BIMSA: Accelerating Long Sequence Alignment Using Processing-In-Memory

Alejandro Alonso-Marín[1,2]*, Ivan Fernandez[1,2], Quim Aguado-Puig[4], Juan Gómez-Luna[3], Santiago Marco-Sola[1,2], Miquel Moreto[1,2]

[1]Computer Sciences Department, Barcelona Supercomputing Center, Barcelona, 08034, Spain.
[2]Departament of Computer Science, Universitat Politècnica de Catalunya, Barcelona, 08034, Spain.
[3]NVIDIA Corporation, Santa Clara, California, US.
[4]Departament d'Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona, Barcelona, 08193, Spain.

*To whom correspondence should be addressed.

Abstract

Motivation: Recent advances in sequencing technologies pose sequence analysis algorithms as fundamental tools for genomics and healthcare research. In particular, sequence alignment is a fundamental building block present in numerous sequence analysis pipelines. Commonly used sequence alignment algorithms are based on dynamic programming and often require quadratic time and memory concerning the sequence length. As a result, classic sequence alignment algorithms fail to scale with increasing sequence lengths and quickly become memory-bound due to data-movement penalties.
Results: Processing-In-Memory (PIM) is an emerging architectural paradigm that seeks to accelerate memory-bound algorithms, bringing computation closer to the data to mitigate data-movement penalties. This work presents BIMSA (Bidirectional In-Memory Sequence Alignment), a PIM-optimized implementation of the state-of-the-art sequence alignment algorithm BiWFA (Bidirectional Wavefront Alignment), incorporating hardware-aware optimizations for a production-ready PIM architecture (UPMEM). BIMSA achieves speedups up to 22.24× compared to state-of-the-art PIM-enabled implementations of sequence alignment algorithms and up to 5.84× compared to the most efficient CPU implementation of BiWFA. In addition, BIMSA exhibits linear scalability with the number of compute units, enabling further performance improvements with upcoming PIM architectures equipped with more compute units.
Availability: Code and documentation will be publicly available after journal acceptance.
Contact: alejandro.alonso1@bsc.es

1 Introduction

The alignment of DNA, RNA, or protein sequences is essential for understanding various biological processes and identifying genetic variations (Alser *et al.* (2022); Churko *et al.* (2013); Roy *et al.* (2018)). These sequences are becoming larger in genomics due to advances in sequencing technologies, reaching up to 100 K bases in some cases (Reuter *et al.* (2015); Schloss (2008); Stephens *et al.* (2015); Frellsen *et al.* (2014); Petersen *et al.* (2019)).

Sequence alignment based on Dynamic Programming (DP), such as Needleman-Wunsch (Needleman and Wunsch (1970)) or Smith-Waterman (Waterman *et al.* (1976); Gotoh (1982)) algorithms, is widely used and effective for aligning short sequences. However, it presents certain challenges, especially when dealing with ultra-long sequences or large-scale data sets. Particularly, DP algorithms present a time and space complexity that grows quadratically with the length of the sequences being aligned, due to the creation and storage of large matrices to store partial solutions during the alignment process. Because of that, these algorithms are not able to scale when aligning large-scale sequence datasets generated by modern sequencing technologies. To face these challenges, Marco-Sola *et al.* (2021) introduced WFA, a novel algorithm that takes advantage of homologous regions between sequences to accelerate the alignment's computation. In essence, WFA computes optimal alignments in $O(ns)$ time and $O(s^2)$ memory, where n is the sequence length and s is the optimal alignment score.

While WFA reduces computational complexity and memory footprint, prior work (Diab *et al.* (2023)) has demonstrated that sequence alignment is a memory-bound application due to its low arithmetic intensity. As a result, existing hardware acceleration solutions (e.g., CPUs, GPUs or FPGAs) (Šošić and Šikić (2017); Daily (2016); Ahmed *et al.* (2019); Sadiq *et al.* (2020); Doblas *et al.* (2023); Aguado-Puig *et al.* (2022b); Balhaf *et al.* (2016); Aguado-Puig *et al.* (2022a); Haghi *et al.* (2021a, 2023, 2021b)) spend most of their execution time waiting for memory requests to be served, frequently leaving their powerful compute units idle. Recently, Marco-Sola *et al.* (2023) introduced BiWFA, which improves the original WFA algorithm by reducing compute time to $O(ns)$ and memory space to $O(s)$. However, our performance characterization of BiWFA reveals that it remains a memory-bound algorithm, especially when aligning long and noisy sequences.

© The Author 2024.

To tackle this problem, Processing-In-Memory (PIM) (Stone (1970); Mutlu *et al.* (2019a,b); Ghose *et al.* (2019); Mutlu *et al.* (2022)) is a paradigm that seeks to alleviate the data movement bottleneck in conventional platforms. The key idea behind PIM is to place compute units as close as possible to the location where data resides. There are already available PIM devices in the market that implement the PIM paradigm, such as UPMEM (UPMEM (2024)). This platform is based on modified DDR4 memory DIMMs that contain a general-purpose DRAM Processing Unit (DPU) next to each memory array. Each DPU can access up to 64MB of data with lower latency and larger bandwidth than a conventional architecture. Prior work has already explored UPMEM implementations of WFA (i.e., Alignment-In-Memory, Diab *et al.* (2023)). We analyze Alignment-In-Memory (AIM) library, the state-of-the-art UPMEM's implementation of WFA, and find that it is limited by the $O(s^2)$ memory requirements when aligning sufficiently long or noisy sequences.

Our **goal** in this work is to *enable fast alignment of sequences of heterogeneous lengths*, ranging from short sequences to ultra-long sequences. To this end, we present BIMSA (Bidirectional In-Memory Sequence Alignment), the first PIM-enabled memory-efficient alignment implementation which exploits the linear memory features of BiWFA to efficiently perform sequence alignment in the PIM units of the UPMEM platform. In summary, this work makes the following contributions:

- We thoroughly characterize state-of-the-art implementations of BiWFA (Marco-Sola *et al.* (2023)) and find that they are bottlenecked by memory in conventional platforms.
- We thoroughly characterize Alignment-In-Memory (AIM) library implementation of WFA (Diab *et al.* (2023)) and find that memory footprint prevents PIM from aligning long sequences.
- We design and implement a PIM-enabled version of BiWFA tailored for the UPMEM architecture, incorporating PIM-aware optimizations to maximize the application's performance.
- We comprehensively evaluate BIMSA in terms of (1) parameter exploration and (2) comparison over state-of-the-art solutions. We demonstrate that it provides up to 22.24× and 5.84× better performance than AIM library and BiWFA-CPU (Marco-Sola *et al.* (2023)), respectively.
- We discuss PIM's challenges when processing heterogeneous real datasets and provide solutions to improve balance in the PIM threads executions.

2 Background

2.1 Sequence Alignment Algorithms

Given two sequences, being Q a query and R a reference of length n and m respectively, classical DP Needleman-Wunch implementations calculate the score of aligning all suffixes between two sequences stored in a $n \times m$ matrix. Calculating the DP-matrix results in a $O(n \times m)$ complexity in compute and space. The operation that represents the calculation of a single DP-matrix cell $M_{i,j}$ is represented in the Equation 1.

$$M_{i,j} = \min \begin{cases} M_{i-1,j-1} + match/missmatch \\ M_{i,j-1} + insert \\ M_{i-1,j} + delete \end{cases} \quad (1)$$

The WFA introduced by Marco-Sola *et al.* (2021) is a sequence alignment algorithm that computes cells of the DP-matrix in increasing order of score. It utilizes a data structure W called the *wavefront*, which keeps track of the most advanced cell on each diagonal of a certain score. Wavefronts dynamically expand as the alignment progress continues until its completion. Their computation involve two main steps, known as *Compute Wavefront* and *Extend Wavefront*:

(1) Compute Wavefront. This step determines each new value of the next wavefront by evaluating the current minimum value among adjacent cells. Each cell calculation follows Equation 2.

$$W_{d+1,k} = \min \begin{cases} W_{d,k-1} + 1 & \text{Insertion} \\ W_{d,k} + 1 & \text{Missmatch} \\ W_{d,k+1} & \text{Deletion} \end{cases} \quad (2)$$

(2) Extend Wavefront. This step comprises the computation of the Longest Common Prefix (LCP) for each diagonal of the wavefront. A single cell extend is represented in Equation 3.

$$W_{d,k} = W_{d,k} + LCP(Q\left[W_{d,k} - k \ldots n\right], R\left[W_{d,k} \ldots m\right]) \quad (3)$$

The WFA improves the classical DP-based algorithm's complexity from $O(n^2)$ to $O(ns)$ in computation and $O(s^2)$ in memory, where s represents the alignment distance. Nonetheless, as stated in Diab *et al.* (2023), WFA encounters challenges in terms of data movement between memory and compute units, categorizing it as a memory-bound algorithm.

BiWFA represents an improvement over the original WFA designed to address memory challenges by reducing the memory complexity from $O(s^2)$ to $O(s)$ while retaining computational complexity at $O(ns)$. The key idea behind BiWFA is to concurrently align the sequences forward and backward-wise using WFA until the two alignments converge. This convergence point is referred to as the *breakpoint*, which is a coordinate of the optimal alignment path. The initial breakpoint provides an operation for the optimal alignment and the distance between the two sequences.

2.2 BiWFA Performance Limitations

The state-of-the-art BiWFA implementation for CPU is WFA2lib (Marco-Sola *et al.* (2021)). To identify its bottlenecks on a server-class CPU, we run experiments aligning four representative datasets produced by real sequencers composed of sequences with different lengths and errors. We perform this characterization on a dual-socket server equipped with two Intel Xeon Silver 4215 CPUs running at 2.50 GHz and DDR4 memory. Each CPU socket has 8 cores and 2 threads per core (a total of 16 hardware threads). Each core is equipped with an L1d and L1i cache (32 KB each), an L2 cache (1 MB), and a shared L3 cache (11 MB).

Figure 1 shows the scalability of BiWFA using a different number of threads and a range of real datasets with different sequence lengths and error rates. The gray dashed line represents the ideal speedup relative to the sequential version. We observe that WFA2lib's BiWFA provides poor scaling compared to the ideal and linear speedup for all datasets. We consider `PacBio.CSS` as a representative example of this limitation, since using 16 threads merely provides $\approx 4\times$ speedup to sequential.

We investigate the cause of BiWFA's poor scaling in CPUs by analyzing the percentage of execution cycles wasted on memory stalls in Figure 2, using the same experimental setup as in Figure 1. In the worst-case scenario, we observe up to 22% of cycles spent on memory stalls, while in the best-case scenario, this percentage drops to 4% (matching the worst and best scaling datasets, respectively). Table 1 provides the percentage of L1 and LLC misses, along with the latency cycles of each load miss and the average number of parallel misses (MLP) for the 16 threads executions. We conclude that the high *Load Miss Latencies* and *LLC Miss Ratios* lead BiWFA to be bottlenecked by data movement.

Key Observation: BiWFA presents a high ratio of costly last level cache misses that prevents scalability. As a consequence, BiWFA is memory-bound in conventional platforms.

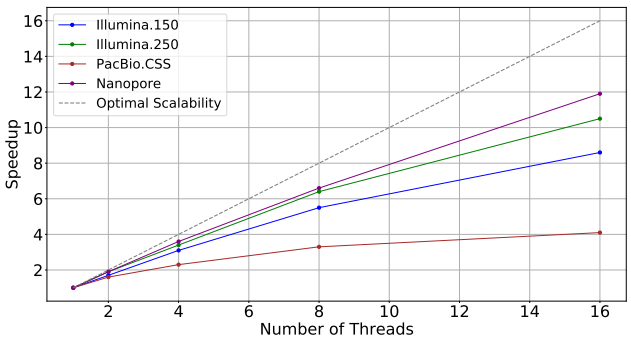

Fig. 1: Scalability of WFA2lib's BiWFA when using a different number of threads on a server with two Intel Xeon Silver 4215 CPUs and DDR4.

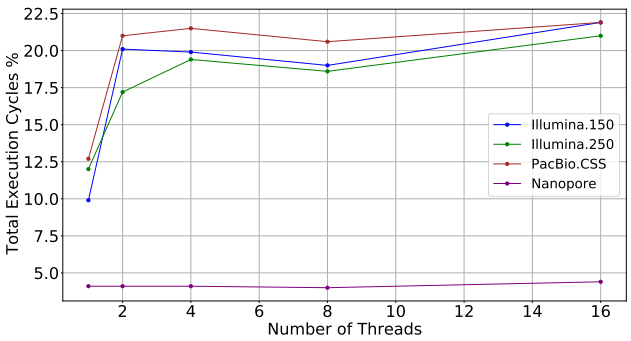

Fig. 2: Percentage of execution cycles spent on memory stalls of WFA2lib's BiWFA when varying the number of threads in the same server as Fig. 1.

Table 1. Miss Level Parallelism, Load Miss Latency, and First/Last Level Cache Miss Percentages of WFA2lib's BiWFA when dealing with real datasets.

Dataset	MLP	Load Miss Latency (cycles)	L1 Miss %	LLC Miss %
`Illumina.150`	7.2	56.4	1.30	95.26
`Illumina.250`	9.2	75.0	1.80	73.22
`PacBio.CSS`	9.8	38.0	8.00	86.81
`Nanopore`	3.6	7.9	0.90	85.73

2.3 Processing-In-Memory Paradigm

Classical *processor-centric* architectures are usually equipped with caches that aim to alleviate the gap between processor and memory speeds. Cache hierarchies effectively hide long-latency memory accesses when the number of operations per byte (i.e., arithmetic intensity) is high.

However, when arithmetic intensity is low, data movement between CPU and memory becomes a bottleneck. To mitigate this bottleneck, the Processing-In-Memory (PIM) (Mutlu *et al.* (2022)) paradigm proposes placing compute units as close as possible to where data is stored, offering higher bandwidth, lower latency, and also lower energy consumption compared to conventional platforms. Although PIM was introduced in the 70s by Stone (1970), commercial PIM platforms became feasible from a technology point of view. Examples of them are UPMEM (Gómez-Luna *et al.* (2022)), Samsung HBM-PIM (Kwon *et al.* (2021); Lee *et al.* (2021)), Samsung AxDIMM (Ke *et al.* (2021)), SK Hynix AiM (Lee *et al.* (2022)), and Alibaba HB-PNM (Niu *et al.* (2022)).

2.4 UPMEM PIM Platform Architecture

UPMEM is a general-purpose PIM platform intended to be integrated on commercial Intel Xeon-based servers. Figure 3 shows a high-level view of our UPMEM setup, showing the different memories and how the pipeline is structured. This platform is composed of x86-based host CPUs, *regular* DDR4 main memory DIMMs, and PIM-enabled DDR4 DIMMs.

Each PIM-enabled Memory DIMM is composed of several PIM Chips. Each PIM Chip comprises DRAM Processing Units (DPUs), which can be regarded as small general-purpose 32-bit RISC processors that provide 24 hardware threads. As a consequence, UPMEM provides two degrees of parallelism: DPUs and threads. Each DPU includes a DRAM bank of 64 MB (MRAM) ❶, 24 KB of Instruction RAM (IRAM) ❷, 64 KB of Working RAM (WRAM) ❸, a register file ❻ and a compute pipeline of 14 stages ❼. Each PIM Chip can be accessed from the host via the DDR4 interface, both for data transferring and compute/control with the Control/Status Interface. When data is transferred to PIM-enabled Memory, it is stored in its MRAM. DPUs are not able to operate directly over the MRAM data, so the DMA Engine moves compute data to the WRAM and the instructions to operate over that data to the IRAM.

In a typical PIM execution flow, the initial step involves reading a block from MRAM, with the block size specified by the programmer. Subsequently, this block is allocated within a pre-defined WRAM data structure. Following the WRAM data allocation, the processor gains access to the WRAM data as an array and can perform computations as needed in the DPU pipeline. Upon completion of the computation, the processed data is written back to MRAM in the same block size. It is important to note that all data transfers to MRAM must be 8-byte aligned.

Current UPMEM platforms work as accelerators, which means that transfers from CPU to UPMEM DIMMs and vice versa are needed. Additionally, DPUs lack direct communication between them. Based on that, communication between DPUs (even if they belong to the same chip) must be *slowly* orchestrated through the host CPU (❹/❺). This fact greatly impacts performance in applications where communication is frequent. However, upcoming UPMEM DIMMs are expected to function as the main memory, removing the need for CPU-DPU transfers.

Key Opportunity: UPMEM platform is a good candidate for applications that (1) have low arithmetic intensity, (2) perform simple operations, and (3) do not require inter-DPU communication.

3 Method

3.1 BIMSA Overview

Sequence alignment applications usually involve comparing thousands or millions of sequence pairs. Each of these alignments can be computed independently, using their own working memory space. Therefore, BIMSA follows a coarse-grain parallelization scheme, assigning one or more sequence pairs to each DPU thread. This parallelization scheme is the best fit when targeting the UPMEM platform, as it removes the need for thread synchronization or data sharing across compute units.

We discard a collaborative parallelization scheme targeting the UPMEM platform since (1) thread synchronization and communication operations are costly (1000s of cycles), (2) communication between compute units has to be done through the host CPU, and (3) BiWFA's parallelism increases progressively during the alignment of a pair of sequences. Therefore, at the beginning of every alignment computation, there is minimal parallelism, which may underutilize the platform.

Adapting BiWFA to the UPMEM programming paradigm poses specific challenges, primarily due to its recursive nature and the limited stack size of DPUs. To tackle these challenges, we opt for unrolling

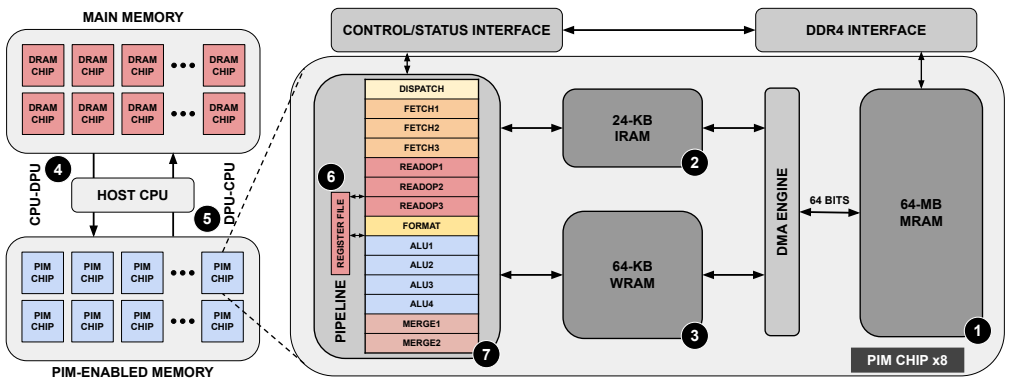

Fig. 3: Typical organization of an UPMEM-based server including one or several host CPU(s), main memory, and PIM-enabled memory (left). The internal components of a PIM chip are depicted on the right based on Gómez-Luna *et al.* (2022).

BiWFA's recursion completely by (1) segmenting sequences using breakpoints, (2) generating additional BiWFA sub-alignments through iteration until a threshold is reached, and (3) generating WFA sub-alignments to finalize the alignment process.

This approach is feasible since we leverage two key properties of BiWFA. First, the breakpoint iteration identifies a coordinate along the optimal path where sequences can be effectively split, separating them into two sections with balanced error. Second, each breakpoint provides the distance for the iteration alignment.

Based on that, we employ the BiWFA algorithm to iterate through sequences, breaking them down into sub-problems until the distance of the sub-problem aligns with our allocated WRAM. These sub-problems, known as *base cases* in our proposal, employ the regular WFA algorithm in which memory space is dictated by the error rate ($O(s^2)$). Thus, we can precisely determine the memory required for a regular WFA within a sub-problem. Once we reach a suitable error distance, we execute the WFA, yielding a partial alignment. Then, partial alignments are stored consecutively in MRAM building the complete one.

3.2 PIM-Aware Optimizations

BIMSA includes eight PIM-Aware optimizations that aim to fully utilize the UPMEM platform, which we describe next.

Irregular Workload Balancing. Sequence alignment datasets can present either homogeneous or heterogeneous sequence lengths and error distributions. For instance, Illumina-generated datasets often have a fixed sequence length (e.g., 136 to 164 bases) and a low maximum error (e.g., 5%). In contrast, datasets such as `PacBio.CSS` can contain sequence lengths ranging from 264 bases to 18218 bases and a wider error range (e.g. 0% to 10%).

The variety in alignment lengths and error rates rarely impacts CPU implementations in terms of load balancing since the number of processing units is usually low. However, it poses two main challenges for PIM implementations where there is no dynamic memory allocation and the number of processing units is high (i.e., thousands). First, MRAM data structure allocation depends on the maximum read length in the input file for BIMSA. Consequently, in cases such as `PacBio.CSS` with 1 million sequences, where the majority of sequences fall within the 100 to 10000 bases range, applications allocate space assuming the maximum sequence length. Second, restrictions on the granularity of parallelism result in application performance being limited by the slowest thread on the slowest DPU. This scenario becomes particularly problematic when a sequence requires more time than the average, leading to overall performance degradation.

We tackle the load balancing in UPMEM with three key mechanisms. First, we input a nominal distance limit for BiWFA wavefronts, which mitigates the MRAM allocation and consumption issues and also interrupts the execution when reached. This approach involves discarding the sequence alignment and flagging the sequence for CPU recovery execution once the PIM kernel completes its task. Second, we assign the alignments dynamically by performing minimal synchronization on a global alignment ID variable. Third, we execute the alignments in batches, allowing the CPU to start recovering the alignments of one batch while the PIM kernel starts computing the next batch.

WFA Kernel Fusion. There are two primary computation steps during WFA execution: the compute operation and the extend operation, typically performed sequentially. We merge them into a single operation, which minimizes the MRAM accesses needed for the steps. To unify these steps, we modify the original execution flow with two key mechanisms.

First, we note that the classical WFA compute implementation leverages loop unrolling to enable vectorization and incorporates boundary checks for computing the limits of the next wavefront. However, loop unrolling is not beneficial in UPMEM since this platform does not provide vectorization support. Because of that, we fuse the compute kernel loop into a single loop by setting the outer wavefronts boundaries to the minimum integer value and choosing the maximum value between the top, center, and bottom surrounding cells.

Second, we observe that in a UPMEM implementation based on the original WFA steps, the compute operation initiates from the lowest wavefront position, progressing upward to calculate the highest position. When the WRAM wavefront structure reaches full capacity, it writes the wavefront back to MRAM, fetching another block from MRAM to refill the cache. This process repeats until the entire wavefront is computed. Once the compute step is done, the extend operation unfolds similarly to the compute operation and traverses the wavefront data, fetching and storing the same data structures from and to MRAM.

This observation motivates us to merge the two kernels and minimize unnecessary MRAM accesses. Our combined compute+extend kernel efficiently reads a block from MRAM for both the current and next wavefronts to execute the compute operation. The compute operation proceeds as usual on the read blocks. Then, instead of writing the block back and moving on to compute the next one when a block is exhausted, we transition into the extend operation on that same block for the next wavefront. Once the extend operation is completed, we write back the block and load the subsequent one for the compute operation. This mechanism has no implications on the subsequent compute block and significantly diminishes the number of MRAM accesses for wavefronts

in the primary kernels by 40%. In the original scheme, the compute wavefront process involved loading two wavefronts, and writing one, while the extend operation consisted in reading one wavefront and writing one. In contrast, the compute+extend operation reads two wavefronts and writes one, effectively reducing the total MRAM accesses from five to three.

Storing Wavefronts in MRAM. We allocate WRAM space to store the wavefronts since they are one of the most accessed data structures and we need to access them as fast as possible. However, accessing MRAM becomes necessary in the breakpoint computation if the wavefronts surpass the allocated space in WRAM. Nevertheless, this flexibility enables us to calculate alignments for sequences up to 100K bases without diminishing the thread count (WRAM allocated space does not change with sequence length and is shared across threads), even though some performance is sacrificed due to MRAM reads. BIMSA only encounters size limitations in the MRAM memory when handling sequences and data structures that surpass 64 MB (i.e., MRAM maximum capacity) on each DPU.

BiWFA Base Cases Using WRAM. To determine the breakpoints, a minimum of four wavefronts is necessary, two for each endpoint of the sequence. As these data structures are unnecessary for the base cases, we repurpose the allocated WRAM initially designated for these wavefronts to perform the WFA. Because of this, the base case memory requirements are calculated to fit into our predefined scratchpad sizes, eliminating the need for MRAM reads and writes during their computation. This involves storing all WFA wavefronts within the space allocated for the other four wavefronts. As a result, the fitting error distance for a base case is $4 \times \sqrt{s}$, ensuring that a base case can always be entirely computed in WRAM, regardless of the size and error rate of the initial sequence.

Base-Case Output Reordering. During base case computation, the alignment output may have variable byte sizes that do not necessarily align with eight-byte boundaries. The UPMEM programming paradigm requires eight-byte aligned accesses to MRAM. To solve this problem, we create auxiliary WRAM and MRAM data structures that store the base-case outputs and write one output after the previous one, already writing them in the right order and storing them consecutively in the MRAM data structure. This way, when all the base cases for a sequence are computed, the complete alignment output is ready to be written back to the CPU without any concatenation operation.

MRAM Reverse Reading. Early stages of BIMSA involve reversing the sequences on the CPU and then transferring the duplicated and reversed sequences to the DPUs. While this simplifies memory reads on the DPU side, it comes at the cost of doubling the sequence's MRAM usage. To tackle this challenge, we implement a reverse sequence reading mechanism. This approach mitigates the risk of hitting the MRAM limit when aligning large sequences and also minimizes transfer times.

Memory Align Functions. Reading a sequence in reverse retrieves WRAM blocks from the end to the start and loads data from MRAM in the opposite direction, moving from the last memory position to the initial one. However, a challenge arises due to the 8-byte alignment requirement. While UPMEM offers functions to align regular reads to MRAM, these functions are not applicable to reverse reading. Consequently, we develop aligned reading functions for both forward and reverse scenarios, ensuring adaptability to different data types for the wavefront beyond 32-bit signed integers. Moreover, we adjust the entire MRAM distribution to accommodate a block size, ensuring compatibility in cases where reverse reading may point to an MRAM address beyond the established limits.

Adaptative Wavefront Transfer. The wavefront structure naturally expands from the initial iteration to the final one until a breakpoint is identified, meaning that earlier iterations require less data movement than later ones. We maximize UPMEM's performance by specifying the size of data transfers from MRAM to WRAM at each iteration of the wavefront. To achieve this, we set the initial transfer size at 8 bytes and increment it by powers of two whenever the wavefront size approaches the transfer size. We stop incrementing when the transfer size matches the size of the WRAM wavefront data structure or we reach the UPMEM transfer limit (i.e., 2048 bytes). Furthermore, since we reinitialize the wavefronts from scratch each time we encounter a breakpoint, the transfer size resets to 8 bytes.

4 Results

4.1 Experimental Setup

We perform the experiments on a UPMEM node with the specifications described in Section 2.2. In addition to regular DIMM modules, this node is equipped with PIM-enabled DIMM modules, which have up to 2556 operative DPUs running at 350 MHz. Each DPU can handle up to 24 hardware threads and is equipped with an MRAM (64 MB), a WRAM (64 KB), and an IRAM (24 KB).

We generate eight simulated datasets for the experimental evaluation, following the methodology of WFA2lib (Marco-Sola *et al.* (2023)). Each simulated dataset is composed of sequences with a fixed length (i.e., ranging from 150, 1000, 10000, to 100000 bases) and error rate (i.e., 5% and 10%). We include the real datasets (i.e., from real genome sequencers) described in Section 2.2, which contain sequences with different lengths and error rates, representing diverse and unbalanced workloads.

We evaluate and compare the performance of BIMSA by selecting the already presented CPU application WFA2lib and the PIM state-of-the-art pairwise alignment application (AIM library, Diab *et al.* (2023)), which implements a gap-affine optimized Needleman-Wunch (NW) algorithm and the classical WFA algorithm for PIM. We report the average of 5 executions discarding the first execution as warm up. For the PIM implementation, we only account for the kernel time since data transfers are expected to be removed in future PIM system generations.

4.2 BIMSA Characterization Using Simulated Datasets

Figure 4 illustrates the performance, in alignments per second, achieved by the state-of-the-art applications and BIMSA across balanced workload datasets. The numbers on top of each bar represent the speedup needed to match the fastest application.

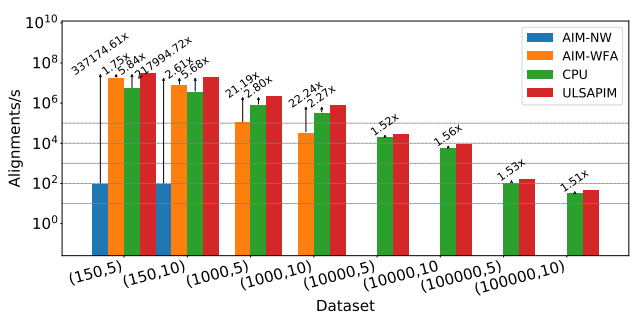

Fig. 4: Alignments per second achieved by AIM (PIM), WFA2lib (CPU), and BIMSA (PIM) when aligning balanced workload datasets. The datasets are refereed as (length,error%). Note that *y* axis is logarithmic.

First, we observe that AIM-NW provides poor performance compared to the rest of the evaluated applications and is not able to deal with large sequence lengths. This is expected since AIM-NW is based on a *traditional* DP algorithm, whose memory footprint is a fundamental limitation in PIM.

Second, we observe that BIMSA outperforms the PIM state-of-the-art (AIM-WFA) across all datasets, achieving up to 22.24× speedup in the best-case scenario. Additionally, while BIMSA handles sequences of

10000 bases or longer, the PIM state-of-the-art is unable to handle them due to insufficient memory allocation space.

Third, comparing BIMSA to the state-of-the-art CPU performance, we observe that BIMSA outperforms CPU implementations by up to 5.84× across all datasets. However, we note that as the sequence length and error percentage increase, BIMSA's performance degrades, achieving 1.50× speedup in the worst-case scenario. This is partly because the highly sequential WFA extend step becomes a major bottleneck.

Key Result I: BIMSA outperforms the state-of-the-art PIM implementations by up to 22.24× and state-of-the-art CPU by up to 5.84× when dealing with homogeneous simulated datasets.

In Figure 5, we examine the scalability of the same datasets as in Figure 4 across different numbers of DPUs. A dashed gray line denotes ideal scalability. We observe that BIMSA's scalability with the number of DPUs is nearly linear, while CPU implementations do not scale well when increasing the number of cores (as demonstrated in Section 2.2, Figure 1).

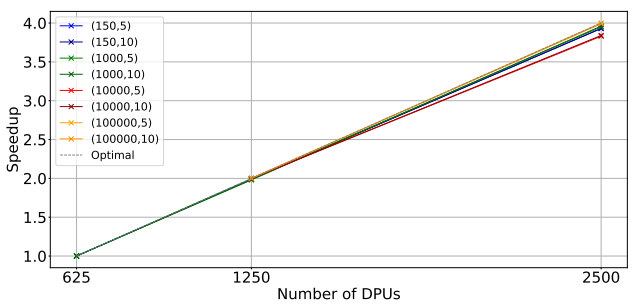

Fig. 5: Scalability with the number of DPUs for BIMSA with balanced workload datasets. The datasets are indicated by (length, error %).

Key Result II: It is expected that next-generation UPMEM systems will double the number of DPUs soon. Thanks to the linear scalability of BIMSA, this will translate into significant performance improvements.

In Figure 6, we analyze the scalability of BIMSA and the same datasets as in Figure 5 by varying the number of threads and fixing the number of DPUs to 2500. We make three observations from it.

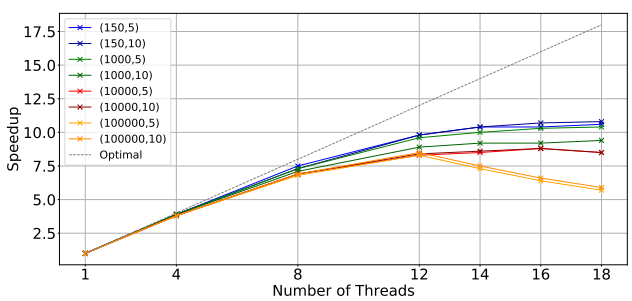

Fig. 6: Scalability with the number of threads for BIMSA with balanced workload datasets. The datasets are indicated by (length, error %).

First, we notice that BIMSA struggles to scale between 12 and the maximum number of threads. This is an expected behavior and matches a limitation of the UPMEM architecture. In this sense, prior work (Gómez-Luna *et al.* (2022)) has demonstrated that PIM applications that fully utilize resources typically scale only up to 11 threads. This limit arises because the DPU processor pipeline can dispatch only one instruction every 11 cycles, corresponding to its 11 pipeline stages.

Second, we find that as the sequence size increases, BIMSA's scalability deteriorates with a lower number of threads. This is attributed to the rise in MRAM accesses when increasing the sequence sizes. Since MRAM can only be accessed by one thread at a time, threads encounter a bottleneck in MRAM access, particularly noticeable with such larger sequence sizes.

Third, we can see how datasets with 100k bases per sequence not only lose scalability but also lose performance regarding the 12-thread execution. This is due to the number of pairs being insufficient to provide even work for all the threads, which shows how unbalanced work hinders UPMEM's performance.

4.3 BIMSA Characterization Using Real Datasets

Table 2 illustrates the execution time in seconds achieved by the state-of-the-art applications across different unbalanced workload datasets. In this analysis, BIMSA consists of two versions: first, an implementation that executes all the alignments in the DPUs (*BIMSA*), and second, an implementation that executes alignments in the DPUs up to a maximum distance threshold and then recovers in the CPU the alignments with higher distance (*BIMSA-Hybrid*). Note that this version includes the transfer times that are required for the CPU to be aware of which alignments have to recover.

We show that BIMSA outperforms the prior state-of-the-art PIM (AIM) across all scenarios, achieving up to 2.08× speedup and processing even the longest sequences datasets that the AIM library cannot. Regarding CPU performance, we observe that BIMSA provides similar performance for the `Illumina` datasets in both versions. Looking at BIMSA when processing long-sequence datasets, we can see how we outperform other CPU implementations aligning `PacBio.CSS` and have similar performance in `Nanopore`. In particular, the CPU recovery optimization (i.e., BIMSA + CPU) helps to reduce the DPU's execution time variability between alignments, making BIMSA able to outperform WFA2lib by 1.21× times in `PacBio.CSS` and 1.95× in `Nanopore`. Note that Illumina datasets are more homogeneous than `PacBio.CSS` and `Nanopore`, resulting in the same execution for BIMSA and BIMSA-Hybrid

Table 2. Execution time in seconds of AIM (PIM), WFA2lib (CPU), BIMSA (PIM) and BIMSA-Hybrid (PIM) when aligning unbalanced real datasets.

Application	`Illumina.150`	`Illumina.250`	`PacBio.CSS`	`Nanopore`
AIM-NW	53.5	149.5	N/A	N/A
AIM-WFA	1.2	2.2	N/A	N/A
WFA2lib	0.8	1.2	9.2	206.1
BIMSA	0.6	1.1	182.4	314.5
BIMSA-Hybrid	0.6	1.1	7.6	105.6

Key Result III: BIMSA outperforms state-of-the-art PIM by up to 2.08× and state-of-the-art CPU implementations by up to 1.95× when dealing with unbalanced datasets, thanks to its recovery mechanism.

5 Discussion

In this work, we introduce BIMSA, a PIM-based implementation of the BiWFA sequence alignment algorithm. BIMSA leverages the real-world general-purpose PIM architecture provided by UPMEM, and it is designed to address the data-penalty challenges that often hinder the scalability of prior sequence alignment implementations.

Our evaluation shows that, when aligning synthetic datasets, BIMSA achieves up to 22.24× compared to the state-of-the-art PIM-based implementation and 5.84× compared to state-of-the-art CPU

implementations. When aligning real datasets, BIMSA achieves speedups up to 2.08× compared with state-of-the-art PIM-based implementations and 1.95× compared against state-of-the-art CPU implementations.

Overall, BIMSA exhibits superior performance than prior state-of-the-art sequence alignment implementations when aligning datasets of different sequence lengths. While larger sequence lengths have an impact on performance, we observe ideal scalability with the number of compute units. At the same time, PIM-based implementations encounter significant challenges when handling heterogeneous datasets mainly due to load unbalance between compute units. This highlights the potential of PIM for optimizations and improvements in this area. Considering that PIM is in its early stages of technology development, BIMSA's results are very promising with plenty of room for improvements in the coming years.

Funding

We thank the UPMEM team for the infrastructure and technical support that greatly helped making possible this project. We also thank the BSC for the financing support.

References

Aguado-Puig, Q., Marco-Sola, S., Moure, J. C., Castells-Rufas, D., Alvarez, L., Espinosa, A., and Moreto, M. (2022a). Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm. *IEEE access*, **10**, 63782–63796.

Aguado-Puig, Q., Marco-Sola, S., Moure, J. C., Matzoros, C., Castells-Rufas, D., Espinosa, A., and Moreto, M. (2022b). Wfa-gpu: Gap-affine pairwise alignment using gpus. *bioRxiv*, pages 2022–04.

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

Alser, M., Lindegger, J., Firtina, C., Almadhoun, N., Mao, H., Singh, G., Gomez-Luna, J., and Mutlu, O. (2022). From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures. *Computational and Structural Biotechnology Journal*.

Balhaf, K., Shehab, M. A., Wala'a, T., Al-Ayyoub, M., Al-Saleh, M., and Jararweh, Y. (2016). Using gpus to speed-up levenshtein edit distance computation. In *2016 7th International Conference on Information and Communication Systems (ICICS)*, pages 80–84. IEEE.

Churko, J. M., Mantalas, G. L., Snyder, M. P., and Wu, J. C. (2013). Overview of high throughput sequencing technologies to elucidate molecular pathways in cardiovascular diseases. *Circulation research*, **112**(12), 1613–1623.

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

Diab, S., Nassereldine, A., Alser, M., Gómez Luna, J., Mutlu, O., and El Hajj, I. (2023). A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics*, **39**(5), btad155.

Doblas, M., Lostes-Cazorla, O., Aguado-Puig, Q., Cebry, N., Fontova-Musté, P., Batten, C. F., Marco-Sola, S., and Moretó, M. (2023). Gmx: Instruction set extensions for fast, scalable, and efficient genome sequence alignment. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1466–1480.

Frellsen, J., Menzel, P., and Krogh, A. (2014). Algorithms for mapping high-throughput dna sequences.

Ghose, S., Boroumand, A., Kim, J. S., Gómez-Luna, J., and Mutlu, O. (2019). Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, **63**(6), 3–1.

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of molecular biology*, **162**(3), 705–708.

Gómez-Luna, J., Hajj, I. E., Fernandez, I., Giannoula, C., Oliveira, G. F., and Mutlu, O. (2022). Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, **10**, 52565–52608.

Haghi, A., Marco-Sola, S., Alvarez, L., Diamantopoulos, D., Hagleitner, C., and Moreto, M. (2021a). An fpga accelerator of the wavefront algorithm for genomics pairwise alignment. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 151–159. IEEE.

Haghi, A., Marco-Sola, S., Alvarez, L., Diamantopoulos, D., Hagleitner, C., and Moreto, M. (2021b). An fpga accelerator of the wavefront algorithm for genomics pairwise alignment. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 151–159.

Haghi, A., Alvarez, L., Front, J., De Haro Ruiz, J. M., Figueras, R., Doblas, M., Marco-Sola, S., and Moreto, M. (2023). Wfasic: A high-performance asic accelerator for dna sequence alignment on a risc-v soc. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 392–401.

Ke, L., Zhang, X., So, J., Lee, J.-G., Kang, S.-H., Lee, S., Han, S., Cho, Y., Kim, J. H., Kwon, Y., *et al.* (2021). Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, **42**(1), 116–127.

Kwon, Y.-C., Lee, S. H., Lee, J., Kwon, S.-H., Ryu, J. M., Son, J.-P., Seongil, O., Yu, H.-S., Lee, H., Kim, S. Y., *et al.* (2021). 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 350–352. IEEE.

Lee, S., Kang, S.-h., Lee, J., Kim, H., Lee, E., Seo, S., Yoon, H., Lee, S., Lim, K., Shin, H., *et al.* (2021). Hardware architecture and software stack for pim based on commercial dram technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE.

Lee, S., Kim, K., Oh, S., Park, J., Hong, G., Ka, D., Hwang, K., Park, J., Kang, K., Kim, J., *et al.* (2022). A 1ynm 1.25 v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE.

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

Marco-Sola, S., Eizenga, J. M., Guarracino, A., Paten, B., Garrison, E., and Moreto, M. (2023). Optimal gap-affine alignment in o (s) space. *Bioinformatics*, **39**(2), btad074.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019a). Enabling practical processing in and near memory for data-intensive computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–4.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019b). Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, **67**, 28–41.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2022). A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pages 171–243. Springer.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, **48**(3), 443–453.

Niu, D., Li, S., Wang, Y., Han, W., Zhang, Z., Guan, Y., Guan, T., Sun, F., Xue, F., Duan, L., *et al.* (2022). 184qps/w 64mb/mm 2 3d logic-to-dram hybrid bonding with process-near-memory engine for recommendation system. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE.

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

Reuter, J. A., Spacek, D. V., and Snyder, M. P. (2015). High-throughput sequencing technologies. *Molecular cell*, **58**(4), 586–597.

Roy, S., Coldren, C., Karunamurthy, A., Kip, N. S., Klee, E. W., Lincoln, S. E., Leon, A., Pullambhatla, M., Temple-Smolkin, R. L., Voelkerding, K. V., *et al.* (2018). Standards and guidelines for validating next-generation sequencing bioinformatics pipelines: a joint recommendation of the association for molecular pathology and the college of american pathologists. *The Journal of Molecular Diagnostics*, **20**(1), 4–27.

Sadiq, M. U., Yousaf, M. M., Aslam, L., Aleem, M., Sarwar, S., and Jaffry, S. W. (2020). Nvpd: novel parallel edit distance algorithm, correctness, and performance evaluation. *Cluster Computing*, **23**(2), 879–894.

Schloss, J. A. (2008). How to get genomes at one ten-thousandth the cost. *Nature biotechnology*, **26**(10), 1113–1115.

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

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big data: astronomical or genomical? *PLoS biology*, **13**(7), e1002195.

Stone, H. S. (1970). A logic-in-memory computer. *IEEE Transactions on Computers*, **100**(1), 73–78.

UPMEM (2024). UPMEM Website. `https://www.upmem.com`.

Waterman, M. S., Smith, T. F., and Beyer, W. A. (1976). Some biological sequence metrics. *Advances in Mathematics*, **20**(3), 367–387.