening in parallel of the batch assignment and DPUs execution. We then employ a simple and well known heuristic for distributing the workload between DPUs of rank. We sort the pairs of sequences by decreasing workload, and keep on assigning the pair with the largest workload to the DPU with lowest workload, until all pairs have been assigned. This simple heuristic is fast to execute and provides a good approximation to the load balancing problem.

4.2 DPU program

4.2.1 Score computation. As stated in 3.2, 3 matrices need to be computed for the recursion. However, to compute the final score ($H_{m,n}$) it is not necessary to keep all the values of these matrices in memory. The computation of $H_{i,j}$, $D_{i,j}$ and $I_{i,j}$ requires only the presence of neighborhood values $H_{i-1,j-1}$, $H_{i,j-1}$, $H_{i-1,j}$, $D_{i,j-1}$ and $I_{i-1,j}$. Thus, instead of storing 3 matrices, only 4 anti-diagonals of size w (the size of the band) need to be kept and updated: the two previous anti-diagonals of H and one anti-diagonal from the I and D matrices.

The memory footprint to compute the score of an alignment between two DNA sequences is reduced to four integer arrays of size w. This means that these arrays can fit the DPU's WRAM, which guarantees fast update.

4.2.2 Traceback algorithm. This procedure aims to provide the optimal alignment between two sequences in the standard format called Compact Idiosyncratic Gapped Alignment Report (CIGAR). The CIGAR is a compact string detailing the position of matches, insertions and deletions. Building the CIGAR for the best alignment consists in two main steps: (1) creation of an auxiliary data structure during the scores computation to retain the path leading to all scores and (2) navigate through this structure to extract the optimal path.

Step *(1):* The auxiliary data structure, denoted BT, has dimensions $(m + n) \times w$. Each row of BT corresponds to an anti-diagonal. Essentially, a cell in BT preserves information about which neighboring cell in H has contributed to achieving the maximum score. This score may originate from $H_{i-1,j-1}$, $I_{i,j}$, or $D_{i,j}$. To represent

this information, we derived a 4-bit encoding scheme: 2 bits are allocated to indicate the origin cell (H with match, H with mismatch, I, or D), while 2 bits indicate whether, in case of a gap (I or D), it corresponds to the initiation of a new gap or the extension of a pre-existing one.

Step (2): Utilizing the BT array, the traceback procedure generates a CIGAR string, reporting the alignment between the DNA sequences. Beginning from the BT cell associated with $H_{n,m}$, the algorithm produces a sequence of elementary operations (match, mismatch, gaps) that corresponds to the transformation of one sequence into the other. The process terminates upon reaching the first anti-diagonal.

4.2.3 Multitasking. As explained in section 2.1, it is necessary to use a minimum of 11 tasklets on each DPU to reach the best possible pipeline utilization (i.e., an instruction per cycle count close to 1). It means that the work assigned to each DPU needs to be parallelized, which can be done using one of the following two strategies : (1) data parallelism at alignment level: each tasklet performs a different alignment concurrently, or (2) data parallelism at the anti-diagonal level: several tasklets share the work associated with the computation of a single alignment. The first approach is the most straightforward to implement, but it also has the largest memory requirement. Indeed it requires to store at least 11 times the data needed for one alignment (i.e., matrices H, D, I, the traceback data BT etc.), which could not fit in the DPU's WRAM. This strategy enables to use a maximum of 8 tasklets in parallel, which is not enough to obtain full pipeline usage. Hence, we instead choose to implement a mix of strategies 1 and 2, where several groups of tasklets work on different alignments in parallel. Having a group of tasklets working on the same alignment is challenging in terms of implementation as tasklets in a group need to be started and ended dynamically while the DPU has no runtime to easily perform these operations (by default, each tasklet is booted at startup and runs till the end of the program). This strategy therefore involves complex synchronization and low-level tasklets management.

The hybrid approach developed uses P pools of T tasklets to simultaneously align P pairs of sequences. Within each pool, a designated tasklet serves as the master, responsible for initializing buffers and shared variables. Once initialized, it orchestrates synchronization among the N tasklets within its pool at the granularity of anti-diagonal computation. Tasklets are assigned distinct segments of the three anti-diagonal vectors to update. This parallelization strategy is possible because the calculation of cells within an anti-diagonal can proceed independently (as described in section 4.2.1). However, it is important to note that the traceback procedure, which entails sequential traversal of the BT table, cannot be parallelized.

4.2.4 Assembly optimization. As outlined in Section 2.1, the Instruction Set Architecture of the DPU includes specialized instructions. However, as of today, the code produced by the compiler does not leverage them at their best in all cases, whereas a gain of a single instruction in a critical part of code can lead to significant performance gains. Hence, to optimize further, the assembly code of critical sections was analyzed and rewritten. Below are described the two classes of assembly optimizations performed:

- Leveraging instructions that are tailored to the specific application domain. For instance, the *cmpb4* instruction, which is a SIMD instruction, can concurrently compare four bytes. This is particularly advantageous when dealing with DNA strings, as it enables efficient comparison on multiple base pairs simultaneously.
- Fused jump instructions involving the combination of arithmetic or logical operations with control flow. Fused jump instructions enhance computational efficiency and reduce the need for separate branching instructions.

These manual optimizations, when applied to the code updating anti-diagonals, resulted in a total of 38% improvement in DPU's code performance.

5 EXPERIMENTATIONS

This section outlines the experiments conducted on an UPMEM PiM server. We evaluate the performance by conducting a comparative analysis against an OpenMP multi-threaded CPU implementation sourced from the minimap2 GitHub repository. This implementation, shared with the KSW2 library, is vector-optimized with SSE instructions and does not include the seeding and chaining steps from the computation, only the N&W step. We are aware of mm2fast [13] claiming 2x speed up using AVX instructions, unfortunately the server described hereafter performed 20% worse. This might be partly explained by the lower CPU core frequencies while using AVX2 and AVX512 instructions on Intel 421X processors. The UPMEM server configuration used includes 256 GB of standard memory and 20 DIMMs of PiM memory, yielding a total of 2560 DPUs running at 350 Mhz. There are 2 sockets equipped with Intel Xeon 4215 CPUs, each housing 16 cores and operating at a peak frequency of 2.5 GHz. The CPU implementation is evaluated on two different servers: A) a dual socket server with the same CPUs as in the PiM server (Intel Xeon 4215, 32 cores operating at 2.5 Ghz, 11MB of L3 cache), and B) a dual socket server with Intel Xeon 4216 CPUs (64 cores operating at 2.1 Ghz, 22MB of L3 cache).

The evaluation is based on several datasets for comprehensive analysis. First, three synthetic datasets, referred in the rest of the paper as S1000, S10000 and S30000, featuring read lengths of 1000, 10000 and 30000 respectively, were generated using the data generator provided in the WFA GitHub repository[18]. Second, two real datasets were considered. The first one comprises 16S Ribosomal RNA sequences, sourced from the NCBI bacterial database (August 2022). Only complete 16S RNA sequences were retained through a curation process. The final dataset contains 9557 sequences. The second real dataset consists of 38,512 sets of PacBio raw reads. Each set is composed of 10 to 30 repeated reads of the same region, characterized by a high error rate and the presence of significant gaps (exceeding 100 bp). Within each set, an all-against-all alignment is performed to generate a consensus sequence. Note that consensus computation is not included in our benchmark.

The evaluations have been done with the number of pools P set to 6, and the number of tasklets per pool T fixed to 4. These parameters values have demonstrated effective pipeline usage across all datasets, achieving utilization rates ranging from 95% to 99%. This implies that, during the DPU execution, the performance is

influenced by internal MRAM-WRAM transfers only to a small extent, with an impact ranging from 1 to 5%. The overall overhead of the host orchestration of DPUs, including pre-processing, transfers to and from DPUs, and post-processing of data compared to the running times of the DPUs is dependent on the dataset. For the smaller reads found in S1000, the overhead reaches 15% while for the longer reads found in S30000, the overhead is less than 0.1%.

5.1 Adaptive vs. Static: Accuracy and Computation

We first look at the execution time of the adaptive N&W formulation against minimap2, which, we recall, implements a highly efficient CPU implementation of the static N&W algorithm. Notably, minimap2 optimizes CPU efficiency through the utilization of query sequence profile[22], a branchless programming strategy, coupled with SSE vectorization to achieve optimal performance. In the context of the DPU implementation, branches are inexpensive and memory usage is the priority, as it is limited. Thus we opted for an adaptive formulation of N&W that does not use a query profile. As stated in section 3, this choice has implications in terms of accuracy and performance. Accuracy, in this context, refers to the precision of sequence alignment, measured as the percentage of correctly aligned sequences in a specific dataset. To establish a baseline for correct alignments, we leveraged minimap2 while disabling the band heuristic, thereby obtaining optimal alignments for reference.

Table 1: Comparison of accuracy percentages across datasets, for static and adaptive band heuristics. The band size is doubled until reaching 100% accuracy on synthetic datasets.

	Static			Adaptive
Band size	128	256	512	128
S1000 (%)	100			100
S10000 (%)	99	100		100
S30000 (%)	89	99	100	100
16S (%)	70	81	85	86
Pacbio (%)	29	62	87	85

As we can see in Table 1, which presents a detailed comparison of accuracy percentages of both static and adaptive band heuristics, the accuracy of the adaptive heuristic is much higher especially for long sequences. We can reduce the band size by four times, thus reducing the amount of computation by the same rate, and retaining the same accuracy. The performance measurement, in the rest of the paper, is assessed under a given accuracy constraint, i.e. the evaluation considers the performance of adaptive and static heuristics with the minimum band size required to provide a given level of accuracy (e.g., 100% or 85%). Note that the associated band size necessary to provide this level of accuracy can be different between the two different heuristics.

5.2 Synthetic datasets pair alignment

The datasets, denoted as S1000, S10000, and S30000, consist of 10 million, 1 million, and 500 thousand pairs of reads for alignment, respectively. As previously mentioned, S1000 consists of reads approximately 1000 bp in size, S10000 features 10,000 bp long reads, and S30000 features 30,000 bp long reads. This dataset being organized by pairs of reads to align, each DPU receives one pair of reads for each alignment to perform. This characteristic makes this setup the most challenging in terms of communication between the host and DPUs when compared to other use cases. Indeed, in other datasets, reads are organized in sets where each read in the set must be aligned with every other read. This means the number of alignments to be performed is quadratic with the number of reads in the set. This therefore increases the ratio of computation to communication.

Table 2: Runtime on the S1000 dataset at 100% accuracy.

	S1000	
	Time (in s)	Speedup
Minimap2 Intel 4215 (32c)	294	1
Minimap2 Intel 4216 (64c)	242	1.2
DPU 10 ranks	560	0.6
DPU 20 ranks	283	1
DPU 40 ranks	146	2

Table 2 shows the comparison between DPU and minimap2 on the S1000 dataset. For this dataset, minimap2 reaches 100% accuracy at a band size of 128. Since the DPU algorithm also requires a band size of 128 for 100% accuracy, the CPU and DPUs perform a similar amount of computation. We can see that on this dataset, the scaling of Minimap2 with an increasing number of cores is quite poor. Indeed, the performance is only 20% better on the Intel 4216 although it has twice as many cores than the 4215. On the other hand, the scaling of the PiM system is quite good, as the performance is nearly doubled when doubling the number of ranks from 20 to 40.

Table 3: Runtime on the S10000 dataset at 100% accuracy.

	S10000	
	Time (in s)	Speedup
Minimap2 Intel 4215 (32c)	744	1
Minimap2 Intel 4216 (64c)	369	2
DPU 10 ranks	502	1.5
DPU 20 ranks	255	2.9
DPU 40 ranks	132	5.6

Table 3 shows the comparison on the S10000 dataset. To maintain 100% accuracy, Minimap2 needs to be configured with a band size of 256, while the DPU still requires a band size of 128. This implies that the CPU is effectively computing twice the number of cells in the DP matrices compared to the DPU. With longer reads, the efficiency of the adaptive heuristic becomes apparent. For this dataset, the pattern favors the second CPU configuration, demonstrating performance twice that of the first CPU configuration. Surprisingly, the Intel 4216 CPU performs twice as fast as the Intel 4215 CPU on

this dataset. Again, we see a good scaling of the DPU implementation, it is almost linear with an increasing number of DPUs (a speedup of 3.8 between 10 to 40 ranks).

Table 4: Runtime on the S30000 dataset at 100% accuracy.

	S30000	
	Time (in s)	Speedup
Minimap2 Intel 4215 (32c)	1650	1
Minimap2 Intel 4216 (64c)	1265	1.3
DPU 10 ranks	755	2.1
DPU 20 ranks	391	4.2
DPU 40 ranks	200	8

For the S30000 dataset, the band size of Minimap2 is set at 512 to reach 100% accuracy. We see similar results in Table 4 in strong scaling, the speedup being 3.7 from 10 ranks to 40 ranks. The comparison between the Intel 4215 and Intel 4216 shows that Minimap2 struggles to scale efficiently on that dataset.

In summary, the experimental results on the synthetic datasets underscore the robust performance of the DPU. Notably, the efficacy of the DPU implementation becomes more pronounced with an increase in read size. This increase in performance can be attributed to the adaptive heuristic, which proves particularly advantageous when dealing with longer reads. In addition, we demonstrate that the PiM architecture scales well with an increasing number of DPUs.

5.3 16S RNA sequence comparison for phylogeny

This experiment seeks to compute an all-against-all comparison between the sequences essential for phylogeny analysis. It involves pairwise sequence alignment within the 16S Ribosomal RNA dataset. In particular, each alignment yields a score without necessitating the alignment itself (CIGAR).

Given the dataset's compact size, which allows it to reside within the MRAM of one DPU, a broadcast mechanism is used to send it to all DPUs. This approach effectively diminishes the pre-processing requirements involved in generating and sending each pair of sequences, and limits the data transfer footprint. Subsequently, each DPU is statically assigned the responsibility of computing a subset of alignments, with each DPU being tasked with the same number of alignments. This simple static assignment of work enables good load balancing for this experiment, with a maximum difference in the execution time of DPUs of the same rank around 5%.

As presented in Table 5, a fully populated UPMEM PiM server outperforms two server-grade CPUs by a substantial margin. Specifically, the PiM server achieves a remarkable 9.3-fold improvement compared to the first CPU configuration. The execution time shows a linear reduction as the number of ranks increases, thus highlighting robust scaling. This linear scaling is possible thanks to the low exchange overhead between the host and the DPU, the dataset being broadcast only once.

Table 5: Comparison of performance on the 16S dataset. The performance is being analysed at accuracy greater than 85% (band size at 512 for minimap2 and 128 on DPU).

	16S	
	Time (in s)	Speedup
Minimap2 Intel 4215 (32c)	5882	1
Minimap2 Intel 4216 (64c)	3538	1.7
DPU 10 ranks	2544	2.3
DPU 20 ranks	1257	4.6
DPU 40 ranks	632	9.3

5.4 Long read alignment for consensus sequence

The objective of this experiment is to align sets of sequences. In each set, individual reads or DNA segments, sequenced from the same region of the genome, are pairwise aligned. This process constitutes the foundational step in constructing a consensus sequence. It is important to note that, in this context, the CIGAR string, which represents alignment, is an indispensable part of the analysis.

The distribution of sets to the DPUs follows the systematic approach of load balancing described in 4.1. In practice, this approach has proved effective in achieving a balanced distribution of the calculation burden across all DPUs.

Table 6: Comparison of the computation time on the Pacbio dataset. Performance was analyzed with an accuracy of over 85%. (band size at 512 for minimap2 and 128 on DPU).

	Pacbio	
	Time (in s)	Speedup
Minimap2 Intel 4215 (32c)	4044	1
Minimap2 Intel 4216 (64c)	2788	1.4
DPU 10 ranks	1882	2.1
DPU 20 ranks	956	4.2
DPU 40 ranks	505	8

Once again, the performance comparison was conducted against the two CPU configuration described in section 5. The UPMEM PiM server consistently outperforms server-grade CPUs by a factor of up to 8, as detailed in Table 6. The task of attaining a good load balancing among DPUs poses a more complex challenge, however the scaling demonstrates remarkable robustness, even with numerous ranks. A minor reduction in performances is observed only when the number of ranks reaches 40.

5.5 Impact of using specific instructions

All benchmarks above have been done using an implementation that contains 26 lines of assembly code for better performance. Those instructions include a cmpb4, which is the only vector instruction of the ISA, that allow the comparison of 4 bytes in one cycle. An other instruction used is a right shift fused with a jump on parity, this allow efficient retrieval and usage of the results from the cmpb4 instruction. Note that the above instructions cannot be targeted by the compiler at the moment. The rest of the assembly is needed due

to the lack of support of C labels inside the inline assembly in the compiler.

Table 7: Speed up of manually optimised vs pure C DPU kernels.

	Datasets				
	S1000	S10'000	S30'000	16S	Pacbio
DPU pure C (sec)	247	207	316	864	806
DPU asm (sec)	146	132	200	632	505
Speed up	1.69	1.57	1.58	1.36	1.59

In Table 7, we can see that manually optimizing the most inner loop of computation can lead to an increase in performance of up to 1.69x. The 16S dataset has the smallest performance increase, this is explained by the fact that it does not compute the traceback, thus there is less to be optimized.

5.6 Power efficiency and cost

To estimate the power efficiency gain, we gathered the power data for the different system part from their specifications and followed the methodology proposed in [8]. System parts includes CPU, DIMMs, chassis, fans and PSU. The Intel 4215 server is estimated as requiring 307W, while the Intel 4216 is estimated to require 337W. The addition of 20 PiM DIMMs requires an additional 460W of power to the Intel 4215, which bring the UPMEM PiM server at 767W. To compute the power per execution, we multiplied the power consumption by the time of execution. As we can see from Table 8, the PiM server consume from 2.4 to 3.7 less power for computing all the alignment of the two real world dataset.

Table 8: Power consumptions of the 40 ranks PiM server compared to Intel servers on both real datasets (in kJ).

	16S	Pacbio
Intel 4215 (kJ)	1805	1241
Intel 4216 (kJ)	1192	939
UPMEM PiM (kJ)	484	387

The cost of the Intel Xeon 4216 server is 11k euros and the current cost of the additional PiM DIMM is 9k euros. We can draw from Table 6 a speed up of 5.5x for an Intel 4216 system with PiM DIMM compared to one without it. This means that to increase the computational power of the server by 5.5x, only an increase of 1.8x in term of cost is necessary. Furthermore, during PiM operations, most of the cores are free to be working on other tasks. Looking ahead, future study could explore heterogeneous computation using both PiM and CPU simultaneously. Using the CPU computational power would add an even greater incentive to integrate a PiM solution.

6 DISCUSSION

In this paper, we demonstrate the first long read alignment library on a real Processing-in-Memory device, going further than previous works that were limited to small reads. It implements a variation of Needleman and Wunsch (N&W), the adaptive banded, most suited for the constraints and strengths of the UPMEM PiM device. We introduce the use of device specific instructions and their impact on performance. We show that our implementation compares favorably against the Minimap2 implementation of N&W, the fastest implementation for our hardware (see section 5), while giving similar results. Our performance gains (up to 9.3x) also translate to power efficiency gains, showing that PiM can effectively reduce the power consumption of read alignment.

Processing-in-Memory architectures fundamentally target accelerating data-intensive applications. With our experiments, we show that a PiM architecture can challenge x86 architectures in the realm of sequence alignment. These results are particularly encouraging considering the fact that UPMEM's PiM system is a new hardware and performance of future PiM device should improve (e.g., frequency increase, DDR5, etc.). More generally, PiM is an emerging paradigm, and we can expect more mature PiM architectures to reach the market in the near future. We believe that this should motivate further work in studying the implementation of different memory-bounded genomics algorithms on PiM systems.

Innovations in CPU have mostly targeted bigger vector units, faster frequencies and higher core counts. PiM on the other hand can bring new benefits for different needs. As described in Section 5.5 even simple instructions are sufficient to improve performance by a substantial margin. Algorithms such as N&W, depend in part on simple instructions such as comparisons, jumps, shifts, etc. The multi-thread design used in UPMEM PiM is also efficient mechanism to saturate the compute unit pipeline without too much effort.

As future work, we plan to expand this work toward mapping reads using PiM, another important application in genomics analysis. In the current work, dispatching reads was a question of load balancing. Mapping is an application where reads must be send to specific DPUs, thus the challenge for homogeneous computations on the same rank will be greater.

ACKNOWLEDGMENTS

This research was funded, in whole or in part, by the French GenoPIM project (ANR-21-CE46-0012) and the BioPIM European Union's Horizon Europe programme for research and innovation under grant agreement No. 101047160.

REFERENCES

[1] Andy S. Alic, David Ruzafa, Joaquin Dopazo, and Ignacio Blanquer. 2016. Objective review of de novo stand-alone error correction methods for NGS data. *WIREs Computational Molecular Science* 6, 2 (2016), 111–146. https://doi.org/10.1002/wcms.1239 arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1239

[2] Rebecca J. Case, Yan Boucher, Ingela Dahllöf, Carola Holmström, W. Ford Doolittle, and Staffan Kjelleberg. 2007. Use of 16S rRNA and rpoB Genes as Molecular Markers for Microbial Ecology Studies. *Applied and Environmental Microbiology* 73, 1 (2007), 278–288. https://doi.org/10.1128/AEM.01177-06 arXiv:https://journals.asm.org/doi/pdf/10.1128/aem.01177-06

[3] Mark J. Chaisson and Glenn Tesler. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application

and theory. *BMC Bioinformatics* 13, 1 (19 09 2012), 238. https://doi.org/10.1186/1471-2105-13-238

[4] Haoyu Cheng, Gregory T. Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. 2021. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods* 18, 2 (01 02 2021), 170–175. https://doi.org/10.1038/s41592-020-01056-5

[5] Jeff Daily. 2016. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics* 17, 1 (10 02 2016), 81. https://doi.org/10.1186/s12859-016-0930-z

[6] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[7] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. 2023. A Framework for High-throughput Sequence Alignment using Real Processing-in-Memory Systems. *Bioinformatics* (03 2023). https://doi.org/10.1093/bioinformatics/btad155 arXiv:https://academic.oup.com/bioinformatics/advance-article-pdf/doi/10.1093/bioinformatics/btad155/49650358/btad155.pdf btad155.

[8] Yann Falevoz and Julien Legriel. 2024. Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture. In *Euro-Par 2023: Parallel Processing Workshops*. Springer Nature Switzerland, Cham, 155–166.

[9] Michael Farrar. 2006. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2 (11 2006), 156–161. https://doi.org/10.1093/bioinformatics/btl582 arXiv:https://academic.oup.com/bioinformatics/article-pdf/23/2/156/49820591/bioinformatics_23_2_156.pdf

[10] Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162, 3 (1982), 705–708. https://doi.org/10.1016/0022-2836(82)90398-9

[11] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. 2019. RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED.2019.8824830

[12] P. Hogeweg and B. Hesper. 1984. The alignment of sets of sequences and the construction of phyletic trees: An integrated method. *Journal of Molecular Evolution* 20, 2 (01 06 1984), 175–186. https://doi.org/10.1007/BF02257378

[13] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. 2022. Accelerating minimap2 for long-read sequencing applications on modern CPUs. *Nature Computational Science* 2, 2 (01 02 2022), 78–83. https://doi.org/10.1038/s43588-022-00201-8

[14] Roman Kaplan, Leonid Yavits, and Ran Ginosasr. 2020. BioSEAL: In-Memory Biological Sequence Alignment Accelerator for Large-Scale Genomic Data. In *Proceedings of the 13th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '20)*. Association for Computing Machinery, New York, NY, USA, 36–48. https://doi.org/10.1145/3383669.3398279

[15] Mikhail Kolmogorov, Derek M. Bickhart, Bahar Behsaz, Alexey Gurevich, Mikhail Rayko, Sung Bong Shin, Kristen Kuhn, Jeffrey Yuan, Evgeny Polevikov, Timothy P. L. Smith, and Pavel A. Pevzner. 2020. metaFlye: scalable long-read metagenome assembly using repeat graphs. *Nature Methods* 17, 11 (01 11 2020), 1103–1110. https://doi.org/10.1038/s41592-020-00971-x

[16] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (05 2018), 3094–3100. https://doi.org/10.1093/bioinformatics/bty191 arXiv:https://academic.oup.com/bioinformatics/article-pdf/34/18/3094/25731859/bty191.pdf

[17] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 25, 14 (05 2009), 1754–1760. https://doi.org/10.1093/bioinformatics/btp324 arXiv:https://academic.oup.com/bioinformatics/article-pdf/25/14/1754/48994219/bioinformatics_25_14_1754.pdf

[18] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. [n. d.]. . https://github.com/smarco/WFA-paper

[19] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. 2020. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics* 37, 4 (09 2020), 456–463. https://doi.org/10.1093/bioinformatics/btaa777 arXiv:https://academic.oup.com/bioinformatics/article-pdf/37/4/456/50359789/btaa777.pdf

[20] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[21] Muhammad Tariq Pervez, Mirza Jawad Ul Hasnain, Syed Hassan Abbas, Mahmoud F Moustafa, Naeem Aslam, and Syed Shah Muhammad Shah. 2022. A Comprehensive Review of Performance of Next-Generation Sequencing Platforms". *Biomed Res Int* 2022 (Sept. 2022).

[22] Torbjørn Rognes. 2001. ParAlign: A parallel sequence alignment algorithm for rapid and sensitive database searches. *Nucleic acids research* 29 (05 2001), 1647–52. https://doi.org/10.1093/nar/29.7.1647

[23] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

[24] Hajime Suzuki and Masahiro Kasahara. 2017. Acceleration of Nucleotide Semi-Global Alignment with Adaptive Banded Dynamic Programming. *bioRxiv* (2017). https://doi.org/10.1101/130633 arXiv:https://www.biorxiv.org/content/early/2017/09/07/130633.full.pdf[2]

[25] Hajime Suzuki and Masahiro Kasahara. 2018. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics* 19 (2018).

[26] Yatish Turakhia, Kevin Zheng, Gill Bejerano, and William Dally. 2017. Darwin: A Hardware-acceleration Framework for Genomic Sequence Alignment. *bioRxiv* (01 2017). https://doi.org/10.1101/092171

[27] Yunhao Wang, Yue Zhao, Audrey Bollas, Yuru Wang, and Kin Fai Au. 2021. Nanopore sequencing technology, bioinformatics and applications. *Nature Biotechnology* 39, 11 (01 11 2021), 1348–1365. https://doi.org/10.1038/s41587-021-01108-x

[28] M.S Waterman, T.F Smith, and W.A Beyer. 1976. Some biological sequence metrics. *Advances in Mathematics* 20, 3 (1976), 367–387. https://doi.org/10.1016/0001-8708(76)90202-4

[29] Haowen Zhang, Chirag Jain, and Srinivas Aluru. 2020. A comprehensive evaluation of long read error correction methods. *BMC Genomics* 21, 6 (21 12 2020), 889. https://doi.org/10.1186/s12864-020-07227-0

Received 2024; revised 2024; accepted 2024