

AMERICAN UNIVERSITY OF BEIRUT

ACCELERATING GENOME ANALYSIS  
USING PROCESSING-IN-MEMORY

by  
SAFAA YOUSEF DIAB

A thesis  
submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
to the Department of Computer Science  
of Faculty of Arts and Sciences  
at the American University of Beirut

Beirut, Lebanon  
April 2022

AMERICAN UNIVERSITY OF BEIRUT

ACCELERATING GENOME ANALYSIS  
USING PROCESSING-IN-MEMORY

by  
SAFAA YOUSEF DIAB

Approved by:

---

Dr. Izzat El Hajj, Assistant Professor  
Computer Science

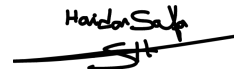
Advisor



---

Dr. Haidar Safa, Professor  
Computer Science

Member of Committee



---

Dr. Wassim El Hajj, Professor  
Computer Science

Member of Committee



Date of thesis defense: April 29, 2022

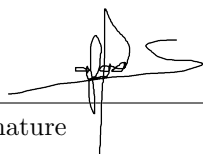
# AMERICAN UNIVERSITY OF BEIRUT

## THESIS RELEASE FORM

Student Name: Di ab Safaa Yousef  
Last First Middle

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of my thesis; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes

- As of the date of submission of my thesis
- After 1 year from the date of submission of my thesis .
- After 2 years from the date of submission of my thesis .
- After 3 years from the date of submission of my thesis .

  
Signature

May 9 2022

Date

# ACKNOWLEDGEMENTS

All praise is due to God.

My greatest thanks have to go to Professor Izzat El Hajj for the support and help he has given me not only throughout this thesis, but over the course of my entire graduate studies. I'm grateful for all that I've learned from him on both the academic and personal levels. This thesis wouldn't have happened without his guidance.

I would like to thank ETH Zürich SAFARI research group, especially Professor Onur Mutlu, Juan Gómez Luna, and Mohammed Alser for their continuous advice and help over the past year. Thanks also to the committee, Professor Haidar Safa and Professor Wassim El Hajj.

I would like to thank my parents, family, and friends for supporting me and giving me the motivation to carry on. I wouldn't be the person I am without their love and support. Finally, I would like to thank Jalal for all his love and support and for always being there when I needed him.

# ABSTRACT

## OF THE THESIS OF

Safaa Yousef Diab for Master of Science  
Major: Computer Science

Title: Accelerating Genome Analysis using Processing-in-Memory

Data movement between memory and the CPU is a bottleneck in data-intensive applications. The cause of this problem is the need for the computing unit to access data frequently in memory through a limited-bandwidth and high-latency memory bus. Processing-in-memory (PIM) architectures solve this problem by bringing computing units closer to the memory on the same memory chip. The aim of this thesis is to show that genome sequence analysis can be effectively accelerated using PIM architectures. We perform high-throughput read pair alignment using the Needleman-Wunsch, Smith-Waterman-Gotoh, GenASM, and the Wave Front Alignment algorithms on a real PIM architecture. The performance was evaluated in terms of speedup and energy against a server-grade multi-threaded CPU baseline. The results show that most PIM implementations can achieve higher throughput for different read lengths and edit distance thresholds. The state-of-the-art algorithm, WFA-adaptive, has up to  $2-3\times$  speedup for all datasets, even when aligning large reads of length 5Kbp and 10Kbp, and achieves even higher speedup when the CPU-DPU communication time is not included.

# TABLE OF CONTENTS

<b>Acknowledgements</b>	<b>1</b>
<b>Abstract</b>	<b>2</b>
<b>1 Introduction</b>	<b>7</b>
<b>2 Background</b>	<b>9</b>
2.1 UPMEM PIM Architecture . . . . .	9
2.2 Genome Sequencing Analysis . . . . .	11
2.3 Pairwise Alignment Algorithms . . . . .	13
2.3.1 Needleman-Wunsch . . . . .	13
2.3.2 Smith-Waterman-Gotoh . . . . .	14
2.3.3 Wavefront Alignment Algorithm . . . . .	16
2.3.4 GenASM . . . . .	18
<b>3 Methods</b>	<b>19</b>
3.1 Overall Workflow . . . . .	19
3.2 Using WRAM vs. MRAM for Alignment Data . . . . .	19
3.3 Needleman-Wunsch DPU Implementation . . . . .	22
3.4 Smith-Waterman-Gotoh DPU Implementation . . . . .	23
3.5 GenASM DPU Implementation . . . . .	24
3.6 Wavefront Algorithm DPU Implementation . . . . .	25
3.6.1 Custom Dynamic Memory Allocator . . . . .	25
3.6.2 WFA using WRAM vs. MRAM for Alignment Data . . . . .	27
<b>4 Evaluation</b>	<b>29</b>
4.1 Experimental Setup . . . . .	29
4.1.1 Baselines . . . . .	29
4.1.2 Evaluation Platform . . . . .	29
4.1.3 Datasets . . . . .	29
4.2 Execution Time Results . . . . .	30
4.2.1 Thread-scale CPU Experiments . . . . .	30
4.2.2 MRAM and WRAM DPU Implementations Comparison . . . . .	32

4.2.3	Thread-scale DPU Experiments . . . . .	34
4.2.4	DPU-scale Experiments . . . . .	34
4.2.5	Full-scale DPU and Full-scale CPU Comparison . . . . .	36
4.2.6	Large Reads DPUs and CPUs Comparison . . . . .	38
4.3	Energy Consumption Results . . . . .	38
<b>5</b>	<b>Related Work</b>	<b>40</b>
<b>6</b>	<b>Conclusion and Future Work</b>	<b>42</b>
<b>A</b>	<b>Abbreviations</b>	<b>43</b>
	<b>Bibliography</b>	<b>44</b>

# ILLUSTRATIONS

2.1	PIM R-DIMM Module [1]	10
2.2	Host attached to UPMEM-DIMMs Illustration [1]	10
2.3	Genome analysis pipeline and read mapping phases [2]	12
2.4	Needleman-Wunsch example and recurrence equation	14
2.5	Smith-Waterman-Gotoh equation	15
2.6	SWG example	15
2.7	WFA equation [3]	16
2.8	WFA example and flowchart	17
3.1	High-throughput read pair alignment on UPMEM	20
3.2	Example of using WRAM for alignment data	21
3.3	Example of using MRAM for alignment data	21
3.4	WFA's data structure example	25
3.5	Custom memory allocator on UPMEM	26
4.1	Execution time CPU thread-scaling	31
4.2	MRAM vs. WRAM DPU implementations timing results	33
4.3	DPU thread-scaling timing results	35
4.4	Number of DPUs scaling timing results	36
4.5	Full-scale DPU and full-scale CPU timing results	37
4.6	WFA adaptive on large reads DPUs and CPUs	38
4.7	Energy Consumption on DPUs and CPUs	39



# TABLES

4.1 Datasets . . . . .	30
------------------------	----

# CHAPTER 1

## INTRODUCTION

Following Moore’s law, the number of transistors in processors has been doubling about every two years, leading to an exponential increase in the power of the processor cores. However, memory performance did not scale comparably which has made the cost of transferring data between the memory and the CPU in some cases more expensive than the computations to be performed on the data [4, 5]. Data-intensive workloads, such as genomics, spend a considerable portion of execution time and energy moving data between memory and the computing units [4, 6].

Processing-in-Memory (PIM) architectures aim to alleviate the data movement bottleneck of existing systems by providing the memory with computing competencies [4, 5, 7]. UPMEM were the first to commercialize a real PIM system [8]. The UPMEM PIM architecture integrates conventional DRAM arrays and general-purpose cores called DPUs into the same chip, which eventually reduces energy and time consumption imposed by data movement. UPMEM PIM is a memory-centric solution that can be used to accelerate memory-bounded applications such as genomics, database index search, compression/decompression, 3D image reconstruction, and many others [1, 9, 10, 11, 12, 13, 14, 15, 16, 17].

Genome sequencing analysis involves data-intensive computational techniques to extract and analyze genomic features such as DNA and RNA sequences. These features play a pivotal role in understanding the properties of species and analyzing out-breaking diseases such as COVID-19. Nowadays, high-throughput sequencing technology is used to extract the sequence of bases that form a DNA fragment [2]. Even with current sequencing technological advancements, it is hard to read the genome as a whole sequence for most organisms. Thus, the DNA is broken into many smaller fragments, and later each fragment is sequenced into small chunks called reads. However, these reads lack information such as the order and the location they originated from. Therefore, genome analysis main goal is to reconstruct the entire genome from many reads. It starts with a computational process called read alignment or read mapping. The goal of read mapping is to find where these reads are most likely to be located by comparing each read

against multiple subsequences of a known reference genome, and detecting the differences between the read and the reference segments at these locations. Then, a variant calling method processes the mapping results to find the best location of the read, and make decisions whether the variations detected are caused by genetic mutations or sequencing errors.

Read mapping is a major bottleneck in the genome analysis pipeline since finding many mappings of these short reads, up to billions of them, while tolerating differences or sequencing errors in each read and performing at a very fast rate is challenging [2]. Due to the large number of reads that need to be mapped, a read mapper performs a large number of data transfers between the processor and main memory which is extremely costly in terms of time and energy [2, 4, 5].

Read mappers compute the sequence alignments using an approximate string matching (ASM) algorithms which use computationally expensive dynamic programming (DP) algorithms. Needleman-Wunsch (NW) [18] and Smith-Waterman (SWG) [19] are widely used DP-based alignment algorithms since they provide the best alignment accuracy. However, these algorithms have quadratic time and memory complexity [20]. Thus, over the years, many heuristics and techniques were developed to improve these algorithms, but most of them sacrifice accuracy for execution time [2, 20]. One of the most recent algorithms is the wavefront alignment algorithm (WFA), which has proven to outperform other state-of-the-art methods while consuming less memory and providing accurate alignments [3].

All of the previously mentioned ASM algorithms are memory-bounded since they have low data reuse. These algorithms have been accelerated using hardware accelerators such as GPUs, FPGAs, and processing-in-memory [2]. For example, GenASM [21] is a related work that improves and accelerates an ASM algorithm called Bitap [22], by implementing the modified algorithm on a customized processing-in-memory hardware. However, the designed framework is a prototype that has been specially designed for the GenASM algorithm, unlike the general purpose real UPMEM-PIM system that we are using.

In this work, we propose to accelerate read mapping by implementing the state-of-the-art pairwise alignment algorithms such as NW, SWG, GenASM, and WFA on the first real PIM architecture, the UPMEM PIM architecture. We implement a high-throughput read alignment framework on the UPMEM-PIM system. We dispatch a large number of read sequences across the DPUs, and within each DPU we have the DPU threads aligning the reads using one of the implemented ASM algorithms. We compare the throughput and energy consumption of these algorithms on the UPMEM system to that on a server-grade multi-threaded CPU system. We show that genome sequence analysis can be effectively accelerated using PIM architectures.

# CHAPTER 2

## BACKGROUND

### 2.1 UPMEM PIM Architecture

Data movement between main memory and processor cores is a bottleneck for many data-intensive workloads. These workloads spend a significant portion of execution time and energy moving data through the high-latency and low-bandwidth memory bus [6, 23]. PIM architectures are a new system paradigm to solve the data movement bottleneck, by integrating computing abilities into memory. The goal of PIM is to perform operations while the data remains in the memory, without the need to move it to the CPU, perform the computation, and then send it back to the memory. If the data can be updated without transfers, time and energy can be saved without affecting the result [4, 1].

The UPMEM PIM architecture is the first real general-purpose processing in DRAM engine [8]. A UPMEM system consists of regular DDR4-2400 DIMM modules, where each module has a large number of DRAM arrays combined with general-purpose processing cores called DRAM Processing Units (DPUs), as shown in Fig. 2.1. UPMEM DIMMs work as a parallel co-processor connected to the main memory of a host CPU (e.g., x86, ARM64, or Power9). A UPMEM system can have up to 20 UPMEM DIMMs plugged into an x86 platform. Each module consists of 16 PIM-enabled chips, and within each chip, there are 8 DPUs. DDR4 DIMMs coexist with UPMEM DIMMs on a server, where the server can perform both regular memory processing and/or PIM. The host side will be responsible of dispatching the input across DPUs, launching the DPU kernels, and retrieving the results.

Fig. 2.2 represents a server with one host CPU attached to multiple UPMEM DIMMs, main memory, and with an illustration of the internal representation of the DPUs.

A DPU has a 32-bit RISC processor that can potentially run at 500 Mhz with a customized Instruction Set Architecture (ISA) [24]. Each DPU has 24 hardware threads, called tasklets, that share a 24KB instruction memory IRAM, 64KB working memory WRAM, and a 64MB main memory MRAM. The CPU transfers



Figure 2.1: PIM R-DIMM Module [1]

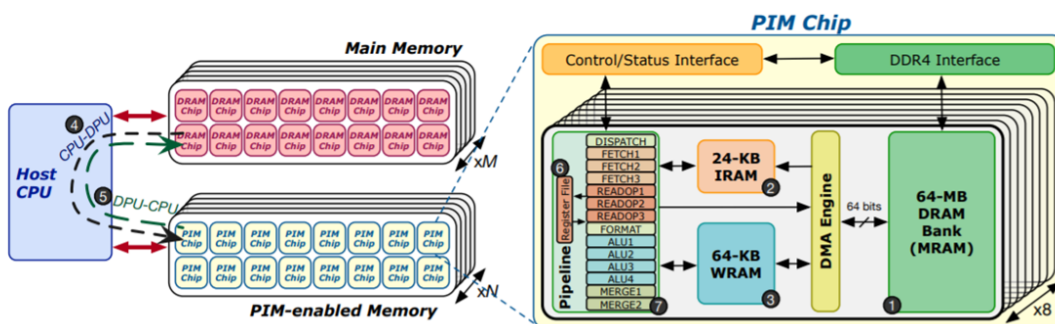


Figure 2.2: Host attached to UPMEM-DIMMs Illustration [1]

data from the host’s main memory to the DPUs’ MRAM, and copies back the results from the MRAM to the main memory when the DPUs finish execution. These transfers can be done in parallel using the programming interface [24] if the buffers sent/retrieved are of the same size among all DPUs. The DPU pipeline consists of 14 stages, as shown in Fig. 2.2, but for the same thread only three of these stages ALU4, MERGE1, and MERGE2 may run in parallel with the DISPATCH and FETCH stages of the next instruction. As a result, the pipeline efficiency is maximized when at least 11 threads are used, because every 11 cycles only one instruction can be dispatched from the same thread [1]. In total, a PIM-enabled server can have up to 2560 DPUs with 160GB MRAM.

UPMEM follows the Single Program Multiple Data (SPMD) programming model, in which threads run the same program but work on distinct data, and can execute alternative control-flow paths at runtime. Within the same DPU, threads can synchronize and share data. Meanwhile, threads across different DPUs cannot synchronize or share data, since DPUs work independently and asynchronously [15]. The host CPU manages intermediate data transfers between DPUs and combines partial results with final findings.

The PIM architecture uses a tool-chain centered on the LLVM-based C-compiler and including Linux drivers for x86 servers. It also has an SDK [24]

that provides a full-featured runtime library for the DPU, libraries to manage the communication from the host to DPUs operations, and LLDB-based debugger.

The provided programming interface allows programmers to program DPUs in the familiar C programming language. However, implementing DPU programs is challenging for multiple reasons. DPUs do not have cache memories, so the transfers between the WRAM and MRAM are explicitly declared. These transfers are DMA instructions that need to be aligned manually. Also, a programmer must ensure an efficient partitioning of the workload across many DPUs and threads within DPUs, so that the DPU pipeline can be fully utilized [1, 24].

The design of UPMEM DIMMs overcomes the data movement bottleneck by integrating processing into DRAM engines. Memory-bounded applications, such as genomics, can be accelerated using the UPMEM PIM architecture, which reduces the cost of data movements [15].

## 2.2 Genome Sequencing Analysis

Genome analysis as defined by Nature [25] is “the identification, measurement or comparison of genomic features such as DNA sequence, structural variation, gene expression, or regulatory and functional element annotation at a genomic scale. Methods for genomic analysis typically require high-throughput sequencing or microarray hybridization and bioinformatics”.

The genome analysis pipeline begins by sequencing a genomic sample into small sequences called reads using high-throughput sequencing machines. HTS technology can sequence millions of DNA samples in parallel [2]. For example, NovaSeq 6000 generates up to  $10^{12}$  sequence bases every 44 hours [26]. These reads vary in length and error rate depending on the sequencing machine used. For example, Illumina sequencing machine outputs short reads of 100-300bp length with low error rate ( $\sim 0.1\%$ ), whereas ONT & PacBio generates large reads of 500-2Mbp with high error rate ( $\sim 15\%$ ) [27]. These reads lack information such as their location and order in the original genome. Therefore, the main goal of genome analysis is to reassemble the entire genome from many reads while accounting for different variations and mutations.

The read mapping phase of the genome analysis pipeline finds one or more potential locations for each read within a known reference genome, based on the similarity between the read and the reference subsequence at these locations. Finally, the read mapping results are processed using variant calling algorithms Fig. 2.3.

Read mapping can be performed in a brute force manner by comparing each read against every subsequence of the reference genome (the human reference genome is about 3.2 billion bases) to find the best similar locations. However, this brute-force approach is expensive. For this reason, read mapping is typically

divided into a three-step procedure as shown in Fig. 2.3. The first two, indexing and filtering, are used to reduce the number of pairwise alignment algorithms that need to be performed between the reference genome segments and each read in the third step.

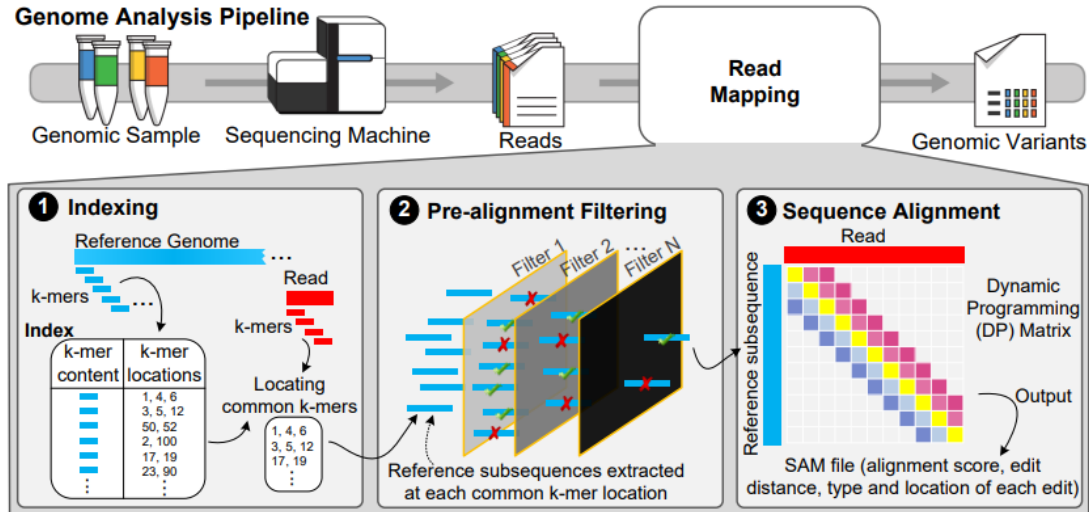


Figure 2.3: Genome analysis pipeline and read mapping phases [2]

In the first step, an index of the reference genome is created using substrings called seeds or k-mers. This step facilitates querying over the full reference genome and helps to reduce the memory footprint by storing the redundant segments of the reference genome only once. The most used technique for indexing the reference genome is hashing [28], where a hash table stores the seeds and their corresponding locations in the reference genome. After building the index, the mapping algorithm extracts the seeds from each read and uses them as a key to query the index structure and retrieve potential mapping locations of each read in the reference genome. The main challenge with indexing is choosing the convenient length and number of seeds. Short seed queries retrieve a large number of mapping positions that need to be aligned, and when long reads are used, requires extracting from each read a large number of seeds. This eventually impacts the number of times the index table is queried and the number of mapping locations retrieved.

After obtaining one or more potential mapping locations of the read within the reference genome, the read mapper computes the similarity between each read and every subsequence at these locations. However, the read and the segments may be similar or dissimilar, even if they share the same seed. Thus, in the second step, pre-alignment filters are used to avoid computationally expensive sequence alignment algorithms on dissimilar sequences. These filters need to quickly decide whether the computationally expensive alignment algorithms are needed or not by estimating the number of edits between two given sequences. If the estimated



number is above a certain edit distance threshold, then they are dissimilar and there is no need to perform the computationally expensive alignment [2].

The last step of reading mapping uses alignment algorithms to verify the similarity and compute the optimal alignment between the read sequence and the reference subsequences at each mapping location. Alignment algorithms can be either DP-based, such as Smith-Waterman and Needleman-Wunsch, or non-DP-based such as Hamming Distance. DP-based algorithms are the most used, as they provide the most accurate alignments [28]. However, DP-based algorithms have quadratic time and memory complexity, so applying them to millions of reads slows down the performance of the entire pipeline. Alignment algorithms can be accelerated either by applying software modifications or by using hardware accelerators to accelerate the DP-based algorithms without changing their behavior [2]. For example, traditional DP-based algorithms are improved by computing bands (diagonal vectors) of the DP table instead of computing the full table, as proposed in Ukkonen’s banded algorithm [29]. Although these heuristics methods enhance the computational time, they deteriorate the accuracy of the alignment. The wavefront alignment algorithm is a recent algorithm that performs better than other state-of-the-art alignment algorithms and provides exact alignments [3]. However, these alignment algorithms are data-intensive and perform more data movements than computation. For this reason, hardware accelerators, such as PIM architectures, are promising for mitigating the memory bandwidth bottleneck. The DP-based algorithms can have the DP matrices stored in memory and the recurrence equation applied within the memory without the expensive data transfers.

## 2.3 Pairwise Alignment Algorithms

The goal of sequence alignment algorithms, also known as approximate string matching ASM, is to compare two sequences and find their similarities and differences. Given a read pair, the alignment algorithm attempts to compute the smallest number of edits required to match two sequences with the best score, where the score is determined by a scoring scheme. The edits can be substitutions, deletions, or insertions. The scoring scheme defines the rules to score an alignment. It takes as an input the match and mismatches scores and the gap penalty. For any two sequences, there is a possibility to find multiple alignments. Thus, the algorithm usually includes a traceback step to obtain the location and type of each edit. In the following sections we explain the alignment algorithms that we will use in our work.

### 2.3.1 *Needleman-Wunsch*

Needleman-Wunsch (NW) [18] computes exact pairwise alignment using dynamic programming. The algorithm finds the best alignment of sequences by keeping



track of the optimized alignments of subsequences. As a result, NW requires quadratic time and memory complexity  $O(mn)$ , where  $m$  and  $n$  are the sequences length.

The algorithm takes as an input the two sequences (read pairs) and the scoring schema including the match, mismatch penalty and gap scores  $\{a, x, e\}$ . The algorithm then outputs the best alignment score and operations. NW uses a linear gap function  $w(n) = n.e$  to compute consecutive gap score, where  $n$  is the gap length and  $e$  is the gap cost. Using linear gap functions does not take into consideration the cost of opening a new gap. It computes the alignment of a read pair  $p = p_1p_2\dots p_n$  and text  $t = t_1t_2\dots t_m$  by tabulating the optimized alignments of the sub-sequences, from (1,1) to (n,m) (Fig. 2.4). For example, a cell  $D_{v,h}$  has the best alignment score of the sub-sequences  $p_{1..v}$  and  $t_{1..h}$ . The algorithm starts by initializing the first row and column of the table as defined by the scoring model. Next, using the recursion equation, the optimal solution is computed and stored in the last cell  $D_{m,n}$ . At the end, the DP-table can be traced back to find the operations of the optimal solution from (m,n) to (1,1), where any cell  $D_{v,h}$  is either the result of a previous mismatch/match (upper diagonal cell), insertion (left cell), or deletion (upper cell).

NW is memory-bounded since to compute each new cell in the DP-table, it needs to load three previous cells while performing little computation to compute the new cell.

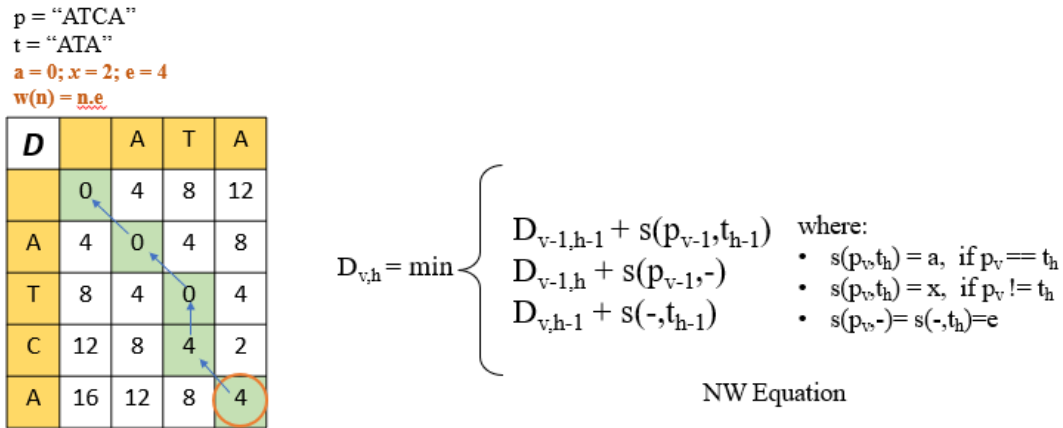


Figure 2.4: Needleman-Wunsch example and recurrence equation

### 2.3.2 Smith-Waterman-Gotoh

Smith-Waterman-Gotoh (SWG) [19] is an improvement of Needleman-Wunsch algorithm. It computes pairwise alignment using dynamic programming under the affine gap model which gives more realistic results. The affine gap model is defined by scoring penalties  $\{a, x, o, e\}$ ,  $a$  and  $x$  are the matching and mismatching score respectively, and a linear function for gap weight  $w(n) = o + n.e$ , where

the weight for a gap of length  $n$  is the sum of a gap opening penalty ( $o$ ) and a gap extension penalty ( $e$ ) multiplied by the length of the gap ( $n$ ). Thus, having a long consecutive gap of insertions or deletions costs less than having multiple small gaps with the same total length, assuming that a single large gap is biologically more likely to occur than many small gaps. The algorithm is widely used due to its ability to provide accurate optimal alignments under the gap affine model [28].

$$\begin{cases} I_{v,b} &= \min\{M_{v,b-1} + o + e, I_{v,b-1} + e\} \\ D_{v,b} &= \min\{M_{v-1,b} + o + e, D_{v-1,b} + e\} \\ M_{v,b} &= \min\{I_{v,b}, D_{v,b}, M_{v-1,b-1} + s(q_{v-1}, t_{b-1})\} \end{cases} \quad \begin{array}{l} \text{where:} \\ s(q_v, q_h) = a \text{ if } q_v = q_h \\ s(q_v, q_h) = x \text{ otherwise} \end{array}$$

Figure 2.5: Smith-Waterman-Gotoh equation

SWG uses three matrices  $M$ ,  $I$  and  $D$  as shown in Fig. 2.6 to compute the minimum score needed to align a query  $p = p_1p_2\dots p_n$  and text  $t = t_1t_2\dots t_m$  from (1,1) to (n,m). The cell  $M_{v,h}$  has the best alignment score achieved for the subsequences  $p_{1..v}$  and  $t_{1..h}$ , the cell  $I_{v,h}$  represents the best score achieved while assuming that the alignment ends with an insertion (gap in  $t$ ), and similarly the cell  $D_{v,h}$  assumes that the alignment ends with a deletion (gap in  $p$ ). Given a scoring schema and using the recurrence equation 2.5 on the three DP tables, the global alignment score be found at the cell  $M_{n,m}$ . Then, the DP tables can be traced back to obtain the optimal solution.

SWG provides more realistic results under the gap-affine model. However, It is more memory-intensive than NW, as it consumes more memory and requires more data movements to get the alignment.

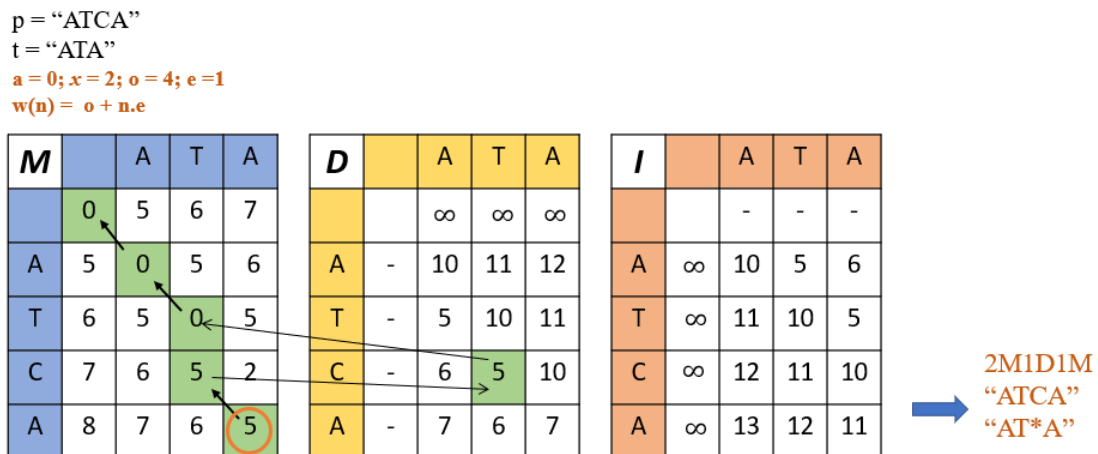


Figure 2.6: SWG example

### 2.3.3 Wavefront Alignment Algorithm

The Wavefront Alignment Algorithm (WFA) [3] is the state-of-the-art exact pairwise alignment algorithm. It computes the alignment of sequences using the gap-affine model. Unlike traditional dynamic programming algorithms, WFA considers the similarity of sequences to compute the alignment efficiently. The algorithm computes increasing-score partial alignments until reaching the optimal alignment. As a result, its time complexity  $O(ns)$  depends on the sequence length  $n$  and the alignment score  $s$ .

WFA improves SWG and redefines its equation in terms of furthest reaching points and wavefronts as shown in Fig. 2.7. The furthest reaching point  $F_{s,k}$  is the DP-cell on the diagonal  $k$  with score  $s$ , which is the furthest from the beginning of the diagonal. Also,  $M_{s,k}$ ,  $I_{s,k}$ , and  $D_{s,k}$  are the offsets of the f.r. point  $F_{s,k}$  in diagonal  $k$  in each of three SWG matrices  $M, I$ , and  $D$  respectively. The  $s$ -wavefront  $WF_s$  is defined as the set of all furthest reaching points with score  $s$  for all  $k$ , and  $M_s$ ,  $I_s$  and  $D_s$  are defined as the components of the wavefront, where  $M_s$  is the set of offsets  $M_{s,k}$  for all  $k$ . The wavefront length is defined by the number of diagonals spanned in the wavefront component. As the score increases, the size of the wavefront components increases, resulting in a better memory consumption  $O(s^2)$ . Also, it takes advantage of extending matching characters along the diagonal. This step plays a major role in accelerating the algorithm. WFA exceeds in performance other other algorithms while using less memory [3].

$$\begin{aligned} \tilde{I}_{s,k} &= \max \left\{ \begin{array}{ll} \tilde{M}_{s-o-e,k-1} & \text{(Open insertion)} \\ \tilde{I}_{s-e,k-1} & \text{(Extend insertion)} \end{array} \right\} + 1 \\ \tilde{D}_{s,k} &= \max \left\{ \begin{array}{ll} \tilde{M}_{s-o-e,k+1} & \text{(Open deletion)} \\ \tilde{D}_{s-e,k+1} & \text{(Extend deletion)} \end{array} \right\} \\ \tilde{M}_{s,k} &= \max \left\{ \begin{array}{ll} \tilde{M}_{s-x,k} + 1 & \text{(Substitution)} \\ \tilde{I}_{s,k} & \text{(Insertion)} \\ \tilde{D}_{s,k} & \text{(Deletion)} \end{array} \right\}, \end{aligned}$$

Figure 2.7: WFA equation [3]

The optimal alignment is found when any furthest point of  $WF_s$  reaches DP-cell  $(n,m)$  with minimal score  $s$ . Figure 2.8 is an illustrative example of the wavefront algorithm. The physical view is how the wavefront components are physically present in memory, and the logical view is a theoretical representation of the DP-matrices. The algorithm takes as an input the read pair and scoring penalties. After initializing the first component, WFA starts by extending the furthest reaching points of the matching characters along the diagonal. After

that, it checks if any of the furthest reaching points of  $WF_s$  reaches the last cell. If so, the score  $s$  is the alignment score, and it performs the traceback. Otherwise, the score is increased, and it computes the new wavefront component using the recurrence equation in Fig. 2.7. Then, the same steps are performed until any of the f.r. points reach the sequence end.

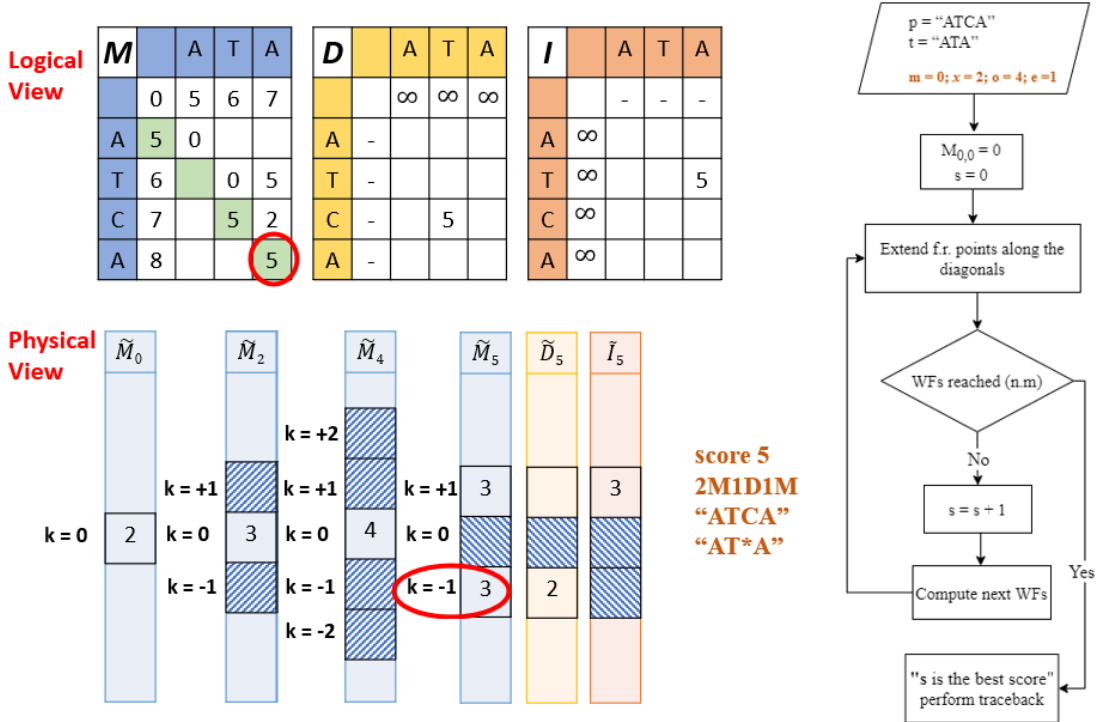


Figure 2.8: WFA example and flowchart

The size of wavefront components grows as the score increases and the component spans over more diagonals as shown in Fig. 2.8. As the size of the wavefront component grows, more execution time is spent processing unpromising paths of the outer diagonals. Thus, a heuristic version of the algorithm, called WFA-adaptive, can be used to avoid unpromising furthest reaching points that are unlikely to lead to the solution. If the distance between the main diagonal and the outer diagonals goes beyond a certain threshold, it prunes them and reduces the wavefront length. WFA-adaptive gives much better performance but may lead to non-optimal alignments.

WFA provides better alignment quality than NW since it uses the gap-affine model. It also has lower time and memory complexity than both NW and SWG. However, WFA is more memory-bounded due to its lower data reuse and the need to perform more memory accesses, particularly for managing its irregular data structures.

### 2.3.4 *GenASM*

GenASM [21] accelerates ASM by designing a near-memory framework. It is based on the Bitap algorithm [22]. Bitap performs approximate string matching using fast bitwise operations, but it fails to align long reads and generate the traceback operations. GenASM improves and accelerates Bitap to support highly parallel and scalable read alignments and adds a bit vector-based traceback algorithm. It designs a near-memory framework to accelerate ASM, which can significantly speed up multiple use cases of genome sequence analysis such as read alignment, pre-alignment filtering, and edit distance calculation.

GenASM takes advantage of the high memory bandwidth and the logic-layer of 3D-stacked memory to design a low-power and area-efficient hardware accelerator that performs the modified highly parallel ASM in the DRAM chip itself. 3D-stacked memory combines DRAM layers with a logic layer that can have computational logic, which interacts with the DRAM cells as well as the processor. This method is known as processing-near-memory (PNM), where processing elements are added or embedded close to or inside the memory [4, 1]. The near-memory hardware used in GenASM is a prototype that has been specially customized to accelerate GenASM. We will accelerate GenASM's algorithm on a real general purpose programmable PIM system.

# CHAPTER 3

## METHODS

### 3.1 Overall Workflow

In our work, we perform high-throughput read alignment on the UPMEM-PIM system. Fig. 3.1 describes our overall workflow. In Step (1), the host CPU loads the read pairs from the input file to the main memory. In Step (2), the reads are evenly distributed and transferred from the host’s main memory to the attached DPUs’ MRAM using parallel transfers [24] to hide the communication overhead. DPU kernels are then launched, and threads in each DPU work independently to avoid the expensive inter-thread synchronization [1]. In Step (3), each DPU thread fetches one read pair at a time from MRAM to WRAM. In Step (4), the DPU thread computes the alignment using an alignment algorithm of interest (NW, SWG, GenASM, or WFA), and extracts the alignment operations using traceback. In Step (5), the DPU thread writes the alignment score and operations to the MRAM. Steps (3), (4), and (5) are repeated by each thread to process the next read. In Step (6), after the DPUs finish execution, the CPU copies the results back from the DPUs’ MRAM. In Step (7), the CPU writes the results to an output file.

In the rest of this Chapter, we describe how each of the different alignment algorithms were implemented on the DPU and the challenges that were faced when implementing it.

### 3.2 Using WRAM vs. MRAM for Alignment Data

Recall that the WRAM has 64KB of memory which is shared among 24 DPU threads. Hence, WRAM can act as a limiting factor to parallelism. For each combination of algorithm, read size, and error rate, we need to find the maximum number of threads such that the algorithm will not run out of WRAM memory. For a given read length and error rate, we can compute the exact amount of memory needed for NW, SWG, and GenASM, and set the number of threads

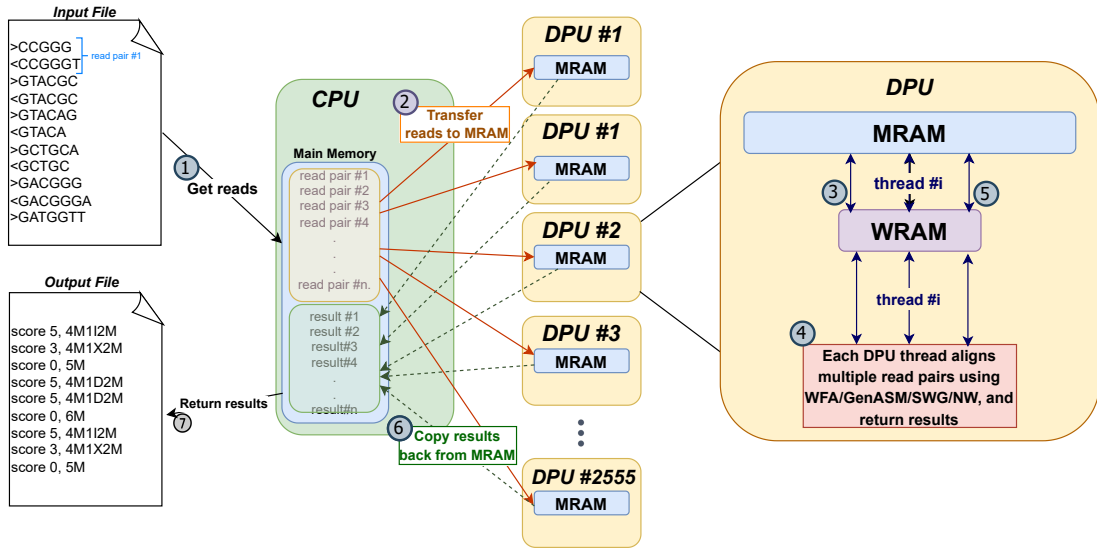


Figure 3.1: High-throughput read pair alignment on UPMEM

accordingly. In the case of WFA, we can only estimate the upper limit since the memory consumed by WFA depends on the alignment similarity, and it varies from one read pair to another even if they have the same read length. As we scale the read length and error rate, more memory will be needed per thread, so we will necessarily use fewer threads.

In cases where reads are large, we will not be able to perform the alignment even for one thread. For example, the NW DP-table will not fit the WRAM capacity if the read length goes beyond 150. For this reason, we have implemented a WRAM and MRAM version for each alignment algorithm. The WRAM version places the algorithm’s data in WRAM only. The MRAM version places data in MRAM, and transfers the data back and forth between the WRAM and MRAM during the computation. The MRAM version allows us to use more threads and align longer reads than the WRAM version, but adds the overhead of the WRAM-MRAM transfers.

Fig. 3.2 shows an example of the WRAM version. Before launching the DPU kernel, we allocate a segment of the MRAM to store the input read pairs and their alignments. We used the `DPU_MRAM_HEAP_POINTER` [24], which defines the base address at which the MRAM can be freely used to perform MRAM dynamic allocation. We define the `MRAM_PTR` variable to keep track of the last available address in MRAM. Each thread gets one read pair at a time from the MRAM to the WRAM, performs the alignment inside the WRAM, and writes the result to the MRAM. The WRAM version avoids the latency of using DMA transfers while computing. However, the WRAM is limited by its capacity (64KB) which prevents fitting the alignment data of many threads. Consequently, the WRAM version utilizes a few DPU threads which can lead to reduced pipeline efficiency.

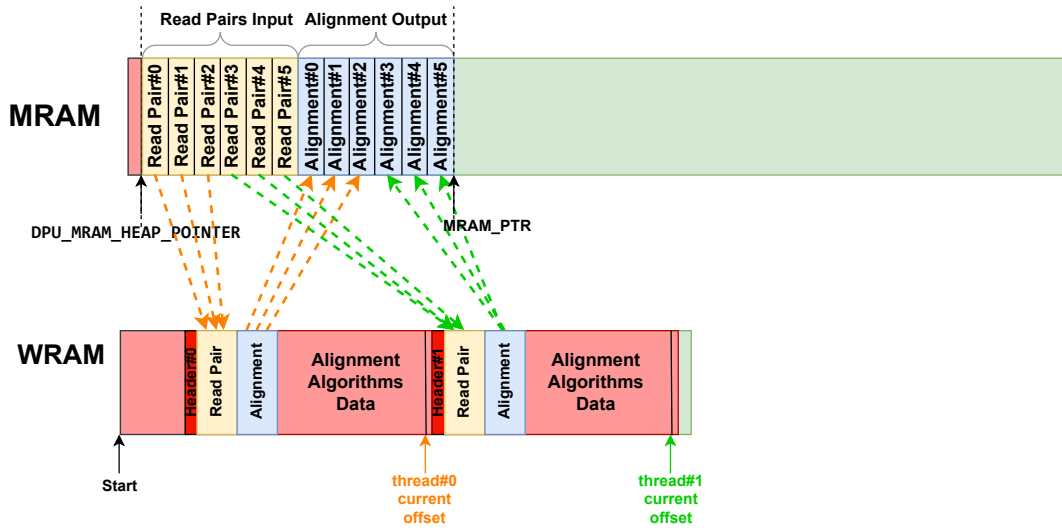


Figure 3.2: Example of using WRAM for alignment data

Thus, the pipeline efficiency ratio depends on the memory usage pattern of each algorithm and the used input read length, e.g. when the read length increase, or the applied alignment algorithm has high memory consumption by design, the WRAM cannot fit the allocated alignment data for a large number of threads, or even for a long read, the alignment data might not fit at all.

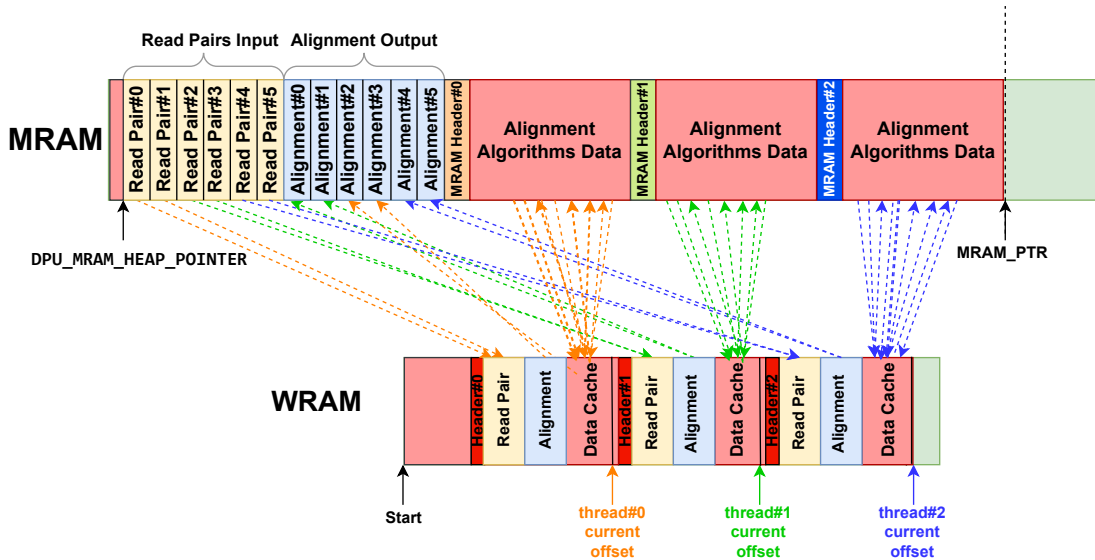


Figure 3.3: Example of using MRAM for alignment data

To overcome the limited capacity of WRAM, we also study storing the alignment data on the slower MRAM space. As shown in Fig. 3.3, similar to the WRAM version, the host allocates the space for the read pairs input and the alignment output in the MRAM. After launching the DPU kernel, each thread



asynchronously allocates and manages its own alignment data segment in the MRAM. The thread is aware of its MRAM segment range, which depends on the number of the used DPU threads and the available MRAM space. The WRAM is segmented across the threads: a WRAM segment holds the needed space to store a read pair input, its alignment output, and a data cache for the required elements of the alignment data. Each thread fetches its read pairs input at a time from the MRAM input segment to its WRAM segment, computes the alignment by actively loading the needed alignment data to its WRAM cache using DMA before updating and writing it back to the thread alignment data segment in the MRAM, and transfers the resulting alignment output to the MRAM. After every alignment iteration, each thread reuse its WRAM and MRAM segment to process the next alignment data.

The MRAM version utilizes more threads, since the size of the data cache is much less than the size of the alignment data, allowing more threads to fit their data in the WRAM even if the read length is large. The size of the WRAM data cache depends on the alignment algorithm used, the input read length, and the edit distance threshold. It uses a large number of DMA transfers to apply the alignment algorithm which adds a latency overhead to the DPU pipeline. However, the latency of the DMA transfers can be mitigated if enough DPU threads are used. We consider the tradeoff of having more DPU threads (MRAM) vs. fewer DMA transfers (WRAM) in our study.

### 3.3 Needleman-Wunsch DPU Implementation

NW uses a DP-table to compute the alignment of sequences as described in Section 2.3.1. The size of the DP-table is  $mn$  where  $m$  and  $n$  are the length of the pattern and text sequences respectively. NW uses the linear gap model to compute the alignment score. In our implementation, we set the scoring penalties parameters to  $a = 0$  (match cost),  $x = 3$  (mismatch cost),  $e = 4$  (deletion/insertion cost), and the data type of the DP-cells to *int16*.

In the NW WRAM version, each thread allocates its DP-table in the WRAM, fills the DP-table, computes the traceback, and send the alignment output to the MRAM. It reuses the DP-table to process the next read pair. We specify the number of DPU threads by computing the memory consumed given the read length. The WRAM memory consumption of each thread includes the DP-table ( $mn * sizeof(int16)$ ), one input read pair, and the traceback operations. We find the maximum number of threads such that the total memory consumption fits in the WRAM capacity. The maximum read length that can be used to fit the DP-table in the WRAM for one thread is 175bp (61KB). Consequently, the NW WRAM version can be only applied for short reads and few number of DPU threads.

We use the NW MRAM version to align longer reads by storing the DP-table

of each thread in the MRAM. Each thread allocates 4 DP-cells in the WRAM which will be used as a data cache. To compute a new cell in the DP-table, 3 DP-cells are loaded from the MRAM to the WRAM cached cells (upper, left, and upper diagonal cells), and the fourth WRAM DP-cell is computed by processing the three loaded DP-cells, then the result is added to the DP-table in the MRAM. Similar approach is used to compute the traceback operations. This process is done  $nm$  times for each thread resulting a large number of small sized DMA transfers, which saturates the DPU pipeline with DMA instructions and lowers its efficiency even if a large number of threads is launched.

Since the DMA transfers are aligned at 8-bytes [24, 1], a DP cell of size `int16` cannot be accessed directly without memory alignment. To overcome this memory alignment issue, we read 8-bytes memory chunks (4 DP-cells) from the DP-table in the MRAM to the WRAM data cache, then we extract the required cell. Consequently, the DP-cell cache size is 8 bytes. The WRAM memory consumption of each thread includes the 4 DP-cell caches, one input read pair, and the traceback operations. We find the maximum number of threads such that the total WRAM memory consumption fits in the WRAM capacity, and we also consider fitting the DP-tables, all input reads and their alignments into the MRAM (64MB).

### 3.4 Smith-Waterman-Gotoh DPU Implementation

SWG provides more realistic results than NW by following the gap-affine model, as shown in Section 2.3.2. It utilizes three DP-tables Matching (M), Insertion (I), and Deletion (D) tables, each of size  $mn$ , to compute the alignment under the gap-affine model. In our implementation, we represent the three DP-tables as one DP-table with each cell defined as a struct of the three elements (`int16`) M, I, and D to reduce the number of data accesses needed to compute the DP-table. We use the following scoring penalties  $a = 0$  (match cost),  $x = 3$  (mismatch cost),  $o = 4$  (deletion/insertion opening cost), and  $e = 1$  (deletion/insertion extension cost).

Similar to the NW WRAM implementation, SWG have the DP-tables of each thread allocated in the WRAM, where the alignment and traceback takes place. SWG consumes more memory as it needs to store the alignment data of 3 DP-tables. As a result, it uses fewer threads and aligns shorter reads. For example, SWG WRAM cannot align reads longer than 100bp due to the limited WRAM capacity.

We provide the SWG MRAM implementation to align longer reads and utilize more DPU threads. This method stores the DP-tables in the MRAM for each thread and allocates the required DP-cells cache in the WRAM to transfer data back and forth between the MRAM and WRAM. The DP-table is filled by actively reading the needed DP-cells from the MRAM to the WRAM cache, computing

the new DP-cell, and writing the result into the MRAM. Similarly, the traceback is performed by transferring the required DP-cells between the MRAM and the WRAM to trace the alignment operations. Consequently, the alignment requires large number of DMA transfers, and applying it for a large number of read pairs by many threads reduces the DPU pipeline efficiency.

NW and SWG DPU implementations can be further optimized in the future work by: (1) using the optimized NW and SWG banded implementations [29] to compute the alignment using bands (diagonals) instead of the full table, (2) parallelizing the alignment computation among multiple threads, (3) and changing the caching method we're using to enable cache reuse, such as caching anti-diagonals instead of cells.

### 3.5 GenASM DPU Implementation

GenASM [21] uses bitvectors to compute sequences alignment. It modifies and adds a traceback method to the bitap alignment algorithm [22], as discussed in Section 2.3.4. GenASM uses the gap-affine model and takes as an input the maximum number of edit distances ( $k$ ) allowed while computing the alignment. We use the following scoring penalties  $\{a = 0, x = 3, o = 4, e = 1\}$ , and we set  $k$  according to the used read length and error rate.

In our WRAM implementation, each thread uses the WRAM to store the pattern bit-mask for each character in the alphabet (A, C, G, and T), two status bit-vectors (R0 and R1) to hold the partial alignment between subsequences of the text and the pattern with maximum number of errors, and four intermediate bit-vectors for each edit case (Matching, Substitution, Deletion, and Insertion). While iterating the text to compute the alignment, GenASM saves the generated intermediate bit-vectors of each text iteration in the traceback matrix to compute the alignment operations later. The traceback matrix is large in size, so we store it in the MRAM, and we use DMA transfers to add and update its elements as needed. GenASM WRAM consumes less memory than the DP-based approaches since it replaces the DP-tables by small-sized bit-vectors, so more threads can fit their data in the WRAM. As a result, GenASM WRAM can be used on longer reads compared to the DP-based WRAM implementations, and with enough threads to utilize the DPU pipeline efficiency.

Although GenASM utilizes the WRAM more efficiently than a DP-based alignment algorithm, the WRAM capacity is still a bottleneck for longer read lengths. We provide an MRAM version where the pattern bit-mask, status bit-vectors, and the traceback matrix are stored in the MRAM. The intermediate bit vectors are allocated in the WRAM since they are of small size (the total size of the bit-vectors is  $m/32$  where  $m$  is the pattern read length) and to reduce the large number of DMA transfers needed to compute the alignment. Therefore, GenASM MRAM version can align longer reads with more DPU threads.

In our work, we don't implement the Divide and Conquer (DQ) method used in GenASM, which can be addressed by a future work. The DQ reduces the memory consumption by dividing the pattern and text into overlapping windows and performing the traceback for each window instead of storing the entire traceback matrix.

### 3.6 Wavefront Algorithm DPU Implementation

WFA computes exact pairwise alignments efficiently using the wavefront components and furthest reaching points under the gap-affine model, as described in Section 2.3.3. It considers sequences similarity while computing, so the time complexity  $O(ns)$  depends on the alignment score  $s$  and read length  $n$ . As the alignment score increases, WFA consumes more memory as it will span over more diagonals. WFA-adaptive provides a heuristic method to reduce the number of the spanned diagonals by eliminating outer diagonals that unlikely lead to the optimal alignment. We provide a DPU implementation for both WFA and WFA-adaptive. Fig. 3.4 shows an example of the data structure that's used to align one read pair using WFA. Unlike NW, SWG, and GenASM, we can't compute the actual memory consumption of the WFA alignments since it's heavily dynamic, and it changes at runtime according to the read pair length, edit distance, location and type of these edits. Consequently, we faced more memory-management challenges while implementing WFA on the UPMEM-PIM architecture.

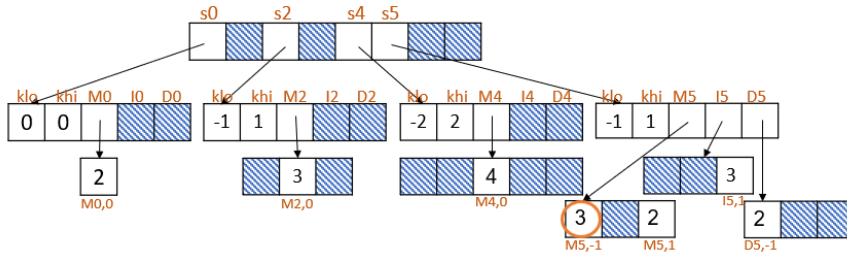


Figure 3.4: WFA's data structure example

#### 3.6.1 Custom Dynamic Memory Allocator

The WFA algorithm uses dynamic memory allocation to allocate the wavefront components because these components vary in size at run time depending on the read length and similarity, as discussed in Section 2.3.3. However, the provided dynamic memory allocators in the UPMEM SDK [24] fall short to meet the requirements of our workloads. The SDK has three memory allocators: fixed-size block, incremental (`mem_alloc`), and buddy memory allocator. The fixed-size block allocator needs to allocate before the beginning of the program a fixed number of blocks with fixed block size, which does not work with the WFA

components that vary at run time. The incremental allocator can allocate the variable size of WRAM segments at run time, but it can only reset the heap memory for all threads since it has no memory deallocation function. Resetting the heap can be a performance bottleneck in parallel read pairs alignment where for every alignment iteration, threads need to synchronize in order to reset the WRAM before starting a new iteration. The buddy allocator has allocation and deallocation functions, but it initializes the heap memory at the beginning of the program with a maximum size of 32KB, which corresponds to half of the WRAM size of 64KB and leads to underutilization of the WRAM capacity. Therefore, we define a custom memory allocator that efficiently utilizes the WRAM capacity and resets memory without thread synchronization.

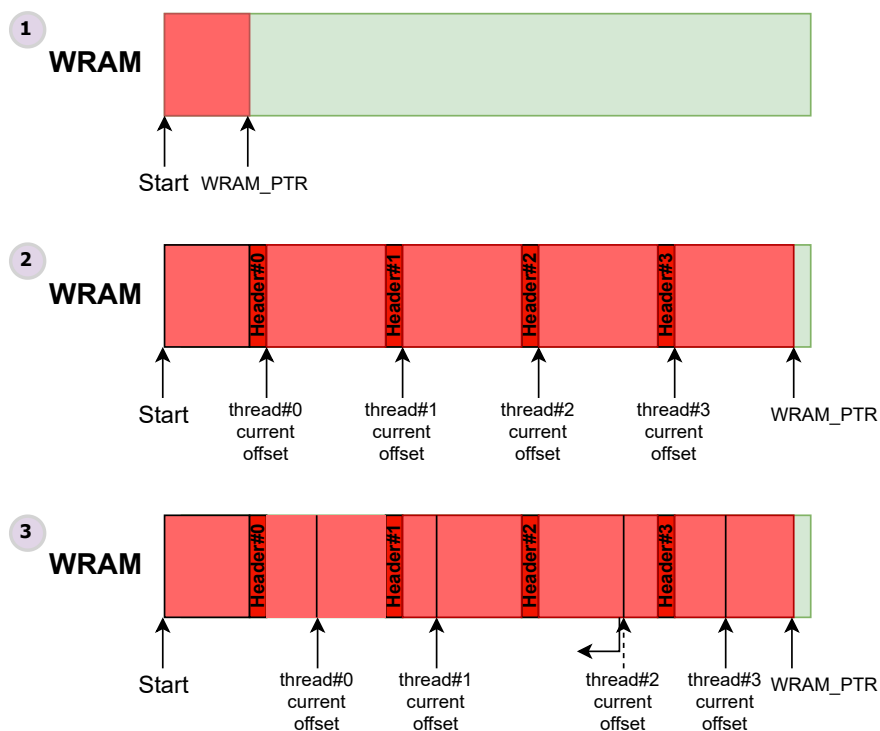


Figure 3.5: Custom memory allocator on UPMEM

The custom memory allocator is a thread-level incremental allocator. Fig. 3.5 shows an example. We use the term `WRAM_PTR` to indicate the WRAM pointer offset, which is only accessible by the SDK memory allocators. At the beginning of the program, a portion of the WRAM is occupied by the stack memory of each thread, whereas the range from `WRAM_PTR` to the end of the WRAM can be used freely (Step 1 in Fig. 3.5). Each thread first allocates its own memory slab in WRAM using `mem_alloc`, which returns the header pointer of the segment (Step 2 in Fig. 3.5). The size of the memory slab is the free range in the WRAM divided by the number of tasklets.

Then, each thread increments its current offset to allocate memory when needed, reducing the overhead of the software-defined memory allocators (Step 3 in Fig. 3.5). Threads can reset their current offset independently after every alignment iteration without synchronization. The custom memory allocator also aligns the segments on the DMA-transfer size, allowing them to be transferred back and forth between the MRAM and WRAM.

The downside of this approach is that it equally assigns all threads with the same amount of memory. So, in case one thread requires a lot more memory and another is under utilizing its memory capacity, the custom memory allocator won't accommodate since it's a per-thread incremental allocator unlike other per-DPU allocators. The custom memory allocator works with use cases where the workload is evenly distributed among DPU threads.

### 3.6.2 *WFA using WRAM vs. MRAM for Alignment Data*

We use the custom memory allocator to allow multiple threads dynamically allocate wavefront components in parallel without the overhead of inter-thread synchronization. In our experiments, we use the following scoring penalties  $\{a = 0, x = 3, o = 4, e = 1\}$ .

Each thread uses its allocated segment in the WRAM to add the wavefront components of one read pair alignment and reuses it to process the next read pair. The size of the wavefront components depends on the maximum alignment score of the input read length and edit distance threshold per read. Given the read length and the maximum alignment score ( $s$ ), we estimate the number of threads needed to align the read pairs without running out of WRAM memory. In the worst case, each thread allocates  $s$  wavefronts and each wavefront has the three wavefront components  $M_s, I_s, D_s$  allocated with their length incrementing every iteration ( $s$  iterations), so the upper limit memory consumption of WFA WRAM will be the summation of this arithmetic sequence. Similarly, the memory consumption of WFA-adaptive is estimated, where WFA-adaptive sets a threshold for the maximum wavefront length, so it uses much less memory than WFA. This method may overestimate the memory consumption but guarantees that threads won't run out of WRAM memory. When the read length and error rate increase, we'll be using fewer threads, and the wavefronts may not fit in the WRAM memory.

Consequently, we store the wavefront components in the MRAM to align longer reads. Each thread uses its MRAM segment to dynamically allocate wavefront components and stores the MRAM addresses of these components in the WRAM so that they can be accessed when required. To compute a new wavefront component  $WF_s$ , the thread first loads from the MRAM to the WRAM: the component of a previous mismatch ( $M_{s-x}$ ), the components of an extended gap ( $I_{s-e}, D_{s-e}$ ), and the components of an opened gap ( $M_{s-o-e}, I_{s-o-e}, D_{s-e}$ ). Then,  $WF_s$  is computed in the WRAM, and the thread allocates space in the MRAM

to transfer the resulted  $WF_s$  from the WRAM to the MRAM. This process is repeated to reach the optimal alignment. The traceback uses DMA transfers to load the required wavefront components from the MRAM and compute the alignment operations. In this approach, the upper limit of the WRAM memory consumption depends on the wavefront length reached when computing the maximum score's wavefront components. We estimate that in the worst case, the maximum wavefront length reached is  $2 * s$  where  $s$  is the maximum score in case of WFA and the max distance threshold in case of WFA-adaptive. The WRAM segment should fit the required wavefront components of maximum wavefront length. WFA and WFA-adaptive MRAM versions align longer reads and higher edit distances with more DPU threads which utilizes the DPU pipeline efficiently. WFA-adaptive can align even longer reads with more DPU threads due to its reduced memory consumption. These WFA DPU implementations can be further optimized in the future work by parallelizing the alignment computation across multiple threads

In the next chapter, we provide a thorough evaluation to study the efficiency of our DPU implementations.

# CHAPTER 4

## EVALUATION

### 4.1 Experimental Setup

#### 4.1.1 *Baselines*

We provide MRAM and WRAM DPU implementations for NW, GenASM, SWG, WFA, and WFA-adaptive alignment algorithms. We compare our DPU implementations against a multi-threaded CPU baseline. We make no modifications to the original CPU implementations where NW, SWG, and WFA implementations are taken from the original WFA repository [3], and GenASM is cloned from its repository [3]. We use OpenMP to align multiple reads in parallel.

#### 4.1.2 *Evaluation Platform*

We evaluate our DPU implementations on a UPMEM system with 2560 DPUs (20 UPMEM-DIMMs) running at 425MHz and 150GB MRAM. We evaluate the CPU implementations on a dual socket Intel<sup>®</sup> Xeon<sup>®</sup> E5-2697 v2 processor, 48 threads in total, with 30GB memory and 60MB L3 cache.

#### 4.1.3 *Datasets*

We use real and synthetic datasets to evaluate our work. The real datasets are short (Illumina [26]) read-reference pairs of length 100bp, 150bp, and 250bp with 0-5% edit distance threshold, generated using minimap2 [30] by mapping the datasets mentioned in Table 4.1 to the human reference genome GRCh37 [31]. We simulate long read pairs of lengths 500bp, 1000bp, 5Kbp, and 10Kbp with 0-5% edit distance threshold, using the synthetic data generator provided in the original WFA repository [3]. Each dataset has 5 million read pairs.



Read Length	Edit Distance%	Description
100	0-5%	Real, Accession# ERR240727 [32]
150	0-5%	Real, Accession# SRR826460 [32]
250	0-5%	Real, Accession# SRR826471 [32]
500	0-5%	Synthetic
1,000	0-5%	Synthetic
5,000	0-5%	Synthetic
10,000	0-5%	Synthetic

Table 4.1: Datasets

## 4.2 Execution Time Results

In this work, we aim to study the efficiency of applying the alignment algorithms on the PIM accelerator. We perform different scaling experiments to detect the peak throughput in which these implementations are efficient. We scale different configurations such as the number of CPU threads for the CPU implementations (Section 4.2.1), and the number of DPU threads per DPU (Section 4.2.3) as well as the total number of DPUs (Section 4.2.4) for the PIM implementations. We compare the performance of the MRAM and WRAM PIM implementations (Section 4.2.2). Finally, we compare the speedups using the best configurations at fullscale (Section 4.2.5).

### 4.2.1 Thread-scale CPU Experiments

We apply high-throughput read alignment on a multi-threaded CPU system. Fig. 4.1 shows the execution time of NW, SWG, GenASM, WFA, and WFA-adaptive when scaling the number of CPU threads from 1 to 48 on all datasets. We make four observations about the CPU thread scaling experiments. First, we observe that when using more than 16 threads, the performance saturates for all runs due to the memory bandwidth limitation. This observation shows that these workloads are memory-bounded and do not scale when increasing the computational power, which motivates the need to use PIM to further accelerate them. Second, as the read length increases, the algorithms spend more execution time on the alignment because the time complexity of these algorithms depends on the read length. Third, the performance of WFA, WFA-adaptive, and GenASM changes relative to the edit distance threshold of the read pairs (ED%) since these algorithms consider the similarity of the read pairs while computing the alignment, as mentioned in Sections 2.3.3 and 2.3.4. Fourth, WFA-Adaptive performs the best even when scaling the read length and ED% due to its efficient memory consumption (Section 2.3.3).

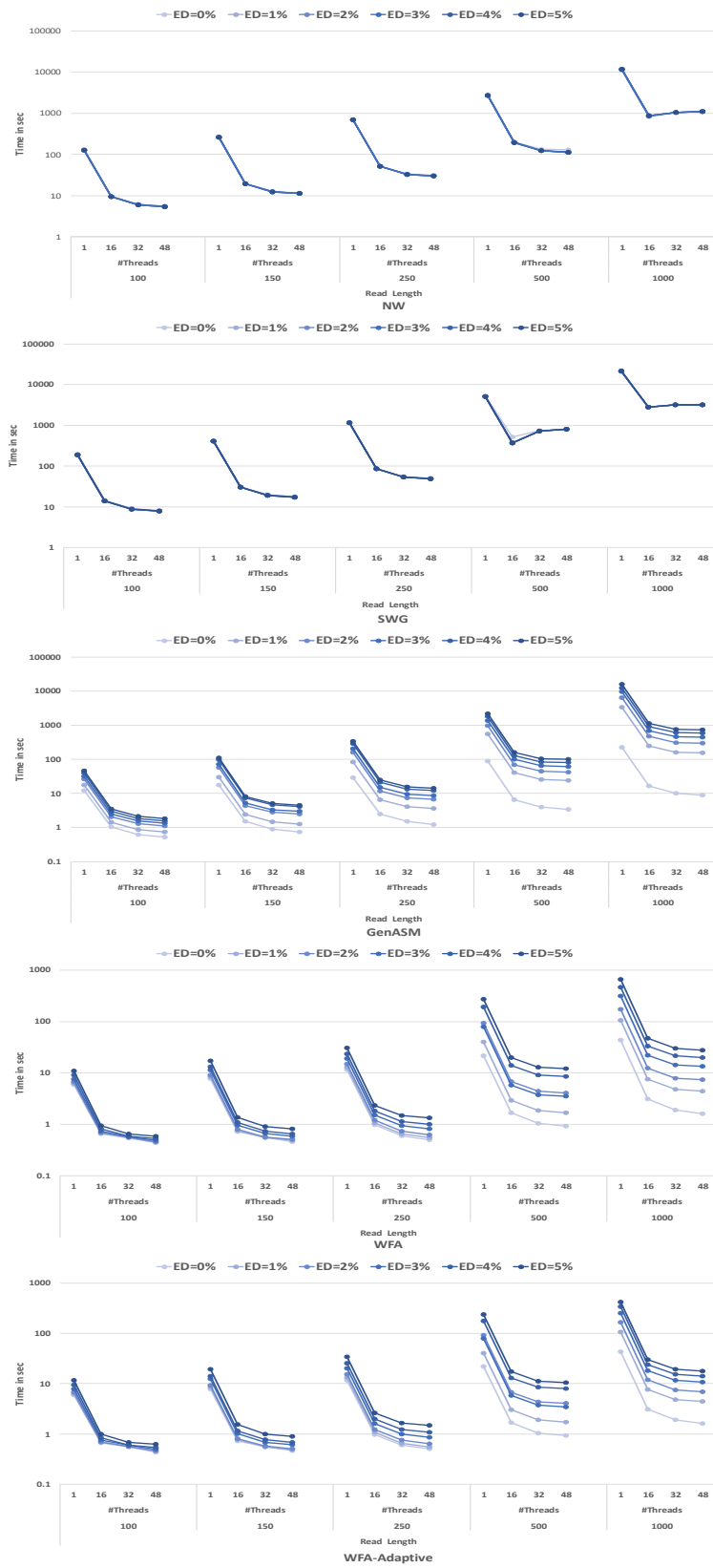


Figure 4.1: Execution time CPU thread-scaling

#### 4.2.2 *MRAM and WRAM DPU Implementations Comparison*

As mentioned in Section 3.2, we provide MRAM and WRAM DPU versions for each alignment algorithm. The MRAM version allows using more DPU threads and aligning longer reads, but it adds the MRAM-WRAM transfers cost. The WRAM version fits all data in WRAM which reduces the transfers latency, but limits the number of threads and read length. Fig. 4.2 shows timing results for MRAM and WRAM versions using 2500 DPUs. We set the max number of threads by estimating the data consumption of each algorithm according to the input read length and edit distance. The DPU-Total time includes time to copy data from and to the DPUs and time to execute the alignments on the DPUs (DPU-Kernel).

Fig. 4.2 shows the results for NW and SWG on short reads only, since the DP-tables do not fit in the WRAM for longer reads. We notice that for NW with read length 100bp, the WRAM version (2 threads) performs better than the MRAM version (24 threads). However, with read length 150bp, the WRAM version is worse than the MRAM version (24 threads) because we are using only 1 thread in the WRAM version in order to be able to fit the DP-table into the WRAM, so we are underutilizing the DPU pipeline [1]. On the other hand, SWG consumes more memory than NW because of the need to store multiple matrices. For SWG with read length 100bp, the WRAM version only uses 1 thread due to its larger memory consumption, and performs much worse than the MRAM version which uses 20 threads.

We show the results of GenASM MRAM and WRAM implementations in Fig. 4.2. GenASM uses bitvectors and bitwise operations to compute the alignment, so it has less memory consumption. Thus, we were able to apply GenASM WRAM for longer reads and to use more than 11 threads. In general, GenASM WRAM performs better than the MRAM version for most of the read lengths and edit distances. The MRAM version performs better for read length 1000 and edit distance greater than 3%, since for these datasets, GenASM WRAM is using lower number of threads (less than 11 threads) leading to under utilization of the DPU pipeline.

WFA and WFA adaptive MRAM versions perform better for read length greater than 100, as shown in Fig. 4.2. The reason behind this is that as the read length and edit distance increase, memory consumption increases, as mentioned in Section 2.3.3, so fewer threads will be used in WFA WRAM, and it won't be applied for longer reads and larger edit distances.



Figure 4.2: MRAM vs. WRAM DPU implementations timing results

### 4.2.3 *Thread-scale DPU Experiments*

We perform DPU thread scaling to study the behavior of the MRAM and WRAM versions and detect the best number of threads where the implementations are efficient. We run the experiments using read lengths 100, 150, 250, 500, and 1000 with a 1% edit distance threshold. We observe that when the number of threads is more than 11, the performance saturates in algorithms such as NW, SWG, and GenASM (Fig. 4.3 since the multi-threaded DPU-pipeline is fully utilized when at least 11 threads are launched [1] (see Section 2.1). We also notice that the performance is getting slightly better when using more than 11 threads in the case of WFA and WFA adaptive (Fig. 4.3). The reason is that WFA has more irregular pattern to access the memory, so there will be more threads queued at the DMA engine. Therefore having a larger total number of threads increases the chance that at least 11 of them are not queued up at the DMA engine and can fill up the DPU pipeline. Other algorithms are more regular so using at least 11 threads is enough to utilize the pipeline.

A thread-level comparison shows that the WRAM version performs better than the MRAM version for all implementations. However, in cases where the WRAM version uses less than 11 threads, the MRAM version performs better. We conclude that the WRAM version performs better than the MRAM version when it uses enough threads, and the MRAM version hides the latency of the data transfers by using more threads.

### 4.2.4 *DPU-scale Experiments*

We provide DPU scaling experiments where we increase the number of the allocated DPUs from 512 to 2500 and fix the number of read pairs, as shown in Fig.4.4. The performance scale linearly as the number of DPUs increases since the read pairs are aligned independently, and fewer read pairs are aligned by each DPU when more DPUs are allocated. It shows that the number of read pairs is doesn't affect the absolute speedup of our experiments because the alignments are done independently.

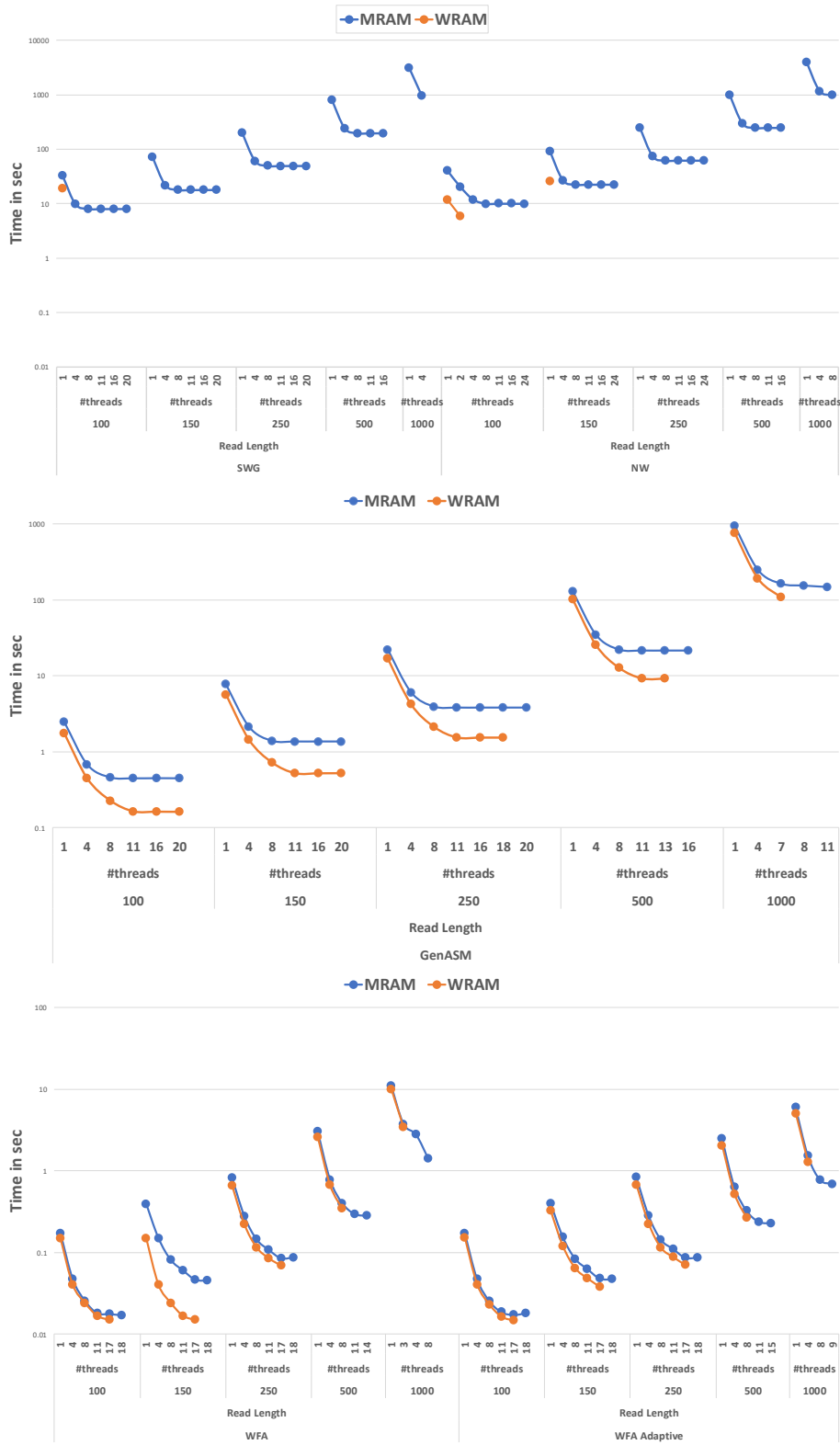


Figure 4.3: DPU thread-scaling timing results

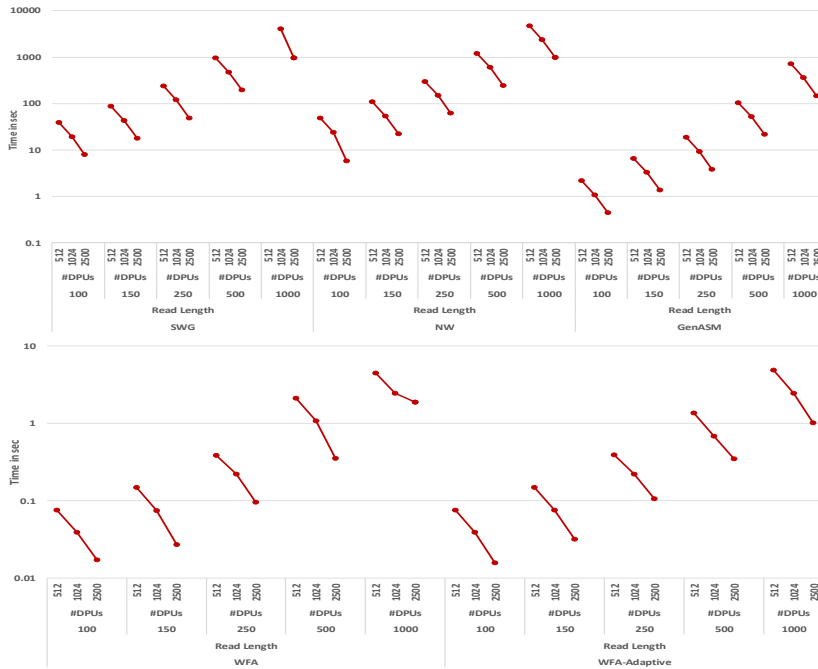


Figure 4.4: Number of DPUs scaling timing results

#### 4.2.5 Full-scale DPU and Full-scale CPU Comparison

We give a full-scale comparison between the DPU implementations and the multi-threaded baseline using the best DPU version and the best CPU configuration applied to all datasets.

Our WFA and WFA-Adaptive implementations have up to 2x speedup (Total) when aligning short reads of length 100 for ED 0-5%, and up to 30x for ED=0, 26x for ED=1%, 12x for ED=2%, 7x for ED=3%, 6x for ED=4%, and 5x for ED=5% when the CPU-DPU copy time is not accounted for (Kernel), as shown in Fig. 4.5. As we increase the read length and edit distance, we can still observe better performance for the Kernel time until we reach read length 1000 and ED greater than 2% where only WFA-adap can reach up to 3x speedup since WFA-adaptive has less memory consumption and use more threads. GenASM (Total) shows up to 3x speedup for read length 100 and up to 2x speedup for longer reads and all ED%. It also achieves up to 2-4x higher speedup for the DPU-Kernel time for all datasets. Due to the low memory consumption of GenASM, it utilizes the DPU pipeline efficiently by launching enough reads.

SWG DPU implementation achieves higher throughput (4x) for read length greater than 500 for both total and kernel time (Fig. 4.5). We observe no speedup for shorter reads and NW on all datasets, even though we are using more than 20 threads. After investigating this behavior, we learned that the DPU pipeline efficiency is almost 25% due to the large number of WRAM-MRAM transfers of small sizes (8 bytes) needed to fill up the DP table.



Figure 4.5: Full-scale DPU and full-scale CPU timing results



### 4.2.6 Large Reads DPUs and CPUs Comparison

WFA-adaptive scales efficiently when aligning longer reads and higher error rates due its low memory consumption, so we apply it on large reads of length 5,000 and 10,000. Fig. 4.6 shows that our implementation keeps scaling up, where it achieves up to 2x better performance (Total) even when increasing the error rate (except for read length 10,000 and ED=5% where we had to use only one thread).

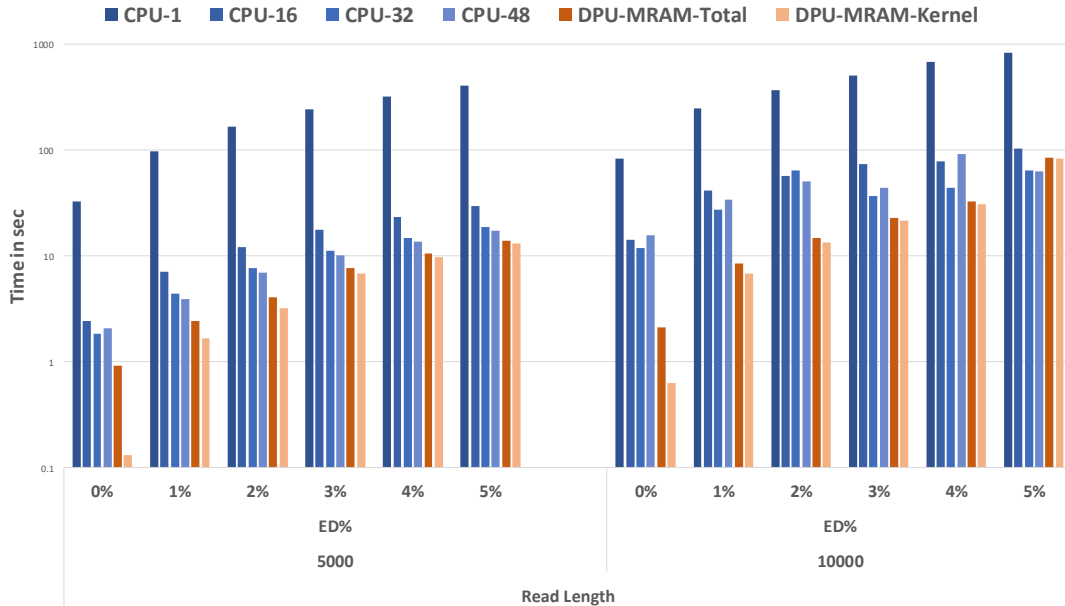


Figure 4.6: WFA adaptive on large reads DPUs and CPUs

### 4.3 Energy Consumption Results

We aim to reduce the execution time and energy consumption spent by the alignment algorithms transferring data between the main memory and the processing unit. We compare the energy consumption of our DPU implementations against the multi-threaded CPU implementations. We measure the energy consumption of the CPU implementations using the *perf* [33] tool. We estimate the DPUs' energy consumption using the power estimator provided by UPMEM [24] since the server we have access to does not currently support energy profiling. To do so, we collect several metrics such as the instructions per cycle, the DMA transfers ratio, the loads and stores ratio, and the average DMA transfer size. We use the performance counters and *dputrace* tool provided by the SDK [24] to collect the metrics. We apply this on read lengths 100, 150, 250 and 500 with ED=5% for the MRAM and WRAM DPU versions.

Fig. 4.7 shows that for read length 100 WFA, WFA-adaptive MRAM versions, and GenASM WRAM have up to 2x and 3x better energy consumption

respectively. SWG MRAM and WFA-adaptive MRAM consumes 4x and 3x less energy respectively for read length 500. We notice that the energy results follow the same trend as the timing results.

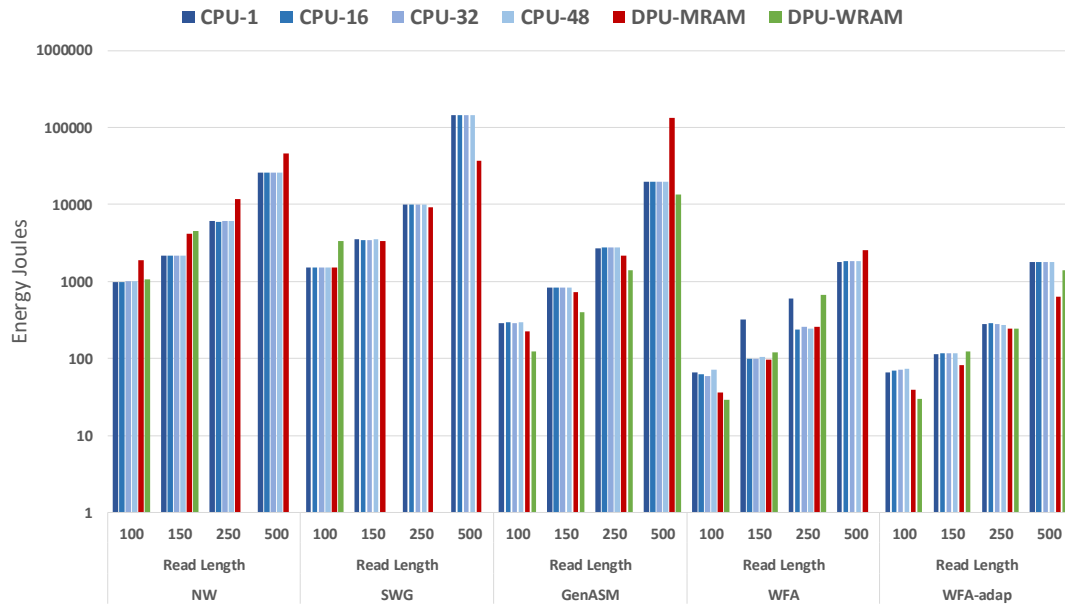


Figure 4.7: Energy Consumption on DPUs and CPUs

# CHAPTER 5

## RELATED WORK

With the current advancements of the high-throughput sequencing machines, genome analysis is bottlenecked by the read mapping phase. Genome analysis can be accelerated using different hardware and software methods [2]. In this section, we discuss recent works that managed to accelerate genome analysis using PIM architecture.

*upVC* [15] introduce an optimized implementation of the variant calling process on the UPMEM-PIM architecture. The variant calling process can be seen as a loop that consists of 4 independent tasks. The first gets the read packets from external storage support. The second dispatches these reads to the DPUs. The third map reads to the reference genome using the DPUs, and the fourth updates a variant calling data structure with the mapping results. This loop was parallelized by pipelining the tasks on the main processor and running the read mapping on the PIM memory concurrently. The read mapping algorithm uses Hamming distance and banded Smith-Waterman to perform the alignment. Their experimentation shows that the mapping task performed on the DPUs consumes the longest time, leading to a high inactivity rate in the CPU. Nevertheless, *upVC* verifies the efficiency of using PIM architecture on mapping and variant calling processes. Our work has different approach to perform read alignment on UPMEM. First, we perform high-throughput read pairs alignment on the UPMEM-PIM system, while *upVC*'s goal is to perform variant calling and read mapping against a reference genome. Also, we implement different alignment algorithms than that used in *upVC*. We implement NW, SWG, GenASM and WFA global alignment algorithms, while *upVC* implements Smith-Waterman local alignment and Hamming-Distance pre-alignment algorithms.

GenASM [21], as discussed in sec. 2.3.4, uses 3D-stack memory to design a near-memory framework and accelerate ASM, which can significantly speed up multiple use cases such as pre-alignment filtering, read alignment, and edit distance calculation. It improves and accelerates the Bitap algorithm to support highly parallel and short and long read alignments, and adds a bit vector-based traceback algorithm.

GRIM-Filter [34] is a novel seed location filtering algorithm that has been optimized to take advantage of 3D-stacked memory systems, which include computing within a logic layer stacked beneath memory layers, to perform efficient processing near memory. Unlike GenASM and Grim Filter, which use customized hardware designs that are still a research prototype, our work use a real programmable processing-in-memory hardware to implement state-of-the-art read alignment algorithms.

Beyond genomics and the UPMEM PIM architecture, many PIM accelerators have been proposed to accelerate a wide variety of memory-bound applications such as machine learning [35, 36, 37, 38, 39, 40, 41, 42], graph processing [43, 44, 45, 46], and sparse matrix computations [10]. Our work focuses on studying pairwise read alignment on the UPMEM PIM architecture.

## CHAPTER 6

# CONCLUSION AND FUTURE WORK

In this thesis, we show that genome sequence analysis can be effectively accelerated using PIM architectures. To do so, we accelerate genome analysis by implementing the memory-bounded alignment algorithms such as WFA, GenASM, SWG, and NW on the first real PIM system.

We provided a custom memory allocator and implemented MRAM and WRAM versions for each algorithm to address the memory management challenges caused by implementing them on UPMEM. Our workload aligns a large number of read pairs in parallel across many DPUs.

We evaluate the performance of these implementations on the PIM system and compared them against a server-grade multi-threaded CPU baseline. The evaluation demonstrates that most of our implementations outperform the multi-threaded CPU implementations in terms of speedup and energy and for different read lengths and error rates.

In our future work, we would like to enable the alignment of longer reads on DPUs by parallelizing the alignment of a single read pair across multiple DPU threads and/or dividing large reads into smaller sequences.

# APPENDIX A

## ABBREVIATIONS

PIM	Processing in Memory
GSA	Genome Sequencing Analysis
HTS	High Throughput Sequencing
ASM	Approximate String Matching
WFA	Wave Front Alignment Algorithm
NW	Needleman-Wunsch
SWG	Smith-Waterman-Gotoh
DP	Dynamic Programming
MRAM	Main Memory
WRAM	Working Memory
IRAM	Instruction Memory
ED	Edit Distance
DMA	Direct Memory Access

# BIBLIOGRAPHY

- [1] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture,” *arXiv preprint arXiv:2105.03814*, 2021.
- [2] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, “Accelerating genome analysis: A primer on an ongoing journey,” *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.
- [3] S. Marco-Sola, J. C. Moure López, M. Moreto Planas, and A. Espinosa Morales, “Fast gap-affine pairwise alignment using the wavefront algorithm,” *Bioinformatics*, no. btaa777, pp. 1–8, 2020.
- [4] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “Processing data where it makes sense: Enabling in-memory computation,” 2019.
- [5] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “A modern primer on processing in memory,” 2020.
- [6] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, “Quantifying the energy cost of data movement in scientific applications,” in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 56–65, 2013.
- [7] W. H. Wen-meì, I. El Hajj, S. G. De Gonzalo, C. Pearson, N. S. Kim, D. Chen, J. Xiong, and Z. Sura, “Rebooting the data access hierarchy of computing systems,” in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–4, IEEE, 2017.
- [8] F. Devaux, “The true processing in memory accelerator,” in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24, IEEE Computer Society, 2019.
- [9] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware,” in *2021 12th International Green and Sustainable Computing Conference (IGSC)*, pp. 1–7, IEEE, 2021.

- [10] C. Giannoula, I. Fernandez, J. Gómez-Luna, N. Koziris, G. Goumas, and O. Mutlu, “Towards efficient sparse matrix vector multiplication on real processing-in-memory systems,” *arXiv preprint arXiv:2204.00900*, 2022.
- [11] D. Lavenier, C. Deltel, D. Furodet, and J.-F. Roy, *MAPPING on UPMEM*. PhD thesis, INRIA, 2016.
- [12] D. Lavenier, C. Deltel, D. Furodet, and J.-F. Roy, *BLAST on UPMEM*. PhD thesis, INRIA Rennes-Bretagne Atlantique, 2016.
- [13] D. Lavenier, J.-F. Roy, and D. Furodet, “Dna mapping using processor-in-memory architecture,” in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1429–1435, IEEE, 2016.
- [14] S. Diab, A. Nassereldine, M. Alser, J. G. Luna, O. Mutlu, and I. E. Hajj, “High-throughput pairwise alignment with the wavefront algorithm using processing-in-memory,” *arXiv preprint arXiv:2204.02085*, 2022.
- [15] D. Lavenier, R. Jodin, and R. Cimadomo, “Variant calling parallelization on processor-in-memory architecture,” *bioRxiv*, 2020.
- [16] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, *et al.*, “A case study of processing-in-memory in off-the-shelf systems,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 117–130, 2021.
- [17] V. Zois, D. Gupta, V. J. Tsotras, W. A. Najjar, and J.-F. Roy, “Massively parallel skyline computation for processing-in-memory architectures,” in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–12, 2018.
- [18] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.
- [19] O. Gotoh, “An improved algorithm for matching biological sequences,” *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.
- [20] M. Alser, J. Rotman, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer, B. Balliu, D. Koslicki, P. Skums, A. Zelikovsky, C. Alkan, O. Mutlu, and S. Mangul, “Technology dictates algorithms: Recent developments in read alignment,” 2020.
- [21] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, *et al.*, “Genasm: A high-performance, low-power approximate string matching



- acceleration framework for genome sequence analysis,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 951–966, IEEE, 2020.
- [22] R. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *Commun. ACM*, vol. 35, p. 74–82, oct 1992.
- [23] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, *Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks*, p. 316–331. New York, NY, USA: Association for Computing Machinery, 2018.
- [24] <https://sdk.upmem.com/2020.3.0/>.
- [25] <https://nature.com/subjects/genomic-analysis>.
- [26] <https://www.illumina.com/systems/sequencing-platforms/novaseq.html>.
- [27] <https://www.pacb.com/technology/hifi-sequencing/>.
- [28] M. Alser, J. Rotman, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer, B. Balliu, *et al.*, “Technology dictates algorithms: Recent developments in read alignment,” *arXiv preprint arXiv:2003.00110*, 2020.
- [29] E. Ukkonen, “Algorithms for approximate string matching,” *Information and Control*, vol. 64, no. 1, pp. 100 – 118, 1985. International Conference on Foundations of Computation Theory.
- [30] H. Li, “Minimap2: pairwise alignment for nucleotide sequences,” *Bioinformatics*, vol. 34, pp. 3094–3100, 05 2018.
- [31] [https://www.ncbi.nlm.nih.gov/assembly/GCF\\_000001405.13/](https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13/).
- [32] <https://www.ebi.ac.uk/ena>.
- [33] V. M. Weaver, “Linux perl\_event features and overhead,” in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, p. 5, 2013.
- [34] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, “Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies,” *BMC genomics*, vol. 19, no. 2, pp. 23–40, 2018.

- [35] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. W. Strachan, M. Hu, S. Williams, and V. Srikumar, “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 14–26, 06 2016.
- [36] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory,” *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, 06 2016.
- [37] L. Song, X. Qian, H. Li, and Y. Chen, “PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, 2017.
- [38] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojevic, “PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, (New York, NY, USA), p. 715–731, Association for Computing Machinery, 2019.
- [39] A. Ankit, I. Hajj, S. r. Chalamalasetti, S. Agarwal, M. Marinella, M. Foltin, J. P. Strachan, D. Milojevic, W.-m. Hwu, and K. Roy, “PANTHER: A Programmable Architecture for Neural Network Training Harnessing Energy-efficient ReRAM,” *IEEE Transactions on Computers*, vol. PP, pp. 1–1, 05 2020.
- [40] S. Huang, A. Ankit, P. Silveira, R. Antunes, S. R. Chalamalasetti, I. El Hajj, D. E. Kim, G. Aguiar, P. Bruel, S. Serebryakov, *et al.*, “Mixed precision quantization for reram-based dnn inference accelerators,” in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 372–377, IEEE, 2021.
- [41] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. El Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, *et al.*, “Hardware-software co-design for an analog-digital accelerator for machine learning,” in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–13, IEEE, 2018.
- [42] P. Bruel, S. R. Chalamalasetti, C. Dalton, I. El Hajj, A. Goldman, C. Graves, W.-m. Hwu, P. Laplante, D. Milojevic, G. Ndu, *et al.*, “Generalize or die: Operating systems support for memristor-based accelerators,” in *2017 IEEE*

*International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2017.

- [43] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “GraphR: Accelerating Graph Processing Using ReRAM,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, 2018.
- [44] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.
- [45] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, 2018.
- [46] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, “GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2019.