Report On

"Domain-Specific Many-Core Processing for Genomic Analysis"

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology
in
Electronics & Communication Engineering

UE21EC461A – Capstone Project

Submitted by:

Abhiram Gopal Dasika [PES2UG21EC003]

Jitesh Kumar Nayak [PES2UG21EC057]

Neha C Waghmore [PES2UG21EC092]

Under the guidance of

Dr. Madhura P

Professor

PES University

January - December 2024

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

"Domain-Specific Many-Core Processing for Genomic Analysis"

is a bonafide work carried out by
Abhiram Gopal Dasika [PES2UG21EC003]
Jitesh Kumar Nayak [PES2UG21EC057]
Neha C Waghmore [PES2UG21EC092]

In partial fulfilment for the completion of seventh semester Capstone Project (UE21EC461A) in the Program of Study -Bachelor of Technology in Electronics and Communication Engineering under rules and regulations of PES University, Bengaluru during the period January - December 2024. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 7th semester academic requirements in respect of project work.

Signature	*Signature*	*Signature*
Dr. Madhura P	Dr. Ajey S N R	Prof.Nagarjuna Sadineni
Professor	**Chairperson**	**Pro Vice Chancellor**

External Viva

Name of the Examiners	Signature with Date
1. _____	_____
2. _____	_____

DECLARATION

We, Jitesh Kumar Nayak, Abhiram Gopal Dasika, Neha C Waghmore, hereby declare that the report entitled, "Domain-Specific Many-Core Processing for Genomic Analysis", is an original work done by us under the guidance of **Dr. Madhura P**, Professor, ECE Department and is being submitted in partial fulfilment of the requirements for completion of 7th Semester course work in the Program of Study, B.Tech in Electronics and Communication Engineering.

PLACE: Bengaluru, KA
DATE:

NAME AND SIGNATURE OF THE CANDIDATES

1. Jitesh Kumar Nayak

2. Neha C Waghmore

3. Abhiram Gopala Dasika

ABSTRACT

Genomic analysis is a computationally intensive process involving multiple stages such as basecalling, read mapping, and sequence alignment. Traditional CPU-based implementations of these tasks suffer from inefficiencies due to memory bottlenecks and high computational complexity. This project explores domain-specific many-core architectures, particularly MemPool, to accelerate genomic processing. By leveraging parallel computing and optimized memory access, we aim to reduce execution time for read mapping and alignment.

ACKNOWLEDGEMENT

We are deeply grateful to **Dr. M. R. Doreswamy**, Chancellor, PES University, **Prof. Jawahar D**, Pro Chancellor – PES University, **Dr. J Surya Prasad**, Vice Chancellor, PES University for providing us with various opportunities and for the consistent encouragement.

We would like to express our immense gratitude to our guide, **Dr. Madhura** P, Department of Electronics and Communication Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE21EC461A - Capstone Project.

We are thankful to the project coordinators, **Dr. Madhura P** and **Prof. Mahesh Awati**, **Prof. Vinay Reddy** for organizing, managing, and helping us with the entire process.

We would also like to extend our gratitude to **Dr. Ajey S N R**, Chairperson, Department of Electronics and Communication Engineering, PES University Electronic City Campus, for all the knowledge and support we have received from the department. We would like to thank **Dr. B.K. Keshavan**, Dean of Faculty, PES University for providing this opportunity and necessary guidance throughout the completion of the project.

Finally, this project could not have been completed without the continual support and encouragement we have received from our family and friends. We are grateful to them as well.

CONTENTS

DECLARATION **3**
ABSTRACT **4**
ACKNOWLEDGEMENT **5**
CONTENTS **6**
CHAPTER 1 - PREAMBLE **1**
1.1 Introduction 2
1.2 Problem Statement 2
CHAPTER 2 - LITERATURE REVIEW **3**
2.1 Accelerating Genome Analysis: A Primer on an Ongoing Journey [1] 4
2.2 A Review of Parallel Implementations for the Smith–Waterman Algorithm [2] 5
2.3 Fast Sequence Alignment Method Using CUDA-enabled GPU [3] 8
2.4 Mempool - A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory[4] 9
CHAPTER 3- BACKGROUND AND PREREQUISITES **11**
3.1 MemPool Platform [4] 12
3.1.1. Architecture Diagrams 12
3.1.2 Tile Interconnect and DMA Engine 12
3.1.3 Local Crossbar Interconnect 13
3.1.4 DMA Engine and Memory Access 13
3.1.5 AXI Port to Interconnect 13
3.1.6 Hybrid Memory Addressing 14
3.2 Genomics and the DNA 15
3.3 Genome Sequencing Pipeline [1] 16
3.4 Read Mapping: The Computational Bottleneck [1] 18
3.5 Brief Overview of Genome Sequencing Algorithms 19
3.5.1 Exact Matching Algorithms 19
3.5.2 Heuristic Algorithms for Genome Alignment 20
3.5.3 Dynamic Programming-Based Algorithms 21
CHAPTER 4- ALGORITHM **23**
4.1 Smith-Waterman Algorithm 24

4.1.1 Steps of the Smith-Waterman Algorithm: 25
4.1.2 Example matrix calculation 27
4.2 Modified Smith-Waterman Algorithm[3] 28
4.2.1 Redefining the Recursive Formula for Parallel Execution 28
4.2.2 Prefix Max Scan Optimization 28
CHAPTER 5 - IMPLEMENTATION 30
5.1 Anti-diagonal Approach: 31
5.2 Row Parallel Method 33
5.2.1 Row-Parallel Speedup Against Row Parallel For Single Core 33
5.2.2 Parallel Methods Speedup Against Serial Implementation 33
5.2.3 Superior Performance of Row-Parallel Over Anti-Diagonal for 8 Cores 34
CHAPTER 6 - RESULTS AND DISCUSSION 36
6.1 Roofline Model 37
CHAPTER 7 - CONCLUSION AND FUTURE WORK 39
REFERENCES 42

CHAPTER 1
PREAMBLE

1.1 Introduction

Genome sequencing has become an indispensable tool in modern biology and medicine, enabling breakthroughs in fields ranging from personalized medicine to evolutionary studies. A fundamental and computationally intensive stage within this process is read mapping, where short DNA sequences (reads) are aligned to a reference genome. The core of this process, sequence alignment, is often the most time-consuming step. Among the various algorithms employed for sequence alignment, the Smith-Waterman algorithm remains a cornerstone due to its ability to guarantee optimal local alignments. This algorithm, however, relies on the construction of a dynamic programming (DP) matrix, which presents a significant computational bottleneck, particularly as genome sizes and sequencing depths increase. To address this challenge, our project explores the potential of parallel computing to accelerate the Smith-Waterman algorithm. Specifically, we are investigating the implementation of a parallelized Smith-Waterman algorithm on the MemPool architecture, a scalable many-core platform known for its low-latency shared memory, to achieve substantial performance improvements.

1.2 Problem Statement

The computational intensity of sequence alignment within read mapping significantly hinders genome sequencing throughput. The sequential nature of the Smith-Waterman algorithm, while providing accurate local alignments, is a primary bottleneck. This project aims to address this challenge by developing a parallel implementation of Smith-Waterman on the MemPool architecture

CHAPTER 2

LITERATURE REVIEW

2.1 Accelerating Genome Analysis: A Primer on an Ongoing Journey [1]

Year of Publication: 2020
Authors:

- Mohammed Alser (ETH Zürich)
- Zülal Bingöl (Bilkent University)
- Damla Senol Cali (Carnegie Mellon University)
- Jeremie Kim (ETH Zürich, Carnegie Mellon University)
- Saugata Ghose (University of Illinois at Urbana–Champaign, Carnegie Mellon University)
- Can Alkan (Bilkent University)
- Onur Mutlu (ETH Zürich, Carnegie Mellon University, Bilkent University)

Abstract: Read mapping, a key step in genome analysis, remains a computational bottleneck despite advances in sequencing technology. This paper surveys algorithmic and hardware-based acceleration techniques, including *processing-in-memory (PIM), SIMD optimizations, FPGA/ASIC accelerators, and heuristic-based methods*. It highlights the challenges of reducing data movement and integrating hardware acceleration into genome analysis pipelines.

Introduction: Genome analysis relies on *read mapping*, where DNA fragments are aligned to a reference genome. The process is slow due to *dynamic programming (DP)-based sequence alignment*, consuming *70%+ of execution time*. The *rapid growth of sequencing data* and *high computational costs* pose significant challenges, particularly in clinical applications like disease outbreak tracking (e.g., COVID-19) and personalized medicine.

Key Contributions:

1. **Read Mapping Process & Challenges**
    - **Steps:** (1) Indexing (locating potential matches) ->
      (2) Pre-alignment filtering (eliminating mismatches) ->
      (3) Sequence alignment (precise alignment using DP).
    - **Challenges:** High computational costs, excessive memory access, and data movement overheads.

2. **Acceleration Techniques**
    - **Indexing Optimization:**
        - FM-index, minimizers reduce storage overhead.
        - Processing-in-memory (PIM) (e.g., RADAR) minimizes data transfer.
    - **Pre-Alignment Filtering:**
        - Pigeonhole Principle (*Shouji*), Base Counting, q-gram filtering (*GRIM-Filter*), Sparse DP (*rHAT*).
    - **Sequence Alignment Acceleration**:
        - **Accurate DP-based**: SIMD (*Parasail*, *KSW2*), GPUs (*GASAL2*, *BWA-MEM2*), FPGAs/ASICs (*GenAx*, *RAPID*), PIM-based (*RAPID*).
        - **Heuristic-based**: X-Drop (*KSW2*, *LOGAN*), Four-Russians Method (*Darwin*), Levenshtein Distance (*ASAP*, *Edlib*).
3. **Future Challenges**
    - **End-to-end acceleration:** Holistic approaches like *Illumina DRAGEN* (FPGA-based) and *NVIDIA Parabricks* (GPU-based) show *48× speedup*.
    - **Reducing data movement:** PIM-based methods (*GenASM*) improve efficiency.
    - **Flexible hardware architectures:** Future designs must adapt to longer, noisier reads.
    - **Optimizing genomic data formats:** Transitioning from FASTQ/FASTA (8-bit per base) to compact 2-bit formats could improve efficiency.

2.2 A Review of Parallel Implementations for the Smith–Waterman Algorithm [2]

Year of Publication: 2022

Authors:

- Zeyu Xia (School of Computer, National University of Defense Technology, China)
- Yingbo Cui (School of Computer, National University of Defense Technology, China)

- Ang Zhang (School of Computer, National University of Defense Technology, China)
- Tao Tang (School of Computer, National University of Defense Technology, China)
- Lin Peng (School of Computer, National University of Defense Technology, China)
- Chun Huang (School of Computer, National University of Defense Technology, China)
- Canqun Yang (School of Computer, National University of Defense Technology, China)
- Xiangke Liao (School of Computer, National University of Defense Technology, China)

Abstract: The Smith–Waterman (SW) algorithm is a widely used method for local sequence alignment, but its high computational complexity makes it time-consuming. This paper reviews parallel acceleration techniques for the SW algorithm, including vector-level, thread-level, process-level, and heterogeneous parallelization (CPU-GPU, FPGA, Xeon Phi). The study classifies existing approaches and highlights large-scale genomic comparisons to guide future research in alignment tool development.

Introduction:

- **Sequence alignment** is crucial in bioinformatics for identifying genetic similarities and functions.
- The Smith–Waterman algorithm provides high accuracy but suffers from **quadratic time complexity** (O(mn)), making it inefficient for large datasets.
- Parallelization techniques have significantly reduced computation time, but research efforts are scattered, making systematic study necessary.
- The paper categorizes parallel approaches and their impact on alignment tools.

Key Contributions:

1. **Parallelization Approaches**
    - **Vector-Level Parallelization:** Uses SIMD instructions for faster matrix operations. Techniques include:

    - **Anti-diagonal layout** (Wozniak) – executes diagonal cells in parallel.
    - **Sequential layout** (Rognes & Seeberg) – partitions sequences into SIMD-friendly segments.
    - **Striped layout** (Farrar) – optimizes data access and dependency resolution.
- **Thread-Level Parallelization:** Uses multi-threading on shared-memory architectures (e.g., OpenMP, Pthreads) to process multiple sequence alignments concurrently.
- **Process-Level Parallelization:** Uses distributed memory systems (e.g., MPI) to divide alignment tasks across multiple computing nodes.
- **Heterogeneous Parallelization:**
    - **GPU-based:** *CUDASW++* (wavefront-based execution), *ADEPT*.
    - **FPGA-based:** *SWIFOLD*, *OSWALD* – energy-efficient for short reads.
    - **Xeon Phi-based:** *SWAPHI-LS*, *XSW* – optimized for high-performance computing clusters.

2. **Performance Analysis & Alignment Tools**
    - *SWIPE* (many-to-one layout) achieves 106.2 GCUPS, outperforming striped CPU-based implementations.
    - *SeqAn* (many-to-many layout) reaches 194.1 GCUPS with AVX512 optimizations.
    - Hybrid architectures (CPU-GPU, CPU-FPGA, CPU-Xeon Phi) further improve alignment speeds:
        - *CUDASW++* (GPU-based) is efficient for large datasets.
        - *OSWALD* (FPGA-based) performs well for small and medium datasets.
        - *SWIMM* (Xeon Phi-based) offers high performance but with higher power consumption.

2.3 Fast Sequence Alignment Method Using CUDA-enabled GPU [3]

Year of Publication: 2014
Authors:

- Yeim-Kuan Chang (Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan)
- De-Yu Chen (Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan)

Abstract: Sequence alignment compares DNA, RNA, or protein sequences to find similarities, a fundamental task in bioinformatics. Traditional dynamic programming-based methods (Needleman-Wunsch, Smith-Waterman) provide optimal results but suffer from high computational cost. This paper proposes a CUDA-enabled GPU acceleration method for the Smith-Waterman algorithm, using a **prefix max scan technique** and on-chip shared memory to optimize computation. The approach achieves a 50x speedup over CPU-based implementations and 2–4x over previous GPU-based methods (CUDASW++).

Introduction:

- Sequence alignment is a key task in bioinformatics for genetic analysis and evolutionary studies.
- The Smith-Waterman (SW) algorithm provides optimal local alignment but has **O(mn) time complexity**, making it inefficient for large datasets.
- GPU computing (CUDA) offers a promising solution due to its massive parallelism and high memory bandwidth.
- This paper redefines the recursive formula of SW to enable parallel row-wise computation and reduces memory access latency using shared memory.

Key Contributions:

1. **CUDA-Based Parallelization of Smith-Waterman**
    - Redefining SW's recursive formula to allow each row to be computed in parallel.
    - Using prefix max scan to minimize dependencies and accelerate computations.

- Optimizing memory usage: Only on-chip shared memory is used, reducing global memory penalties.

2. **Performance Evaluation & Comparisons**
   - **Compared against:**
     - CPU-based SW (single-threaded)
     - CUDASW++ (two different GPU-based kernels)
   - **Results:**
     - 50x speedup over CPU implementation
     - 2–4x faster than CUDASW++
     - Handles database sequences up to 8192 bases efficiently

2.4 Mempool - A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory[4]

Year of Publication: 2023
Authors:

- Samuel Riedel (Integrated Systems Laboratory, ETH Zurich, Switzerland)
- Matheus Cavalcante (Integrated Systems Laboratory, ETH Zurich, Switzerland)
- Renzo Andri (Independent Researcher, Zurich, Switzerland)
- Luca Benini (Integrated Systems Laboratory, ETH Zurich, Switzerland; Department of Electrical, Electronic and Information Engineering, University of Bologna, Italy)

Abstract: MemPool is a scalable manycore architecture designed to support hundreds of RISC-V cores with a low-latency shared L1 memory, addressing the challenge of efficient parallel processing. It features a 256-core system based on the **RV32IMAXpulpimg "Snitch" cores**, optimized interconnects, an advanced DMA engine, and an efficient instruction path. MemPool offers high performance (229 GOPS, 180 GOPS/W), minimal execution stalls (≤2%), and low-latency (≤5 cycles L1 access). It provides a flexible programming model supporting bare-metal, OpenMP, and Halide runtimes.

Introduction:

- Modern computing workloads require efficient parallel architectures for applications such as machine learning, genomics, and computational photography.
- Scaling shared L1 memory clusters beyond a few cores is traditionally infeasible due to latency bottlenecks and interconnect complexity.
- MemPool is a **256-core manycore system** that achieves scalability by implementing:
    - A multi-banked, low-latency L1 scratchpad memory.
    - Optimized interconnects for high memory bandwidth.
    - Efficient DMA engines to facilitate data streaming.

Key Contributions:

1. **MemPool Architecture & Design**
    - Uses 256 lightweight RISC-V Snitch cores, each featuring **application-tunable functional units**.
    - Implements an **efficient L1 memory interconnect** for low-latency data sharing.
    - Supports **hierarchical AXI-based interconnects** to facilitate high-throughput memory access.
    - Introduces a **distributed DMA engine** to efficiently move data between system memory and cores.
2. **Performance Optimization**
    - L1 Memory Access:
        - 5-cycle latency in the absence of conflicts.
        - Hybrid addressing scheme to reduce inter-tile memory requests.
        - Optimized instruction cache (L0 + L1) for reduced stall cycles.
    - Execution Efficiency:
        - High IPC (0.96) with minimal execution stalls.
        - Compute-intensive applications achieve **229 GOPS and 180 GOPS/W efficiency**.
3. **Programming & Runtimes**
    - Bare-metal runtime for direct control and optimization.
    - OpenMP support for parallel programming.
    - Halide integration for high-level application development in image processing & ML.

CHAPTER 3
BACKGROUND AND PREREQUISITES

3.1 MemPool Platform [4]

We use Mempool which is a scalable many-core RISC-V platform with low-latency shared L1 memory, as our architectural simulator

3.1.1. Architecture Diagrams

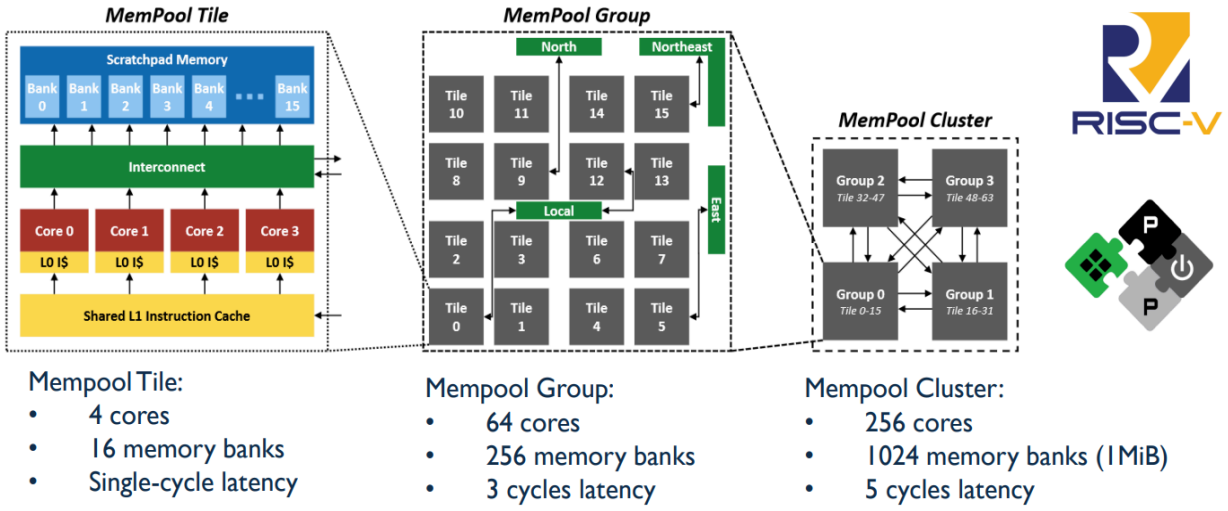

Fig 3.1 - Top Level Architecture Diagram [4]

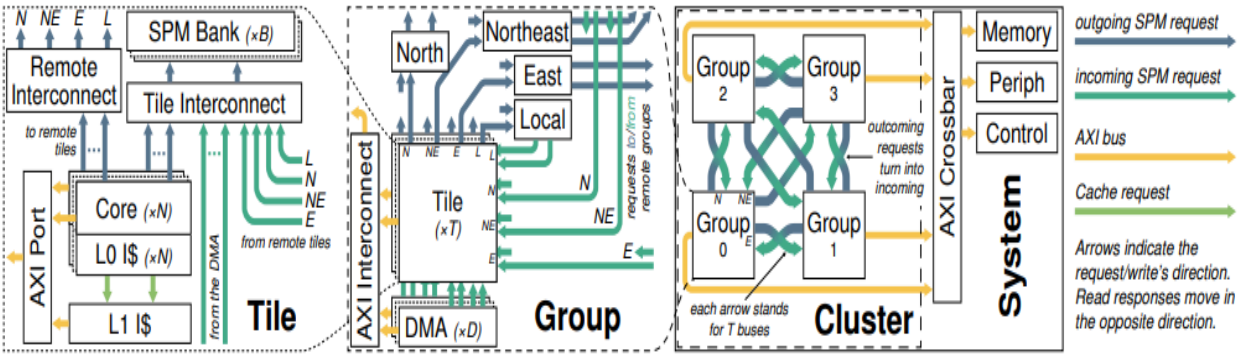

Fig. 3.2 - Detailed Architecture Diagram [4]

3.1.2 Tile Interconnect and DMA Engine

The MemPool architecture employs a tile-based design, where each tile features a shared AXI master port. This port facilitates access to higher-level main memory, peripherals, and control registers, serving both the processing cores within the tile and the instruction cache refill mechanism.

3.1.3 Local Crossbar Interconnect

A local crossbar within each tile acts as an interconnect block, enabling communication between the processing cores and the tile's scratchpad memories. This crossbar provides a low-latency pathway for data exchange within the tile, crucial for efficient inter-core communication and data sharing.

3.1.4 DMA Engine and Memory Access

Each tile integrates a Direct Memory Access (DMA) engine, which plays a pivotal role in optimizing data transfer operations. The DMA engine possesses the following key characteristics:

- **Independent Memory Access:** The DMA engine is equipped with its own set of registers and control mechanisms, allowing it to directly access memory without CPU intervention.
- **AXI and Local Crossbar Access:** The DMA engine can access both the AXI interconnect, providing access to external memory and peripherals, and the local crossbar, facilitating data transfer to and from the tile's scratchpad memories.
- **Performance Enhancement:** By offloading data transfer tasks from the CPU, the DMA engine significantly improves system performance, reducing CPU overhead and latency.
- **Memory Management:** The DMA engine may also handle memory allocation and deallocation for data modules, which are devices requiring direct memory access. This feature optimizes memory usage and minimizes data transfer latency.

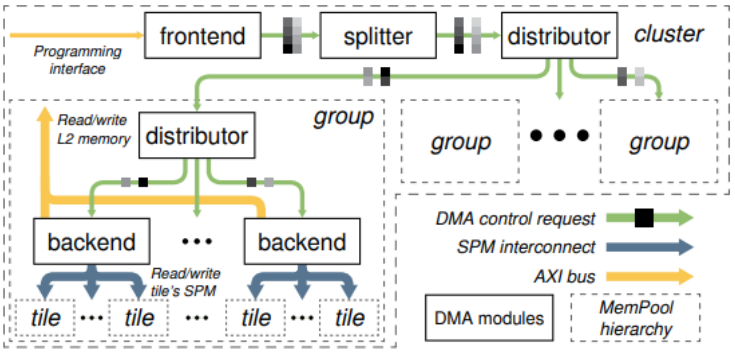

Fig 3.3 - Memory Access Flow [4]

3.1.5 AXI Port to Interconnect

Each tile is further connected to the overall system interconnect via an individual AXI port, enabling seamless communication with other tiles and

system components. This architecture promotes scalability and efficient data flow within the MemPool system.

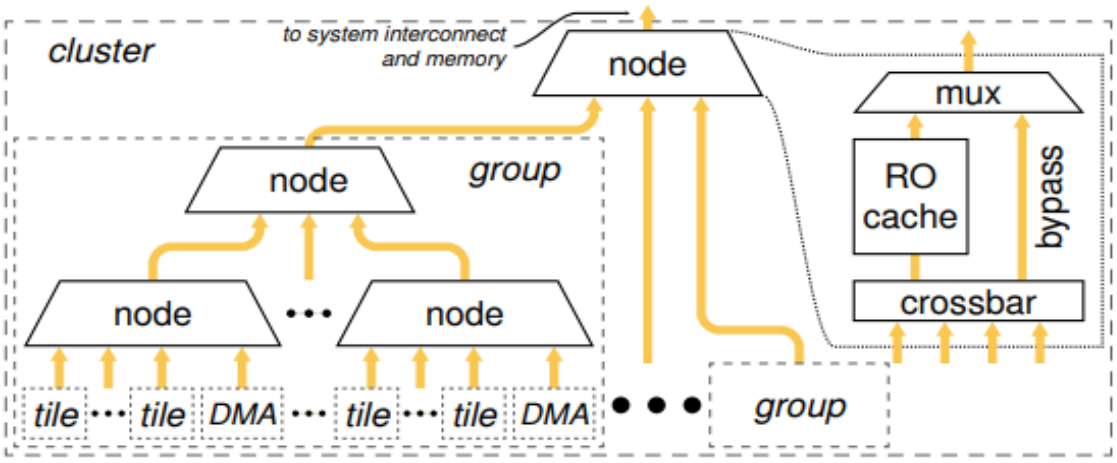

Fig 3.4 - AXI Interconnect Hierarchy

3.1.6 Hybrid Memory Addressing

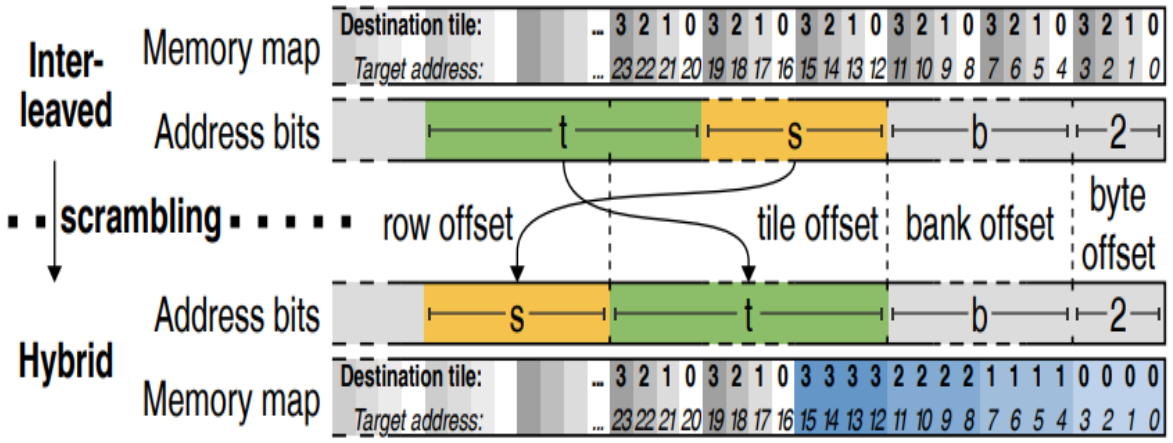

Fig 3.5 - Hybrid Memory Addressing

In the MemPool architecture, memory is divided into banks, and accessing contiguous locations can incur latency due to bank switching when those locations span bank boundaries. To address this, each bank is designed with dedicated sequential regions. This ensures that consecutive data can often be accessed within a single bank, minimizing the need for bank switching and its associated latency. By optimizing for spatial locality, where sequential data access is common, the architecture improves memory access efficiency and overall performance.

3.2 Genomics and the DNA

Deoxyribonucleic acid (DNA) is the molecular foundation of heredity, encoding the genetic instructions required for the development, function, and perpetuation of all known living organisms and many viruses. It serves as a stable yet dynamic repository of biological information, ensuring precise genetic transmission across generations while allowing for evolutionary adaptation. The structural organization of DNA is fundamental to its function, with its distinctive double-helical conformation enabling efficient storage, replication, and expression of genetic material.

DNA consists of two antiparallel polynucleotide strands, each composed of a linear sequence of nucleotides. Each nucleotide comprises a deoxyribose sugar, a phosphate group, and a nitrogenous base, which collectively establish the structural and functional integrity of the molecule. The four nitrogenous bases - adenine, guanine, cytosine, and thymine - adhere to strict complementary base-pairing rules, with adenine forming two hydrogen bonds with thymine and cytosine forming three hydrogen bonds with guanine. This complementarity is critical for replication fidelity and the conservation of genetic information.

The genome of an organism encompasses its complete DNA sequence, organized into chromosomes and compacted through interactions with histone proteins. Within this genomic framework, genes serve as discrete units of heredity, encoding instructions for protein synthesis through a highly regulated process involving transcription and translation. Transcription yields messenger RNA (mRNA), which conveys genetic information to ribosomes, where translation facilitates the precise assembly of amino acid sequences into functional proteins. This central dogma of molecular biology underscores the hierarchical relationship between genotype and phenotype, governing cellular processes and organismal traits.

The ability of DNA to self-replicate is essential for cellular proliferation and genetic continuity. During replication, the double helix unwinds, and each strand serves as a template for the synthesis of a complementary strand, a process mediated by DNA polymerases and associated enzymatic complexes. The semiconservative nature of this mechanism ensures high-fidelity duplication, although replication errors and external mutagens may introduce

mutations. While some mutations are deleterious, others contribute to genetic variability and evolutionary adaptation, shaping the diversity of life.

DNA is not a static entity but rather a dynamic substrate for regulatory mechanisms that modulate gene expression. Epigenetic modifications, including DNA methylation and histone modifications, influence chromatin accessibility and transcriptional activity without altering nucleotide sequences. These regulatory processes enable cellular differentiation, response to environmental stimuli, and heritable phenotypic variation. The interplay between genetic and epigenetic factors is central to developmental biology, disease pathogenesis, and evolutionary biology.

3.3 Genome Sequencing Pipeline [1]

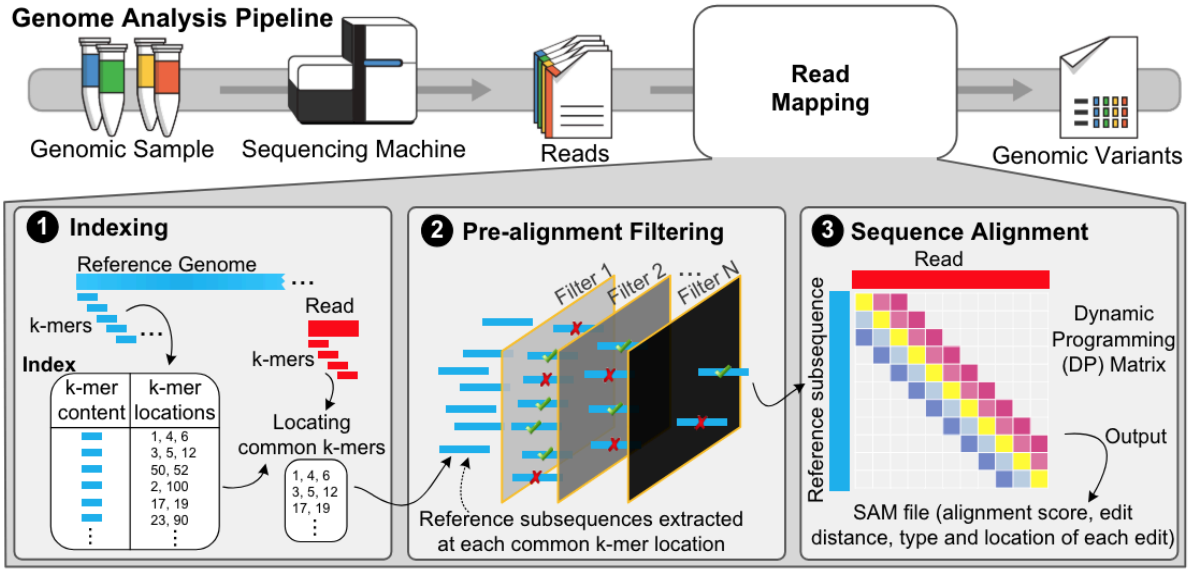

Fig 3.6 - The Genome Analysis Pipeline [1]

Genome sequencing is a multi-stage computational and laboratory process that deciphers the order of nucleotides (A, T, G, and C) in a DNA molecule. The sequencing pipeline consists of several key stages: DNA extraction and preparation, sequencing, base calling, pre-processing, read mapping, variant calling, and downstream analysis. Each stage involves both biological and computational techniques that ensure high accuracy and efficiency in interpreting genomic data.

The steps in the pipeline are as follows:

1. **DNA Extraction and Preparation:**
   The process begins with extracting DNA from a biological sample, such as blood, saliva, or tissue. Since DNA exists in long strands, it must be broken into smaller pieces to make sequencing manageable. Special molecules called adapters are attached to these fragments, helping them bind to a surface where sequencing can take place. The prepared DNA is then amplified to ensure there is enough material for sequencing.

2. **Sequencing: Reading the DNA Fragments:**
   In this step, the actual sequencing takes place. The DNA fragments are processed through a sequencing machine that determines the order of their building blocks (nucleotides). Since DNA is double-stranded, sequencing may involve reading both strands multiple times to improve accuracy. The result of this stage is a large collection of short or long DNA sequences, referred to as reads, which represent small portions of the full genome.

3. **Base Calling: Converting Signals into DNA Sequences**
   The sequencing process produces raw signals rather than direct DNA letters. These signals are processed to determine the correct sequence of nucleotides (A, T, G, and C). Errors may occur during this step, so confidence scores are assigned to each nucleotide to indicate how reliable the sequence is.

4. **Pre-processing: Cleaning the Data**
   Since sequencing can introduce errors, the raw reads must be processed to remove unwanted portions. This involves:
   - Filtering out low-quality reads that have too many errors.
   - Trimming unwanted sequences, such as extra adapters.
   - Correcting errors, if needed, to improve the accuracy of the reads.

   This step ensures that only high-quality DNA sequences are used in the next stages.

5. **Read Mapping: Placing Reads in the Genome**
   Since sequencing produces small fragments, these must be arranged in the correct order to reconstruct the full genome. This is done by comparing each read to a reference genome, which is a known sequence of the same species. The reads are placed in the most likely position based on how well they match. This step can be challenging if there are repetitive regions in the genome, where multiple locations look similar. The result of this process is a structured alignment of all the reads, showing where they belong in the full genome.

6. **Variant Calling: Detecting Differences**
   Once the genome is assembled, the next step is to identify variations—differences between the sequenced genome and the reference genome. These variations can be:
   - Single-letter changes in the DNA sequence.
   - Small insertions or deletions of DNA fragments.
   - Larger structural changes, such as missing or duplicated sections.

   Identifying these differences is essential for understanding genetic diversity, inherited diseases, and mutations linked to specific traits.

3.4 Read Mapping: The Computational Bottleneck [1]

Read mapping, the process of aligning short DNA sequences (reads) to a reference genome, has been identified as a significant computational bottleneck in genome sequencing pipelines. Studies have consistently highlighted the time-consuming nature of this step, particularly as sequencing depths and genome sizes increase. Within read mapping, sequence alignment stands out as the most demanding task.

3.5 Brief Overview of Genome Sequencing Algorithms

Genome sequencing algorithms fall into three main categories:

- **Exact Matching Algorithms:** Prioritize precise matches.
- **Heuristic Algorithms:** Trade accuracy for speed.
- **Dynamic Programming Algorithms:** Guarantee optimal alignments.

3.5.1 Exact Matching Algorithms

1. **Naive String Matching**

The naive string matching algorithm compares a query sequence (read) against the reference genome at every possible position.

Advantages:
- Simple and straightforward implementation.
- No preprocessing of the reference genome is required.

Disadvantages:
- Computationally inefficient for large genomes, with a time complexity of **O(mn)** (where m is read length and n is genome length).
- Impractical for modern sequencing applications.

Due to its inefficiency, naïve string matching is rarely used in genome sequencing.

2. **Knuth-Morris-Pratt (KMP) and Boyer-Moore Algorithms**

These algorithms enhance naive string matching by precomputing failure functions (KMP) or utilizing mismatch heuristics (Boyer-Moore) to skip unnecessary comparisons.

Advantages:
- Faster than naive matching, with time complexity O(n + m) (KMP) or O(n/m) on average (Boyer-Moore).
- Efficient for exact substring searches.

Disadvantages:

- Unsuitable for handling sequencing errors or mutations, as they rely on exact matches.

While effective for small-scale searches, these methods are inadequate for large-scale sequencing due to their inability to tolerate mismatches and indels.

3.5.2 Heuristic Algorithms for Genome Alignment

To address the computational cost of exact matching, heuristic-based algorithms provide approximate but faster solutions.

1. **BLAST (Basic Local Alignment Search Tool)**

BLAST identifies highly similar regions by breaking the query sequence into short words and searching for matches in the reference genome.

Advantages:

- Significantly faster than dynamic programming algorithms.
- Effective for detecting conserved sequences and homologous regions.
- Optimized for large-scale databases.

Disadvantages:

- Produces approximate alignments and may miss weak matches.
- Ineffective for highly divergent sequences.
- Trades accuracy for speed, limiting its utility for variant detection.

BLAST remains widely used for genomic searches but is not ideal for precise alignment in genome sequencing.

2. **FASTA Algorithm**

FASTA is similar to BLAST, relying on k-mer matching to identify regions of similarity before local alignment.

Advantages:

- Faster than BLAST for certain applications.
- Useful for efficiently finding highly similar sequences.

Disadvantages:

- Less sensitive than BLAST in detecting weak similarities.
- Not optimal for whole-genome alignment.

Both BLAST and FASTA are efficient for database searches but are insufficient for full genome sequencing, where precision is essential.

3.5.3 Dynamic Programming-Based Algorithms

Dynamic programming (DP) provides optimal sequence alignment by constructing a scoring matrix and identifying the best alignment path.

1. **Needleman-Wunsch Algorithm (Global Alignment)**

Needleman-Wunsch performs global alignment, aligning the entire length of two sequences.
In a *m* x *n* matrix, each value are defined by F_{ij}, where

$$F_{ij} = max \begin{cases} F(\nwarrow) + S(match/mismatch) \\ F(\uparrow) + S(indel) \\ F(\leftarrow) + S(indel) \end{cases}$$

And S(match/mismatch) and S(indel) represent the substitution table values for the specific conditions.

Advantages:

- Guarantees optimal alignment.
- Handles mismatches and gaps through scoring penalties.

Disadvantages:

- Computationally expensive, with **O(mn)** time and space complexity.
- Unsuitable for aligning short sequencing reads against long reference genomes.

Due to the fragmentary nature of sequencing reads, Needleman-Wunsch is rarely used in practice.

2. **Smith-Waterman Algorithm (Local Alignment)**

The Smith-Waterman algorithm finds the best matching subsequence between a query and a reference genome.

Advantages:

- Highly accurate, finding the best local matches even with sequencing errors.
- Handles mutations, insertions, and deletions.
- Well-suited for genome sequencing due to its local alignment approach.

Disadvantages:

- Computationally intensive (**O(mn)** time complexity).

- Requires significant memory and processing power.

Despite its computational cost, Smith-Waterman is widely used for short-read alignment, particularly when accuracy is paramount.Genome sequencing demands efficient algorithms for aligning short DNA reads to reference genomes. While heuristic methods offer speed, they lack the precision needed. Dynamic programming algorithms provide optimal alignments, with Smith-Waterman being preferred for its accuracy and adaptability.
Hence, we select to parallelize the Smith-Waterman algorithm as it is the most-used alignment algorithm.

CHAPTER 4

ALGORITHM

4.1 Smith-Waterman Algorithm

The Smith-Waterman algorithm employs a dynamic programming approach by constructing a scoring matrix and performing a traceback to identify the optimal local alignment.

Key Characteristics:

1. **Local alignment:** Finds the best matching subsequence instead of aligning the entire sequence.

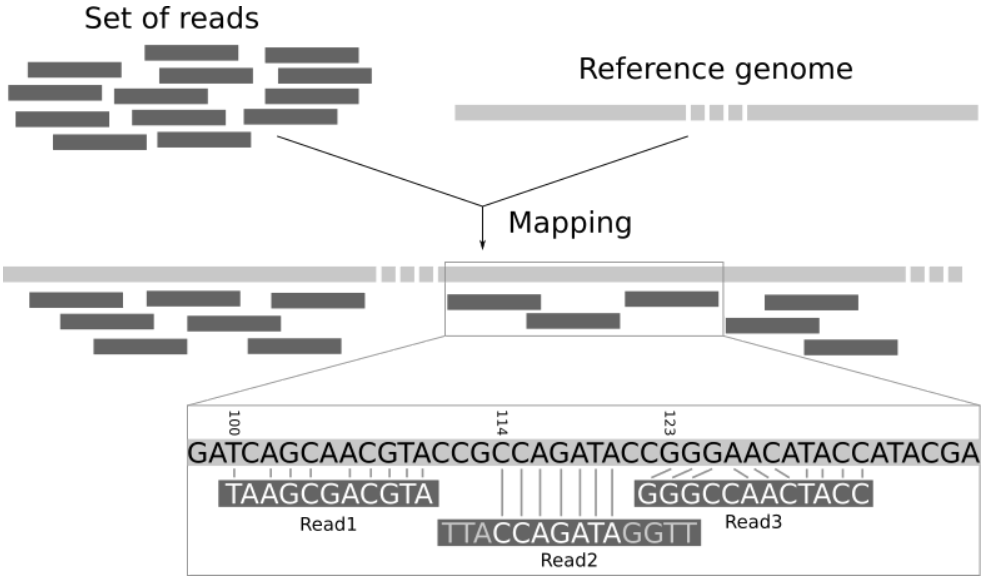

Fig 4.1 - Local Alignment

2. **Affine gap penalties:** Assigns different penalties for opening and extending gaps. The affine gap penalty combines the components in both the constant and linear gap penalty, taking the form $A + B \cdot (L - 1)$. This introduces new terms, A is known as the gap opening penalty, B the gap extension penalty and L the length of the gap

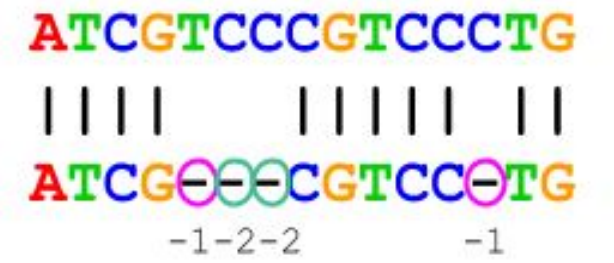

Fig 4.2 - Affine Gap Penalties

3. **Backtracking:** Identifies the highest-scoring local alignment by tracing back the matrix.

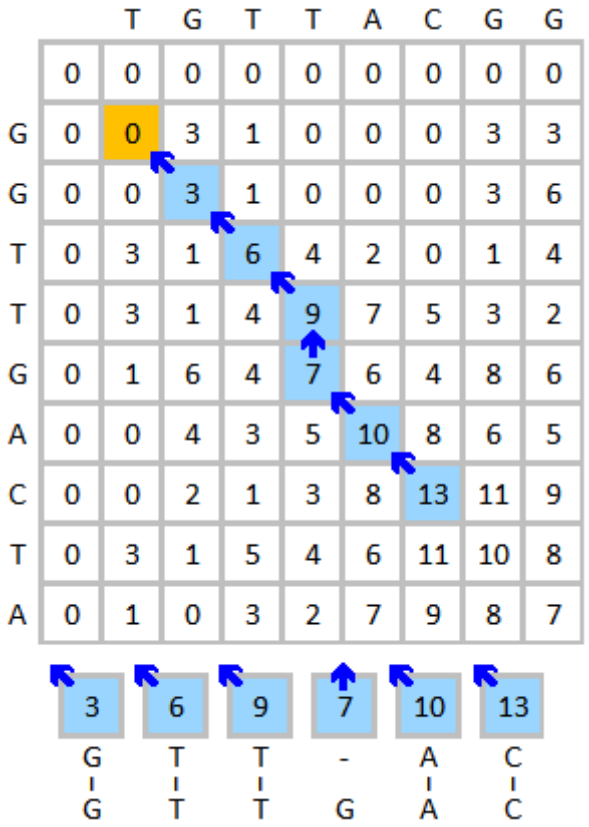

Fig 4.3 - Trace-back stage

4.1.1 Steps of the Smith-Waterman Algorithm:

Step 1: Initialization

A scoring matrix H of size $(m+1) \times (n+1)$ is created, where

- m is the length of the first sequence.
- n is the length of the second sequence.

The first row and first column are initialized to zero because local alignment does not penalize unmatched regions at the start, and allows alignments to start anywhere in the sequences.

$$H_{i,0} = 0 \quad \forall i \in [0, m]$$
$$H_{0,j} = 0 \quad \forall j \in [0, n]$$

Eqn 4.1 - Initial Values

Step 2: Matrix Filling

Each cell H(i,j)H(i,j) is computed using the following recurrence relation:

$$H_{i,j} = \max \begin{cases} 0, & \\ H_{i-1,j-1} + S(A_i, B_j) & \text{(Match/Mismatch)} \\ H_{i-1,j} - g & \text{(Insertion)} \\ H_{i,j-1} - g & \text{(Deletion)} \end{cases}$$

Eqn 4.2 - Cell computation

Where:

- $S(x_i, y_j)$ is the substitution score for aligning characters x_i and y_j, with a match reward (s) and mismatch penalty (p), given by:

$$S(A_i, B_j) = \begin{cases} +s, & \text{if } A_i = B_j \text{ (match)} \\ -p, & \text{if } A_i \neq B_j \text{ (mismatch)} \end{cases}$$

Eqn 4.3 - Substitution Function

- g is the gap penalty (for insertions/deletions).
- The 0 value ensures local alignment, preventing negative scores from propagating.

Step 3: Identifying the Optimal Alignment

- The highest-scoring cell in the matrix represents the end of the best local alignment.
- A traceback is performed from this highest-scoring cell until a zero is encountered.

Step 4: Traceback and Alignment Reconstruction

- Starting from the highest-scoring cell, the optimal path is traced back using the same scoring rules, in the direction of maximum values:
  - ↖ - (i - 1, j - 1) - Match/Mismatch
  - ↑ - (i - 1, j) - Insertion
  - ← - (i, j - 1) - Deletion
- This process reconstructs the best local alignment.

Computational Complexity:

The Smith-Waterman algorithm has a time complexity of ***O*(*mn*)** and a space complexity of ***O*(*mn*)**, making it computationally expensive for large sequences.

4.1.2 Example matrix calculation

Consider the following two sequences:

➔ A: “ACGT”

➔ B: “AGT”

and let the points be as follows:

- Match reward s = 2
- Mismatch penalty p = 1
- Gap penalty g = 2

The matrix is created as follows:

	0	A	G	T
0	0	0	0	0
A	0	–	–	–
C	0	–	–	–
G	0	–	–	–
T	0	–	–	–

Fig 4.4 - Empty Example Matrix

For the (1,1) position, we compute as follows:

$$H_{1,1} = \max \begin{cases} 0 & = 0 \\ H_{0,0} + S(2/1) & = 2 \\ H_{0,1} - 2 & = -2 \\ H_{1,0} - 2 & = -2 \end{cases}$$

Eqn 4.4 - Example Calculation of $H_{(1,1)}$

Following this, the matrix is filled as follows:

	0	A	G	T
0	0	0	0	0
A	0	2	0	0
C	0	0	1	0
G	0	0	2	0
T	0	0	0	4

Fig 4.5 - Filled Example Matrix

The final alignment is:

A C G T
A G T

Fig 4.6 - Final Example Alignment

4.2 Modified Smith-Waterman Algorithm[3]

4.2.1 Redefining the Recursive Formula for Parallel Execution

The main challenge preventing parallelization of the formula is that each $H_{(i,j)}$ depends on three preceding values, making direct parallel computation difficult. Instead of computing the matrix diagonally, the recurrence relation is redefined so that one entire row can be computed in parallel.

$$H_{(i,\ j)} = max \begin{cases} 0 \\ H_{(i,\ j-1)} + p \\ H_{(i-1,\ j)} + p \\ H_{(i-1,\ j-1)} + SbtCost \end{cases}$$

	-	A	T
-	0	0	0
A	0	1	0

Eqn 4.5 - Original relation with highlighted dependencies

$$H'_{(i,\ j)} = max \begin{cases} 0 \\ H_{(i-1,\ j)} + p \\ H_{(i-1,\ j-1)} + SbtCost \end{cases}$$

$$H_{(i,\ j)} = max \begin{cases} H'_{(i,\ j)} \\ H_{(i,\ j-1)} + p \end{cases}$$

	-	A	T
-	0	0	0
A	0	1	0

Eqn 4.6 - Separation of dependencies

4.2.2 Prefix Max Scan Optimization

The bottleneck in computing each row lies in evaluating:

$$H_{(i,\ j)} = max \begin{cases} H'_{(i,\ j)} \\ H_{(i,\ j-1)} + p \end{cases}$$

Eqn 4.7 - Original Row Computation formula

To accelerate this, the prefix max scan method is applied, which computes:

$$y_i = \max(x_1, x_2, \ldots, x_i), \quad \forall i \in \{1, 2, \ldots, n\}$$

$$Input = [a,\ b,\ c,\ d]$$

$$Output = [max(a,a),\ max(a,b),\ max(a,b,c),\ max(a,b,c,d)]$$

Eqn 4.8 - Prefix Max Scan

Using a parallel logarithmic-time scan algorithm:

- **Step 1:** Compare adjacent elements and store the max.
- **Step 2:** Repeat for higher-order neighbors, doubling the step size each time.
- The total complexity is ***O*(log*n*)** instead of ***O*(*n*)** per row.

CHAPTER 5

IMPLEMENTATION

The algorithm reveals potential opportunities for parallelization along the anti-diagonal. Initially, we attempt to parallelize the Smith-Waterman algorithm along the diagonal.

5.1 Anti-diagonal Approach:

Consider a system with p processing cores and k elements in a single anti-diagonal. The number of active cores is determined as $\min(p, k)$, ensuring that $p \leq k$. The number of active cores is dynamically adjusted according to the length of the diagonal, which initially increases and then decreases.

The computational workload is evenly distributed among the available cores for processing the anti-diagonal. Synchronization is enforced using barriers after processing each anti-diagonal containing more than one element.

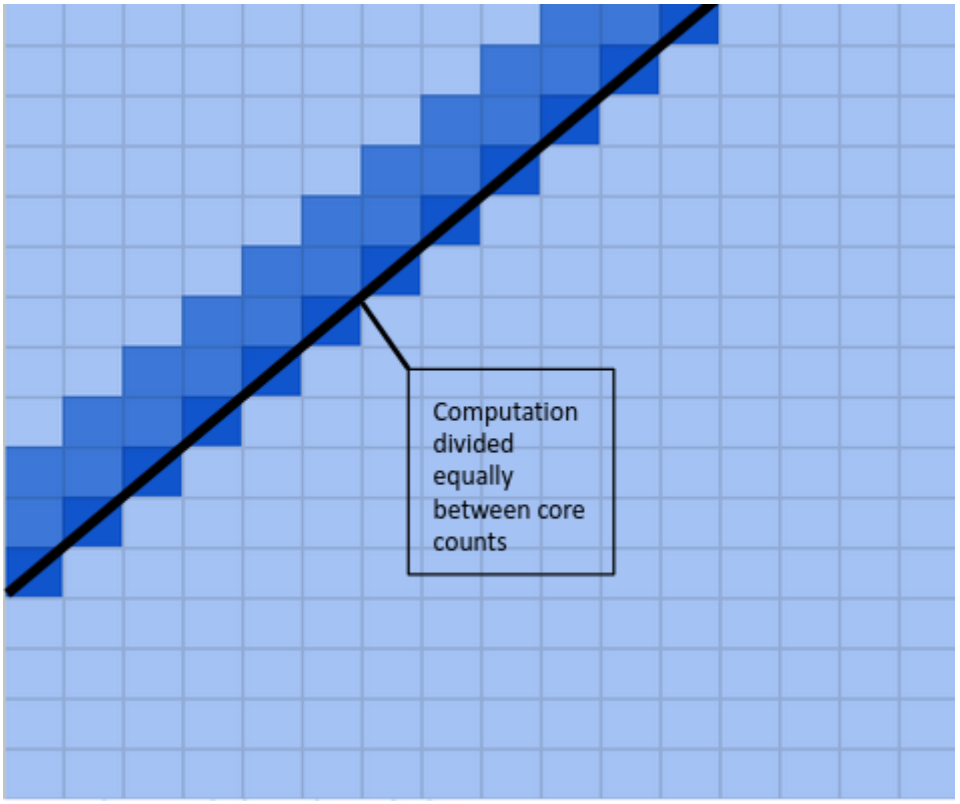

Fig 5.1 - Diagonal Computation Diagram

The obtained results and speedups for the anti-diagonal method, corresponding to query-reference character lengths of 20×40, 30×70, and 40×100, are presented below.

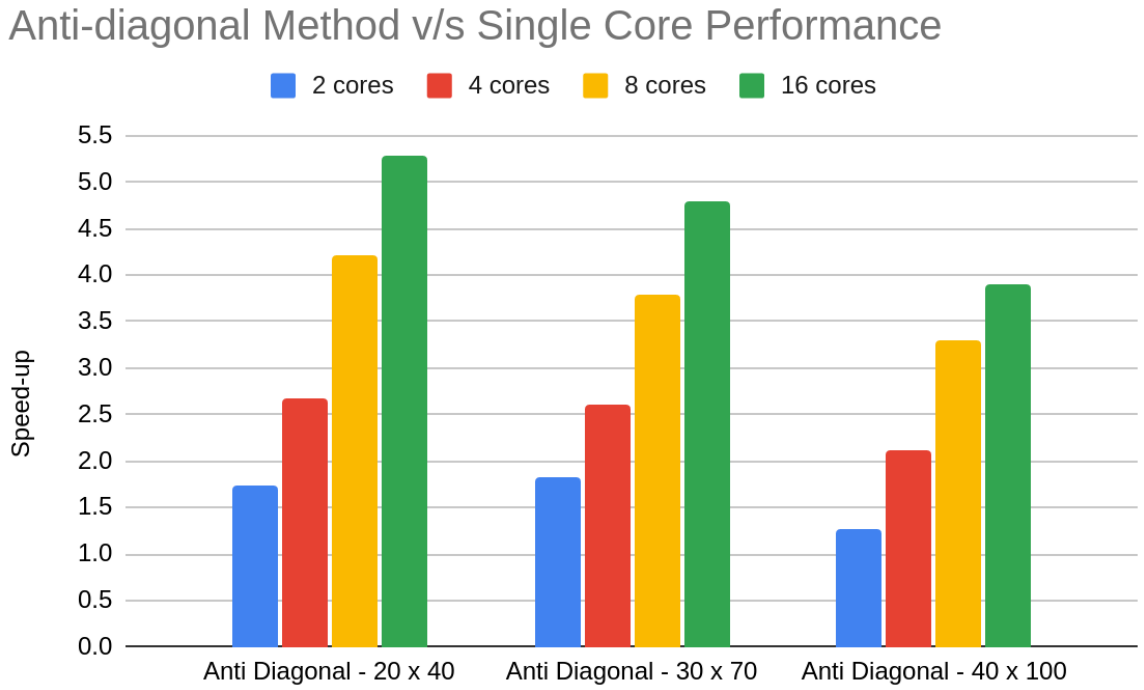

Fig 5.2 - Anti-diagonal Method v/s/ Single Core Performance

With the anti-diagonal method, we achieve an approximate speedup of 5x on 16 cores compared to executing the same method on a single core. This indicates that the method does not scale efficiently.

The primary limitation of this approach is that the diagonal size increases to a maximum before decreasing, necessitating core stalling when the number of elements in the diagonal is smaller than the available core count. This results in computational inefficiency.

Since sequence alignment typically involves aligning a smaller query against a larger read, the row sizes are generally substantial. Therefore, processing entire rows at once could enhance core utilization and improve overall efficiency.

5.2 Row Parallel Method

To mitigate the inherent sequential dependency between adjacent elements in the traditional Smith-Waterman algorithm, an intermediate computational stage is introduced. This modification enables the parallel computation of the initial segment of the scoring matrix row. This adaptation of the algorithm is detailed in Section 4.2.

5.2.1 Row-Parallel Speedup Against Row Parallel For Single Core

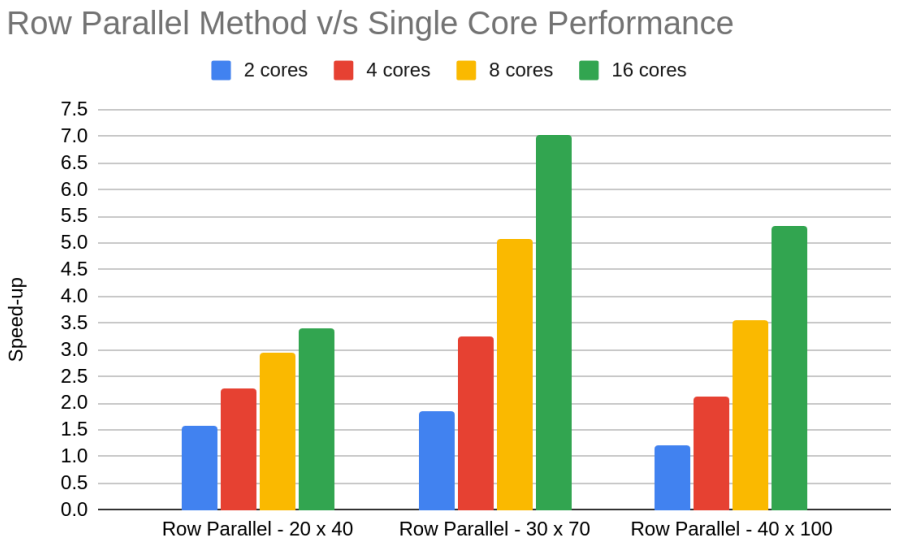

Fig 5.3 - Row Parallel Methods v/s Single Core Performance

5.2.2 Parallel Methods Speedup Against Serial Implementation

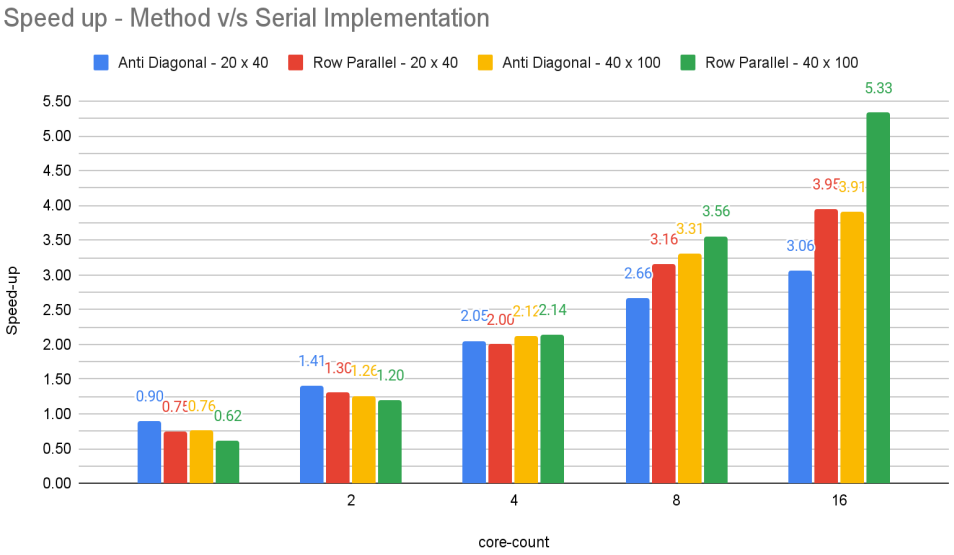

Fig 5.4 - Parallel Methods Speedup against Serial Implementation

We observe that the anti-diagonal method exhibits slightly better performance than the row-parallel approach up to 4 cores. However, for 8 cores and beyond, the row-parallel method outperforms the anti-diagonal approach.

5.2.3 Superior Performance of Row-Parallel Over Anti-Diagonal for 8 Cores

	Row Parallel	Anti Diagonal
Total instructions	28873	14053
Total cycles	12409	14705
Cycles to compute rows/diagonals where 1+ cores are stalled	None(no stalled cores)	2516 {diagonals (0-6) & (52-59)} [56 elements]
Cycles to compute above elements using all 8 cores	None(no rows where cores are stalled)	535

Total cycles	12409	12724

- For core counts (cc) ranging from 1 to 4, the anti-diagonal method outperforms the row-parallel approach due to lower total kernel instruction counts: 14,305 (Anti-Diagonal), 17,061 (Row-Parallel), and 9,101 (Serial). However, as the core count increases, performance deteriorates due to core stalls.
- The initial and final seven diagonals cause one or more cores to stall, accounting for 2,516 cycles, while the remaining diagonals, where all cores are fully utilized, require 11,542 cycles.
- Both the Anti-Diagonal and Row-Parallel methods take approximately 570–600 cycles to compute 40 elements using eight cores. Optimizing the initial and final diagonals (56 elements) to approximately 500 cycles could reduce the total Anti-Diagonal cycles to around 12,724, bringing it closer to the Row-Parallel execution time of 12,409 cycles.
- As the core count increases (e.g., cc=16cc = 16), the number of stalled cores grows across more diagonals (e.g., diagonals 0–14), further degrading the performance of the anti-diagonal approach.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 Roofline Model

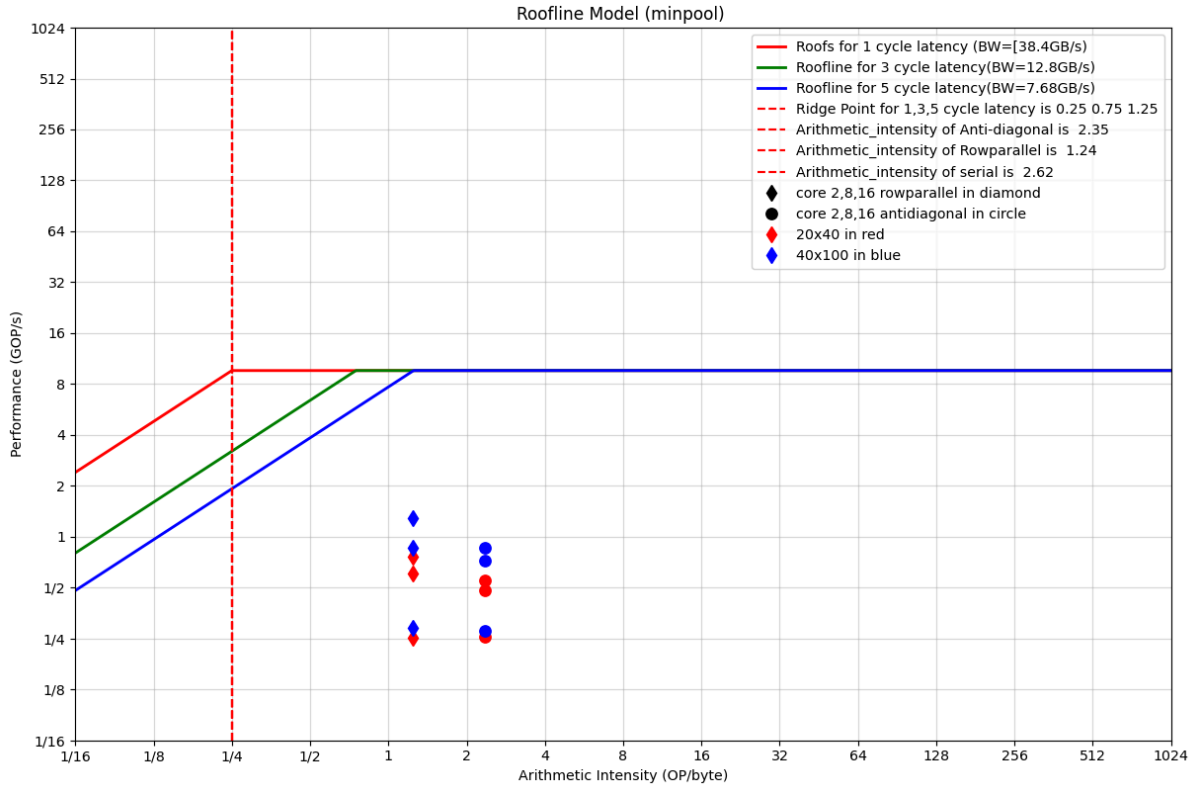

Fig 6.1 - Roofline Model

From the roofline model, we determine that the arithmetic intensity of the anti-diagonal method is 2.26, whereas the row-parallel method has an arithmetic intensity of approximately 1.24. The ridge point for a 5-cycle latency is 1.25, indicating that the row-parallel method is slightly memory-bound.

However, in terms of throughput, the row-parallel method achieves 1.29 GOP/s, compared to 0.856 GOP/s for the anti-diagonal method. This suggests that despite its lower arithmetic intensity, the row-parallel approach delivers higher throughput while remaining close to the compute-bound region, making it the optimal algorithm.

The values to generate the roofline diagram are as follows:

DIMENSION 20x40	Anti-Diagonal	Row Parallel	Serial
Total Memory Instructions	2524	4491	1672
Total Compute Instructions	11781	12570	6628
Total Instructions	14305	17061	8300
Byte L/S	773	500	814
Half-Word L/S	1283	3196	831
Word L/S	468	795	27
Total Bytes Required	5211	10072	2584
Arithmetic Intensity	2.260794473	1.248014297	2.56501548

Table 6.1 - Arithmetic Intensity calculation - 20 x 40 Matrix

DIMENSION 40x100	Anti-Diagonal	Row Parallel	Serial
Total Compute Instruction	54967	60518	32866
Total Memory Instruction	12535	22812	8217
Total Instruction	67502	83330	41083
Byte L/S	3953	2194	4062
Half-Word L/S	7427	17238	4118
Word L/S	1155	3380	37
Total Bytes Required	23427	50190	12446
Arithmetic Intensity	2.346309813	1.205778043	2.640687771

Table 6.2 -Arithmetic Intensity calculation - 40 x 100 Matrix

	Row Parallel 20x40	Anti-Diagonal 20x40	Row Parallel 40x100	Anti-Diagonal 40x100
Compute instructions	12570	11781	60518	54967
Cores	**Throughput**	**Throughput**	**Throughput**	**Throughout**
1	1.44E+08	1.63E+08	1.48E+08	1.66E+08
2	2.51E+08	2.54E+08	2.90E+08	2.77E+08
4	3.86E+08	3.71E+08	5.15E+08	4.65E+08
8	6.08E+08	4.81E+08	8.58E+08	7.25E+08
16	7.61E+08	5.52E+08	1.29E+09	8.56E+08

Table 6.3 - Throughput Calculation Values

CHAPTER 7
CONCLUSION AND FUTURE WORK

This project explored the implementation and optimization of the Smith-Waterman algorithm for genomic sequence alignment on the MemPool many-core architecture. We investigated two primary parallelization strategies: the anti-diagonal approach and the row-parallel method.

Our initial exploration of the anti-diagonal method demonstrated a modest speedup, reaching approximately 5x on 16 cores compared to a single-core implementation. However, this approach suffered from inherent limitations due to the varying size of the anti-diagonals, leading to significant core idle time and reduced scalability.

In contrast, the row-parallel method, which effectively mitigated dependencies within rows, exhibited superior performance, particularly at higher core counts. By strategically introducing an intermediate stage, we were able to compute substantial portions of each row in parallel, maximizing core utilization. Our performance analysis revealed that while the anti-diagonal method provided a slight advantage at lower core counts, the row-parallel method consistently outperformed it for 8 cores and beyond. This improvement is attributed to the elimination of core stalling, as evidenced by the reduced total cycle count and the absence of idle time observed in the row-parallel implementation.

The comparative analysis of instruction and cycle counts between the anti-diagonal and row-parallel methods further underscored the efficiency of the row-parallel approach. Notably, the row-parallel method achieved a lower total cycle count with a higher instruction count, indicating a more efficient utilization of computational resources.

Future Work:

1. **Hybrid Approaches:**
    - Investigate hybrid parallelization strategies that combine the strengths of both anti-diagonal and row-parallel methods. For example, dynamically switching between strategies based on the size of the alignment matrix could potentially yield further performance improvements.
2. **Integration with Genomic Pipelines:**
    - Integrate the optimized Smith-Waterman implementation into a complete genomic analysis pipeline to evaluate its impact on end-to-end performance.

- Explore the implementation of other algorithms within the genomic sequencing pipeline on the mempool platform.

3. **Increasing Throughput**

   - Use other algorithmic optimizations such as vectorization and prefetching to further increase throughput of the row-parallel

REFERENCES

[1] M. Alser, Z. Bingöl, D. S. Cali, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[2] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A Scalable Manycore Architecture With a Low-Latency Shared L1 Memory," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 143-162, 2023.

[3] T. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation," *BMC bioinformatics*, vol. 12, no. 1, p. 221, 2011.

[4] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "MemPool: A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory," *IEEE Transactions on Computers*, vol. **72**, no. **12**, pp. **1-16**, 2023. doi: 10.1109/TC.2023.3307796.

[5] Ł. Ligowski, W. Rudnicki, and P. Wojciechowski, "Parallel implementation of the Smith-Waterman algorithm for CUDA-enabled GPUs," in *Parallel Processing and Applied Mathematics*, 2013, pp. 105-115.

[6] L. Sandes and E. Alba, "Parallelization of the Smith-Waterman algorithm on multicore processors," *The Journal of Supercomputing*, vol. 68, no. 1, pp. 143-162, 2014.

[7] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gómez-Luna, A. Boroumand, A. V. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 951–966, 2020.

[8] D.-H. Park, J. Beaumont, and T. Mudge, "Accelerating Smith-Waterman alignment workload with scalable vector computing," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 661–668, 2017.

[9] R. Barnes, "A review of the Smith-Waterman GPU landscape," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2020-152*, 2020.

[10] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007.

[11] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[12] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "GPU accelerated Smith-Waterman algorithm for large-scale sequence alignment," *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, p. 79, 2006.

[13] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270–1281, 2007