SEQUENCE-TO-GRAPH ALIGNMENT ON A PROCESSING-IN-MEMORY SYSTEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Ömer Yavuz Öztürk
September 2025

Sequence-to-Graph Alignment on a Processing-In-Memory System
By Ömer Yavuz Öztürk
September 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Can Alkan(Advisor)

Ayşegül Dündar Boral

Mehmet Somel

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

SEQUENCE-TO-GRAPH ALIGNMENT ON A PROCESSING-IN-MEMORY SYSTEM

Ömer Yavuz Öztürk
M.S. in Computer Engineering
Advisor: Can Alkan
September 2025

Genome graphs are used to represent the genetic information and variation of a population rather than a single individual. The sequence-to-graph alignment (SGA) problem can be defined as finding the best match between a query sequence and a genome graph. SGA algorithms are expected to suffer from a memory bottleneck due to the irregularity of graph representation, and benchmarks confirm more than 50% memory boundness found in some applications. Therefore, the SGA alignment problem can benefit from processing-in-memory (PIM) technologies.

PIM is an upcoming non-von Neumann architecture that allows computing near the main memory without the need to utilize the memory bus for data transfers to the CPU and back. One of the currently available PIM technologies developed by UPMEM is an architecture consisting of thousands of DPUs (DRAM Processing Units) that allow for much faster and energy-efficient memory access.

Our contribution includes SEGAPIM, the design of a pipeline that partitions and distributes a genome graph across DPUs, directs short reads to the relevant DPUs according to their seed locations, and performs alignment solutions within DPUs. Although these distribution and routing components are part of the envisioned full system, the implemented portion of our work focuses on calculating alignment scores for each seed of each read using an adaptation of the graph wavefront alignment algorithm (GWFA) under very limited memory, followed by collecting and finalizing the results on the host CPU.

We present comparisons for accuracy and run-time of our implementation to

state-of-the-art tools such as vg and GraphAligner, and conclude that the PIM architecture at hand provides a promising speed-up and energy-saving for the SGA problem.

Keywords: Genome, Sequence-to-Graph Alignment, Processing-in-Memory.

ÖZET

BELLEK-İÇİ-İŞLEM SİSTEMİ ÜZERİNDE DİZİ-ÇİZGE HİZALAMASI

Ömer Yavuz Öztürk
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Can Alkan
Eylül 2025

Genom çizgeleri, tek bir birey yerine bir popülasyonun genetik bilgisini ve varyasyonlarını temsil etmek amacıyla kullanılmaktadır. Dizi-çizge hizalama (Sequence-to-Graph Alignment, SGA) problemi ise, bir sorgu dizisinin bir genom çizgesi üzerinde en uygun eşlemesinin bulunması olarak tanımlanır. Ancak çizge temsillerinin düzensiz yapısı, SGA algoritmalarını bellek darboğazına duyarlı hale getirmekte ve yapılan karşılaştırmalara göre bazı uygulamalarda %50'nin üzerinde belleğe bağımlılık gözlemlenmektedir. Bu nedenle, SGA probleminin bellek-içi-işlem (Processing-in-Memory, PIM) teknolojilerinden fayda görmesi beklenmektedir.

PIM mimarisi, verilerin CPU'ya aktarılması için bellek veri yolunun kullanılmasındansa, hesaplamaların ana belleğe yakın konumda yapılmasına olanak tanıyan, von Neumann dışı güncel bir mimaridir. UPMEM tarafından geliştirilen mevcut bir PIM çözümü, her biri bellek içinde çok daha hızlı ve verimli çalışabilen binlerce DPU'dan (DRAM Processing Unit) oluşmaktadır.

Katkımız olan SEGAPIM, genom çizgesinin DPU'lar arasında dağıtılması, kısa tipte sorguların tohum konumlarına göre ilgili DPU'lara yönlendirilmesi ve dizi-çizge hizalama işleminin DPU'lar içerisinde gerçekleştirilmesini içeren bir program tasarımıdır. Tüm bunlar tasarıma dahil olmalarına rağmen hayata geçirdiğimiz uygulamanın asıl odağı, sınırlı bellek koşullarına uygun biçimde her sorgunun her tohumunun çizge dalga cephesi hizalama algoritması (Graph Wavefront Alignment, GWFA) kullanılarak hizalama puanlarının hesaplanması ve elde edilen sonuçların ana CPU'da birleştirilmesidir.

Önerilen yaklaşımın doğruluk ve çalışma süresi açısından vg ve GraphAligner gibi güncel araçlarla karşılaştırmaları sunulmakta; mevcut PIM mimarisinin SGA

problemi için umut verici bir hızlanma ve enerji tasarrufu sağladığı sonucuna ulaşılmaktadır.

Anahtar sözcükler: Genom, Dizi-Çizge Hizalama, Bellek-İçi-İşlem.

Acknowledgement

First and foremost, I extend my deepest gratitude to my family, whose unwavering support has been the foundation of this journey. My mother and father not only encouraged me at every step but also carried countless burdens so that I could remain focused on my work. On many nights, they stayed awake just to keep me company at the computer, reminding me that I was never alone in this effort.

I am sincerely thankful to my advisor, Can Alkan, whose patience endured even in the face of my slow progress and whose guidance kept me moving forward. I am also grateful to the committee members, who listened carefully to my thesis defense and asked their questions with kindness.

I wish to thank Akmuhammet, Berkan, and Zülal from AlkanLab. Akmuhammet and Berkan's willingness to review my work and offer feedback greatly improved the clarity of my narrative, while Zülal's generosity in sharing her own experiences with the subject provided valuable insight. I feel especially indebted to Akmuhammet for his friendship and his selfless character, always ready to help and placing others before himself.

I am grateful to my friends Alper and Batuhan, who shared countless study hours with me. Especially to Batuhan, I owe many quiet evenings of companionship; even without obligation, he chose to sit beside me, reminding me of the strength found in solidarity.

I also thank my friends Asım, Kerem, Arman, Sami, Safa, Kuzen, Erkuzen, and many others I cannot name here, who made the years I spent at Bilkent more worthwhile. I hope never to lose touch with any of you as life moves forward.

To the friends who encouraged me with reverse psychology, insisting I would never finish on time, I thank you as well for keeping me motivated in your own way.

Finally, I dedicate these words to my one-year-old nephew, Erdem, whose smile has the power to dissolve every fatigue and whose presence makes my world infinitely brighter.

To those whose presence could not be named here but whose love has quietly guided me, I am equally grateful.

This work was funded by The BioPIM Project, supported by the European Union's Horizon Programme for Research and Innovation under grant agreement No. 101047160.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Bioinformatics Background

1.1.1 Genomics

Genomics is the study of the structure, function and evolution of genomes, which are the complete set of hereditary material of an organism [1]. Unlike genetics, which often focuses on the inheritance of individual traits or specific genes, genomics takes a broader perspective by analyzing entire genomes at once. This view enables researchers to investigate large-scale biological questions such as the genetic basis of complex diseases, evolutionary relationships among species, and the molecular mechanisms driving cellular processes. As the volume of genomic data has grown, computational methods have become indispensable for managing, analyzing, and interpreting this information.

1.1.2 Genome Sequencing

The foundation of modern genomics is genome sequencing, the process of determining the order of nucleotide bases (A, C, G, T) in DNA. Advances in sequencing technologies have led to a dramatic reduction in cost and turnaround time. The Human Genome Project [2], completed in 2003, required more than a decade of international collaboration and billions of dollars in funding. Today, high-throughput sequencing platforms can sequence a human genome in days at a fraction of the cost [3].

However, sequencing technologies typically produce error-prone partial reads rather than one complete contiguous sequence. These technologies are broadly classified into short- and long-read approaches. Short reads (50–300 bases) generally exhibit very low error rates, often below 1% (e.g., the Illumina platform error rate is below 0.4% [4]). Long reads (more than 1000 bases), in contrast, tend to be much noisier, with raw error rates often in the range of 10%–20% depending on the platform [5].

1.1.3 Genome Reconstruction

To construct the genetic sequence of an individual from their sequenced reads, these reads must be computationally assembled like puzzle pieces or aligned to some previously obtained reference, which suggests the loci of these reads, the former being more costly than the latter. The accuracy and efficiency of these secondary analysis algorithms are critical for turning raw sequencing data into comprehensible biological insights.

1.1.3.1 Sequence-to-Sequence Alignment

Traditional alignment methods map sequenced reads to a single *linear reference genome*, which is called linear alignment or sequence-to-sequence alignment (SSA). Classical sequence-to-sequence alignment implementations are based on dynamic programming (DP) algorithms (i.e., Needleman-Wunsch [6], Smith-Waterman [7]) that work in quadratic time and memory complexity. Given that, it is not feasible to use the full reference genome. Therefore, each read is compared with only subsequences of the reference, typically using seed-and-extend heuristics (Section 1.1.3.1.3) combined with DP to produce a best-scoring alignment.

In practice, we do not expect perfect matches between reads and the reference. This is due to two main sources of discrepancy:

1. **Sequencing errors**: Substitutions, insertions, or deletions (indels) introduced by the sequencing process. These errors do not reflect the true underlying DNA and their rate may change depending on the sequencing technology used. Shorter reads (∼150 bases) generally have lower error rates (∼0.1–1%).

2. **True genomic variants**: Genuine differences between the subject's genome and the reference genome, such as single nucleotide polymorphisms (SNP) and structural variants (SV) [8].

Sequencing errors and SNPs are often tolerated by sequence aligners that allow some mismatches or indels, and the goal is to find the alignments with the least difference. In some cases, the types of variation are given unequal importance, but in our case we use the most basic *edit distance* metric.

1.1.3.1.1 Edit Distance Edit distance, also known as the Levenshtein distance [9], is a string metric that measures the minimum number of single-character operations (indels and substitutions) needed to transform one string into another.

Formally, for strings $a = a_1 a_2 \dots a_m$ and $b = b_1 b_2 \dots b_n$, the Levenshtein distance $d(i, j)$ is defined recursively as:

$$d(i,j) = \begin{cases} i, & j = 0, \\ j, & i = 0, \\ \min \begin{cases} d(i, j-1) + 1 \\ d(i-1, j) + 1 \\ d(i-1, j-1) + \delta(a_i, b_j) \end{cases}, & \text{otherwise.} \end{cases}$$

where $\delta(a_i, b_j) = 0$ if $a_i = b_j$, and 1 otherwise.

Figure 1.1 provides a visualization of this concept. Each cell of the dynamic programming table represents the edit distance between the prefixes $a_1 \dots a_i$ and $b_1 \dots b_j$.

In the context of read alignment, our goal is to find the minimum value of $d(i, |q|)$ over all reference positions i, where $|q|$ denotes the length of the query sequence. We refer to this quantity as the *optimal alignment score*.

1.1.3.1.2 Wavefront Alignment (WFA) Wavefront Alignment is a fast and efficient algorithm, which is basically an optimization on sequence-to-sequence alignment under some constraints. Unlike traditional alignment methods that fill the entire dynamic programming matrix (which can be too large for long sequences), WFA focuses only on computing the parts of the matrix that matter, specifically the cells with alignment scores up to a maximum threshold or up to the optimal alignment score.

Theoretically, given an arbitrary scoring metric, it is not possible to find the guaranteed optimal alignment score without computing all DP cells. WFA achieves this under the constraint that the match score be equal to zero and the

		G	A	C	T	T	A	G	A	A	C
	0	1	2	3	4	5	6	7	8	9	10
G	1	0	1	2	3	4	5	6	7	8	9
A	2	1	0	1	2	3	4	5	6	7	8
T	3	2	1	1	1	2	3	4	5	6	7
T	4	3	2	2	1	1	2	3	4	5	6
A	5	4	3	3	2	2	1	2	3	4	5
C	6	5	4	3	3	3	2	2	3	3	4
A	7	6	5	4	4	4	3	3	2	3	3

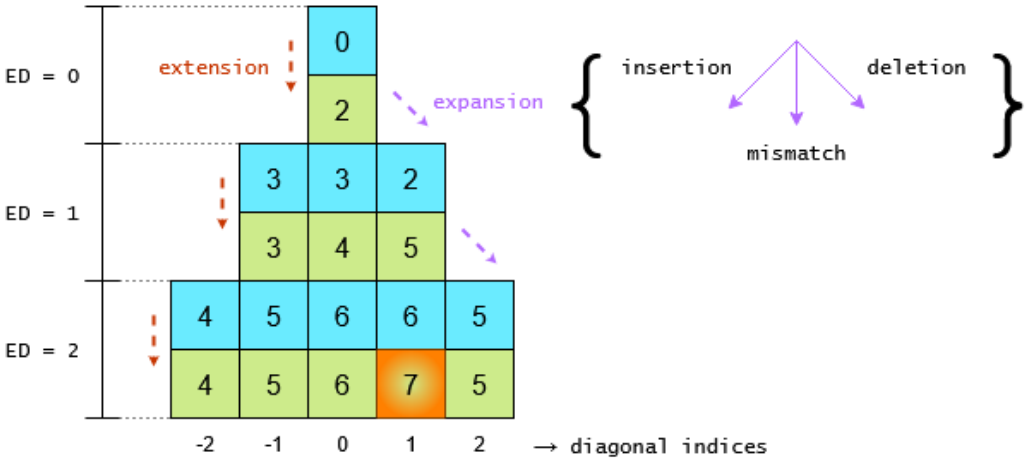
Figure 1.1: Populated DP table for two sequences using edit distance as the scoring metric. Minimum value in the last row (end of the query) represents the edit distance of the best alignment. The colored path represents the trace of the optimal alignment.

substitution and indel scores be of the same sign (positive without loss of generality). The edit distance metric falls right into this constraint, and therefore WFA becomes applicable.

In essence, where traditional methods compute all possible alignments between each possible prefix of two sequences, WFA intelligently computes only the ones that count. As a result, it avoids the heavy quadratic computation of classic methods and runs much faster, especially when the sequences are similar. WFA uses an array to track the progress of the furthest-reaching cell in each diagonal, as shown in Figure 1.2b.

		G	A	C	T	T	A	G	A	A	C
	0	1	2	3	4	5	6	7	8	9	10
G	1	0	1	2	3	4	5	6	7	8	9
A	2	1	0	1	2	3	4	5	6	7	8
T	3	2	1	1	1	2	3	4	5	6	7
T	4	3	2	2	1	1	2	3	4	5	6
A	5	4	3	3	2	2	1	2	3	4	5
C	6	5	4	3	3	3	2	2	3	3	4
A	7	6	5	4	4	4	3	3	2	3	3

(a) Computed wavefront cells shown on the otherwise fully populated DP table. Blue cells are extended along exact matches to produce green cells, and green cells are expanded on neighbors to produce blue cells, as the algorithm switches between these two phases.



(b) Progress array of the tracked furthest cells of increasing edit distances, for the same case as given in Figure 1.2a.

Figure 1.2: WFA algorithm and data visualization.

1.1.3.1.3 Seed-and-Extend Strategy

Most state-of-the-art aligners adopt a *seed-and-extend* [10] approach to balance speed and accuracy:

1. **Seeding:** Exact or near-exact short matches (seeds) are identified between the query read and the graph. Seeding can be performed using minimizers, k-mers, or other hashing-based strategies. These seeds provide anchor points that drastically reduce the search space.
2. **Extension:** Each seed is then tested for complete alignment using an alignment algorithm. Most of the time, a seed is not at the exact start or the exact end of a read, so the extension occurs in both directions.

The advantage of this method is that it avoids performing expensive full dynamic programming across the entire graph. Instead, computation is focused on the regions most likely to produce correct alignments. Some filtering techniques can be applied to eliminate seeds that are produced by repeating short sequences in the genome. Even after filtering, there can be large numbers of seed locations per read, which need to be extended to see if they result in actual alignments.

1.1.3.2 Reference Bias in Linear Alignment

When the subject's genome diverges from the reference, aligners may fail to produce an optimal mapping. This phenomenon is known as *reference bias* [11]. Reads containing true variants may be misaligned, forced into suboptimal placements, or discarded entirely. The result is that real biological differences are underrepresented or even missed, which is particularly problematic in highly diverse genomic regions.

1.1.4 Pangenomes

To address the problem of reference bias, researchers have developed the concept of *pangenome* [12], which integrates multiple reference genomes into a unified

representation that captures the full genomic diversity of a population. Genome graphs are a natural formalism for pangenomes, as they encode alternative alleles, structural variants, and haplotype diversity as branching paths in a graph.

By incorporating many possible genomic sequences into a single structure, genome graphs substantially increase the likelihood that each read finds a correct alignment. Even if one reference path diverges from the subject's genome, another path may capture the true sequence, enabling an accurate placement of the read. This richer representation improves the accuracy of sequence alignment and variant discovery, particularly in highly polymorphic or structurally complex regions of the genome.

In summary, while sequence-to-sequence alignment against a linear reference remains widely used, its inherent bias limits accuracy in diverse populations. Pangenomes, representable as genome graphs, provide a principled solution that reduces misalignment, mitigates reference bias, and offers a richer substrate for downstream genomic analysis.

1.1.4.1 Graphical Fragment Assembly (GFA) Format

The *Graphical Fragment Assembly (GFA)* format has become the standard for representing genome graphs. It is widely supported by graph alignment tools such as `vg map` and GraphAligner.

One of the advantages of the GFA format is that it is human-readable, unlike more compact binary formats such as variation graphs used in the `vg` toolkit. This makes GFAs easier to inspect and manipulate, especially during the development pipeline.

A GFA file encodes two primary elements that is of our concern:

- **Segments (S):** Each segment corresponds to a contiguous DNA sequence, identified by a unique segment ID.

- **Links (L):** Links connect segments and define the topology of the graph by specifying their adjacency and orientation (forward or reverse). Depending on the orientations of the connected segments, a link may also represent a connection to the reverse complement of one or both of the segments. In the general format, segments may represent overlapping loci, and their differences over these loci can be explicitly represented by the links connecting each pair.

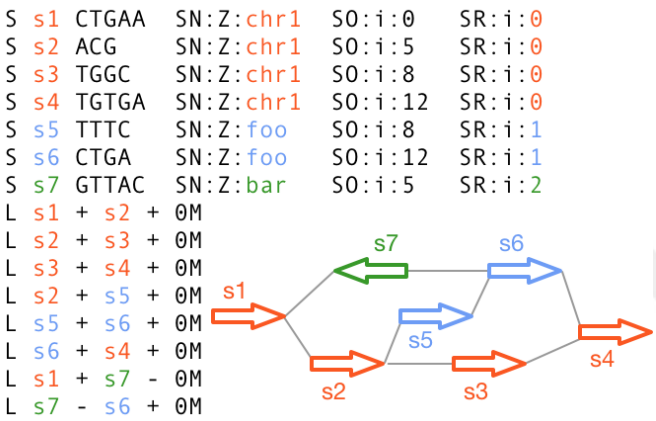


Figure 1.3: rGFA file format and its visualization [13]. All links represent a connection without overlap ("0M"). The example contains mostly positive-to-positive oriented links, except the ones connecting to segment $s7$. The path going through $s1 \rightarrow s7 \rightarrow s6$ describes the sequence CTGAA→GTAAC→CTGA, notice the reverse complement of $s7$ is included, instead of the original.

1.1.4.1.1 rGFA Subset In this work, we specifically focus on the **rGFA** subset of the format. The rGFA specification disallows overlaps between segments, which means that links only indicate connectivity and orientation, but not explicit overlaps. This restriction makes rGFA more intuitive and lightweight to work with while still preserving all the information required for read alignment. By avoiding overlap fields, rGFA eliminates potential ambiguities in graph interpretation and simplifies preprocessing.

1.1.5 Sequence-to-Graph Alignment

Sequence-to-graph alignment (SGA) generalizes the classical SSA by replacing the linear reference string with a graph-structured reference.

In the context of SGA, the reference is a labeled graph $G = (V, E)$ whose nodes (segments) carry DNA strings and whose edges (links) encode adjacencies. The alignment objective is to find a path through G whose spelled sequence best matches the query under a chosen scoring model (e.g., edit distance). This generalization introduces several fundamental differences.

- **Irregular state space.** The DP frontier is shaped by the topology of G rather than a rectangular grid. Node lengths vary, branching factors can be high, and cycles may be present, creating data-dependent control flow and heterogeneous work.
- **Memory behavior.** Whereas SSA often maintains a band or a small set of DP rows, SGA must track active states across many nodes and edges. The number of concurrently active positions and thus the memory footprint depend on query length and graph connectivity in that region, making memory demands less predictable and typically larger. In addition, reverse complements of the node sequences must be stored along with the originals in order to eliminate additional processing during alignment.
- **Navigation overhead.** Advancing along the reference involves node/offset tracking and edge traversals (including orientation), in contrast to unit cost index increments in linear alignment. This adds pointer chasing and conditional control flow that are unfriendly to caches and SIMD (Single Instruction Multiple Data).

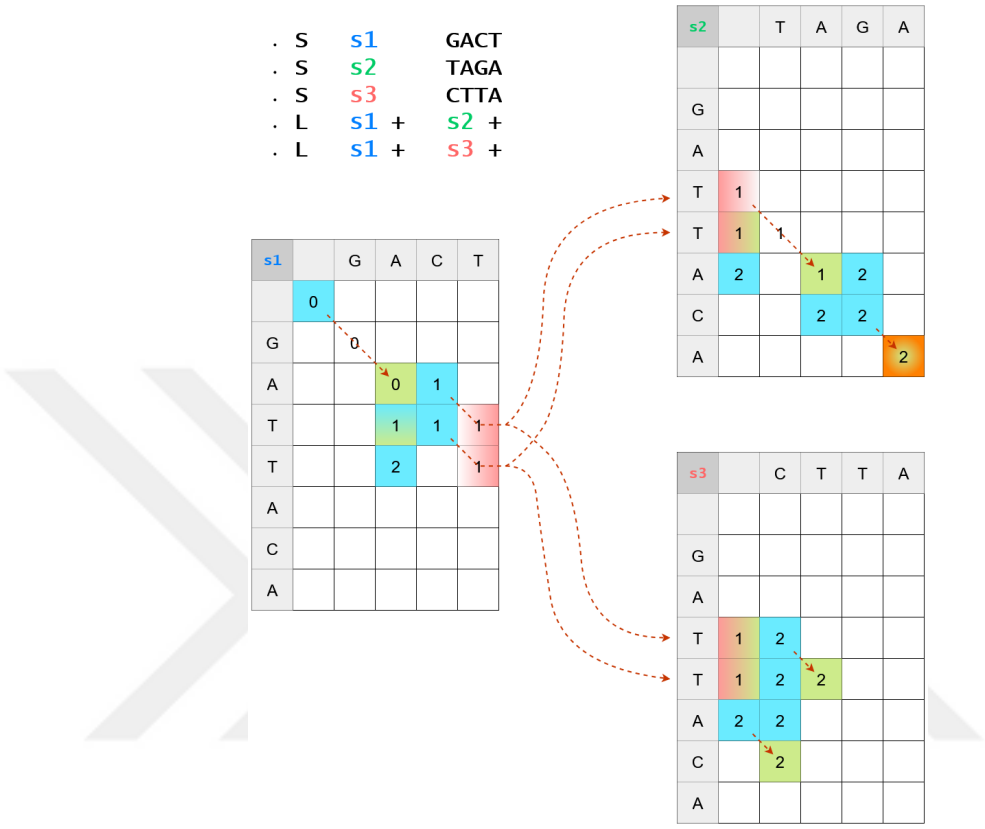
Unpredictable memory demands for SGA solutions suggest that these solutions may be prone to memory boundness (more on this in Section 1.2.1).

1.1.5.1 Graph Wavefront Alignment (GWFA)

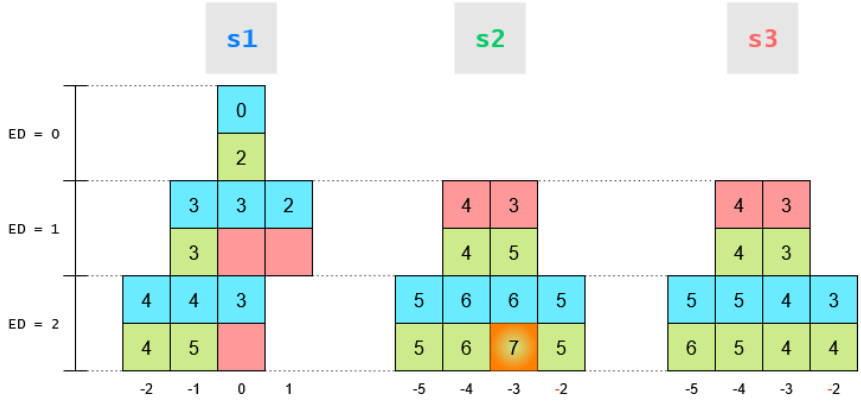
The Graph Wavefront Alignment (GWFA) algorithm [14] extends the WFA algorithm, which was originally built for SSA, to graph-structured references. When a wavefront reaches the boundary of a segment, it is terminated on the original segment and is copied to neighboring segments with the same progress score.

In WFA, tracked diagonals are consecutive and in the same virtual DP table, so they can be stored in a structure as simple as an array. However, in GWFA, each segment requires a separate virtual DP table, and the tracked diagonals in the same virtual DP table cannot be assumed to be consecutive. In addition, the structure and size of these wavefronts are difficult to predict before actually progressing through the algorithm. In short, the GWFA process is more complex and less predictable than that of WFA.

To handle irregular wavefront structures, GWFA stores its wavefront cells in a hash table keyed by a tuple of the segment and the diagonal index, which maps to the corresponding progress score.



(a) Computed wavefront cells shown on the graph DP tables. Reddish cells imply edge traversals. Blue cells are extended along exact matches to produce green cells, and green cells are expanded on neighbors to produce blue cells, as the algorithm switches between these two phases.



(b) Progresses of the tracked furthest cells per segment sequence with increasing edit distances, for the same case as given in Figure 1.4a.

Figure 1.4: GWFA algorithm and data visualization.

1.2 Memory Bottlenecks and Architectural Responses

1.2.1 Memory Boundness

The term *memory-bound* defines a computer program that loses a significant portion of its instruction cycles waiting for memory accesses.

In a conventional CPU system, the CPU cores and DRAM banks are separate modules. When the CPU requires a piece of data, it fetches this piece of data either from DRAM banks or from the cache if the same piece of data was used recently. The former is an expensive operation, especially when we consider that different cores have to share the same means that allow them to access memory. The additional waiting time caused by this resource sharing increases with the number of cores running concurrently. This issue becomes more pronounced when a program is data-intensive, meaning that it constantly requires fresh data or frequently accesses pieces of memory far apart. In this case, communication between the cores and memory costs more time and energy than performing actual computations on the cores, and memory access becomes the primary bottleneck of a program.

1.2.1.1 Benchmarks on Sequence-to-Graph Alignment Tools

To evaluate the memory boundness of typical graph alignment logic, we analyzed two state-of-the-art graph alignment programs, which are GraphAligner and *vg map*.

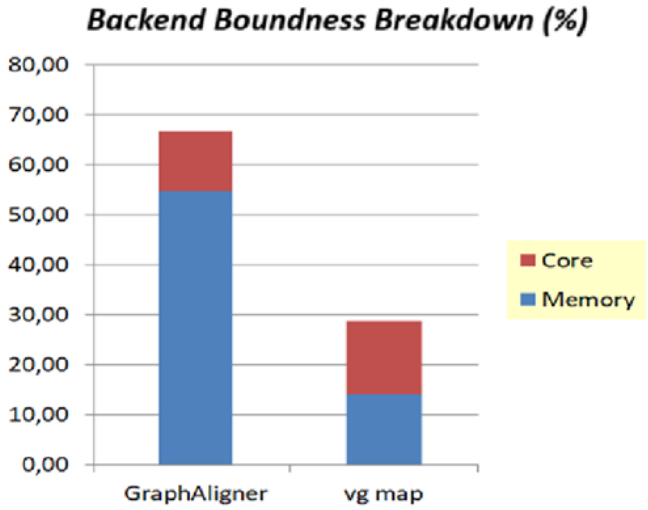


Figure 1.5: Memory boundness and core boundness (together forming backend boundness) level breakdown of GraphAligner and *vg map* obtained with `toplev` from `pmu-tools` [15] using 8 threads. GraphAligner exhibits strong memory-bound behavior even at low thread counts, while vg map appears less memory-bound under the same conditions.

	vg map (64 threads)	
Function	CPU Time	% Memory Bound
gssw_sw_sse2_byte	89021 s	37.2
__memset_avx2_unaligned_erms_rtm	8151 s	34.3
sdsl::coder::elias_delta::decode_prefix_sum	6415 s	25.3
sdsl::rank_support_il<(unsigned char)1, (unsigned int)512>::rank1	4054 s	78.4
sdsl::bits::read_int	3954 s	92.1

Figure 1.6: Top time-consuming functions of *vg map* with 64 threads profiled using Intel VTune Profiler [16]. The Smith-Waterman kernel (`gssw_sw_sse2_byte`) dominates runtime, and shows 37.2% memory boundness. This indicates that VG becomes increasingly memory-bound as the thread count increases.

Figure 1.5 shows that, at 8 threads, GraphAligner is already dominated by memory stalls, indicating an intrinsic memory-bound behavior that does not depend on aggressive parallelism. Under the same conditions, *vg map* exhibits a smaller memory-bound fraction, suggesting a more balanced mix of core execution

and memory stalls at low thread counts.

When *vg map* is scaled to 64 threads (Figure 1.6), the Smith-Waterman kernel, which is the dominant contributor to total runtime, shows 37% memory boundness, which is high for the primary compute kernel. As concurrency increases, this level of memory sensitivity in the main kernel is sufficient to push overall performance toward a memory-bound regime; the remaining routines (other functions in the figure) also exhibit memory stalls, but account for a smaller share of time. In short, *vg map* transitions from moderately balanced at low thread counts to clearly memory-bound at scale, reinforcing that large-scale SGA is constrained by memory access rather than pure arithmetic throughput.

1.2.2 Processing-in-Memory (PIM)

Processing-in-memory (PIM) [17] is an architecture paradigm that aims to mitigate the latency caused by memory access bottleneck by moving the computation closer to where data resides, inside or near the memory chips themselves. PIM reduces the growing disparity between the speed of processors and the bandwidth limitations of memory access.

In conventional systems, CPUs or GPUs repeatedly fetch data from off-chip DRAM through narrow and power-hungry channels. For workloads dominated by data movement rather than computation, such as genomics or graph analytics, this results in cores spending most of their time stalled on memory operations. Common acceleration methods, such as SIMD, do not address this memory bottleneck. PIM mitigates it by embedding lightweight processing units directly in memory banks, therefore reducing the need for long-distance data transfers.

This architectural variation brings two key benefits:

- **Lower latency and higher effective bandwidth.** Because data are processed where they are stored, fewer round-trip requests over the memory bus are required. This increases effective memory throughput and reduces

waiting time.

- **Energy efficiency.** Data movement across chip boundaries is a major contributor to energy cost in modern systems [18]. By minimizing such transfers, PIM significantly improves performance-per-watt for memory-bound workloads.

1.3 UPMEM PIM Architecture

The first commercially available PIM system is the UPMEM architecture [19]. UPMEM augments standard DRAM modules with thousands of lightweight *DRAM Processing Units* (DPUs), each embedded inside a DRAM chip. Each DPU contains its own small working memory (WRAM, 64 KB) and a larger main memory (MRAM, 64 MB), as well as a simple instruction pipeline, enabling it to run general-purpose programs directly where the data are stored. WRAM provides low-latency access suitable for active computations, while MRAM serves as the high-capacity storage area for sequences, graphs, and intermediate data. They require careful management of WRAM buffers and explicit scheduling of MRAM transfers for optimal performance.

From a programming perspective, a UPMEM-equipped system exposes both a conventional CPU host and a large number of DPUs. Data can be partitioned across DPUs, with computations that are memory-bound on a CPU instead executed in parallel within memory. However, the limited WRAM capacity means that algorithms must be restructured to fit their working sets into 64 KB per DPU, often requiring compact data structures and careful reuse of buffers. Despite these constraints, the architecture is a candidate for accelerating sequence-to-graph alignment, a workload whose performance is constrained primarily by memory access rather than raw arithmetic throughput.

1.3.1 Communication and Memory Limitations

A key limitation of the UPMEM architecture is the absence of inter-DPU communication. Each DPU operates in isolation, with no direct channels for data exchange or synchronization; all coordination must pass through the host CPU. Although this simplifies hardware design by avoiding complex interconnects, it shifts the burden of coordination to the host and can create bottlenecks in communication-intensive workloads.

Another fundamental constraint is the limited memory per DPU. Each DPU provides only tens of kilobytes of low-latency working memory (WRAM) and a modest amount of higher-latency main memory (MRAM). This makes it difficult to fit large or irregular data structures in the address space of a single DPU, requiring careful partitioning of input data and compact in-memory representations.

Although this small-memory design reduces the classical memory access bottleneck by bringing computation closer to data, it significantly increases programming complexity. Memory layout, buffer reuse, and stack sizing all require explicit management. The challenge is compounded by the lack of inter-DPU communication: each DPU must operate entirely on its own partition of the data. To achieve scalable performance without becoming host-bound, data structures must therefore be divided so that each DPU can progress independently with minimal CPU-mediated transfers.

1.3.2 Absence of Bounds Checking

In DPU WRAM, the run-time libraries do not enforce bounds checking or prevent buffer overflows. As a result, every write operation must be carefully managed by the programmer to ensure that it does not corrupt data outside the allocated region. This can be good because it increases flexibility, but it also makes it harder to implement a memory-safe program.

1.3.3 Scalability

In the UPMEM architecture, when equally sized problems are distributed across DPUs with little or no inter-DPU synchronization, the execution time remains approximately constant as the size of the dataset and DPU count grow together [20].

To achieve this, all necessary data must be placed in each DPU's local memory. This motivated our graph partitioning strategy, which will be explored in Section 2.1.3, ensuring each subgraph and its batch of queries fit within WRAM and MRAM.

Sustained CPU-DPU bandwidth also scales with the number of DPUs, though sublinearly. For 64 DPUs, the aggregate sustained bandwidth is 20.13× higher for CPU→DPU transfers and 38.76× higher for DPU→CPU transfers [20] compared to single DPU transfers. This will be the motivation behind the scaling of our test results in Section 3.1.4 to have a fairer comparison.

1.3.4 Parallelization Inside DPU

There are 16 hardware threads per DPU in an UPMEM system. At least 11 of these threads must be used to fully utilize the pipeline [20]. This decreases the amount of memory per thread. In a realistic case, with as much data used in common among the threads as possible, the total memory, including a stack and allocatable memory private to a thread, is around 4-5 KB at most.

1.3.5 Energy Usage

The programs that perform better in UPMEM PIM in terms of performance also consume less energy [20]. This is related to the energy wasted by memory bottlenecks, the source of execution time, and the energy consumption is the

same after all.

In summary, The architectural trade-offs of UPMEM simultaneously alleviate the fundamental memory access bottleneck while introducing new constraints that strongly shape algorithm design. These realities motivate specialized strategies for partitioning, memory management, and parallelization, which are critical for implementing complex workloads such as sequence-to-graph alignment on PIM systems.

1.4 Related Work

1.4.1 SeGraM

Several hardware acceleration efforts have been proposed for sequence-to-graph alignment, one of which is SeGraM, a domain-specific accelerator that targets sequence-to-graph alignment [21]. SeGraM implements a specialized processing pipeline designed to reduce the cost of dynamic programming on graph-structured references. They report significant speed-ups over CPU-based baselines, highlighting the potential of hardware acceleration for this problem class.

However, SeGraM is an accelerator with a rigid hardware pipeline. Although it demonstrates performance gains, being a hardware designed only for this purpose, it sacrifices programmability and flexibility. In contrast, our design explores a general-purpose PIM device, programmable with multithread capabilities, only under some memory and communication constraints.

These distinctions emphasize the novelty of evaluating sequence-to-graph alignment on a PIM platform rather than on a custom accelerator.

1.5 Motivation

Given that sequence-to-graph alignment is dominated by data movement rather than arithmetic intensity, Processing-in-Memory (PIM) systems such as UPMEM DPUs provide a promising alternative. By moving computation closer to where the data reside, PIM architectures can reduce memory access latency, alleviate bandwidth pressure, and improve energy efficiency. SEGAPIM is proposed as a sequence-to-graph alignment pipeline that exploits these architectural advantages.

1.6 Thesis Statement and Contributions

This thesis investigates the feasibility of accelerating sequence-to-graph alignment for short-read sequencing data in UPMEM's Processing-in-Memory architecture. We named the implementation **SEGAPIM** (**Se**quence-to-**G**raph **A**lignment with **P**rocessing-**I**n-**M**emory). The main contributions are as follows.

1. A memory-efficient adaptation of the Graph Wavefront Alignment (GWFA) algorithm for UPMEM DPUs.
2. A preprocessing pipeline design that prepares the input set to fit within per-DPU memory constraints.
3. An implementation of memory layouts and data transfer strategies tailored to WRAM and MRAM organization.
4. Performance and accuracy evaluation of SEGAPIM against state-of-the-art CPU-based graph aligners (GraphAligner and *vg map*).
5. Performance and accuracy evaluation of SEGAPIM with different parameter sets.
6. An analysis of scalability and bottlenecks, highlighting the trade-offs of PIM-based sequence-to-graph alignment.

7. Projected speedups with a single rank of UPMEM PIM (64 DPUs), achieving **1.4–2.5**× acceleration over *vg map* and over **2.7**× compared to GraphAligner, when the baselines are executed with 64 threads.

1.6.1 Rationale for Choosing GWFA

Our choice of GWFA was motivated by both theoretical and practical considerations. On the theoretical side, its asymptotic optimality and relatively compact memory footprint make it an attractive candidate for evaluating how well a PIM architecture handles graph alignment workloads. On the practical side, previous work benchmarking various linear sequence-to-sequence alignment algorithms on the UPMEM PIM system found that WFA consistently outperformed other state-of-the-art methods in terms of throughput and scalability [22]. Building on this observation, we hypothesized that its graph-based extension, GWFA, would similarly showcase the strengths and limitations of PIM in handling more complex alignment tasks.

Chapter 2

Method

In this thesis, we present SEGAPIM, an adaptation of graph wavefront alignment on a real-world processing-in-memory architecture, namely UPMEM DPU. The pipeline of SEGAPIM encompasses various stages, including the preprocessing of input data by the host CPU, repeatedly loading chunks of this input data into PIM-enabled memory, repeatedly performing sequence-to-graph alignments in PIM-enabled memory, and finally, after all queries are done, the collection and postprocessing of results by the host CPU.

2.1 Pre-processing

2.1.1 Graph Orientation Normalization

As mentioned in Section 1.1.4.1, links in rGFA format can specify connections for reverse complements of segments. However, this adds to the complexity of the program because there arises a potential need for the reverse-complemented sequences of segments in runtime, which we then have to either have in store or compute on the go. Even though this is not a big issue for CPU-based aligners, it

is not affordable for a low-memory DPU adaptation. Therefore, we wish to eliminate this complexity by normalizing the orientations of all reads, which means all links become positive-to-positive oriented, and the storage requirement of the graph is reduced.

To achieve this, all segments should be able to be divided into two distinct sets, where the links with positive-to-positive or negative-to-negative orientations only connect the segments in the same set, and the links with positive-to-negative or negative-to-positive orientations only connect segments from different sets. We argue that this should be achievable for most segments in the majority of genome graphs. If there are segments and links conflicting with this property, they can be treated as special cases, and their reverse complements could be stored with probably negligible sacrifice of memory.

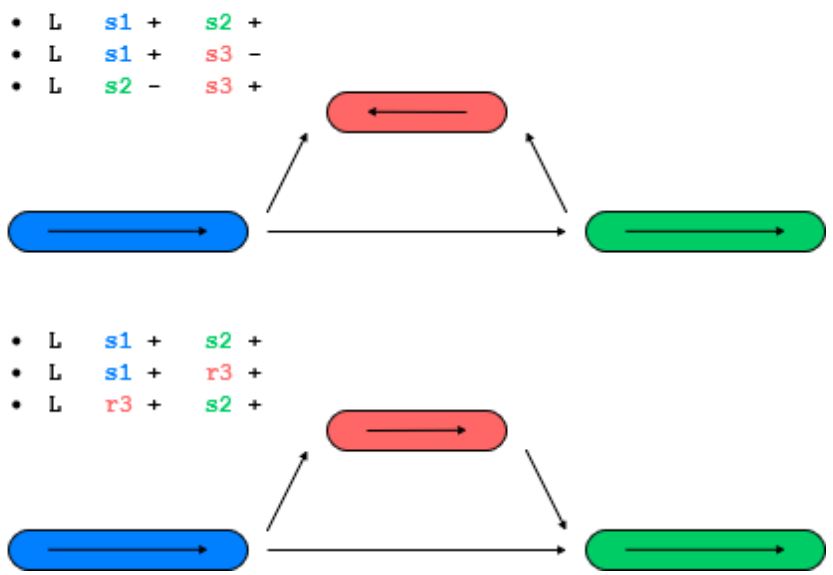


Figure 2.1: Link set of a normalizable rGFA, and its normalized form. Segments are separated into two sets like $s1, s2$ and $s3$, then the segments in the latter set are reversed ($r3$) along with orientations on the related links.

The strategy of normalization basically is to put the first segment in one of the two sets and discover which set each segment belongs to by going through all links in the graph, skipping conflicting links to later account for them. After putting all segments in two distinct sets, reverse-complement all segments in one

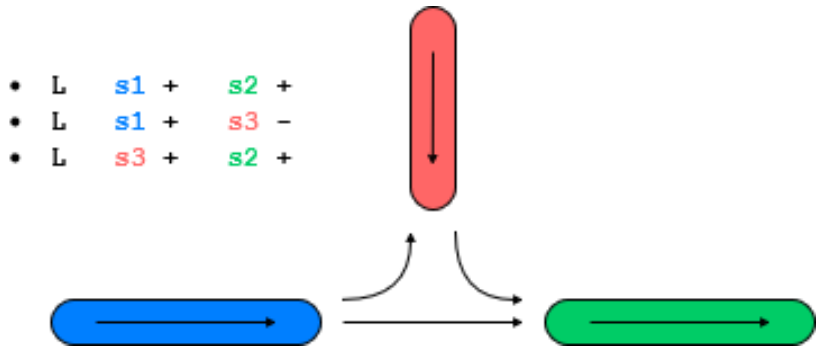


Figure 2.2: Link set of a non-normalizable rGFA. Segments are not separable into two such sets.

set, and also reverse their orientations in their links. Then, for the links with negative-to-negative orientations, change the order of segments and also revert their signs, preserving the same meaning with a different representation. In the end, all links in the graph will have positive-to-positive orientations, other than those expected to be negligible exceptions.

Figure 2.2 shows an example of a set of links to which we cannot cleanly apply normalization. In real-world directed acyclic graph (DAG) examples, we do not expect to see such structures, as at least one of the segments conflicts with the idea of each segment representing only one strand of the DNA.

2.1.2 Two-Bit Representation of Bases

Leveraging the fact that DNA sequences consist of only four bases (A, C, G, T) and encoding each base using two bits, we achieve a 4× reduction in memory usage compared to ASCII encoding. The conversion exploits the binary structure of ASCII codes by extracting specific bits without requiring conditional branching. Although this compact representation reduces memory usage, base comparisons require additional bit manipulation during alignment because of the inability to compare individual sets of bits rather than bytes.

2.1.3 Graph Partitioning with Overlaps

When multiple DPUs are used, the graph must be partitioned into smaller subgraphs. Each DPU stores and processes queries that correspond only to its assigned subgraph. To preserve all possible walks of the length and to avoid losing alignments at subgraph boundaries, an overlap of at least *read length + maximum edit distance* must be enforced between consecutive partitions.

2.1.4 Seeding with GraphAligner

The current implementation of SEGAPIM makes use of a modified GraphAligner to perform minimizer-based seeding. By modification, the resulting seeds are sorted according to the segment ID they correspond to, and seeds having the same segment ID are sorted according to the offset of the seed on the segment. In this way, consequent seeds are likely to correspond to the same piece of the genome graph, and therefore the memory locality for the query-directing stage and also the alignment stage is increased.

It is worth noting that while it is ideal to have all exact matches, including those traversing graph links, GraphAligner uses a seeding strategy that finds only the exact matches that are fully contained in a single-segment sequence, arguing that it is a negligible loss.

2.1.5 Query Directing

After all the above steps, only the direction of queries/seeds to DPUs that hold the related subgraphs remains. This can be done on the host partition by populating a hash table in the partition stage, then using this hash table to efficiently obtain the target subgraph (or subgraphs if the seed points to an overlapping region) for the seed. The said hash table would take a segment ID as input and return a compact map that links the index intervals of the segment to their corresponding

subgraphs. This allows each segment to be queried only once for all seeds it contains, enabling efficient determination of the target DPUs for those seeds.

For the expected general case where the whole genome graph could fit in the full UPMEM DPU system at one go, multiple threads on the CPU side can split all seeds among themselves and determine the target subgraphs for each seed in parallel. This can be trivially adapted to a larger genome graph case. Then, each DPU transfer buffer can be populated with the corresponding seeds. Finally, the buffers will be transferred in parallel to the DPUs of the same rank.

2.2 PIM Memory Management

2.2.1 WRAM Layout

WRAM hosts every type of memory that would be required in DPU run-time, including synchronization structures, thread stacks, and the rest, which is freely allocatable memory.

Since WRAM does not support memory deallocation once allocated, careful planning of memory layout is essential. If memory is allocated in small chunks, fragmentation can quickly render large portions of memory unusable, even if sufficient free space exists in total. To address this, our design enforces contiguous allocations and reuses fixed-size buffers across query batches.

We reserve the allocatable space in WRAM in this given order:

1. The subgraph that the queries next in line will refer to,
2. Successful alignment results and uncomputed queries,
3. Queries (read sequences and their seed locations),
4. Wavefront queues for each thread,

The subgraph in WRAM is updated according to the queries that are processed at the time. It can include multiple segments if seeds point close to the boundaries of a segment, or it can be just a linear base sequence if seeds point to inside regions.

For the results section, we do not store any information on the queries that we decide are not aligned. We only store the alignment scores of the successful ones along with their order in MRAM, so that the host program can interpret this information later rather than using a larger space for the results. We also write the uncomputed queries that have failed their computation due to memory limitations to this section of the memory along with their order in MRAM. The motivation is to report to the user that there were some difficulties regarding the alignment of the reported query, and we do not know what score it would get. We report this so that the user does not confuse it with an unaligned query. This feature prevents false reports of alignment or misalignment, making the program more reliable.

2.2.1.1 Results & Queries Memory Layout

Alignment results are written within a synchronized section of the program, where minimizing overhead is critical. To avoid repeated checks on whether the result buffer has been exhausted, we organized the WRAM layout so that the results region is placed immediately before the queries region. This design removes the need for per-write depletion checks inside the critical section.

In principle, avoiding these checks could raise concerns about race conditions that could corrupt query data. However, in practice, this risk is negligible. Result structures are much smaller than query structures, and as queries are processed, the result pointer advances monotonically upward. By giving the query pointer a sufficient 'head start', which is the allocated result space length, the result pointer cannot realistically catch up, even under concurrency.

The consecutive placement of these two regions in WRAM is therefore a deliberate trade-off: it avoids unnecessary control logic and synchronization costs, while maintaining safe and predictable memory behavior.

2.2.2 MRAM Layout

An overview of data structures that can reside in MRAM is as follows:

1. Memory addresses and quantities of the following structures in MRAM,
2. Graph structure including segment information, segment sequences, and links,
3. Queries in sorted order,
4. Alignment results, sharing the same memory space with queries,

2.2.2.1 Memory Structure of Graph

Segments are stored in an array including their IDs, lengths, edge array pointers, and sequence pointers; followed by incoming edges and outgoing edges (each edge appears twice in total); and finally segment sequences. The adjacency list structure to store the edges increases the edge access performance since all edges will be needed each time, and it also uses less memory.

2.2.2.2 MRAM Queries & Results Memory Reuse

Queries are loaded into WRAM in batches, and their corresponding results are written back to MRAM only once the entire batch has been processed. This design enables the reuse of the same MRAM region for both queries and results. Unlike in the WRAM case, here, no race conditions arise. Data transfer between WRAM and MRAM is handled by a single thread, so the result pointer

is guaranteed to trail behind the query pointer. As a result, queries are safely overwritten only after they have been fully processed, eliminating redundancy while maximizing memory efficiency.

2.2.2.3 Padding for Memory Alignment

Due to the limitation of DMA transfers between WRAM and MRAM, which requires each transfer address and size to be a multiple of 8 bytes, we add padding in some of the structures to complete their size to a multiple of 8, so that they can be accessed without any overhead. This introduces a little fragmentation, but the trade-off is made considering the relatively less strict memory management requirements of MRAM compared to WRAM.

2.3 DPU Workflow

The workflow of our program is designed to make use of thread-based parallelism as much as possible while remaining within memory limitations. It starts by fetching the memory addresses and the quantities of segments and queries from MRAM into WRAM. Then, the execution alternates between single-threaded data fetching and multi-threaded alignment processing phases, until all queries in MRAM are done with. An overview of this workflow is given in Figure 2.3.

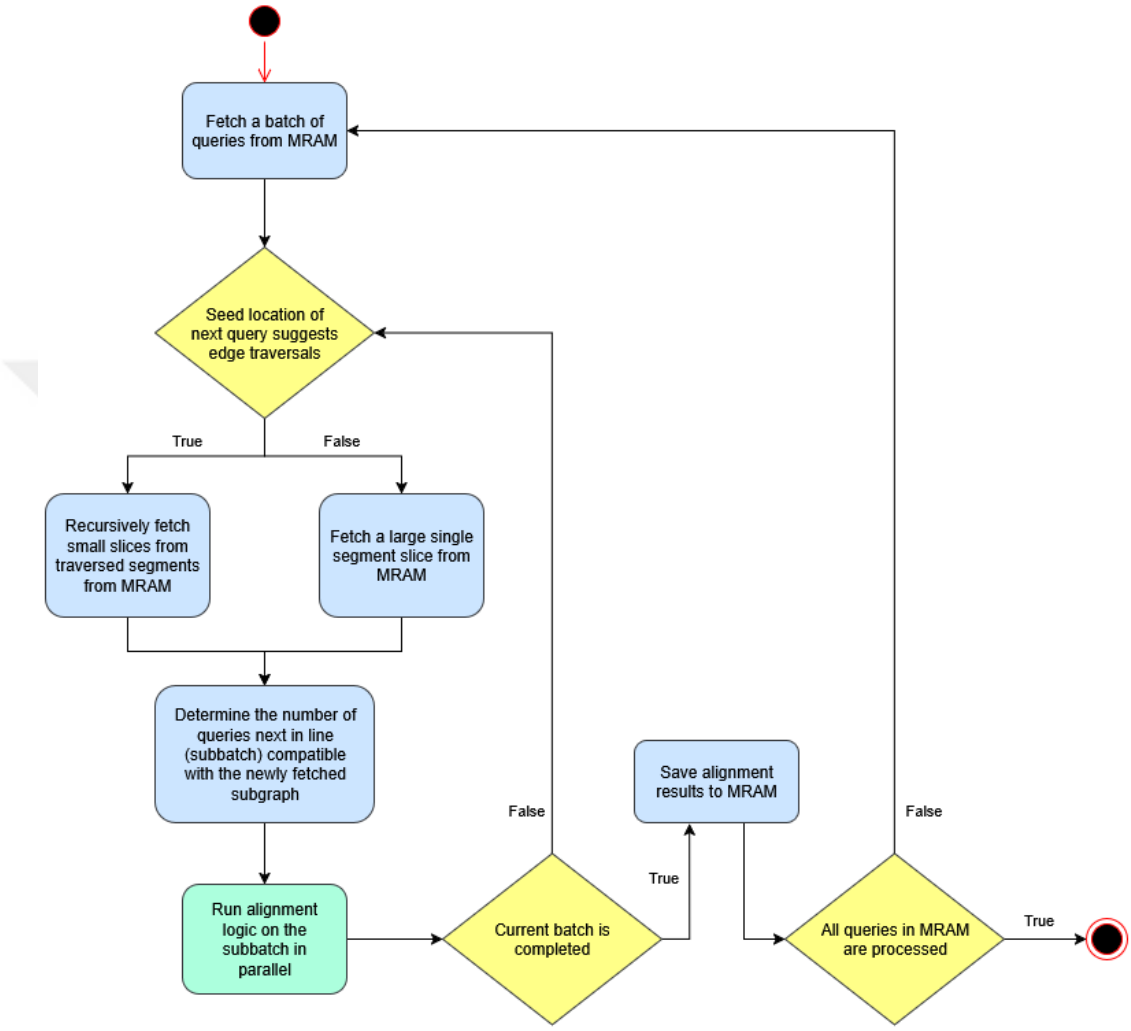


Figure 2.3: Main workflow inside a DPU that is followed after the initial parameter loading from MRAM. The turquoise node represents the multithreaded part; the other nodes, which mostly handle MRAM-WRAM transfers, are executed by a single thread.

2.3.1 Query Loading

A batch of queries is transferred to WRAM, and the corresponding portion of the graph is loaded as needed. Once both the queries and the relevant subgraph are resident in WRAM, threads execute the alignment stage in parallel, after which

the workflow advances to the next batch as long as there is more.

2.3.2 Subgraph Loading

After a batch is loaded, the subgraph is updated according to the needs of the next query in the line. There are three cases that we can face:

1. The seed points to the inner regions of a long segment: We load a subsequence of the segment covering the region around the pointed index. Since there is no graph complexity required in this alignment, we can use the full size of subgraph memory to store a subsequence that is as long as possible.

2. The seed points close to one boundary of a long segment: We load a sufficient prefix or suffix of the current segment's sequence and also recursively traverse the incoming or outgoing edges of segments up to a sufficient walk length and transfer their sequences along with metadata to WRAM.

3. The seed points to a short segment: We load the whole sequence of the current segment and also recursively traverse both incoming and outgoing edges of segments up to a sufficient walk length and transfer their sequences along with metadata to WRAM.

2.3.3 Batch Selection

After a new batch load or a subgraph change, the number of queries in line that are compatible with the current subgraph is determined using the bisection method, given that the seeds belonging to the same segment are in sorted order according to seed offset on the segment. In this way, we can ensure that all the data that will be needed for the parallel execution of a subbatch are already in WRAM, and there is no need for WRAM-MRAM transfers that would bring in additional synchronization overhead.

2.3.4 Parallel Query Processing

At this stage, a call for parallel execution of threads is made. Each thread picks a query from the subbatch in a synchronized manner and executes it before picking another one. The alignment results are stored in the corresponding section of the WRAM in a synchronized manner. Synchronization barriers are used to guarantee that all threads start and finish this stage simultaneously, so that subsequent steps can safely reuse shared memory regions.

2.3.5 Result Write-back

After all queries in the current batch are processed subbatch by subbatch, the alignment results are transferred to MRAM in the same space as queries without overwriting data that will be used in the future, as discussed previously.

2.4 GWFA on DPUs

2.4.1 Extension/Expansion Cycle

The GWFA algorithm is based on two main phases, between which it alternates.

1. **Extension:** Each cell attempts to advance along its diagonal, incrementing progress as long as bases match. If the bases on the query are depleted, it means that an alignment has been found. If the bases on the segment are depleted first, the extension continues on the neighboring segments.
2. **Expansion:** When a cell cannot be extended further, it spawns three new cells on its own diagonal and on neighboring diagonals, representing mismatches, insertions, and deletions. Each new cell maintains the correct segment slice reference, diagonal index, and query progress. Duplicate cells

corresponding to the same diagonal are undesirable and are eliminated whenever possible to reduce WRAM usage and avoid redundant computation.

The algorithm first starts with extension, and as long as there is no alignment yet and also the maximum score is not yet reached, it switches to the expansion step while increasing the alignment score by 1 each time. The algorithm ends without an alignment if none of the progress scores of the wavefront diagonals reach the length of the query when the score exceeds the maximum allowed edit distance. By switching between extension and expansion, the algorithm efficiently explores all feasible alignments of the query against the segment slice while maintaining a bounded memory footprint.

2.4.2 Wavefront Queues

Within WRAM, each thread maintains two queues of wavefront cell structures that represent progress on a diagonal. Each cell encodes three pieces of information: a pointer to the related segment slice, the diagonal index, and the progress along the query, similar to how the GWFA algorithm works.

In each step, which is an extension or an expansion, one of the queues is used as the input of that step, which we also will call the primary queue.

There are two types of extension: traversing and non-traversing. The former case occurs when the end of the segment sequence is reached, and alignment continues to the neighboring segment sequences. In this case, the resulting wavefronts, which point to the diagonals of different segments, are added to the secondary queue. On the other hand, wavefront cells resulting in non-traversing extensions are updated in their spot in the primary queue. After processing all original cells in the primary queue, elements in the secondary queue are appended at the end of the primary queue, which becomes the output of the extension step. This strategy of accumulating two types of wavefront in separate queues allows

us to secure the locality of wavefront cells in the queues, which is handy for the elimination of duplicate cells (Section 2.4.3) in the expansion step.

The expansion step generates up to three new cells per input cell and almost always increases the total number of cells. Unlike extension, expansion frequently depends on the state of neighboring diagonal cells that are also undergoing expansion. In other words, the formation of a new wavefront during expansion depends not only on the progress along its own diagonal, but also on the progress of adjacent diagonals. We present a method we call the Shift-Compete-Append method, described in Figure 2.4. To handle the dependency on neighbors, cells are processed in sequence along consecutive diagonal indices within the same segment. For each new cell, we compare candidate progress values from up to three sources. To enable this, we temporarily store the old progress values of the two most recently processed cells, provided that they belong to the same sequence and correspond to the immediate first and second neighboring diagonals. This short-term memory allows us to consistently select the most advanced progress among competing candidates, eliminating potential duplicates. Each resulting wavefront cell is then added to the secondary queue.

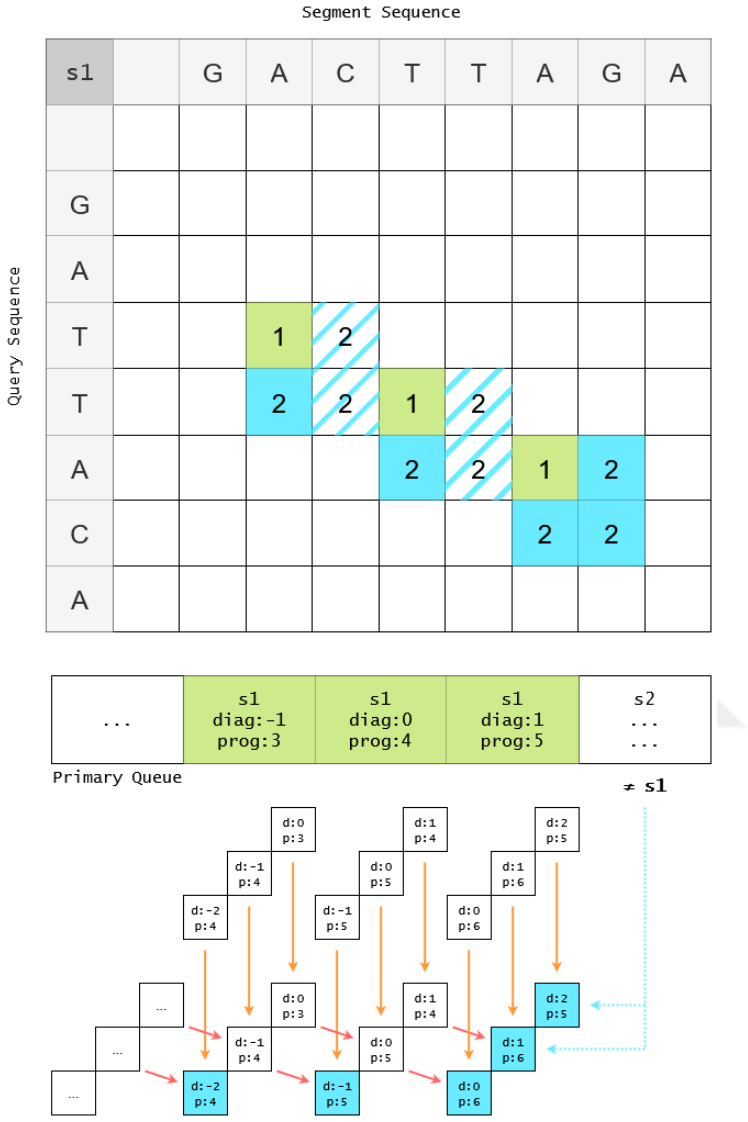


Figure 2.4: An example of the Shift-Compete-Append method used for expansion. Cells with consecutive diagonals (d) are expanded without having duplicate children. A state of three, resembling the three expansion routes (insertion, mismatch, deletion), is carried forward as queue elements are processed. Orange and pink arrows represent the flow of candidate values for the state; the candidate with the maximum progress score (p) takes over. The insertion state is appended to the secondary queue after each comparison. This goes on until the segments differ or the diagonal value of the next element in the queue is not +1 or +2 of the last processed diagonal, after which the mismatch and the deletion states are also appended to the secondary queue.

The extension step returns the primary queue as the output, while the expansion step returns the secondary queue as the output. Therefore, after each iteration of extension and expansion together, the queues are switched. This forces the queues to have symmetrical requirements. Making one of them larger and the other smaller according to need is not a viable option because both of their lengths should be sufficient for either role.

The expansion step does not track query or segment boundaries and unconditionally produces new cells. All bound checks are performed only in the extension routine.

The choice of queues over more complex structures, such as the hash table, which is used by GWFA, is dictated by the severe memory constraints of WRAM. After reserving space for segment slices, query batches, and thread stacks, a dynamic hash table would exceed the available memory. However, the queue approach also has a disadvantage: Multiple wavefront cells that represent the same diagonal within the same segment may accidentally be created, as the key uniqueness property of hash tables cannot be leveraged in simple queues.

2.4.3 Duplicate Elimination

During the wave expansion phase, multiple wavefront cells may converge on the same segment and on the same diagonal. To avoid unnecessary memory usage and redundant computation, the algorithm attempts to eliminate such duplicates by retaining only the cell with the greatest progress along the query. Duplicate elimination relies on the assumption that potential duplicates, or their parent cells, appear consecutively in the queue. When two consecutive cells correspond to the same segment slice and nearby diagonals, the algorithm can detect the potential redundancy and automatically discard the less advanced cell. However, if potential duplicates are separated in the queue, either by cells from other segments or by distant diagonals within the same slice, the detection mechanism cannot operate, and duplicate cells may persist. Although this imperfection means that duplicate elimination is not guaranteed in all cases, the impact is expected to

be minor. In single-segment alignments, where no edge traversals occur, there will be no duplicates to begin with. However, when traversals introduce new segments, their corresponding wavefront cells are inserted adjacently in the queue, increasing the likelihood that duplicates are still detected and removed.

2.4.4 Graph Traversal

Each wavefront cell operates on a particular segment or segment slice loaded into WRAM. Extension along the current diagonal proceeds until a mismatch is encountered or the segment boundary is reached. When the boundary of a segment is reached, the outgoing or incoming edges are traversed to neighboring segments, enabling the wavefront to continue across the graph. Each loading step before the parallel execution of queries ensures that all necessary segment slices reside in WRAM for the current batch, avoiding run-time stalls.

2.4.5 Pruning

The algorithm comes with the option of pruning. In pruning mode, after each extension step, the highest progress is calculated among all wavefront cells. Then, at the beginning of each expansion step, the progress of each wavefront cell is compared with the highest progress, and the cells that fall behind by over a given threshold are eliminated. This mode may result in suboptimal alignments depending on how large this threshold is. Though in a small memory system such as this, it might be necessary to store and compute only the promising wavefront cells, to make it lighter on the queue size limitation.

2.4.6 Seed Skipping as Fallback

Despite duplicate elimination and pruning strategies, in some exceptional cases, the number of wavefront cells can exceed the maximum capacity of a queue.

When the available thread memory is insufficient to store all wavefront cells, particularly in densely connected regions of the graph, certain seeds may be skipped. In such cases, SEGAPIM marks the corresponding reads in the output file to indicate that potentially better alignments might have been omitted due to memory constraints. As a result, the user may decide to run the program again with a higher memory space per thread.

2.5 User-Configurable Parameters

List of Parameters:

1. **Input-Dependent Parameters:** Determined directly by the characteristics of the input data set.
    - **Read Length**
    - **Seed Length**
2. **Logical Parameters:** Heuristic choices that balance accuracy, speed, and memory usage.
    - **Maximum Edit Distance (ED) Score**: Defines the maximum penalty a read may accumulate. The algorithm abandons the query once this threshold is exceeded. Smaller thresholds improve runtime, but risk discarding valid alignments with higher penalty scores.
    - **Pruning Threshold**: Specifies the maximum difference in progress allowed compared to the best performing wavefront diagonal. Paths falling behind by more than this threshold are pruned to save time and memory.
3. **WRAM Memory Management Parameters:** Control how WRAM is allocated for different structures.
    - **Subgraph Memory Size**: Amount of WRAM dedicated to storing subgraph slices fetched from MRAM.

- **Queries Memory Size**: Size reserved for queries. This parameter only influences performance (through batch size) and does not affect alignment accuracy.
- **Per Queue Memory Size per Thread**: Maximum number of wavefront cells that each thread can track simultaneously.
- **Stack Size of Thread 0**: Thread 0 handles recursive subgraph loading from MRAM to WRAM. Because this process involves deeper recursion, Thread 0 requires a larger stack than other threads.
- **Default Stack Size per Thread**: Stack space available to other threads during alignment. Needed primarily when traversing dense regions of the graph, but recursion depth is usually shallower than in subgraph loading.

In addition to core parameters such as the maximum edit distance, all WRAM allocations are explicitly defined by the user. The program exposes these memory sizes as configurable parameters, allowing the user to tailor memory usage to the characteristics of the dataset and desired trade-offs. Some examples are as follows:

- If a high maximum edit distance is required, the user should allocate more space to the wavefront queues. A larger queue enables the storage of more concurrent wavefronts, reducing the need for pruning and increasing the likelihood of processing each query successfully.
- If the target graph contains densely connected regions, recursive calls on the thread stack become frequent during both graph loading and extension. In such cases, specifying a larger per-thread stack size ensures that deep recursions can be handled without overflow.
- If many reads are expected to fail to align within the maximum edit distance, less memory can be reserved for the result section. This frees additional space for other structures, optimizing overall WRAM utilization.

Chapter 3

Experiments and Discussion

3.1 Experimental Setup

In our experiments, we evaluated SEGAPIM using only a single DPU out of the 2560 available in the system. The main reason we did not scale to multiple DPUs is the absence of a robust graph partitioning algorithm tailored for this purpose. However, we argue that single-DPU experiments remain representative because the algorithm itself is inherently scalable. In a full system deployment, each DPU would process a distinct portion of the input without requiring inter-DPU communication, which matches the parallelization model of the UPMEM PIM architecture, where the host CPU is only involved in data loading, data fetching, and a post-process that is agnostic to the number of DPUs used. The data transfer steps that occur between DPU binary executions are taken into account when the performance is compared to that of the state-of-the-art tools.

3.1.1 Datasets

We conducted experiments on the MHC-57 graph [23], a pangenome graph representation of the human Major Histocompatibility Complex region constructed

from 57 haplotype sequences. The MHC region on chromosome 6 contains the most polymorphic gene cluster of the entire human genome [24]. This property fits the purpose and methodology of our tests, since we wish to acquire a good representation of alignment on a variation-rich complete genome graph by using only the most computationally challenging fraction of it.

We used *vg sim* [11] to simulate two read sets on this graph, with error rates 0.5% and 2%. The 0.5% error set reflects Illumina-like short reads with typical sequencing error rates [25], while the 2% error set combines sequencing errors with single nucleotide polymorphisms (SNPs), providing a more challenging scenario for testing the limits of the algorithm.

3.1.2 Baselines

We ran all baseline experiments on a compute node equipped with four Intel Xeon E7-4830 processors (2.13 GHz) providing 64 CPU cores and 500 GB of main memory, running Ubuntu Linux (kernel version 5.15.0).

We used GraphAligner and *vg map*, which run solely on the CPU, to compare their alignment accuracies and execution times to those of SEGAPIM runs with various parameter values on various datasets. For GraphAligner, we used the same minimizer seeding parameters as SEGAPIM. For *vg map*, we used parameters that would match the edit distance scoring metric.

We also tried the same input set on AStarix [26] and the original GWFA, which is maintained as a part of `gfatools` [13] under the `ed` command. Unfortunately, the AStarix runs resulted in segmentation faults. On the other hand, `gfatools ed` is only capable of alignments that start from the first character of a given segment since it is only a proof-of-concept implementation. Therefore, we could not include these tools in our baselines.

3.1.3 Comparison to Baselines

We used scripts to compare the accuracy of the baseline program and SEGAPIM. For each pair of compared runs, we report alignment rates, the distribution of alignment scores found, the fraction of the reads that have a matching score between two sides, the fraction of the reads that performed better in one than the other, and the average score difference in the fraction of different scores.

We do not know of a more reliable alignment tool like edlib, but one that works for graphs. We also do not accept GraphAligner or *vg map* as ground truths.

In addition, we measured the execution times of the CPU baselines with varied thread counts and compared them to the single-DPU SEGAPIM execution times and also estimated larger-scale execution times.

3.1.4 Scaling Execution Times to Full DPU System

When projecting to 2560 DPUs, we use different transformations on different parts of the SEGAPIM execution. These parts are the data transfer to DPUs, the binary execution on DPUs, and the data transfer from DPUs, on which we have separate time measurements.

The projection of the binary execution time is simple. Assuming an equal division of data among DPUs, we can use the linear performance scale property and divide the single DPU execution time by 2560 to predict the theoretical execution time on the full system.

Data transfer times can be projected by adjusting data transfer times according to the bandwidth improvements introduced with parallel data transfers from and to DPUs in Section 1.3.3. In this way, we can provide a fairer estimate of the throughput achievable on a large scale.

Parallel data transfers to or from DPUs can be performed only within the same

rank. A rank includes 64 DPUs, with 40 ranks in the system. Therefore, for a large dataset that would be divided and transferred from the host CPU to the DPUs, we can achieve a speed-up factor of 20.13×, but not more, since transfers to different ranks cannot be parallelized. Similarly, we can achieve a speedup of 38.76× for data transfers from the DPUs to the host CPU.

When estimating the performance of a single rank or a full 40 rank DPU system, since both our program and the baseline aligners process the same amount of data, data transfer time projection can be done by dividing the transfer times of the single DPU version by the throughput improvement factor of the parallel data transfer, divide by 38.76 for graph and query loading, and divide by 20.13 for result collecting.

We also have to count the execution time of GraphAligner seeding, since it is a crucial part of our pipeline, along with the host-side preprocess and postprocess.

Therefore, the final formula for the projected execution time is:

$$T_{\text{projected}} = T_{\text{GA-seeding}} + T_{\text{host-pre}} + \frac{T_{\text{CPU}\to\text{DPU}}}{20.13} + \frac{T_{\text{DPU-exec}}}{N_{\text{DPU}}} + \frac{T_{\text{DPU}\to\text{CPU}}}{38.76} + T_{\text{host-post}}$$

for a single rank system ($N_{\text{DPU}} = 64$) or a 40 rank system ($N_{\text{DPU}} = 2560$).

3.2 Experiment Results and Interpretations

3.2.1 Alignment Accuracy Comparisons Among GraphAligner, *vg map*, and SEGAPIM

We compared the alignment scores produced by GraphAligner, *vg map*, and SEGAPIM on our two datasets. We picked the SEGAPIM parameters that give the best balance of accuracy and speed for each read set.

The detailed score distribution is presented in Table 3.2a, followed by the summary statistics are presented in Table 3.2b.

Table 3.1: Score distribution and pairwise comparison of SEGAPIM (max ED 5, prune 5) versus GraphAligner and *vg map* on 1M reads. Scores above 9 are merged into the 10+ bucket. Percentages are over total reads. Dataset: average 0.5% error rate over the MHC57 genome graph.

(a) Score distribution among GraphAligner, *vg map*, and SEGAPIM

ED Score	GraphAligner %	*vg map* %	SEGAPIM %
0	55.0154	55.3088	55.2952
1	26.4318	30.7531	31.1399
2	11.4794	10.7021	10.5503
3	4.1127	2.6399	2.4822
4	1.8104	0.4975	0.4369
5	0.5643	0.0834	0.0632
6	0.1625	0.0131	0.0000
7	0.0540	0.0018	0.0000
8	0.0262	0.0002	0.0000
9	0.0227	0.0000	0.0000
10+	0.3166	0.0001	0.0000
Total	99.9960	100.0000	99.9677

(b) Pairwise comparison summaries (SEGAPIM vs others)

Metric	vs GraphAligner	vs *vg map*
Reads compared	999,677	999,677
Equal scores	897,047 (89.734%)	992,057 (99.238%)
SEGAPIM has better alignment	102,629 (10.266%)	7,533 (0.754%)
Other has better alignment	1 (0.000%)	87 (0.009%)
Average difference in unequal scores (SEGAPIM – Other)	-1.90	-0.99

3.2.1.1 Interpretation of Accuracy Comparisons

Although SEGAPIM could not find any alignment for a small portion of the reads due to a much higher number of errors than average, it outperformed the baselines on the alignments it could report. In general, SEGAPIM achieved significantly superior accuracy compared to baseline CPU programs, especially GraphAligner, finding better alignment scores for up to 10% of reads.

Table 3.2: Score distribution and pairwise comparison of SEGAPIM (max ED 7, prune 7) versus GraphAligner and *vg map* on 1M reads. Scores above 9 are merged into the 10+ bucket. Percentages are over total reads. Dataset: average 2% error rate over the MHC57 genome graph.

(a) Score distribution among GraphAligner, *vg map*, and SEGAPIM

ED Score	GraphAligner %	*vg map* %	SEGAPIM %
0	4.8947	4.9789	4.9765
1	14.5314	15.0203	15.1614
2	21.7643	22.6505	22.9379
3	21.8628	22.5508	22.7773
4	16.6440	16.7464	16.7563
5	10.3223	9.9182	9.7578
6	5.4442	4.9168	4.7160
7	2.5620	2.0849	1.9138
8	1.0922	0.7557	0.0000
9	0.4609	0.2670	0.0000
10+	0.4184	0.1105	0.0000
Total	99.9972	100.0000	98.9970

(b) Pairwise comparison summaries (SEGAPIM vs others)

Metric	vs GraphAligner	vs *vg map*
Reads compared	989,970	989,970
Equal scores	899,957 (90.908%)	960,760 (97.049%)
SEGAPIM has better alignment	89,948 (9.086%)	29,060 (2.934%)
Other has better alignment	65 (0.007%)	150 (0.015%)
Average difference in unequal scores (SEGAPIM – Other)	-1.65	-1.07

The reads are included in the average score difference if both sides found an alignment for them. Therefore, in both of these experiments, among the aligned reads, SEGAPIM found better edit distance scores than GraphAligner and *vg map*, respectively, on average. This could stem from the heuristics these aligners apply in their algorithms, but it could also be caused by bugs.

We observe that a small fraction of the reads were aligned with a higher score by *vg map*. Upon closer inspection, these cases consistently correspond to alignments that span multiple segments, i.e., alignments crossing graph links. A likely explanation is the seeding strategy of GraphAligner that we used in our pipeline, which generates seeds exclusively from exact matches that are fully contained within a single segment. Consequently, alignments requiring seeds that cross segment boundaries may be missed.

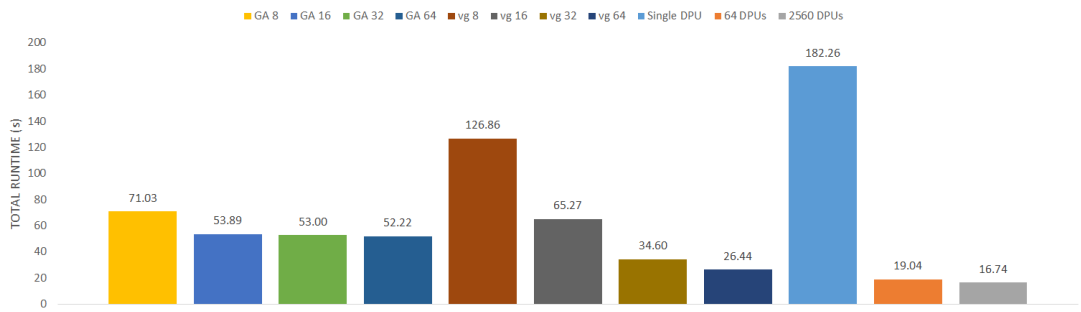
Another potential contributing factor could be the seed-skipping strategy employed by SEGAPIM. However, in the subset of reads for which SEGAPIM could not find the true optimal alignment and which we manually inspected, none of the lower-scoring SEGAPIM alignments was marked as skipped.

Although we cannot fully exclude seed skipping as a factor, these observations suggest that the most likely explanation for these missed multi-segment alignments is the inherent limitation of GraphAligner's seeding mechanism, rather than SEGAPIM's memory-related seed skipping.

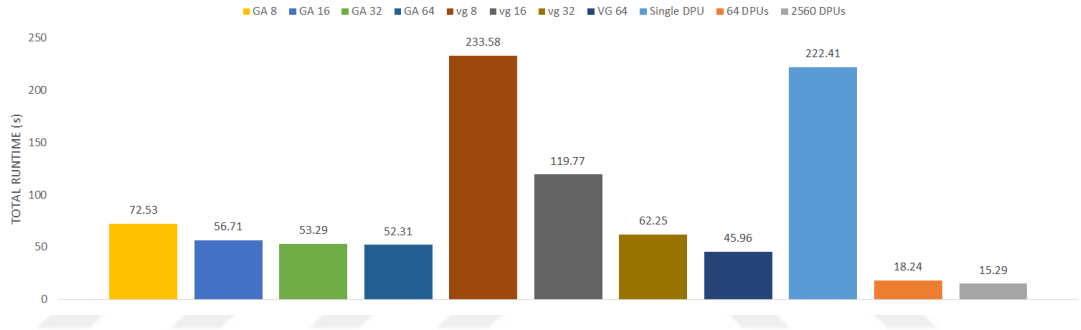
3.2.2 Execution Time Comparisons Among GraphAligner, *vg map*, and SEGAPIM

We measured the execution times of GraphAligner and *vg map* with various numbers of threads. We also calculated the projected larger-scale execution times for the same SEGAPIM runs that we used for the accuracy comparison.

GraphAligner execution time does not get a huge improvement when more threads are included, which is expected considering its observed strong memory



(a) 0.5% Error Reads, Maximum ED 5, Prune Threshold 5.



(b) 2% Error Reads, Maximum ED 7, Prune Threshold 7.

Figure 3.1: GraphAligner and *vg map* total execution times on varied number of threads, along with SEGAPIM total execution time and its projections to larger scales.

boundness. On the other hand, the performance of *vg map* scales almost linearly with the increased number of threads, except that its memory boundness starts to show itself between 32-64 threads. Both of these observations align with the benchmarks presented in Section 1.2.1.

A full 2560-DPU UPMEM system is projected to be extremely powerful and cost-effective. Although a single DPU already achieves performance comparable to *vg map* running on 8 CPU threads, the performance of projected scalings far exceeds the baseline CPU programs. It is difficult to envision *vg map* closing this gap without a fundamental reduction in the memory boundness of today's CPUs, even when using very large thread counts.

The degree of parallelization offered by a full UPMEM system is exceptionally high. In practice, execution is bottlenecked not by the DPUs themselves but by

relatively trivial host-side tasks such as file parsing and seed grouping. The benefits of scaling from 64 to 2560 DPUs will become more evident on substantially larger datasets.

Finally, the observed reductions in execution time relative to CPU baselines also imply improved energy efficiency, as discussed in Section 1.3.5.

3.2.3 Execution Time Comparisons Among SEGAPIM Runs

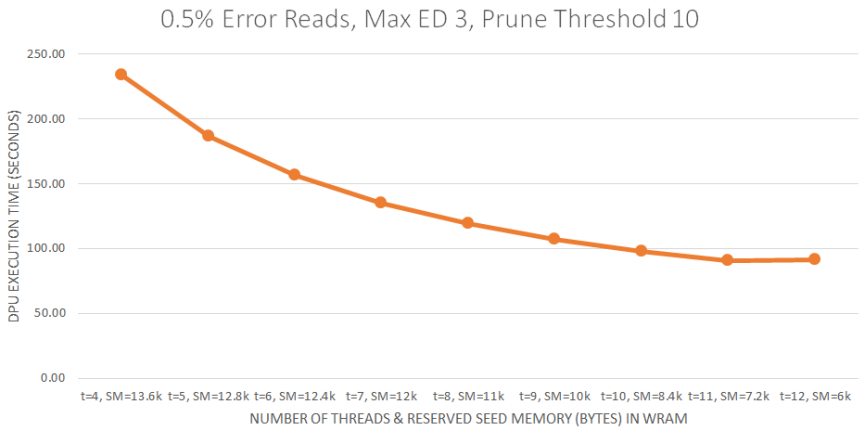
3.2.3.1 Number of Threads

For both read sets, we performed experiments with different thread counts (Figure 3.2). With an increase in the number of threads in which other critical parameters remain unchanged, we had to decrease the size of the memory in WRAM that is reserved for queries and their results.

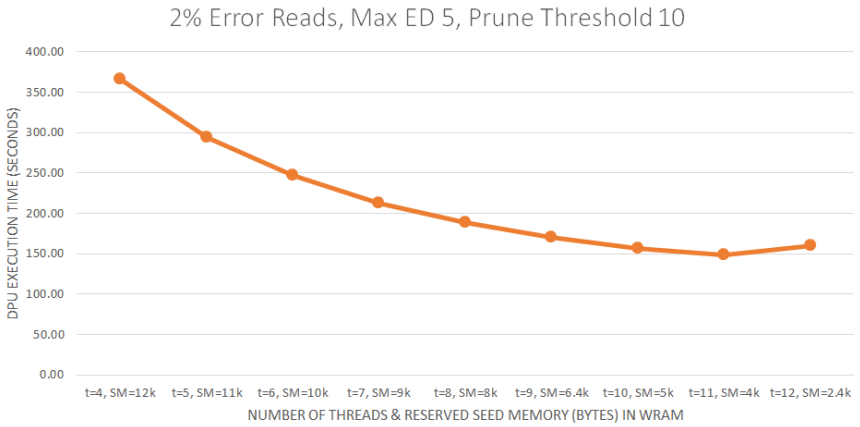
We see that using 11 threads gives the least DPU execution time, which is the general behavior of the UPMEM system, as also stated in Section 1.3.4. With 11 threads, the pipeline is already utilized to the maximum. Compared to that, using 12 threads causes more memory usage by threads in total, and therefore less memory is reserved for queries and their results. Hence, the more granular the transfers between WRAM-MRAM are, the more cycles it takes. In addition, the subbatch detection mechanism works in logarithmic time complexity, which is sublinear, and, therefore, the granularity also decreases its overall performance.

3.2.3.2 Maximum Edit Distance Score

We compare SEGAPIM runs with different maximum edit distance values (Figure 3.3). The computational cost of each query depends on the minimum of the actual edit distance of the query and the threshold we determine. We chose the



(a) 0.5% Error Reads, Maximum ED 3, Prune Threshold 10.



(b) 2% Error Reads, Maximum ED 5, Prune Threshold 10.

Figure 3.2: DPU execution times with varied number of threads.

read set with the highest error rate because we did not want the computational cost to be capped by low actual error rates.

We observe that the execution time increases superlinearly according to the maximum edit distance, even though the actual error rates should have pulled it towards a sublinear rate. This shows how increasingly heavy the workload can become when the maximum edit distance threshold is increased, theoretically backed by the fact that in every expansion step, a cell could singlehandedly produce three new cells after a traversal through graph edges.

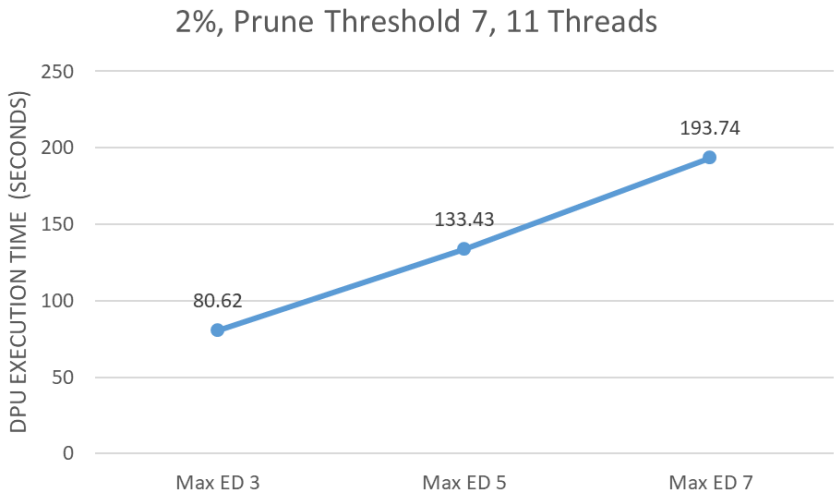


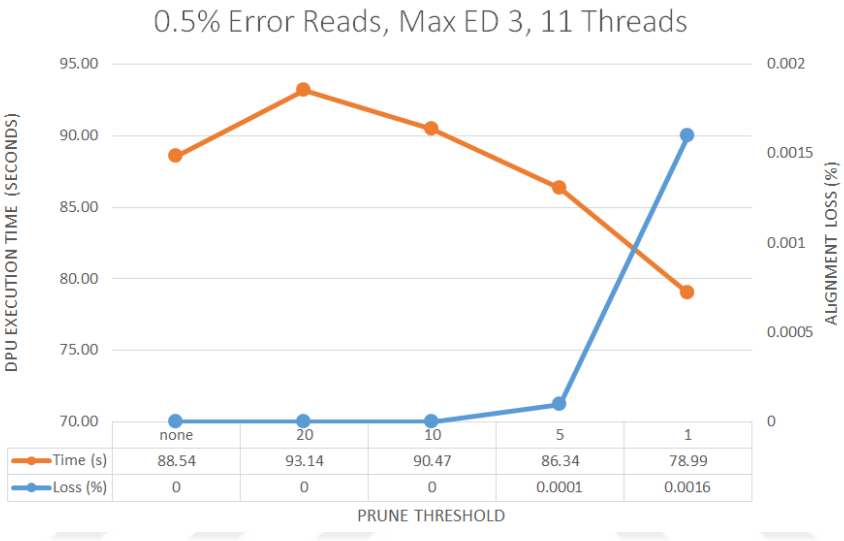
Figure 3.3: DPU execution times with varied maximum edit distance threshold.

3.2.3.3 Prune Threshold

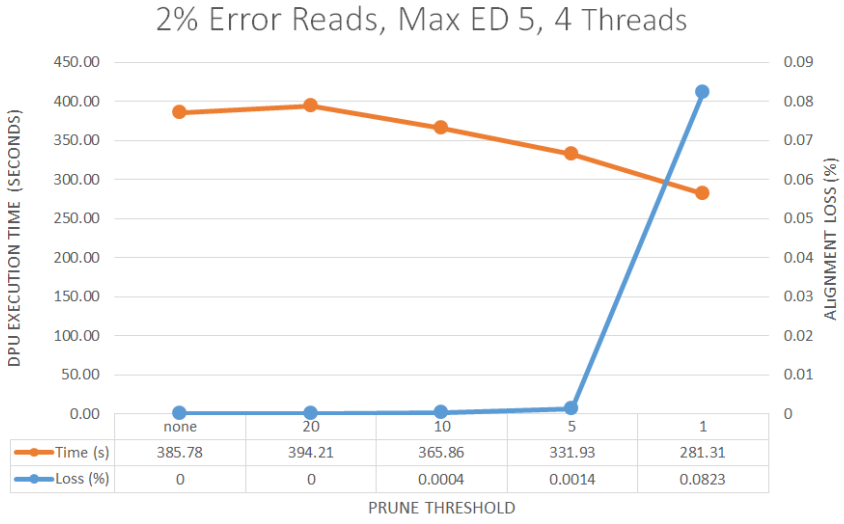
We measured execution times and alignment loss rates with different prune thresholds (Figure 3.4). "None" represents the case where no pruning logic was applied (i.e., the prune threshold is infinite). The alignment loss denominator is the number of successful alignments that would be possible if no pruning were applied. We used only four threads for the second set of experiments because applying a weak pruning does require a larger wavefront cell queue size, and therefore the number of threads had to be low.

Pruning the wavefront cells that fall behind by 20 or more appears to be ineffective. It does not eliminate many cells and causes a worse performance than when no pruning was applied. This is because the program now computes the progress of the most advanced cell and also compares the progress of every cell to the most advanced one in each step, without enough payback in performance.

Using a prune threshold of 1 causes some alignments to be lost and also possibly causes some alignments to have worse final edit distance scores. This threshold should be chosen by the user carefully according to the performance and accuracy requirements.



(a) 0.5% Error Reads, Maximum ED 3, 11 Threads.



(b) 2% Error Reads, Maximum ED 5, 4 Threads.

Figure 3.4: DPU execution times with varied number of prune thresholds.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

The purpose of this work was to evaluate the suitability of PIM architectures for the acceleration of SGA algorithms.

To achieve this, we chose to adapt the GWFA algorithm because of its relatively low memory requirements which UPMEM DPU, the PIM system at hand, forces on us.

We designed and implemented SEGAPIM, compared it to memory-bound CPU baselines, from which our motivation stemmed, and also evaluated its accuracy and performance trade-offs in itself.

As a result, we have found that the UPMEM PIM system is quite suitable for such acceleration, also suggesting that PIM is suitable for this type of workload in general. In fact, we believe that the drawbacks of PIM will be alleviated further in the near future, considering that it is a relatively young area of research and will be better recognized, especially in the context of data-intensive applications.

Furthermore, the proposed design and implementation of SGA are not limited to the UPMEM platform. The memory-aware strategies developed in this work, such as compact data representations, explicit buffer reuse, and careful stack management, are broadly applicable to other low-memory environments, including embedded accelerators, edge devices, and specialized high-throughput sequencing platforms. By demonstrating that complex alignment tasks can be adapted to operate effectively under strict memory constraints, our work highlights a general approach that extends beyond PIM systems.

4.2 Future Work

Several directions remain open for improving and extending SEGAPIM:

- **Graph partitioning and multi-DPU scaling:** The current implementation runs on a single DPU. Future work should focus on robust graph partitioning and seed directing strategies that enable a fully parallel multi-DPU version.
- **Linear model for single-segment seeds:** A simplified linear alignment mode for seeds that cannot possibly extend through the graph edges could reduce branching overhead and greatly improve performance.
- **Automatic memory parameter tuning:** Many memory management variables must currently be set by the user. Automating a selection of default values based on input parameters would improve usability.
- **Integrated seeding pipeline:** At present, seeds are written to and read from intermediate files. Merging the seeding stage directly into the program would reduce I/O overhead.
- **Host-side parallelism:** The seed organization stage on the host currently runs serially. Parallelizing this step could remove a bottleneck and better complement the high parallelism of DPUs.

- **Accuracy–performance trade-offs:** Implementing modes that can adaptively balance speed and accuracy, such as not allowing alignments at the graph boundaries, could reduce overhead for a great majority of queries, although it would sacrifice a small minority.

Bibliography

[1] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. G. Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, and et al., "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[2] F. S. Collins, M. Morgan, and A. Patrinos, "The human genome project: lessons from large-scale biology," *Science*, vol. 300, no. 5617, pp. 286–290, 2003.

[3] N. H. G. R. Institute, "Dna sequencing costs: Data." NHGRI Genome Sequencing Program, 2022. Cost per genome falls to approximately $600 for broad throughput centers.

[4] M. Quail *et al.*, "A tale of three next generation sequencing platforms," *BMC Genomics*, vol. 13, no. 341, 2012.

[5] H. Zhang *et al.*, "A comprehensive evaluation of long read error correction," *BMC Genomics*, vol. 21, no. 1, p. 709, 2020.

[6] 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.

[7] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[8] T. . G. P. Consortium, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, pp. 56–65, Oct. 2012.

[9] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, p. 707, February 1966.

[10] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, pp. 3363–3369, 07 2004.

[11] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin, B. Paten, and R. Durbin, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature Biotechnology*, vol. 36, pp. 875–879, Sept. 2018.

[12] National Human Genome Research Institute, "Pangenome — Talking Glossary of Genomic and Genetic Terms." `https://www.genome.gov/genetics-glossary/Pangenome`, 2025. Accessed: 2025-09-17.

[13] H. Li, X. Feng, and C. Chu, "The design and construction of reference pangenome graphs with minigraph," *Genome Biology*, vol. 21, no. 1, p. 265, 2020.

[14] H. Zhang, S. Wu, S. Aluru, and H. Li, "Fast sequence to graph alignment using the graph wavefront algorithm," 2022.

[15] A. Kleen, "pmu-tools." `https://github.com/andikleen/pmu-tools`, 2023. Performance monitoring tools for Intel processors.

[16] Intel Corporation, *Intel VTune Profiler*, 2023. Version 2023.2.

[17] O. Mutlu, S. Ghose, J. Gómez-Luna, R. Ausavarungnirun, M. Sadrosadati, and G. F. Oliveira, "A modern primer on processing in memory," 2025.

[18] 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," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), p. 316–331, Association for Computing Machinery, 2018.

[19] UPMEM, *UPMEM Processing-in-Memory (PIM) Architecture*, 2023. Available at `https://www.upmem.com/`.

[20] 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," 2022.

[21] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi, G. Singh, J. Gómez-Luna, N. A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "Segram: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, p. 638–655, ACM, June 2022.

[22] S. Diab, A. Nassereldine, M. Alser, J. Gómez-Luna, O. Mutlu, and I. E. Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," 2023.

[23] H. Li, "Sample graphs and sequences for testing sequence-to-graph alignment," 2022. Data set.

[24] P. Cruz-Tapias, J. Castiblanco, and J.-M. Anaya, "Major histocompatibility complex: Antigen processing and presentation," in *Autoimmunity: From Bench to Bedside* (J.-M. Anaya, Y. Shoenfeld, A. Rojas-Villarraga, *et al.*, eds.), ch. 10, Bogota, Colombia: El Rosario University Press, 2013. Available online.

[25] N. Stoler and A. Nekrutenko, "Sequencing error profiles of illumina sequencing instruments," *NAR Genomics and Bioinformatics*, vol. 3, no. 1, p. lqab019, 2021.

[26] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, and M. Vechev, "Astarix: Fast and optimal sequence-to-graph alignment," in *Research in Computational Molecular Biology (RECOMB 2020)* (R. Schwartz, ed.), vol. 12074 of *Lecture Notes in Computer Science*, pp. 104–119, Springer, Cham, 2020.

Appendix A

Code

The source code will be available on `https://github.com/BilkentCompGen/` under the name SEGAPIM.