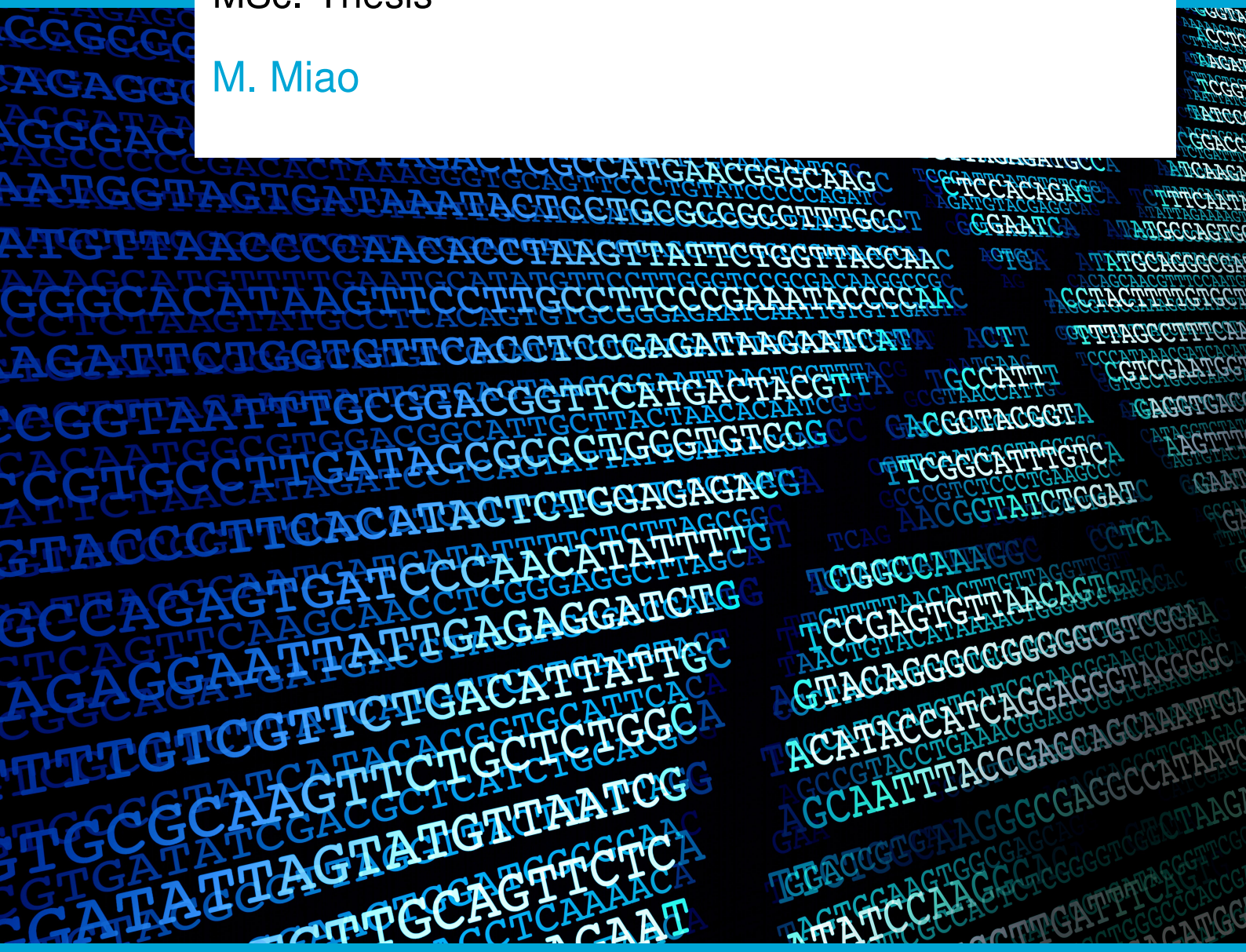
CIM-Architecture for Acceleration of DNA Pre-Alignment Filters

MSc. Thesis

M. Miao



TU Delft Delft University of Technology

April, 2023

CIM-Architecture for Acceleration of DNA Pre-Alignment Filters

MSc. Thesis

by

M. Miao

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday April 24, 2023 at 1:00 PM.

Student number:	4684206
Project duration:	December, 2021 – April, 2023
Thesis committee:	Dr.ir. J.S.S.M. Wong, CE, TU Delft, supervisor
	Dr.ir. M.A.Z. Zuniga, ENS, TU Delft
	MSc. T. Shahroodi, CE, TU Delft

Cover photo adapted from [1]

TU Delft

Abstract

Due to recent developments in DNA sequencing technology, there is a growing abundance of available genomic data. To process this information for use in fields such as healthcare and forensics, raw sequencing data have to be processed using computationally intensive algorithms. Currently, one of the major bottlenecks in this processing pipeline is the alignment step, which makes use of dynamic-programming algorithms. To reduce computation times, numerous solutions have been proposed aimed at reducing the execution time of the alignment step. This is done either by accelerating alignment itself using hardware accelerators and heuristics or by reducing the amount of input data through the use of pre-alignment filters. The algorithms associated with the latter solution are less computationally intensive than DP-based alignment, which reduces the end-to-end alignment time.

Currently, pre-alignment filters are effective to the point where the alignment bottleneck is shifted to the filtering step. Therefore, the filters are accelerated on hardware solutions such as GPUs and FPGAs. While these solutions show orders of magnitude improvement in execution times, they are insufficient for removing the filtering bottleneck entirely. The performance of these hardware accelerators is limited by the rate at which data can be supplied. As a solution, we propose a CIM-based accelerator to reduce data-movement overheads between the host device and the accelerator. Additionally, this architecture makes use of emerging non-volatile memories to perform Boolean operations directly within its memory elements. In doing so, it can exploit parallelism in the algorithms to achieve higher throughput.

In this work, we explore commonly found operations in existing pre-alignment filters and devise ways to implement them on the CIM-architecture. The proposed architecture is flexible in supporting multiple pre-alignment filters and a wide range of input data. The functionality of the architecture is verified through simulation and its effectiveness is tested using real data sets.

Using this architecture, we can achieve improvement in end-to-end execution time over the state of the art ranging from 7.2x to 119.6x for the evaluated data sets, while also achieving a reduction of up to 59% and 79.7% in chip-area and power consumption, respectively.

Furthermore, the provided work offers a platform for the development of future pre-alignment filtering algorithms to further improve performance.

Acknowledgements

This thesis concludes my time as an Embedded Systems student at TU Delft. After several challenging but ultimately very rewarding months, I am now proud to present my thesis work.

First and foremost, I want to express my gratitude towards Michael Shahroodi and Stephan Wong for their excellent advice and guidance throughout the duration of the thesis. I am very appreciative of your willingness to go out of your way to help whenever needed. Your input has helped me greatly in improving not only this thesis work but also my skills in academic research as a whole.

Also, I would like to thank the other members of the Q&CE research group for providing such a welcoming atmosphere, and for making my time spent working on the thesis all the more enjoyable.

Last but not least, I am incredibly grateful to my family and friends for their overwhelming love and support throughout my years of study. The great experience I have had during my time at TU Delft is in no small part thanks to you.

Michael Miao
Delft, April 2023

Contents

List of Figures

List of Tables

List of Abbreviations

A Adenine
ADC Analog-Digital Converter
ALU Arithmetic Logic Unit

BG Bank-Group
bp Base-Pair
BV Bit-Vector

C Cytosine
CAM Content Addressable Memory
CCS Circular Consensus Sequence mode
CIM Computation In Memory
CIM-A Computation In Memory Array
CIM-P Computation In Memory Periphery
CLR Continuous Long Read mode
CMOS Complimentary Metal Oxide Semiconductor
CNM Computation Near Memory
CPU Central Processing Unit

DAC Digital-Analog Converter
DEMUX De-multiplexer
DIMM Dual In-line Memory Module
DNA DeoxyriboNucleic Acid
DRAM Dynamic Random Access Memory

eNVM Emerging Non-Volatile Memory

FIFO First-In-First-Out
FN False Negative
FP False Positive
FPGA Field Programmable Gate Array
FSM Finite-State Machine

G Guanine
GPU Graphics Processing Unit

HBM High-Bandwidth Memory
HM Hamming Mask
HRS High Resistance State
HRT Horizontal Routing Track
HW Hardware

I/O Input/Output

LRS Low Resistance State
LSB Least-Significant Bit

MAGIC Memristor Aided loGIC
MSB Most-Significant Bit
MTJ Magnetic Tunnel Junction
MUX Multiplexer

NW Needleman-Wunsch algorithm

ONT Oxford Nanopore Technology
OS-TCAM Output-Select Ternary Content Addressable Memory

PacBio Pacific Biosciences
PCIe Peripheral Component Interconnect express
PCM-RAM Phase-Change Memory RAM
PD-TCAM Pattern-Detect Ternary Content Addressable Memory

RAM Random Access Memory
ReRAM Resistive RAM
RRAM Resistive RAM
RTL Register Transfer level

SA Sense Amplifier
SA Sub-Array
SBS Sequencing By Synthesis
SHD Shifted Hamming Distance
SHM Shifted Hamming Mask
SIMD Single-Input-Multiple-Data
SMRT Single-Molecule Real-Time sequencing
SNR Signal-Net Routing
SRAM Static Random Access Memory
SRS Speculative Removal of Short-matches
SS SneakySnake
STT-MRAM Spin-Transfer Torque Magnetic RAM
SW Software
SW Smith-Waterman algorithm

T Thymine
TCAM Ternary Content Addressable Memory
TN True Negative
TP True Positive
TSV Though-Silicon Via

VLSI Very Large Scale Integration
VRT Vertical Routing Track

WB Write Buffer
WPB Writes-Per-Bank

XBAR Crossbar

Chapter 1

Introduction

1.1. Motivation

Over the past decades, DNA sequencing has been used in an increasingly wide range of applications such as healthcare and forensics [17–19]. With recent advancements in DNA sequencing machines, sequencing has become more and more affordable and thereby more accessible [20, 21]. Consequently, there has been a massive increase in the amount of data available for processing [22]. To deal with the sheer amounts of data produced, the processing is often offloaded to large compute clusters [23, 24]. Even then, the computation can take a long time and can use a lot of energy. Because of this, numerous efforts are made to reduce the computation time and simultaneously increase energy efficiency.

One of the major bottlenecks in converting raw sequencing data into user-intelligible information is the alignment stage [25–27]. Sequencing machines generally cut up samples of DNA into small pieces called reads, which are translated to a computer-interpretable sequence of bases. During the alignment stage, the obtained DNA read is compared to various sub-sequences of a reference genome using dynamic programming algorithms such as the Needleman-Wunsch algorithm or Smith-Waterman algorithm [28, 29]. These algorithms are computationally intensive as their complexity scales quadratically with the length of the reads [26]. There are two main research directions aimed at alleviating this bottleneck. The first of these methods is to accelerate the alignment process itself. Research in this field includes mapping the alignment algorithms onto accelerators such as GPUs or FPGAs, achieving improvements in execution time by exploiting parallelism [30–32]. The other method of acceleration is pre-alignment filtering; a technique that uses heuristics to remove obviously-wrong matchings, thereby reducing the number of sequence pairs that need to be aligned. Several such filters have been put forward in prior works, showing orders of magnitude improvement in terms of execution time [6–10, 33]. Research in both methods is orthogonal, and can therefore be combined to achieve even greater results. The focus of this work is on pre-alignment filtering.

With the current state-of-the-art, pre-alignment filtering – in certain use cases – manages to reduce the alignment execution time to the point where the bottleneck is shifted to the filtering process itself [6–10, 34]. This causes a need for an accelerated pre-alignment filter. Previous works have implemented such accelerators on GPU and FPGAs, which manage to accelerate the filtering by orders of magnitude [7–10, 34]. However, the performance of these accelerators is limited by the rate at which data can be supplied to them [8, 10]. Additionally, this data movement between host and accelerator causes a large portion of the total energy consumption due to the energy-hungry busses over which the data is transmitted [8, 10].

Promising avenues for improvement of pre-alignment filtering are Computation Near Memory (CNM) and Computation In Memory (CIM). These paradigms do not employ a traditional Von Neumann architecture, but instead perform computation where the data resides, eliminating the need for data transfer. The use of such architectures is currently widely researched in fields such as artificial intelligence for the acceleration of neural networks, and other stages of the DNA sequence processing pipeline [35–37]. These kinds of applications are particularly suitable because of the relatively small amount of computations performed on large amounts of data, and the use of primarily simple instructions. In these aspects, pre-alignment filtering algorithms share the same properties, and could potentially also benefit from similar architectures.

In addition to reducing data-movement overheads, CIM provides the possibility to perform Boolean operations directly in the memory elements themselves [38, 39]. Most notably, this is explored in the field of Emerging Non-Volatile Memory technologies (eNVM) [13, 14, 37]. Unlike conventional memories such as

DRAM, SRAM, and flash, which store data as an electric charge, eNVMs store data as a resistance state [40]. This property can be leveraged to perform logic operations without needing to read the memory contents of the operands. By applying carefully selected voltages to certain memory elements, the resulting current contains information on the data stored in the memory cells.

As the industry reaches the limits of conventional memories in terms of density scaling [41, 42], the viability of eNVMs as an alternative is being explored, as they offer potential benefits such as greater densities, access speeds, and non-volatility [43–45]. Provided that the overheads added by implementing a CIM architecture on these memories are small, accelerators could be implemented at a relatively small cost.

Combining the need to reduce the alignment bottleneck, and the prospect of acceleration using CIM architectures, this thesis explores the possibility of mapping pre-alignment filtering algorithms onto CIM architectures.

1.2. Problem statement

The research in this thesis is aimed at answering the following research question:

How can pre-alignment filtering algorithms be implemented on a CIM-architecture, and what are the benefits of doing so in terms of area, power, and performance compared to the current state-of-the-art?

The main research question can be broken down into the following sub-questions:

- What operations are commonly found in pre-alignment algorithms?
- How can these shared operations be accelerated within a CIM architecture using eNVMs?
- What design considerations and trade-offs need to be made to efficiently map common operations found in pre-alignment algorithms on a CIM architecture?
- What improvements can be achieved using the proposed architecture compared to the state-of-the-art?

1.3. Research goals

To provide an answer to these questions, the project has to reach the following goals:

- Creation of a fair comparison platform for existing pre-alignment filtering algorithms.
- Identification of common operations shared amongst pre-alignment filtering algorithms.
- Creation of an architecture to support the common operations, with minimal overhead w.r.t. a conventional memory architecture.
- Mapping of multiple algorithms onto the same proposed architecture, with support for a wide range of input datasets and edit-distances.
- Evaluation of the proposed architecture and comparison with the state-of-the-art in terms of performance, power, and area.
- Providing considerations for future works concerning CIM and/or pre-alignment filtering.

1.4. Methodology

To achieve the research goals, the following 5-step methodology is employed. First, existing works are benchmarked and profiled using the same datasets to provide a fair platform for comparison, and to establish a baseline for comparison to the proposed design. From the profiling, we identify common operations shared by the pre-alignment algorithms. Next, a low-level architecture is devised which supports the common operations. Building on this, we propose a high-level architecture that maps the algorithms onto the architecture and takes care of the distribution of input data and the processing of output data. Combining these two components, an analytical model is built that evaluates the proposed architecture in metrics such as the ex-

pected execution time, memory capacity, and power for various hardware configurations. Following this, the architecture is implemented using VHDL, and its functional behavior is verified through simulation. Furthermore, the accuracy of the CIM implementation of the algorithms is compared to their software counterparts. The digital components of the design are then quantitatively evaluated through synthesis, and the analog components are evaluated by building on prior works. Combining the quantitative evaluation results of the analog and digital parts of the design, the optimal parameters for the design are found with design-space exploration. Finally, the proposed architecture is compared to state-of-the-art accelerators.

1.5. Thesis overview

The remainder of the thesis is structured as follows. In Chapter 2, background information is provided, giving an overview of the DNA sequencing pipeline, CIM techniques, and eNVMS. Following this, Chapter 3, provides a summary of existing works and presents the benchmarking/profiling results. Also, the identification of the common operations is discussed. In Chapter 4 different CIM-logic styles are discussed, exploring their capabilities and limitations. Also, existing pre-alignment algorithms are adjusted to account for the limitations of the chosen CIM-logic style. In Chapter 5 a complete architecture is proposed which supports the common operations, and two of the evaluated algorithms are mapped to the architecture. Chapter 6 will go over the implementation details of the architecture, as well as an analytical model that characterizes its resource requirements and performance. This is followed by an evaluation of the achieved results and a comparison with the state-of-the-art in Chapter 7. Finally, a conclusion is drawn in Chapter 8, discussing the main findings and contributions of the thesis and providing research directions for future works.

Chapter 2

Background Information

This chapter is aimed at providing background information to help understand the purpose and contribution of this thesis project. First, we provide a wider context on the purpose of DNA sequencing in Section 2.1. Secondly, a brief summary of the main steps of the DNA-sequence analysis pipeline is provided in Sections 2.2 to 2.5, giving an overview of what steps a DNA sequence needs to go through to transform raw sequencing data into useful information. This is followed by an introduction to Computation In Memory (CIM) in Section 2.6, which explains the concept of CIM, the differences with conventional computing methods, and how it can be used for the project at hand. Here we also touch upon emerging memory technologies (eNVM), and introduce the concept of a memristor, and how its properties can be leveraged to act as a memory or processing element. Additionally, we present the most prevalent implementations of this technology. Finally, we provide a conclusion in Section 2.7, summarizing the main findings of this chapter.

2.1. DNA

This section briefly explains what DNA sequencing entails and what it can be used for. It also presents the main steps required to transform a sample of DNA into user-interpretable information.

DNA, or DeoxyriboNucleic Acid, is a biochemical molecule that acts as the main carrier of genetic information of living organisms and viruses [46]. DNA is present in the cells of an organism and is stored as a multitude of separate stands. A single strand of DNA consists of two chains, each containing many millions of nucleotides, which are intertwined in the characteristic double helix structure. A nucleotide is made up of one of four nitrogen bases; Adenine (A), Cytosine (C), Guanine (G), or Thymine (T). The chemical structure of these nitrogen bases makes it so that an A-base attaches to a T-base, forming a base pair. The same holds for C-bases and G-bases. This property makes it so that the two chains of the DNA are complementary to each other: if one strand contains an A-base, the other strand will contain a T, and vice versa [47].

These chains of nucleotides are interesting to examine as they provide information on the holder of the DNA. The sequence in which base pairs occur within the DNA strands is unique for every individual organism and is called the genotype. Certain parts of the genotype (genes) can affect the cell processes of the organism, which ultimately determine properties such as the appearance, development, and behavior of the organism, also known as the phenotype. Small differences in the genotype can lead to huge differences in the phenotype of an organism. For example, 99.9% percent of the DNA of any pair of unrelated humans is identical, while they might look and behave completely differently [48].

It is exactly these differences that make it interesting to examine the DNA, as this information allows us to draw conclusions on where a DNA sample originates from, and what biological properties the holder might possess. For instance, since DNA is unique per individual, we can compare two samples of DNA to determine whether they originate from the same source [48]. This is used in forensics [49], for instance, where DNA collected at the scene of a crime is compared to that of a suspect to determine whether they might have been present when the crime took place. Another way in which DNA sequencing can be used is to compare the DNA of one individual to that of many others. For example, in healthcare, the DNA of a patient can be examined to determine the likelihood of them developing a certain disease [50]. If the DNA of the patient contains a pattern in the DNA that is commonly found in patients of a certain illness, but not in healthy individuals, it might suggest that the patient has an increased probability of developing the same disease. In recent years, data from many different people have been studied, and while DNA is unique per individual, it is for a large

part identical for every human. This allows us to compile a baseline DNA, called the human genome, from which the DNA of an individual differs only slightly [51, 52]. Using this genome, we can determine what parts of the DNA determine certain character traits by correlating the genotype and phenotype of many samples.

There are several steps involved in transforming a raw DNA sample into useful information, each introducing challenges along the way. An overview of this pipeline can be found in Figure 2.1. Here, the processes marked in green are the main focus of this thesis work. In the following section we present each step and the most prevalent techniques in each step. We also touch on the sequencing step, as this has implications for the input data to the system.

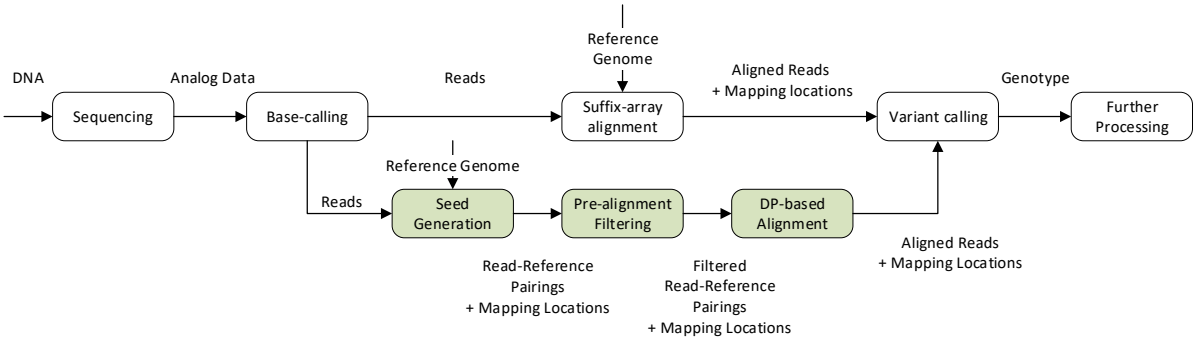


Figure 2.1: An overview of the sequencing pipeline from raw DNA samples to useful information.

2.2. Sequencing

The first step in extracting information from DNA is the sequencing of the DNA sample. DNA sequencing is the process of reading the order of the nucleotides within the DNA and translating it to a computer-interpretable form. This is performed using so-called sequencing machines which use various different methods for obtaining the order of the nucleotides, each with its own advantages and disadvantages. While the underlying physical principles used for obtaining the sequences are different, one commonality is that they all split the sampled DNA into shorter fragments called reads.

Here, we will discuss the basic principles of obtaining reads for the most widely adopted sequencing technologies and promising recently developed technologies. This helps understand what kinds of datasets are examined in the later stages of the genome analysis, and what factors should be taken into account.

2.2.1. Illumina

Illumina [53] is currently the most widely adopted sequencing technology as around 80% of all sequencing data is obtained using this technology [54]. It is classified as a second-generation sequencing technology that makes use of so-called Sequencing By Synthesis (SBS). In this process, the double-stranded reads of the input DNA are split into two complementary strands. These single-stranded reads are then attached to a chip called a flow cell with a so-called adapter. Then, the reads are cloned through a process called bridge amplification, such that clusters of the same read are formed. Following that, primers are added to the flow cell, which attaches to the reads. Also, a solution containing free nucleotides is applied. These nucleotides are chemically altered to be fluorescent, such that each of the 4 types of nucleotide lights up with a different wavelength when exposed to a laser. Through a biological process, the primer can attach a single complementary nucleotide to the read. For example, if the read starts with an A-base, the primer will attach a fluorescent T-base to it. After each cycle, the unattached bases are washed away, and a laser is applied. The wavelength of the light emitted by the newly attached fluorescent base is identified by a camera to determine what base was added to the strand. This process is repeated several times to obtain the full sequence of base pairs of the original read. The whole process of obtaining Illumina reads is illustrated in Figure 2.2.

However, during this process, there are several points of failure. Sometimes the primer is unable to attach a free nucleotide to the read during a cycle. In this case, when the laser is applied, the previous base pair is detected instead. Conversely, it is also possible that more than one base is added during a cycle, thereby effectively skipping a base, in which case the wavelength of the next base in the sequence is detected. The effects of these defects are diminished by the fact that the read is cloned multiple times, such that the majority

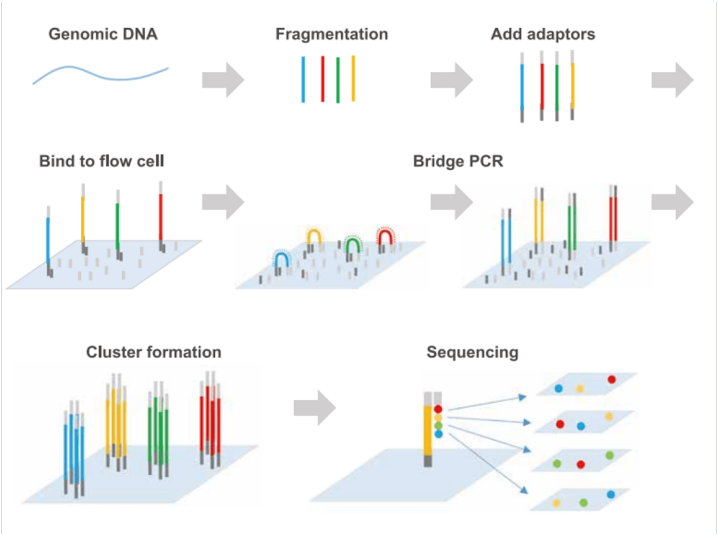


Figure 2.2: Steps involved in sequencing using Illumina. Adapted from [2].

of the reads still emit the proper wavelength. The correct base can then be determined in a post-processing step called base-calling, which converts the analog inputs into digital sequences of base-pairs. However, since every cycle builds upon the same read, the errors accumulate to the point where the number of errors is so large that no consensus can be called with confidence. This makes it so that Illumina reads are highly accurate (>99%) for producing short reads [55], but unsuitable for producing long reads. For this reason, Illumina reads are generally limited to a length of around 50 to 300 base pairs (bps) [56].

2.2.2. Single-Molecule Real-Time Sequencing (SMRT/PacBio)

The second most widely adopted sequencing technology is the third-generation technology Single-Molecule Real-Time sequencing (SMRT) by Pacific Biosciences (PacBio) [57]. This method, like Illumina, makes use of Sequencing by Synthesis. Before samples are added to the machine, adapters are added to the reads. These are different from the ones used in Illumina in that they split the original read but keep the two parts together in a circular structure, as can be seen in Figure 2.3.

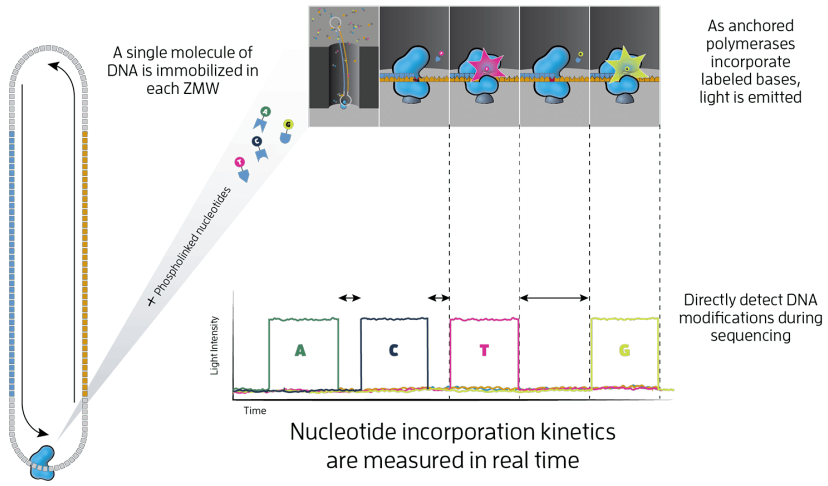


Figure 2.3: Sequencing using PacBio. Adapted from [3].

At its core, SMRT has a chip with small wells called zero-mode waveguides. Rather than directly attaching one end of the read to the chip, the circle is attached to a DNA-polymerase molecule which resides at the bottom of each well. Here, the polymerase attaches a fluorescent free nucleotide to the circular strand, which illuminates with a different wavelength depending on what nucleotide is attached. Since the nucleotides are

attached at the bottom of the wells, and the small width of these wells, only the wavelength of the attached cell is detected. This eliminates the need for washing away the remaining nucleotides, as is the case for Illumina. Also, this allows the reads to be processed in real-time. The reads processed in this way have an accuracy of around 80%, which is lower than Illumina, partially because there are no cloned sequences to average out with. To improve accuracy, the same read is fed through the polymerase multiple times, providing multiple readings collected over time. Then, base-calling is performed on the captured data to achieve accuracies over 99%. This Circular Consensus Sequence mode (CCS) is, however, limited in the size of the reads it is able to produce (~3000bps) [56,58]. Therefore, the user can also opt to forego this procedure by using the Continuous Long Read mode (CLR), which produces reads of up to 100kbps, at a lower accuracy (~80%) [56, 58].

2.2.3. Oxford Nanopore Technologies

Lastly, we discuss Oxford Nanopore Technology (ONT) [59], which is a relatively new sequencing technology, that operates in a fundamentally different way as compared to the two previously discussed. At its core, ONT uses nanopores, which are proteins normally found in the membranes of cells. These normally act as a gateway for molecules to enter and leave the cell. In ONT these nanopores are placed on an electrically resistive synthetic polymer. An electrical potential is applied over the nanopore, resulting in a flow of ions through it. Any molecule passing through the narrowest part of the nanopore will disrupt this ionic current. This disruption rate of flow of this ionic current can be measured to determine what passed through the molecule, as is shown in Figure 2.4.

For DNA sequencing, an enzyme is attached to a DNA sequence, which gradually feeds the DNA strand through the nanopore. Since every type of nucleotide has a different size, the disruption of the ionic current varies according to which nucleotide is present at its narrowest part. This causes different ionic currents through the nanopore over time, from which the sequence of nucleotides in the read can be determined. However, since the changes in current are small, this method is prone to errors due to noise, producing an accuracy of ~98%. A benefit of this approach is that there is no deterioration of the accuracy of the readout, which allows reads to be very long (>100kbps) [56].

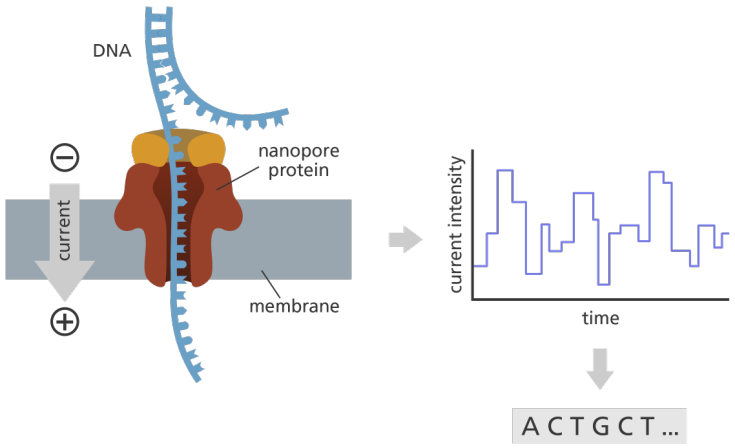


Figure 2.4: Sequencing using ONT. Adapted from [4].

2.2.4. Evaluation of Sequencing Technologies

Comparing these three sequencing technologies, we can distinguish two defining features of the reads they produce: accuracy and read length. The accuracy of a sequencing machine is described as the percentage of base pairs it has correctly extracted from a DNA sample, and the read length as the number of base pairs that constitute a read.

Since the aim of sequencing is comparing samples of DNA to find differences/similarities between them, it is important to be able to differentiate between actual differences between the samples and sequencing errors. Sequencing errors are defined as differences between the extracted read and the DNA sample that originate from deficiencies in the sequencing technologies. For this reason, it is favorable to have highly accurate reads.

As for read length, it is mostly favored to have reads that are as long as possible. This is for two main reasons. The first of these reasons is for the reconstruction of the DNA. To reconstruct the original DNA sequence from separate reads, several reads need to be overlapped to find out what reads come from what parts of the DNA in a process called read mapping. Like fitting pieces of a puzzle together, it is easier to do this with fewer long reads than many short reads. Secondly, the probability of a short read aligning with multiple parts of a reference genome is much higher than is the case with long reads. This makes finding the source of a mutation in the DNA much harder with short reads.

For these reasons, the industry is moving towards long-read sequencing. However, most currently available data is produced by Illumina sequencing machines and is thus in the form of short reads. This motivates us to find solutions for analysis tools that can support short reads as well as long reads. We summarise the read length, accuracy, and amount of available data of the respective technologies in Table 2.1.

Table 2.1: Comparison between the most widely used sequencing technologies

Sequencing technology	Max. Read-length	Accuracy	Share of available data (2020) [54]
Illumina	350bps	>99%	~82%
PacBio (CCS)	30kbps	>99%	~10%
PacBio (CLR)	>100kbps	~80%	
Oxford Nanopore	>100kbps	~98%	~3%

2.3. Read mapping

After sequencing and base-calling, the third step toward alignment is the mapping phase. To determine where a read originates from, it needs to be compared to every possible location of the reference genome. Due to the sheer size of the reference genome (>3 billion bps for the human genome), this would be a computationally prohibitively intensive task. To add to that, the reads might contain edits that further complicate this process. Edits are defined as the differences between two strings. In genomics, these refer to the differences between the read and reference sequence and can be classified into three categories: substitutions, insertions, and deletions. We speak of substitutions when the base at a certain position of the read sequence is different from the one found at the same position in the reference sequence. Insertions refer to the case where the read contains a base pair that is not found in the reference sequence. Conversely, a deletion is when the read does not contain a base pair that is found in the reference. Insertions and deletions are often referred to under the collective term indel.

Because of the computationally intensive nature of mapping, methods have been developed which reduce the search space in the reference genome [33]. Solutions that implement this can be classified into two main categories: suffix-array and seed-and-extend methods [33].

2.3.1. Suffix-array

Another method of mapping the read to the reference is the suffix-array method [60, 61]. The reference genome is pre-processed by creating an index in the shape of a suffix tree. In this tree, every node has 4 outgoing edges, each representing one of the four types of base pairs. Each node represents a list of locations in the reference genome. If we traverse the tree from root to leaf and concatenate all of the bases represented by the edges, we create a k-mer (in which 'k' depends on the depth of the tree). The leaf we end up at contains all locations in the reference genome at which this k-mer is present.

During runtime, the location of a read sequence can be found by traversing the tree by following the path indicated by subsequent bases of the read sequence. We then find all locations at which the read can be found in the reference genome in the list at the eventual leaf. An example of this is provided in Figure 2.5 where a 4-mer is mapped onto a reference of 14 bps. When traversing the tree according to the bases in the given example, two mapping locations are found at the 1st and the 9th base of the reference, which corresponds to the searched pattern.

Actual read-mappers of this type make use of the Burrows-Wheeler Transform [61] and the Ferragina-Manzini index [62] to implement the suffix tree for better memory efficiency while maintaining a similar underlying principle.

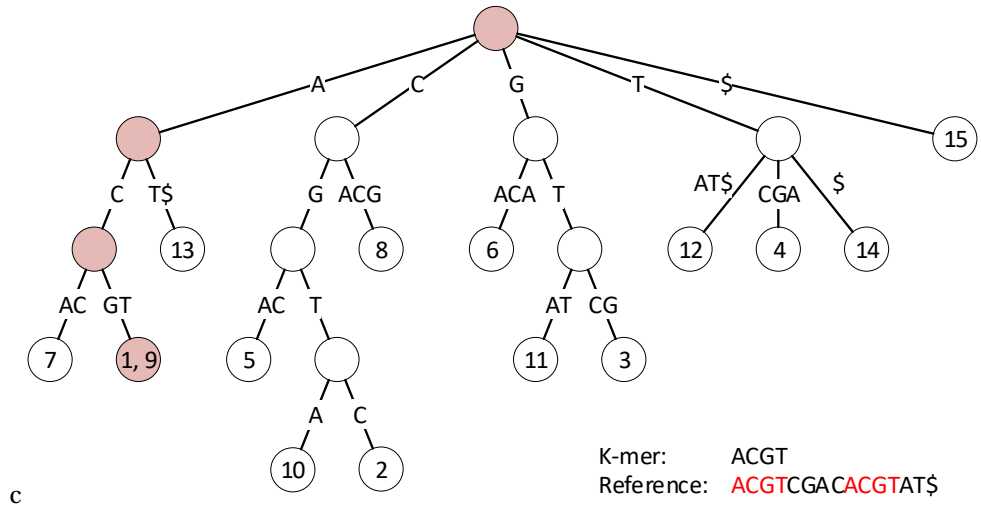


Figure 2.5: Example of a suffix tree for the mapping of a 4-mer onto a small reference of 14 bps.

2.3.2. Seed-and-Extend

The seed-and-extend method aims at reducing the search space by leveraging the fact that in order for the read to be similar to the reference at a certain location (i.e., have few edits), they must share smaller sections (k-mers, where 'k' indicates the number of bases in the segment) that are identical [33]. Such a matching location is called a seed. The locations of seeds in the reference genome can be found by pre-processing the reference genome. In this process, all locations at which a certain k-mer occurs, are looked up and saved. These locations are generally stored in a hash table in which the k-mer is used as an index. We can then take a k-mer from the read sequence and index the hash table to find all possible mapping locations. This thus reduces the search space from the entire genome to only the bases surrounding the seed. The seeding process is illustrated in Figure 2.6.

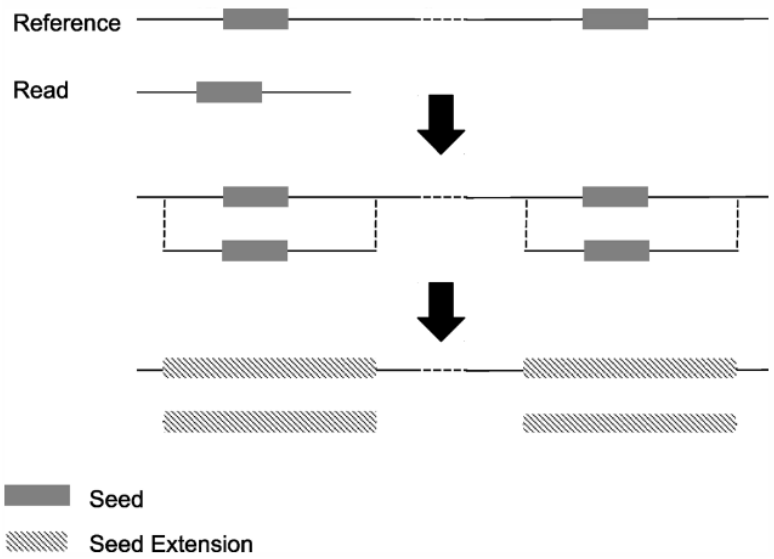


Figure 2.6: Illustration of the seed-and-extend process. Adapted from [5].

After finding all seeds associated with a read, the seed is extended to the bases in the reference surrounding the seed. In this process, we find the exact mapping of the read onto the reference locations, including insertions, deletions, and substitutions by means of alignment. Alignment is performed using dynamic-programming algorithms such as the Needleman-Wunsch algorithm (NW) or Smith-Waterman algorithm (SW). We will dive into these algorithms in more detail in the following section.

2.3.3. Evaluation of mapping techniques

While both methods fulfill the same purpose, they have distinct advantages and disadvantages. For one, the suffix-array-based mappers are generally faster than their seed-and-extend counterparts [33]. This is mainly due to the computationally intensive alignment step in seed-and-extend mappers. On the other hand, seed-and-extend-based mappers have better sensitivity when the read contains errors. As long as any seed of the read does not contain any errors, the read will still be mapped correctly to the reference. Contrast that to suffix-array mappers, where an error in the read leads to completely different paths in the suffix-tree, and therefore wrong mappings [33]. Due to the latter advantage, the focus of this work is on seed-and-extend-based mappers.

2.4. Sequence Alignment

After the seeding phase using a seed-and-extend-based mapper, the second step of seed-and-extend is alignment. Given a pairing of a read and a reference sequence obtained through seeding, alignment aims to find the most likely location of where the read has originated from while taking edits into account. If the number of found edits exceeds a user-defined threshold, the read-reference pair is deemed invalid, as the differences between the two sequences are too great. As mentioned before, the exact mapping of the read to the reference is achieved with dynamic programming algorithms, which form the main bottleneck in the mapping process. As examples of this, we present the NW and SW algorithms, explaining how they work and why they are so computationally intensive.

2.4.1. Needleman-Wunsch Alignment

The Needleman-Wunsch algorithm (NW) [28] is used for finding global alignments. This means finding the optimal alignment of the entire read and the reference sequence. This algorithm is mainly used when the read and the reference sequence have the same length and have relatively few edits. It is a dynamic programming algorithm, which means that a large problem is split into smaller sub-problems that are solved recursively, by applying the notion of divide-and-conquer. The NW algorithm consists of two major steps: grid construction and back-tracing.

Grid Construction

During grid construction, all possible alignments are evaluated based on a scoring system. First, the algorithm creates an MxN matrix where the dimensions correspond to the length of the read and the reference, respectively. The row and column headers of this so-called similarity grid are filled with the bases of the read and the reference respectively. The grid is then initialized as can be seen in Figure 2.7(a), when using the example given in Table 2.7. Following this, starting from the top left corner, the bases of the read and reference are compared one by one, and their matching is assigned a score. The scoring could for example be +1 for a match (read and reference have the same base), and -1 for a mismatch (substitution or indel). The scores assigned to matches and mismatches can be altered depending on how much importance is associated with a substitution, insertion, or deletion. This score is added to the three adjacent cells, and the lowest value is taken as the local alignment score for the evaluated cell. This step is evaluated cell-by-cell, row-by-row until the final score, or global alignment score is found at the bottom-right corner, as is shown in Figure 2.7(b). If the alignment score is high enough (the number of edits is below the edit-distance threshold), we proceed to the back-tracing step. The edit-distance is a metric that describes the similarity of two strings and is defined as the minimum number of edits required to turn one string into the other.

The construction of the grid has dependencies on the previous iterations of the algorithms and therefore creates difficulties in extracting parallelism.

Table 2.2: Example read-reference pairing and the correct alignment.

Read Sequence	ACGTTGTCTG
Reference Sequence	ACGGTTTCGA
Correct Alignment	ACG-TTGTCTG- ||| || || | ACGGTT-TC-GA

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

(a)

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

(b)

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

(c)

Figure 2.7: Example of the Needleman-Wunsch algorithm aligning the read-reference pairing in Table 2.2. Alignment happens in three steps: (a) grid initialization, (b) grid construction, and (c) back-tracing.

Back-tracing
Using the filled grid, the back-tracing step is performed starting at the bottom-right cell of the similarity grid. The cells to the top, left, and top left of this cell are compared, and the cell with the highest score is considered to be part of the trace, as demonstrated in Figure 2.7(c). Then, this step is repeated for every cell that is added to the trace, until the cell at the top left has been reached. A diagonal trace suggests that there was a match, and an upward or leftward trace denotes that there has been an insertion or deletion, respectively. The direction that is chosen at every step determines the optimal alignment.

2.4.2. Smith-Waterman Alignment
The Smith-Waterman algorithm (SW) [29] is a variation of the Needleman-Wunsch algorithm that performs local alignment. Unlike global alignment, which searches for the optimal alignment of the entire sequence, local alignment aims at finding similar segments within the two sequences. Like NW, SW first creates a similarity grid. It follows the same procedure for filling the cells as NW, but any negative cells are given a score of 0. During the back-tracing step, the algorithm starts at the cell with the highest local alignment score, rather than in the bottom left cell. The rest of the trace-back procedure is the same as in NW, but it ends when the first 0 has been encountered. The segment over which the trace-back happens indicates the most similar region of the two sequences.

2.4.3. Acceleration of Alignment
Using either of these algorithms, the number of cells that need to be evaluated scales quadratically ($O(mn)$) with the length of the read/reference [7–10, 33]. Therefore, the alignment becomes increasingly more computationally demanding as we increase the read length. With the industry moving towards sequencing technologies with increasingly long reads, alignment becomes a major bottleneck in the processing pipeline. Therefore, the acceleration of the alignment step is a widely studied field of research.

Numerous studies have been conducted that aim at improving the execution time of the alignment algorithms themselves. This includes acceleration implementing the algorithms on dedicated hardware such as GPUs or FPGAs [63, 64], which improves performance by exploiting parallelism. Another way is using heuristics to approximate alignment [65], where accuracy is sacrificed for improved performance.

2.5. Pre-alignment Filtering
Besides accelerating alignment itself, a different approach is pre-alignment filtering [7–10, 34]. Here, the aim is to reduce the number of pairings that need to be evaluated by alignment. This is achieved by approximating the edit-distance between the read and reference and removing pairings with an edit-distance that greatly exceeds the alignment threshold, as is demonstrated in Figure 2.8. This is effective as most pairings generated by the seeding process are dissimilar (contain many edits), while there is only one accepted mapping [6–10, 34].

The edit-distance estimation of pre-alignment filters can be performed using less computationally demanding algorithms, thereby decreasing the end-to-end execution time of alignment (i.e., filtering time + alignment time). Research in this method of improving alignment times is orthogonal to the acceleration of alignment algorithms. This means that improvements to one can be combined with those of the other to achieve an even greater benefit.

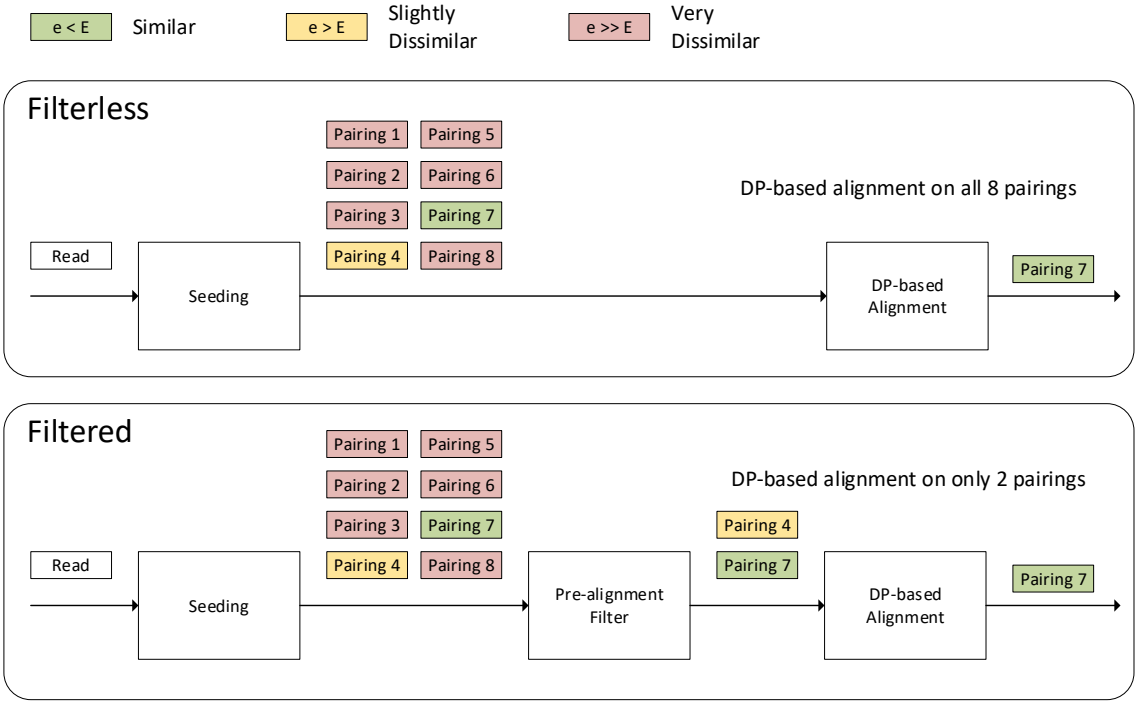


Figure 2.8: Demonstration of the concept of pre-alignment filtering.

Numerous works have been published in the field of pre-alignment filtering [6–10,34]. With the current state-of-the-art, pre-alignment filtering can achieve orders of magnitude improvements to alignment times by filtering out the majority of wrong mappings.

2.5.1. Evaluation of Pre-alignment Filters

The effectiveness of pre-alignment filters is determined by two main factors that influence the end-to-end alignment time: filtering accuracy and execution time.

Filtering Accuracy

The accuracy of the algorithm is defined by its ability to distinguish wrong mappings (negative mappings) from correct mappings (positive mappings). The accuracy can be defined by four metrics: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). These metrics are defined as follows:

- **True Positive**: mappings that have an edit-distance **lower** than the edit-threshold, and are **accepted** by the pre-alignment filter.
- **True Negative**: mappings that have an edit-distance **higher** than the edit-threshold, and are **rejected** by the pre-alignment filter.
- **False Positive**: mappings that have an edit-distance **higher** than the edit-threshold, but are **accepted** by the pre-alignment filter.
- **False Negative**: mappings that have an edit-distance **lower** than the edit-threshold, but are **rejected** by the pre-alignment filter.

Here the edit-distance is defined by the number of edits found by alignment with the Needleman-Wunsch algorithm. An ideal pre-alignment filter has a false-positive rate and a false-negative rate equal to 0%. This

means that it does not discard any mappings that would result in a correct alignment, and that any mappings that do pass the filter will also be aligned within the edit-threshold. However, this has not been achieved in prior works. Most prior works target a zero false negative rate, to avoid any loss of information. Therefore, the edit-distance estimation is often on the conservative side, meaning that when the algorithm will never over-estimate the number of edits. Due to this, the filters – in most cases – let through some incorrect mappings, leading to a non-zero false-positive rate. The development of new algorithms is mostly aimed at reducing this false-positive rate, whilst keeping the non-zero false-negative rate.

Filtering Time
Another important metric for evaluating pre-alignment filters is the execution time of the algorithm. Since we add an additional processing step to the pipeline, we must make sure that the reduction in alignment time is greater than the execution time of the filtering step. To minimize the effect on the end-to-end execution time, the pre-alignment filter should take as little time as possible. For this reason, these algorithms are often implemented on hardware accelerators such as FPGAs or GPUs.

However, increased performance often comes as a trade-off against accuracy. As a consequence, an inaccurate filter with a large false-positive could still outperform an accurate filter, provided that its execution time is low enough. We therefore must evaluate the effectiveness of the filter by its end-to-end execution time.

In the remainder of this section, we present the main concepts used in current state-of-the-art pre-alignment filters, going into more detail for those that are implemented in this thesis work (Shifted Hamming Distance (SHD) [14] and SneakySnake [10]). We discuss these works in order of publication.

2.5.2. Shifted Hamming Distance

The first of the discussed works is Shifted Hamming Distance (SHD) [6]. This algorithm is developed for seed-and-extend mappers and specifically designed with short reads in mind, which are the same length as the reference sequence (i.e., paired-end). All operations involved in the algorithm are chosen such that they can be efficiently accelerated using SIMD instructions. The algorithm is based on two main observations.

1. If two strings differ by e edits, then all non-erroneous characters of the strings can be aligned in at most e shifts.

2. If two strings differ by e edits, then they share at most $e+1$ identical sections.

These observations are incorporated in the algorithm, which exists of two main parts: Shifted Hamming Mask (SHM) and Speculative Removal of Short-matches (SRS), which are illustrated in Figure 2.9. The first step is to create the shifted Hamming masks, which are based on the first observation. Here the read and the reference sequence are compared base-pair by base-pair using a series of XOR operations. Doing so results in a vector of '0's and '1's, where a '1' indicates a mismatch between the two sequences. Without shifting the sequences, this allows us to detect substitution errors, but not insertions or deletions.

To also detect indels, the reference sequence is compared to shifted versions of the read sequence, with shifts ranging from $-e$ to $+e$, where e is the edit-distance threshold. The left-shifted version of the read will align with parts of the reference after deletion has occurred, while the right-shifted sequences can detect insertions in the same way. This results in $2e+1$ Hamming masks. These intermediate results are then combined to form a final bit-vector using a bit-wise AND operation with all Hamming masks. This way, a '0' at a given position in the reference of any of the Hamming masks will propagate to the final bit vector. In the final bit-vector, the number of mismatches is represented as the number of '1's present in the vector.

However, using this approach, the authors encounter the problem that not all '0's come from actual matches in between the reference and the read. Since there are only 4 types of base pairs, there is a probability of $\sim 25\%$ that a base of the read matches with the reference due to random chance. This type of '0's are referred to as spurious zeros. The authors make the observation that the probability of the occurrence of a sequence of spurious zeros decreases exponentially with the length of these short sequences. The probability that a sequence of zeros greater than n bits is spurious is calculated as 0.25^n. It is found that any segments shorter with fewer than 3 subsequent zeros have a considerable probability of being caused by spurious matches ($\sim 31\%$). For this reason, SHD removes these short sequences from the Hamming mask using SRS. SRS checks

for every possible segment of 4 base pairs within the Hamming mask if it is either "1001" or "101", which indicates the presence of a short sequence of '0's. If the pattern is detected, the segment is amended to "1111" or "111", respectively. In this way, the short sequences are not propagated to the final bit vector.

However, SRS introduces problems of its own when two edits occur within 3 base pairs of each other. In this case, SRS will amend the '0's while it should not. This can cause more '1's to be present in the final bit-vector than expected. Therefore, to avoid having false-negatives, the number of edits is counted conservatively. In the final bit-vector, if any sequence of "1111" or "111" is detected, this could be caused by false amendments. Therefore, even though more '1's are detected, they are only counted as 2.

The whole process of SHD is illustrated in Figure 2.9, and the pseudo-code is provided in Algorithm 1.

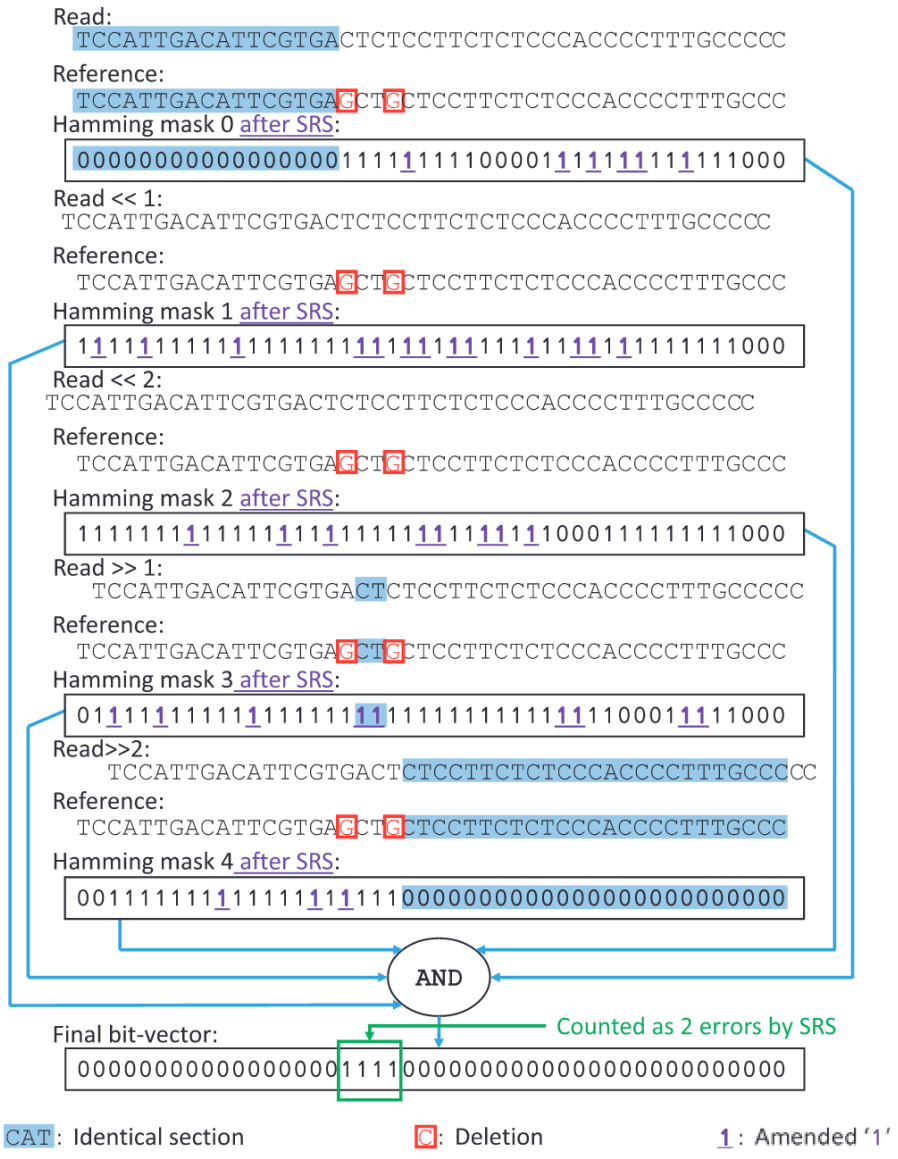


Figure 2.9: Visualization of the SHD algorithm. Adapted from [6].

Using SHD, a low false-positive rate is achieved for small edit-distance thresholds, while maintaining a zero-false-negative rate. Because of this, several magnitudes of improvement to end-to-end execution time can be achieved compared to alignment without any pre-alignment filter, and without the loss of correct pairings.

However, this approach to filtering has several disadvantages. While the algorithm has zero false-negatives, its false-positive rate is high compared to other works for input data containing a high number of edits. One

Algorithm 1 Shifted Hamming Distance

Input: Read, Ref, E
Output: Accept

```
function ComputeHammingMask(Seq1, Seq2, L)                          ▷ Bitwise-XOR
    for i ∈ {0 ... L−1} do
        HM[i] ← Seq1[i] == Seq2[i]                                  ▷ Pairwise-OR (HW)
    end for
    return HM
end function

function SRS_amend(HM, L)                                           ▷ Short Pattern Detection
    for i ∈ {0 ... L−3} do
        if HM[i ... i+2] == "101" then
            Amended_HM[i ... i+2] ← "111"
        end if
        if HM[i ... i+3] == "1001" then
            Amended_HM[i ... i+3] ← "1111"
        end if
    end for
    return Amended_HM
end function

function SRS_count(HM, L)
    Count ← 0
    while i < L do
        if HM[4i ... 4i+3] == "0000" then
            Count ← Count
        else if HM[4i ... 4i+3] ∈ {"0101","0110","1001","1010","1011","1101"} then
            Count ← Count + 2
        else
            Count ← Count + 1
        end if
        i ← i + 4
    end while
    return Count
end function

L ← length(Read)
Final_BV ← SRS_amend(ComputeHammingMask(Read, Ref, L))

for i ∈ {1 ...E} do
    for j ∈ {0 ... L−1} do
        ShiftedRead[j] ← Read[j] << i
    end for
    Final_BV ← Final_BV & SRS_amend(ComputeHammingMask(ShiftedRead, Ref, L))   ▷ Bitwise-AND
    for j ∈ {0 ... L−1} do
        ShiftedRead[j] ← Read[j] >> i
    end for
    Final_BV ← Final_BV & SRS_amend(ComputeHammingMask(ShiftedRead, Ref, L))   ▷ Bitwise-AND
end for

edits ← SRS_count(Final_BV)                                         ▷ Multi-operand Addition
return (edits <= E)                                                 ▷ Integer Comparison
```

factor is that, as the permissible edit-distance increases, the number of shifted Hamming masks that are evaluated increases linearly. Thereby, the probability of the occurrence of a sequence of more than 3 spurious '0's increases too. Errors caused in this way are not propagated to the final bit-vector, leading to an underestimation of the edit-distance. Another factor is the conservative counting of errors. As more shifts are evaluated, the probability that edits are located close to each other increases too. SHD then assumes that these errors are caused by amendment, and underestimates the edit-distance.

Despite its disadvantages, the algorithm can still be used in specific use cases. As of the time of writing, the majority of DNA sequence data is in the form of highly accurate short reads, for which SHD maintains a low false-positive rate.

To reduce the execution time of SHD, the bit-wise operations of SHD are accelerated by SIMD operations, which improves its execution time by orders of magnitude. A disadvantage of this is that the number of operations that are able to be performed in parallel is limited to the supported SIMD width of the processor. This limits this solution to short reads only. In another work [34], the same algorithm is accelerated using FPGA, which removes this limitation and further improves the performance of the system.

2.5.3. MAGNET

Another pre-alignment filter is MAGNET [7], which builds on the two observations made in SHD. Like SHD, it creates a set of shifted Hamming masks, but the final bit-vector is calculated differently. MAGNET foregoes the SRS process and instead makes use of the observation that the correct alignment always contains all the longest non-overlapping uninterrupted sub-sequences of zeros. The algorithm is implemented in 4 steps. First, shifted Hamming masks are created by performing a base-pair-wise XOR operation between the reference and the read shifted by $-$ to $+e$ positions. Out of all of these shifted Hamming masks, the longest uninterrupted sequence of '0's is found and propagated to the final bit-vector. Then, the bits to the left and right of this sub-sequence are considered errors and are represented by a '1' in the final bit vector. These bits are excluded from consideration in further steps of the algorithm. These first two steps are repeated for the remainder of the shifted Hamming masks to the left and right of the original longest sequence. The process is repeated recursively until there are no more sub-problems to be solved or segments are smaller than 2 bases. Finally, like SHD, the number of '1's in the final bit-vector is counted to obtain the approximate edit distance between the read and reference sequences. This process is illustrated in Figure 2.10, and the pseudo-code of the algorithm is provided in Algorithm 2.

Figure 2.10: Illustration of the MAGNET algorithm. Adapted from [7].

Compared to SHD, this filter has a much lower false-positive rate, leading to a shorter alignment phase. However, the algorithm is more computationally intensive, meaning that the filtering stage takes more time. Additionally, this filter has a non-zero false-negative rate, which causes some correct mappings to be removed.

This algorithm is accelerated using an FPGA in [9]. Here the base-pair-wise boolean operations are made parallel, and multiple problems are run at the same time. In doing so, the execution time is reduced by three orders of magnitude with respect to the software implementation. However, due to the irregular sub-problem sizes, which are determined during runtime, the implementation has to use a combination of fixed problem size and masking. This approach requires a lot of resources, which limits the number of parallel processes on the FPGA. Despite this, the throughput of the system is limited by the data transfer rate of the PCIe link between the host and FPGA.

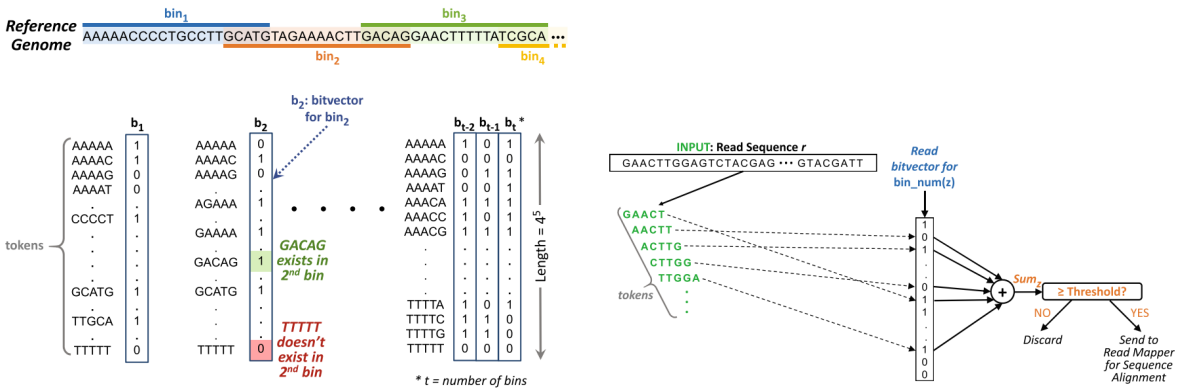
Algorithm 2 MAGNET

```
Input: Read, Ref, E
Output: Accept
  function ComputeHammingMask(Seq1, Seq2, L)                          ▷ Bitwise-XOR
    for i ∈ {0 ... L − 1} do
      HM[i] ← Seq1[i] == Seq2[i]                                      ▷ Pairwise-OR (HW)
    end for
    return HM
  end function

  function ConsecutiveZeros(HammingMasks, L)                          ▷ Longest Zero Sequence
    ## Finds and returns left and right index of longest zero sequence.
    return [left_index, right_index]
  end function

  function ExEn(HMs, GlobalMaskBegin, GlobalMaskEnd, L)
    if MaskBegin < MaskEnd then
      [left_index, right_index] ← ConsecutiveZeros(HMs, L)
      MagnetMask ← MagnetMask & ExEn(HMs, GlobalMaskBegin, left_index, L)   ▷ Bitwise-AND
      MagnetMask ← MagnetMask & ExEn(HMs, right_index, GlobalMaskEnd, L)
    end if
    return MagnetMask
  end function

  L ← length(Read)
  for i ∈ {−E ...E} do
    for j ∈ {0 ... L − 1} do
      ShiftedRead[j] ← Read[j] << i
    end for
    HammingMasks[i] = computeHammingMask(ShiftedRead, Ref, L)
  end for

  for i ∈ {0 ... L − 1} do
    MagnetMask ← 0
  end for
  MagnetMask ← ExEn(HammingMasks, 0, L − 1, MagnetMask, E)
  edits ← popcount(MagnetMask)                                        ▷ Multi-operand Addition
  return (edits <= E)                                                 ▷ Integer Comparison
```

2.5.4. GRIM-filter

GRIM-filter [8] is fundamentally different from the other pre-alignment filtering algorithms. Rather than using the pigeon-hole principle, this algorithm compares the presence of patterns occurring in the read and reference sequences. This algorithm pre-processes the entire genome and creates metadata about segments of the genome. To do this, the genome is split into overlapping bins of several hundred base pairs. The bin is then scanned for the presence of a so-called tokens, which are defined as all possible sub-sequences of 5 base pairs that can be created with the four different base-pair types. This creates a $4^5 = 256$ bit long bit-vector of 'existence bits' where a '1' or '0' represents the presence or absence of a particular token in the bin, respectively. This process is illustrated in Figure 2.11(a), and the pseudo-code is given in Algorithm 3.

During run-time, a similar bit-vector is created for the read sequence. This sequence is then compared to that of the reference bins that contain the seed locations. The number of patterns that occur in both bit-vectors is counted and compared to a threshold which is calculated using the edit-distance threshold. If the two bit-vectors are sufficiently similar, the filter passes the mapping. This process is illustrated in Figure 2.11(b), and the pseudo-code is given in Algorithm 4.



(a) (b)

Figure 2.11: A demonstration of (a) the bit-vector creation, and (b) read filtering steps in GRIM-filter. Adapted from [8].

Algorithm 3 GRIM-filter: bit-vector creation

Input: Reference, BinSize, Overlap, TokenSize
Output: BitVectors

```
j ← 0, k ← 0
while j <= length(Reference) − BinSize do
    Bin ← Reference[j ... j + BinSize - 1]
    for Token ∈ {"AAAAA", "AAAAC", ..., "TTTTT"} do          ▷ Short Pattern Detection
        for i ∈ {0 ... BinSize-TokenSize-1} do
            if Bin[i ... i + TokenSize − 1] == Token then
                TokenIndex ← hash(Token)
                BitVector[k][TokenIndex] ← 1
            end if
        end for
    end for
    j ← j + BinSize − Overlap
    k ← k + 1
end while
return BitVectors
```

Algorithm 4 GRIM-filter: filtering step

Input: BitVectors, Read, SeedLocation, E, TokenSize
Output: Accept

```
L ← length(Read)
BinNum ← ⌊Seedlocation/(BinSize − Overlap)⌋
Count ← 0
for i ∈ {0 ... L − TokenSize − 1} do
    TokenIndex ← hash(Read[i ... i + TokenSize − 1])          ▷ Short-pattern Detection
    Count ← Count + BitVectors[BinNum][TokenIndex]            ▷ Multi-operand Addition + Bitwise-XOR (HW)
end for
Threshold ← L − (TokenSize − 1) − TokenSize ∗ ⌈L ∗ E⌉
return Count >= Threshold                                     ▷ Integer Comparison
```

Since most previous works have encountered limitations in acceleration due to data-movement overheads, this algorithm is specifically designed for a computation-near-memory architecture (this concept is further elaborated in a later section). The algorithm is mapped onto a 3D-stacked memory architecture. This implementation allows multiple bins to be evaluated at the same time, by retrieving the existence-bit from the bit-vectors of multiple bins that correspond to the token using Though-Silicon Via (TSV). The existence bits are moved to a processor die that resides close to the memory elements, where each bit is compared to the existence bit of the read-sequence in parallel.

2.5.5. Shouji

In this work [9], the authors provide an FPGA implementation of the previously discussed MAGNET algorithm, as well as present a novel FPGA accelerated algorithm, Shouji. Like some other works, this algorithm uses the pigeon-hole principle. Shouji operates in three main steps: neighborhood map creation, identifying diagonally consecutive matches, and filtering out dissimilar sequences.

The neighborhood map is akin to shifted Hamming masks, with the main difference being that the shifted Hamming masks are placed on the diagonal of a grid. The Hamming mask between the reference and the non-shifted read sequence is placed on the main diagonal, with right and left-shifted versions to the top-right and bottom-left, respectively. The neighborhood map is used to identify the consecutive matches using a sliding window approach. For every sub-sequence of n base-pairs (n being the window size) along the diagonals of the neighborhood map, the longest pattern of uninterrupted '0's is found and greedily considered part of the final bit vector, as demonstrated in Figure 2.12. The sliding window is then moved by a single column for which the same process is repeated. In the end, this will produce a final bit vector with a length equal to that of the read sequence. The number of '1's in the final bit-vector is counted to determine the minimum number of errors of the mapping, which is compared to the edit-distance threshold to determine whether it should be accepted or not. The pseudo-code for this algorithm is provided in Algorithm 5.

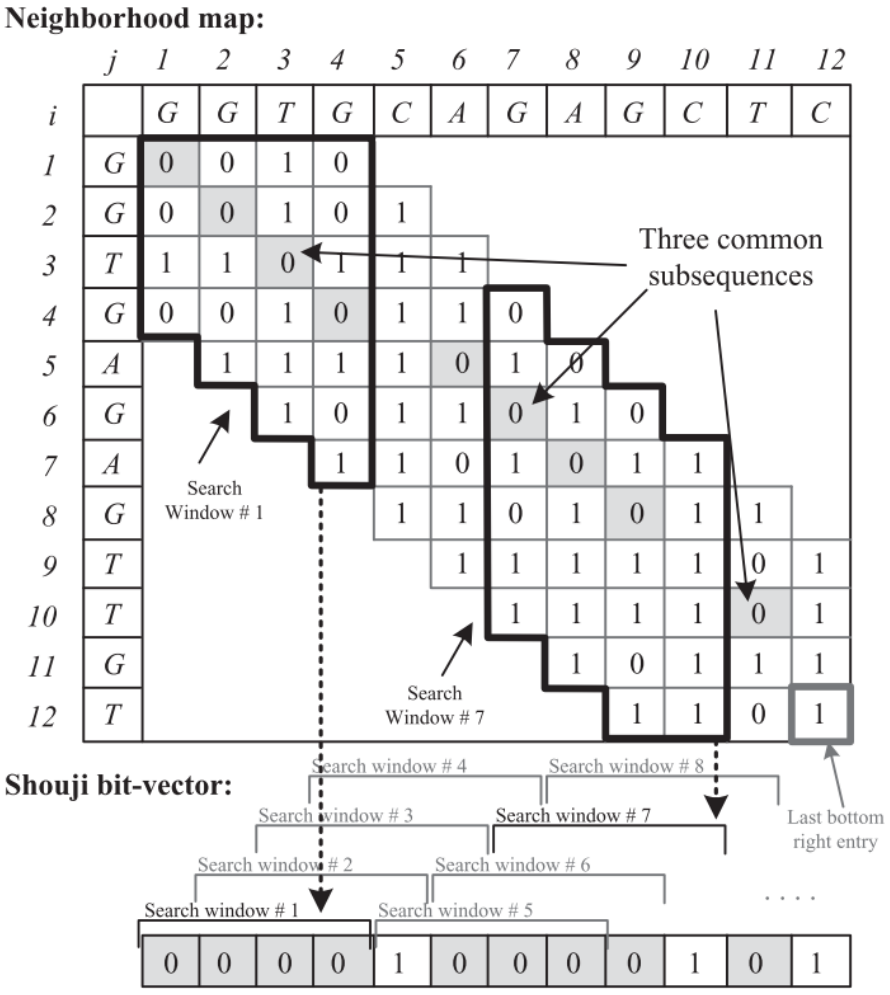


Figure 2.12: Illustration of the algorithm used by Shouji. Adapted from [9].

This algorithm is specifically designed to be accelerated on an FPGA. The sliding window approach makes it so that the search space of each iteration is limited to 2^n patterns, which are implemented using look-up tables on the FPGA. Like other FPGA-based accelerators, Shouji is limited by the rate at which data (read-reference pairings) is supplied to the FPGA.

Algorithm 5 Shouji

Input: Read, Reference, E
Output: Accept

```
function CZ(BitVector)                                   ▷ Multi-operand Addition (HW)
    Count ← 0
    for i ∈ 0 ... Length(BitVector) do
        if BitVector[i] == 0 then
            Count ← Count + 1
        end if
    end for
    return Count
end function

L ← length(Read)
for i ∈ {0 ... L − 1} do                                 ▷ Bitwise-XOR
    for e ∈ {−E ... E} do
        if Read[i] == Reference[i + e] then              ▷ Pairwise-OR (HW)
            NMap[i, i + e] ← 1
        else
            NMap[i, i + e] ← 0
        end if
    end for
end for
for i ∈ {0 ... L − 1} do
    FinalBV[i] ← 1
end for

Z ← "0000"
for i ∈ {0 ... L − 1} do                                 ▷ Short-pattern Detection
    for j ∈ {1 ... E} do
        if CZ(UpperDiagonal[j]) > CZ(LowerDiagonal[j]) then
            Z ← UpperDiagonal[j]
        else if CZ(UpperDiagonal[j]) == CZ(LowerDiagonal[j]) then
            if NMap[i + j, i] == 0 then
                Z ← UpperDiagonal[j]
            else if NMap[i, i + j] == 0 then
                Z ← LowerDiagonal[j]
            end if
        else
            Z ← LowerDiagonal[j];
        end if
    end for
end for
if CZ(NMap[i : i + 3, i : i + 3]) > CZ(Z) then
    Z ← NMap[i : i + 3][i : i + 3]
end if
if CZ(Z) > CZ(FinalBV[i : i + 3]) then
    FinalBV[i : i + 3] ← Z;
end if
return CZ(FinalBv) >= L − E                              ▷ Integer Comparison
```

2.5.6. SneakySnake

SneakySnake (SS) [10] is the most recently published work discussed in this thesis. This work finds similarities between the approximate string-matching problem and the signal-net routing (SNR) problem. The SNR problem is originally used for connecting two terminals on a grid with a path with as few obstacles as possi-

ble. SneakySnake draws parallels between the SNR problem and the approximate string-matching problem and uses those to perform pre-alignment filtering.

Signal Net Routing Problem
The signal net routing algorithm is aimed at finding the optimal way of connecting two terminals on a VLSI chip. It views the chip as a grid of interconnected vertical and horizontal routing tracks (VRT and HRT, respectively), with the processing components on the chip represented as obstacles. These obstacles are placed on the routing tracks between two interconnection points of VRTs and HRTs. For a signal to pass from one terminal to the other, it needs to traverse these obstacles, which introduce a certain propagation delay. The aim of the algorithm is to find the path between the two terminals with as little propagation delay as possible.

SneakySnake uses the same concept to determine the lowest number of edits between two sequences. In essence, the algorithm consists of two main steps: grid creation and grid traversal. In its most basic form, SneakySnake creates a grid with horizontal and vertical interconnects, in which the horizontal tracks represent shifted Hamming masks, which are created similarly to the way described in SHD and MAGNET. This is illustrated in Figure 2.13(a). In this maze, matches between the sequences ('0's in the shifted Hamming masks) are analogous to the processing elements in the SNR problem, except in this case, they can only be placed on the horizontal tracks. The algorithm starts by looking for the longest uninterrupted sub-sequence of '0's in the grid starting from the left of the maze and ending at the first obstacle it finds. This sub-sequence is considered part of the optimal alignment, and the obstacle is considered a mismatch between the read and reference and is referred to as a 'checkpoint'. From this checkpoint, the algorithm is iterated, starting at the horizontal position of the mismatch, as can be seen in Figure 2.13(b). This process is repeated until the end of the grid has been reached, or the number of errors encountered by the algorithm exceeds the edit-distance threshold. In the actual implementation of the algorithm, the grid is not constructed in its entirety, but it only evaluates the sub-sequences until it encounters the first obstacle, which reduces the computational load, as is shown in Figure 2.13(c). The pseudo-code of this filter can be found in Algorithm 6.

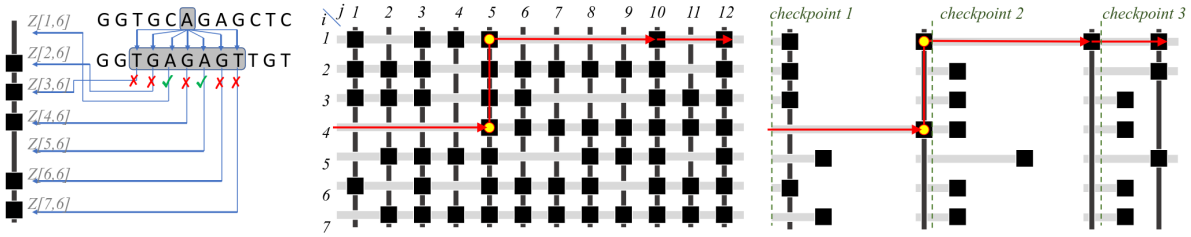


Figure 2.13: Illustration of (a) the grid-construction, (b) grid traversal, and (c) optimization of the SneakySnake algorithm. Adapted from [10].

This approach guarantees zero false negatives while promising fewer false positives than other works. It proves to be sufficiently effective at filtering wrong mappings that in some cases the pre-filtering time exceeds the alignment time. In those cases, the end-to-end execution time benefits greatly from the acceleration of pre-filtering. Therefore, SneakySnake is implemented on two different acceleration platforms: GPU and FPGA.

The GPU implementation is identical to the original implementation and utilizes thread-level parallelism to compute multiple pairings concurrently. On the GPU, each thread computes a single matching separately in the same way it is done on the CPU. The FPGA implementation is different in that it splits up one problem into multiple non-overlapping sub-problems which are solved separately. The shifted Hamming masks are split up into several parts, and the minimum edit distance of each part is calculated and added up to obtain the final result. This approach is necessary to limit the search space to a dimension known at design time such that the overall design can be simplified. This modularity also enables the accelerator to be scaled up to larger sequence lengths. The main downside of this is that the accuracy of the filter is degraded since it only finds locally optimal segments.

As is the case for other hardware-accelerated pre-alignment filters, the throughput of the accelerators is limited by the rate at which data is supplied to the accelerator.

Algorithm 6 SneakySnake

Input: Read, Reference, E
Output: Accept

```
Checkpoint ← 0
Count ← 0
L ← length(read)
for e ∈ {−E ... E} do                                   ▷ Bitwise-XOR
    for j ∈ {0 ... L − 1} do
        HM[e][j] ← Read[i] == Reference[i + e]          ▷ Pairwise-OR (HW)
    end for
end for
while (Checkpoint < L) & (Count <= E) do
    MaxZeros ← 0
    for e ∈ {−E ... E} do                               ▷ Longest Zero Sequence
        ZeroCount ← CLZ(HM[e], Checkpoint)
        if ZeroCount > MaxZeros then
            MaxZeros ← ZeroCount
        end if
    end for
    Checkpoint ← Checkpoint + MaxZeros
    Count ← Count + 1                                   ▷ Multi-operand addition (HW)
end while
return (Count <= E)                                     ▷ Integer Comparison
```

Furthermore, SneakySnake is claimed to be suitable for both short and long reads, which sets it apart from other works. However, the accelerators have only been implemented as a proof of concept on short read lengths of 100bps, and can therefore not be used for large sequence lengths.

2.5.7. Evaluation State-of-the-Art Pre-alignment Filters

With the current state-of-the-art, pre-alignment filtering can achieve orders of magnitude improvements to alignment times by filtering out the majority of wrong mappings. In fact, in certain use cases, the filter manages to reduce the alignment time to such a degree that the throughput of the pipeline is limited by pre-alignment filtering itself. To alleviate this bottleneck, pre-alignment filters have been implemented on hardware platforms such as GPUs or FPGAs to improve execution time (summarized in Table 2.3). By doing so, orders of magnitude improvement have been achieved (this is elaborated in detail in Chapter 3). However, this is in some cases still not enough to overcome the filtering bottleneck in any of the prior designs. A major problem with most filtering accelerators is that their throughput is limited by data-movement overheads [7, 9, 10]. In these systems, data is moved from the main memory to the host CPU, from where it is transferred to the accelerator.

Table 2.3: Overview of the algorithm implementations.

Algorithm	Software	Hardware	Zero FN-rate	Long-read Capable
SHD	MATLAB	FPGA	Yes	No
MAGNET	MATLAB	FPGA	No	No
Shouji	C/C++	FPGA	Yes	No
GRIM-filter	C/C++	HBM2	No	No
SneakySnake	C/C++	FPGA & GPU	Yes	Yes

Another commonality between the prior works is that most of these works have a similar approach to evaluating the performance of the presented designs. However, the exact datasets and hardware used for evaluation differ between the works, creating an unfair platform for comparison. A unified evaluation procedure is still missing from literature.

2.6. Computation-In-Memory

In recent years, CPU processing speeds have been increasing steadily according to Moore's law [66]. From 1986 to 2000, CPUs achieved annual increases of around 50% in terms of processing speed. Memory, on the other hand, has not seen such rates of improvement, at only around 10% annually. This growing disparity between CPU and memory speed can cause processing to be held up by the memory, which is called the "memory wall" [67]. For data-intensive tasks, such as many genomics or artificial intelligence applications, this wall has been reached, as the overall throughput of the processing elements is bottlenecked by the inability to supply them with data [68]. As a response to the limitations imposed by the memory wall, research is being conducted into finding more data-efficient paradigms such as Computation Near Memory (CNM) or Computation In Memory (CIM) [69].

In this section, we discuss what these new paradigms entail, and explain how they differ from more traditional computing methods. We explore current CNM/CIM solutions and their capabilities and limitations. Furthermore, we introduce the concept of memristors and explore how they can be used for computation in memory.

2.6.1. Von-Neumann Architecture

To understand why the memory wall exists in the first place, we first discuss how current processors work. Most current computing systems employ a so-called Von-Neumann architecture. This architecture consists of five main components [70]. Firstly, the Central Processing Unit (CPU) is the component that performs logic and arithmetic operations on data. In its most basic form, it consists of an Arithmetic Logic Unit (ALU), and processor registers. Data stored in the processor registers are used as inputs to the ALU, which performs basic operations such as boolean operations or arithmetic operations such as addition or subtraction. The type of operation performed by the ALU is determined by instructions that are given by the control unit. This component supplies instructions based on the contents of its instruction register and program counter. It is also responsible for loading in data from the main memory to the processor registers. The main memory stores all instructions and data required for the execution of a program. The main memory is volatile, meaning that it loses its contents when power is cut off from the system. Therefore, data is stored in mass storage, from which it is loaded into the main memory. Lastly, Von-Neumann architectures have Input/Output (I/O) devices, which form the interface between the computing system and the outside world (humans or other computing systems). The structure of the Von-Neumann architecture is shown in Figure 2.14.

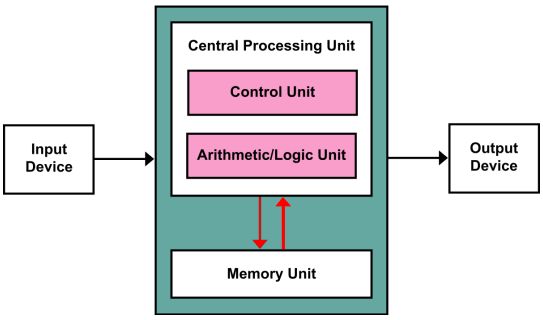


Figure 2.14: Illustration of a basic Von-Neumann-Architecture. Adapted from [11]

The memory wall occurs between the main memory and the central processing unit as the transfer happens over buses causing additional latency and power consumption [68]. This is indicated by the red arrows in Figure 2.14.

2.6.2. Computation-Near-Memory

One of the ways to diminish the effects of the memory wall is to use Computation-Near-Memory. In this paradigm, processing units are added to the periphery of existing memory elements (usually DRAM). Examples of this approach are Monolithic3D [71] and UPMEM [72]. In Monolithic3D, 3D-stacked memories are augmented by adding processing layers to the memory die. UPMEM creates processing-capable DIMMs by placing general processing units close to the memory dies, which work alongside conventional DRAM modules. Both architectures benefit from the large memory bandwidth that far exceeds the bandwidths of conventional architectures such as GPUs or FPGAs.

2.6.3. Computation-In-Memory

Another approach is to use the analog properties of the memory cells themselves to perform computations, which is referred to as Computation In Memory (CIM). This allows CIM-based architectures to forgo the need to read out the operands of the instruction entirely. Several works have been proposed that implement CIM using currently ubiquitous technologies such as SRAM [59, 73] and DRAM [74, 75]. However, as we reach the limits of memory density that can be achieved with conventional memories [76], most research goes out toward CIM implementations in new memory technologies [77]. These Emerging Non-Volatile Memories (eNVMs), are characterized by their use of memristive devices (memristors) as a means to store data rather than charges, as is the case for SRAM and DRAM. eNVMs show promising improvements over conventional memories such as lower power consumption and higher densities [12, 77].

In this section, we will discuss the concept of a memristor, and compare the most common implementations of this technology. Furthermore, we explore how their physical properties can be used to store data and how they can be used to enable CIM.

2.6.4. Memristive Devices

Memristive devices or memristors are two-terminal passive devices that relate electric charge and flux linkage and are considered the fourth fundamental circuit component alongside resistors, inductors, and capacitors [12, 40]. It voltage-current relation is non-linear, as the device's resistance changes based on the current that has previously flown through it. The current-voltage curve of a memristor takes the shape of a pinched hysteresis loop as can be seen in Figure 2.15.

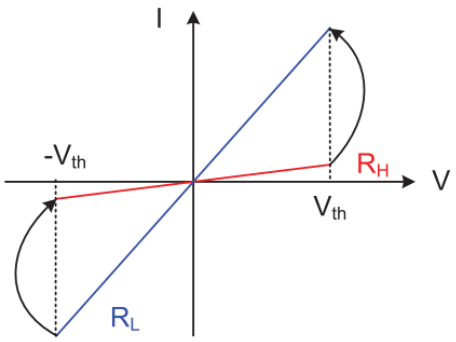


Figure 2.15: The voltage-current relation on memristive devices. This curve shows the pinched hysteresis loop that characterizes memristors. Adapted from [12].

Here it can be seen that the memristor has two stable resistance states: a High Resistance State (HRS) and a Low Resistance State (LRS). The device can transition between these states based on the electrical potential applied over its terminals. When a positive voltage exceeds a certain threshold, V_{set} or V_{th}, the device changes from its high-resistance state to its low-resistance state. When exposed to a negative voltage that exceeds V_{reset} or $-V_{th}$, it changes back to the high-resistive state. Any voltage between these values does not influence the resistance of the device, which allows it to 'store' its resistance state. This property of the memristor can be leveraged to use it as a memory element by representing a binary '0's and '1's as the HRS and LRS, respectively. This resistance state is also retained when power is removed from the device, which makes it a non-volatile memory.

The data in the memristor cell can be accessed by exploiting Ohm's law. By applying a predetermined read-voltage over the device (that does not exceed the threshold voltages), and sensing the resulting current, it is possible to deduct the resistance value of the memristor.

The underlying physical process of memristive devices differs between implementations. Here, we discuss the three most common memristor technologies, Phase-Change Memory RAM (PCM-RAM), Spin-Transfer Torque Magnetic RAM (STT-MRAM), and Resistive RAM (ReRAM/-RRAM).

PCM-RAM
Phase-Change Memory RAM gets its memristive properties by changing the phase of chalcogenide material [12, 16, 45]. This material has two physical states: a crystalline state and an amorphous state. In the crystalline state, the PC material has a lattice structure, which has a low resistance, and in the amorphous state, the material is unstructured and has a high resistance. Transitions between the two phases can be achieved by applying heat generated by a current through the device. Crystalline material can be turned into its amorphous state by applying a short but intense spike of heat, while the crystalline state is obtained through a longer exposure to a lower heat. It is also possible to have states between these two extremes, which have a resistance between the LRS and HRS. This allows for multiple bits of information to be stored on a single cell. PCM-RAM has reached a relatively mature state compared to the other memristor technologies and has shown good compatibility with existing CMOS processes. However, this technology suffers from several disadvantages [12]. For one, the writing process requires high writing currents to achieve the required heat output and is slow compared to the other memristor technologies. Also, this technology suffers from drifting in the amorphous state, which requires additional circuit components to compensate for it.

STT-MRAM
Spin-Transfer Torque Magnetic RAM works by using Magnetic Tunnel Junctions (MTJs), which consist of two magnetic layers separated by a thin insulating layer [12, 44]. One of the magnetic layers has a fixed magnetic orientation and is therefore called the "fixed" layer. The other magnetic layer is the "free" layer and has an adjustable magnetic orientation. The orientation of this layer can be changed by applying a current through it. The electrons in this current transfer some of their angular momentum over to the free layer, changing its magnetic orientation. The resistance of the device depends on whether the magnetic orientations of the two layers are parallel to each other, the device is in its LRS, and when the orientations oppose each other, the device is in its HRS. This type of device has a greater switching speed compared to PCM-RAM and is also relatively mature [12]. However, it has worse CMOS compatibility due to the MTJs requiring multiple layers on the die.

ReRAM
Resistive RAM employs a metal-insulator-metal structure [12,16,43]. Here an insulating material, often HfO_2 is placed between two metal electrodes. This device changes its resistance state by the creation and destruction of an electrically conductive filament through the insulating material. By applying a voltage to one side of the device, a conductive filament is formed from one electrode to the other. This decreases the resistance of the device and puts it in the LRS. Conversely, applying a current in the opposite direction destroys this filament, putting the device back in the HRS. This technology has the advantage of being compatible with current CMOS technology [12]. Also, the ratio between the HRS and LRS is greater than those of the other memristor devices, making distinguishing between the resistance states easier. Furthermore, it has a high switching speed and can achieve high densities. The endurance of these types of devices varies greatly in literature, with reported ranges of $10^6 - 10^{12}$ cycles.

A comparison between the main characteristics of state-of-the-art implementations of these three technologies can be found in Table 2.4 [16]. Here the cell area is expressed in feature size.

Table 2.4: Overview of the characteristics of the most commonly used eNVMs. [16]

	PCM-RAM	STT-MRAM	ReRAM
Cell area	~ 4-30 F^2	~6-50 F^2	**~4-12 F^2**
Multi-bit capability	**yes**	no	no
Read time	~10-100 ns	~10-100 ns	~10-100 ns
Write time	~50-100 ns	~25-100 ns	**~10-100 ns**
Endurance	**1E15**	1E9	1E6 - 1E12
Write energy	**~0.1 pJ**	~10 pJ	**~0.1 pJ**

Crossbar Structure
Due to their comparatively high densities, low power consumption, and non-volatility, eNVMs are considered a replacement for DRAMs and SRAMs in current computing systems. Proposed memory architectures usually place the memristor in a $n \times m$ crossbar configuration [16]. In this configuration, a grid is created with n horizontal word lines and m vertical bit lines. The cross-points of these lines are connected using a crossbar

switch. This structure allows every memristor cell to be accessed by applying voltages on the word lines and bit lines for the targeted cell.

The crossbar switch can have multiple different topologies, such as 1R, 1T1R, 2T1R, etc. Here the 'R' denotes the number of memristive elements per cell, and 'T' is the number of transistors per cell. The transistors are placed in series with the memristive elements to reduce leakage currents in the case that the ratio between HRS and LRS is small. Peripheral circuitry is added to the crossbar, such as DACs/write-drivers and ADCs/sense-amplifiers to interface between digital compute systems and the analog crossbar. These peripheral components often are the main contributors in terms of area and power overheads in crossbar architectures. Therefore, resources are often shared between multiple rows/columns using what is called interleaving. For example, a single sense-amplifier can be used for sensing multiple columns of the crossbar by multiplexing its inputs. A basic illustration of a 1T1R array – one of the most common crossbar topologies – is provided in Figure 2.16.

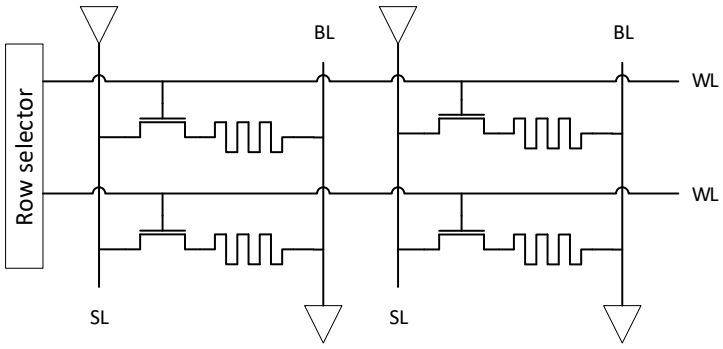


Figure 2.16: Basic topology of a 1T1R array.

2.6.5. CIM using eNVMs

The memristive devices and the crossbar structure enable the implementation of logic/arithmetic operations using CIM. Most notably, the possibility of implementing vector-matrix-multiplication is being explored for the acceleration of Neural Networks [35, 36]. Another focus of CIM research is to enable massively parallel boolean operations [13, 78], which are commonly used in research fields such as bio-informatics [37], which relates to this work. There is a multitude of ways to enable such operations, which differ in the way inputs are provided to the crossbar, and how the results can be retrieved. In this section, we explore the most prominent approaches of boolean logic in CIM, and we provide examples of how they are implemented.

CIM-Array

The first approach to enabling CIM in crossbars is Computation In Memory Array (CIM-A). In this approach, the input operands of the boolean operation are provided as the resistance states of two or more cells in the same column of a crossbar array. Voltages are applied to the targeted operand cells, and the resulting currents (following Ohm's law) are combined (Kirchhoff's law) and used as input to an output memory cell. Depending on the resistive states of the input cells, the current may or may not be large enough to trigger a switch of resistance states in the output cell. The result of the operation can then be obtained by performing a regular read operation on the output cell. This approach has the advantage that the output is stored in the output cell, which can therefore be used as an input for a subsequent operation. It does, however, in some cases, require multiple read/write operations to the output cell to obtain the result of a single boolean operation. This is detrimental to the latency of the computation and reduces the endurance of the output memory cell. Several works have been published proposing different solutions for implementing CIM-A. These include MAGIC [79], Snider [80], IMPLY [78], and Stateful-logic [81].

CIM-Periphery

The second approach is Computation In Memory Periphery (CIM-P). Similar to CIM-A, this method stores the input operands of the instruction as cells in the memory array, and voltages are applied to the targeted word lines to sensitize the operands. The difference is that the resulting current is not used as an input to the output cell but is directly sensed in the sense-amplifier in the periphery of the array. The current level is compared to reference currents and, depending on the outcome of the comparison, an output is generated

in the form of a digital signal. This approach has the advantage of obtaining the results in a single cycle. Also, since there is no output cell, there is no loss in endurance. However, this also means that the result is stateless, meaning that the result is lost when the system is deactivated. Furthermore, this approach requires the modification of peripheral circuitry, which increases area overheads. CIM-P has been implemented in multiple works including, Pinatubo [82] and Scouting logic [14].

To illustrate the differences between the two solutions, we give an example of performing a NOR operation between two operands using MAGIC (CIM-A) and Scouting-Logic (CIM-P), as this operation is supported by both works.

Memristor Aided Logic
Memristor Aided loGIC (MAGIC) supports NOR, OR, AND, NAND, and NOT operations. However, of these operations, only NOR and NOT can be implemented in a crossbar structure without significant changes, while the other operations can only be used as stand-alone logic.

The NOR operation in MAGIC is performed in two steps. First, the output cell is initialized to the HRS, which is achieved using a regular write operation. The two operands are initialized to HRS to represent a logic '0' and LRS to represent a logic '1'. Then, the word line of the output cell is grounded, and the word lines of the operand cells are set to a voltage V_0 by their respective word-line drivers. In doing so, the crossbar can be abstracted to a circuit equivalent as found in Figure 2.17.

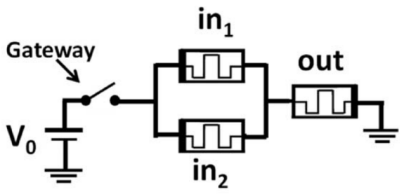


Figure 2.17: Equivalent circuit for a 2-operand NOR in MAGIC. Adapted from [13]

In this equivalent circuit, the input operands act as parallel resistances, which are in series with the output cell. With the assumption that HRS » LRS, if either input cell is in an LRS, the equivalent resistance of the input cells is close to the LRS value. When both inputs are '0' (HRS), the equivalent resistance is HRS/2. The voltage over the output cell can be obtained as a voltage division over the equivalent resistance and the output cell resistance.

For a correct gate operation, the output cell is reset if the voltage over the output cell exceeds $V_{T,reset}$, which should only be the case if both inputs are '0'. For that reason, the input voltage V_0 is chosen as described in Equation 2.1 [79].

$$2V_{T,reset} < V_0 < \frac{R_{HRS}}{2R_{LRS}} * V_{T,reset} \tag{2.1}$$

The result is stored in the output cell, which is in LRS for logic '1' and in HRS for logic '0'. This result can be read through a regular read operation.

Scouting Logic
As an example for CIM-P, we provide the process for computing a NOR operation using Scouting logic. While Scouting logic formally only supports OR, AND, and XOR operations, we observe that a NOR operation is equivalent to an AND operation with inverted input operands. The authors observe that during a read operation, the current that results from applying a read voltage to the input of the cell is compared to a reference current in the sense-amplifier. If the crossbar current is greater than the reference, it is deduced that the cell is in LRS, and HRS otherwise. To do a logic operation, the two operands are stored as resistance values in the same column of a crossbar. Then, both cells are activated, which in effect act as two resistances in parallel. It is found that with two operands, there are three distinct equivalent resistance levels: $R_{HRS}/2$ (two '0' inputs), $R_{LRS}/2$ (two '1' inputs), and $R_{HRS}//R_{LRS} \approx R_{LRS}$ (one '1' and one '0'). Consequently, there are three

current levels determined by V_0/R_{eq}. By picking two reference currents in the sense-amplifier between the three resistance levels, the three input combinations can be differentiated. These levels are pictured in Figure 2.18.

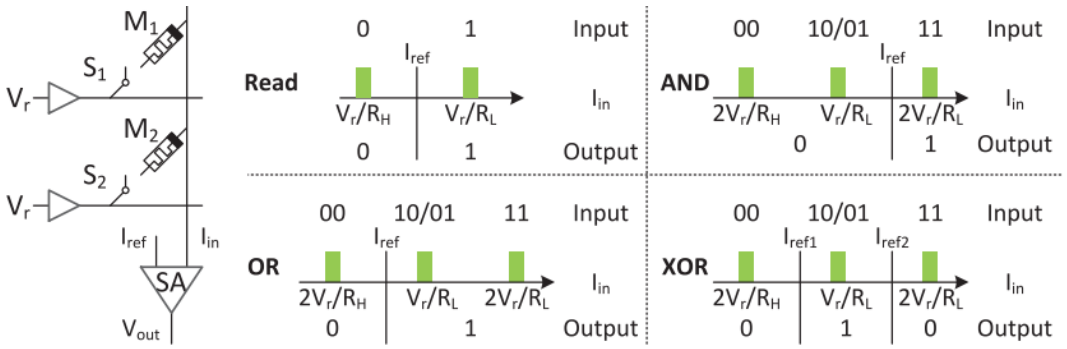


Figure 2.18: Reference current levels in Scouting logic. Adapted from [14].

2.6.6. Memristor-based TCAM

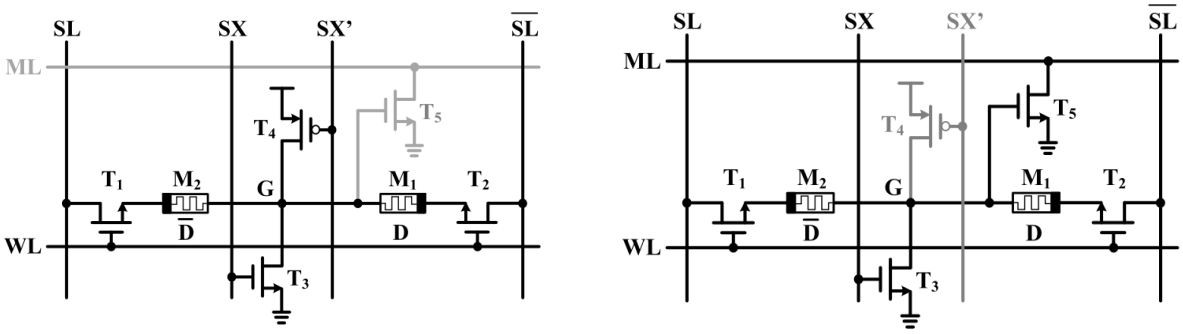
Lastly, we discuss the Memristor-based Ternary Content Addressable Memory (TCAM). A TCAM is a form of associative memory that compares input data against a table of stored data and returns the address of the table entry that matches the input data. TCAMs differ from binary Content Addressable Memory (CAM) in that it can store a 'X'-value rather than just '0' and '1'. This 'X' denotes a "don't care"-value which produces a match regardless of whether the input-data bit is a '0' or '1'. Currently, TCAMs are most commonly implemented using SRAM cells, but recent research has gone out to explore the possibility of implementing TCAMs using eNVMs such as PCM-RAM, STT-MRAM, and ReRAM [15, 83–86]. These solutions claim a reduction in cell area, search energy, and standby power consumption over SRAM-based TCAMs.

In this section, we discuss one such design based on ReRAM which is presented by Lin et al. [15]. This design makes use of a 5T2R structure, where each cell has two ReRAM cells that store information on the '0' and '1' programming state of the TCAM cell. To this five CMOS transistors are added for control during the write and search operations. The structure of the TCAM cell can be found in Figure 2.19.Figure 2.19(a) shows the programming scheme for the three cell states, Figure 2.19(b) the circuit in programming mode, and Figure 2.19(c) the circuit in search mode. To program the memristor cells, programming voltages are applied over SL and $\overline{SL}$, in a two-cycle process, while the voltage at node G is controlled by SX and $\overline{SX}$.

D	
0	HRS
1	LRS

Content	D	$\overline{D}$
X	0	0
0	0	1
1	1	0

(a)



(b) (c)

Figure 2.19: (a) Definitions of binary and ternary data, (b) ReRAM-based TCAM cell in write mode, and (c) ReRAM-based TCAM cell in search mode. Adapted from [15].

For the search operation, the match-line ML is pre-charged, and the search pattern is driven onto SL and $\overline{SL}$. For a 'search-0' operation, these values are set to low and high respectively, while SX is set to low. The applied voltages are selected such that the voltage at node G, which results from the voltage division over the two memristors, exceeds the threshold-voltage of T_5, only when M_2 and M_1 are set to values other than LRS and HRS (mismatch), respectively. The values of SL and $\overline{SL}$ are reversed for a 'search-1' operation. If a mismatch occurs, T_5 is activated, and ML is discharged to ground. If all cells in the row have matches, ML maintains its pre-charged value. Peripheral circuitry is added to sense the charge level of ML, based on which it is determined whether the input pattern matches the TCAM entry.

2.7. Conclusions

In this section, we provide background information for topics related to this thesis. We explain the concept of DNA sequencing and discuss what pre-alignment filtering entails, and how it fits into the sequencing pipeline. We explore current state-of-the-art pre-alignment filters and find that the different ways of testing performance lead to an unfair comparison between the different works. This calls for a unified testing methodology to make sure the algorithms are evaluated fairly.

Furthermore, we find the shortcomings of the current accelerators. Most notably, the data movement between the host device and accelerator is reported to be a major bottleneck to which computation-near-memory or computation-in-memory are identified as possible solutions.

Also, we introduce the concepts of CNM/CIM and explain common methods of implementing boolean operations using this paradigm. We introduce the concept of memristors and emerging non-volatile memories, explaining their underlying physical processes, advantages/disadvantages, and how they can be used for CIM. Lastly, we discuss the concept of a TCAM, and how it can be implemented using memristors.

Chapter 3

Benchmarking & Profiling

From the main findings of Chapter 2, we identify CIM as a potential avenue for the improvement of the performance of pre-alignment filters. To effectively create a CIM architecture specifically for the acceleration of pre-alignment filtering, we first have to determine what algorithms must be implemented and what operations are required to support them. Furthermore, we need to establish a baseline for performance in terms of accuracy and performance to which we compare the proposed CIM design.

From the examination of existing pre-alignment filters in Section 2.5, we found that different works employ different input data sets and hardware configurations to evaluate their designs. In Section 3.1, we describe the process of creating a basis for a fair comparison between existing works in terms of accuracy, filtering time, and end-to-end processing time, using the same data sets and hardware for the evaluation of each of the previous works. The results of this evaluation are presented in Section 3.2.

Additionally, we identify operations commonly found in existing pre-alignment algorithms and show their relative contributions to the overall filtering time in Section 3.3. Here we also categorize the works and conclude what operations should be supported, and what algorithms are most suitable for acceleration using CIM architectures. Finally, we summarize and draw conclusions on the main findings of this chapter in Section 3.4.

3.1. Testing Methodology

As alluded to in Chapter 2, existing works make comparisons between their respective algorithms and previous works. However, the datasets used for the evaluation of the algorithms differ between the various works. These differences are in the raw DNA-read data used, seeding methods, and alignment algorithms/configurations. To add to that, the hardware used for evaluation is vastly different, which paints a skewed picture when evaluating execution times. To fairly evaluate the performance of the various works, we implement the algorithms with the exact same software and hardware setup. In this section, we discuss how we generate the datasets used to evaluate the various works.

3.1.1. Test datasets

Human Genome

For this project, we focus on mapping reads to the human genome, as the vast majority of sequencing data is targeted at human DNA samples. To this end, we use the human genome assembly GRCh38 [87], which is the most recent assembly at the time of writing.

Short Reads

The first dataset category is aimed at evaluating the performance of the pre-alignment algorithms for short reads. These reads typically range from 75 to 300 base pairs in length and compose the vast majority of currently available sequencing data. These reads are most often obtained using the Illumina sequencing technology and therefore have a high accuracy of over 99%. To evaluate the performance of the algorithms for this kind of read, we use two sample sets, ERR240727_1 [88] and SRR826471_1 [89] which are both Illumina reads and have read lengths of 100 and 250 base pairs, respectively.

For seed generation, we use MrFAST [90], a commonly used read-mapper specifically designed for short reads. The source code is modified such that it outputs the seed location, read sequence, and reference

sequence for every potential mapping location. Here, the location of the seed is given as the base-pair offset of the seed with respect to the start of the read. We use this method to produce 30 million pairings used to evaluate the pre-alignment filters. For this type of sequence data, the edit-distance threshold is typically set between 2-5% of the read-length [7,9,90]. This allows us to distinguish true mappings, where edits only occur from sequencing errors or genetic mutations, from wrong mappings, where the seed extension is dissimilar from the read.

All of the examined algorithms have been verified to work on sequence data similar to these two, with the exception of GRIM-filter, which has only been implemented for 100 bp reads.

Long reads
The second category of reads is long reads of 10-100 kbps in length. These kinds of read lengths are representative of reads obtained using PacBio or ONT sequencing technologies. Unlike Illumina short reads, these reads are not uniform in length. As we are interested in examining performance across the range of read lengths specifically, we use simulated reads with a fixed read length rather than real sequence data. This is done using PBSIM, a model-based simulator, which can generate reads similar to real PacBio reads for a given read length and sequencing accuracy. We use these reads as input for Minimap2 [82], a state-of-the-art read-mapper that supports long reads. The output of Minimap2 only contains the seed location, which we use to retrieve the reference sequence corresponding to the read using SAM-tools [91].

With this method, we create a total of four datasets, with different read lengths and sequencing accuracies. We use a low sequencing accuracy of 80%, with sequences ranging from 10 kbp to 100 kbp in length, to represent PacBio CLR reads. For this type of read, the edit threshold for alignment should be at a minimum of 20% of the read length to account for sequencing errors. We evaluate this dataset up to 25% to also account for variations in the genome. Similarly, to simulate accurate reads – such as those obtained using PacBio CCS and ONT – we set the sequencing accuracy to 98%, again with read lengths of 10 kbp and 100 kbp. Here, we focus on edit-distance thresholds of 2-7%. Of each of the datasets, we generate 100 thousand and 10 thousand pairings for read-lengths of 10 kbp and 100 kbp, respectively. Currently, the only pre-filtering algorithm that supports these long reads is the CPU implementation of SneakySnake. An overview of the datasets can be found in Table 3.1 and Figure 3.1.

Table 3.1: Overview of the Evaluated Datasets

Dataset name	Seq tech.	Seq. Acc.	Read length	No. of Pairings	Algorithm support
Short_100bp	Illumina	99%	100 bp	30 million	all
Short_250bp	Illumina	99%	250 bp	30 million	all but GRIM-filter
Long_10kbp_inacc	CLR	80%	10 kbp	100 thousand	SneakySnake
Long_100kbp_inacc	CLR	80%	100 kbp	10 thousand	SneakySnake
Long_10kbp_acc	CCS/ONT	98%	10 kbp	100 thousand	SneakySnake
Long_100kbp_acc	CCS/ONT	98%	100 kbp	10 thousand	SneakySnake

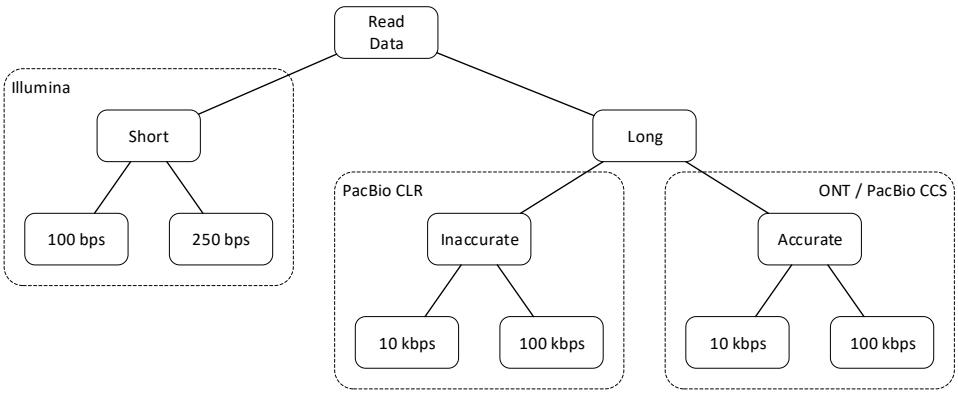


Figure 3.1: Overview of the tested read data sets.

3.1.2. Evaluation Strategy

As a baseline for alignment, we use a state-of-the-art software-based alignment algorithm, Edlib [92]. We examine edit-distance thresholds in a range of 0-10% and 0-25% of the total read length for the short and long reads, respectively. If the algorithm finds that the alignment is within the defined threshold, we declare the pairing accepted. We perform alignment and record which pairings are accepted for every edit-distance threshold, and we measure the total alignment time. To reduce the effects of statistical outliers and measurement error, we take an average of 5 runs, discarding the highest and lowest execution time recorded.

To evaluate the performance of the pre-alignment filters we look at three key metrics. Firstly, accuracy is evaluated, as this determines the reduction of the alignment time. Here we are particularly interested in the false-negative rate and the false-positive rate. Also, the filtering time is important, as it is desirable to minimize the execution time that is introduced by adding another step to the pipeline. For this, we perform the same averaging as described for alignment. Lastly, we look at the end-to-end execution time. This is defined as the sum of the filtering time and the alignment time after the removal of pairings through pre-alignment filtering. This metric combines accuracy and filtering time and is the most important for evaluating the performance of the filter.

3.1.3. Hardware Specifications

All algorithms are run on CPU, with the exception of SneakySnake, which is evaluated for both CPU and GPU. While some works also provide FPGA accelerated works, they require a specific Virtex®-7 FPGA VC709 Connectivity Kit, which is not available for evaluation for this project. Therefore, to get a ballpark estimate of the performance of these works, we include execution times as reported by their respective authors. Furthermore, the performance of the HBM2 implementation of GRIM-filter is obtained through a simulation, of which neither the process nor the performance data are provided by the author. Therefore, this implementation is not considered in this comparison. An overview of the hardware specifications can be found in Table 3.2.

Table 3.2: Hardware specifications of the test setup

Component	Specification
CPU	Intel® Xeon® CPU E5-2680 v4 @ 2.40GHz (single thread)
GPU	Nvidia Tesla K80
FPGA	Virtex®-7 FPGA VC709 Connectivity Kit

3.2. Benchmarking Results

In this section, we discuss the benchmarking results in the three main performance metrics for both short and long sequences.

3.2.1. Short Read Accuracy

Total Positive Rate

First, the datasets are evaluated using alignment without any pre-alignment filters. This gives an impression of the composition of the datasets and gives a benchmark against which the pre-alignment filters can be compared. The positive rate of alignment, as evaluated by Edlib, is shown in Figure 3.2. Here it can be seen that the vast majority of all generated pairings (>99.99%) in the region of interest (highlighted in green) exceed the edit-distance threshold. This means that without a pre-alignment filter, alignment is performed unnecessarily for most pairings, which clearly shows the need for pre-alignment filters.

False-positive Rate

The false-positive rate is the measure of how many pairings pass the filter but are rejected by alignment and ideally should be as close to 0% as possible. The FP-rates of the examined algorithms for short reads can be found in Figure 3.3. Here it can be seen that for all algorithms the false-positive rate increases with the edit-distance threshold. Furthermore, we observe that SneakySnake shows the (shared) lowest false-positive rate for all examined edit-distance thresholds.

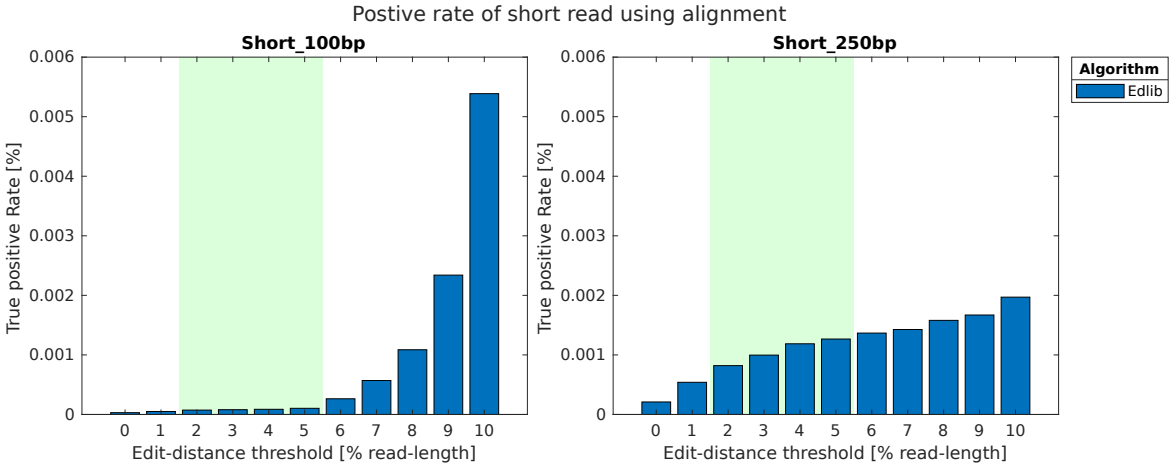


Figure 3.2: Positive rate of alignment for short reads.

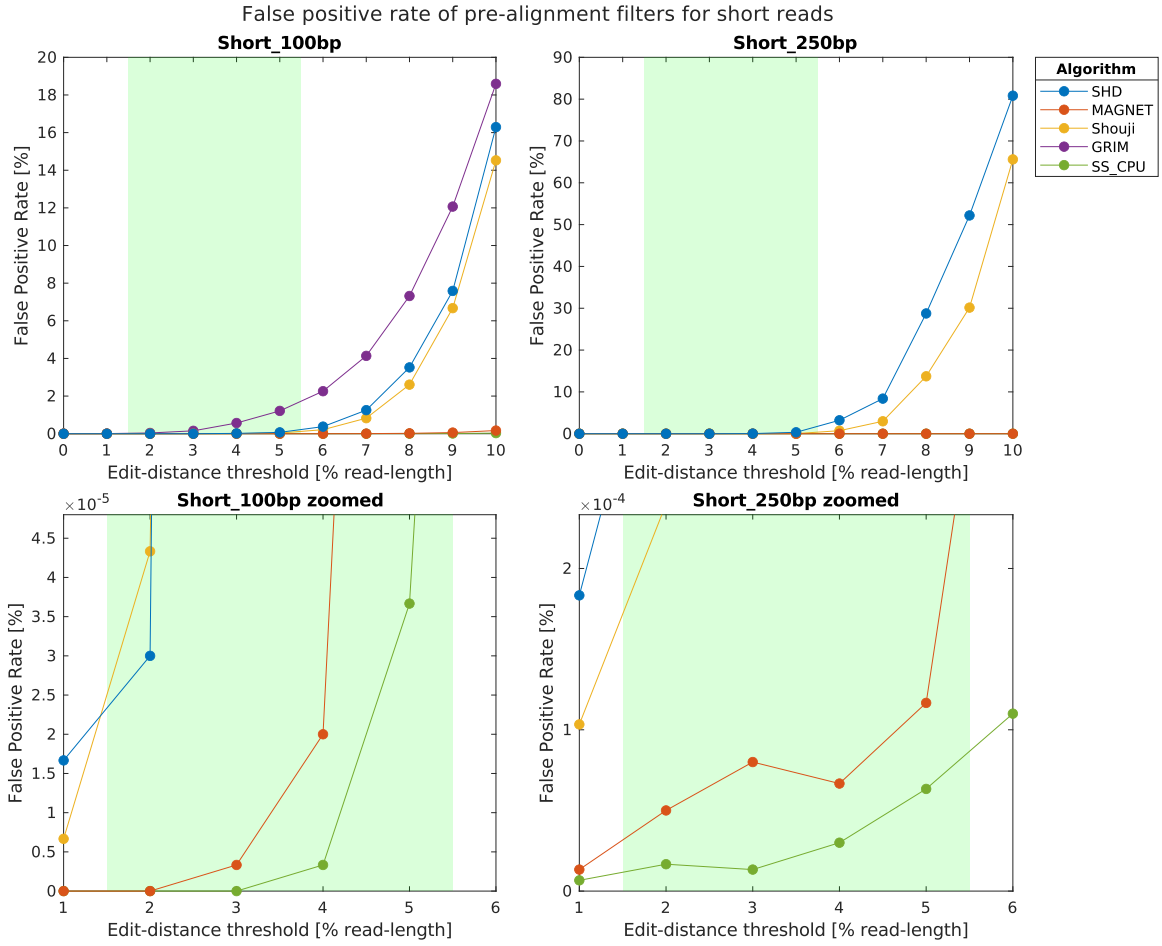


Figure 3.3: False positive rate of the examined pre-alignment filters for short reads. The bottom figures more clearly show the false-positive rate of the most accurate filters in the region of interest.

False-negatives
The absolute number of false-negatives of each algorithm can be found in Figure 3.4. For the examined datasets, only GRIM-filter shows any false negatives in the Short_100bp dataset. Note, however, that while MAGNET shows a zero false-positive rate in the examined datasets, the approach taken in MAGNET does not guarantee the absence of false negatives. This means that these filters will remove mappings that would otherwise be accepted by alignment, leading to the loss of genomic information.

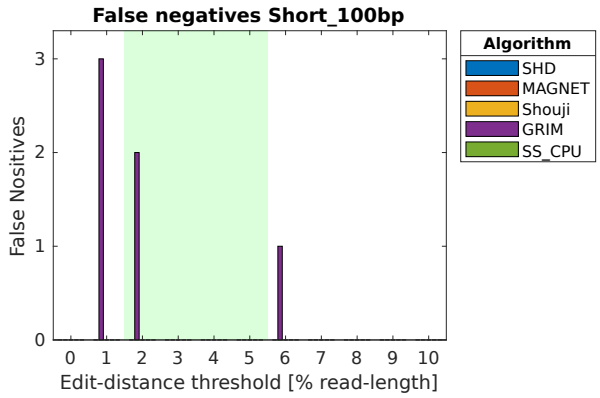


Figure 3.4: Absolute number of false negatives per algorithm for the Short_100bp dataset.

3.2.2. Short Read Filtering Time

In Figure 3.5, it can be seen that in most algorithms, there is a steady increase in filtering time as the edit-distance threshold increases. Here the exception is GRIM-filter, which remains largely constant for edit-distances over 0%. Additionally, it can be seen that there is an increase in filtering time when increasing the read length for all algorithms. Furthermore, we observe that the algorithms implemented in MATLAB (SHD and MAGNET) have a far worse filtering time than those implemented in C/C++ (several orders of magnitude), and rely on acceleration for performance. Lastly, we observe that SneakySnake has the best non-accelerated performance for all evaluated datasets and edit-distance thresholds, which can be attributed to the fact that, unlike other Hamming-mask-based algorithms, it does not evaluate the entire length of the Hamming masks.

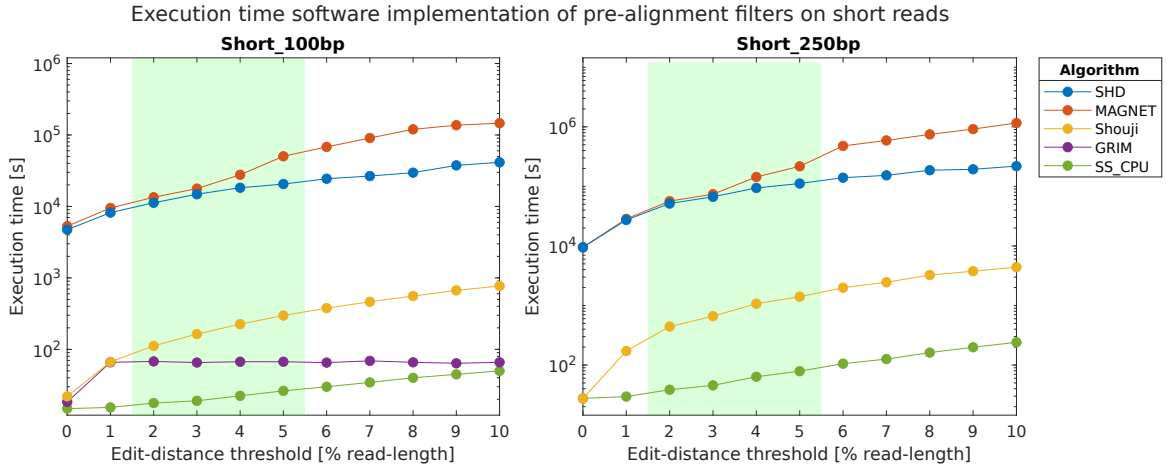


Figure 3.5: Execution time of the software implementation of the pre-alignment filters on short reads

To reduce the impact of pre-alignment filtering on the end-to-end execution time, the filters have all been implemented on hardware accelerators. We compare the algorithms' performance based on their respective accelerators in Figure 3.6. Note that the FPGA-based algorithms have been implemented for 100 bp reads only, while the GPU implementation of SneakySnake (SS-GPU) has been extended to support 250 bp reads as well. Furthermore, the performance of FPGA-based algorithms (SHD, MAGNET, Shouji, and SneakySnake) are all limited by the throughput of the PCIe connection, and therefore show the same performance figure

and a constant execution time. For SS-GPU it can be seen that the execution time scales exponentially with the edit-distance threshold. Despite this, the throughput is still greater than that of the FPGA-based solutions in the tested edit-distance range.

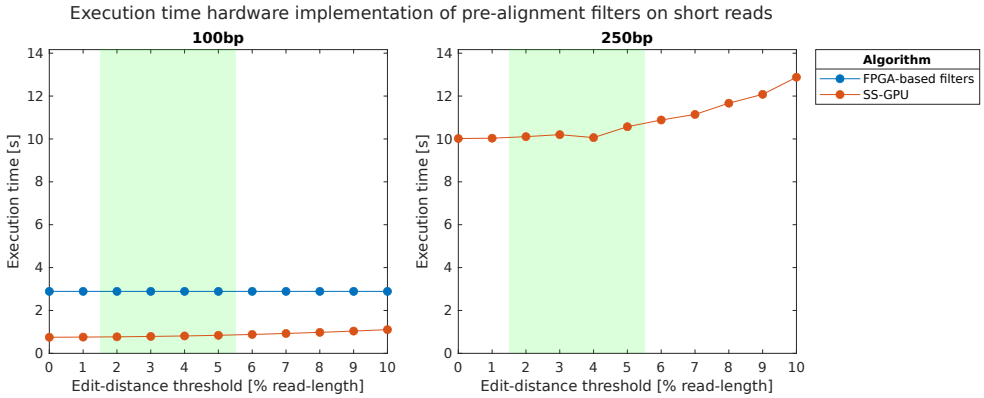


Figure 3.6: Execution time of the hardware implementation of the pre-alignment filters on short reads.

3.2.3. Short Read End-to-end Execution Time

The end-to-end execution time is defined as the sum of the filtering time and the alignment time after filtering. The alignment time is affected by the accuracy of the algorithm, as that determines the number of evaluated read-reference pairs that have to be evaluated by alignment. The alignment time is measured for all datasets by running Edlib over only those pairings that have passed the filter (i.e., true positives and false positives). We compare the end-to-end execution time of Edlib without filtering, with software filtering, and with hardware-accelerated filtering. In Figure 3.7 we only show the filters with the best performance (i.e., SS-CPU and SS-GPU for software and hardware, respectively).

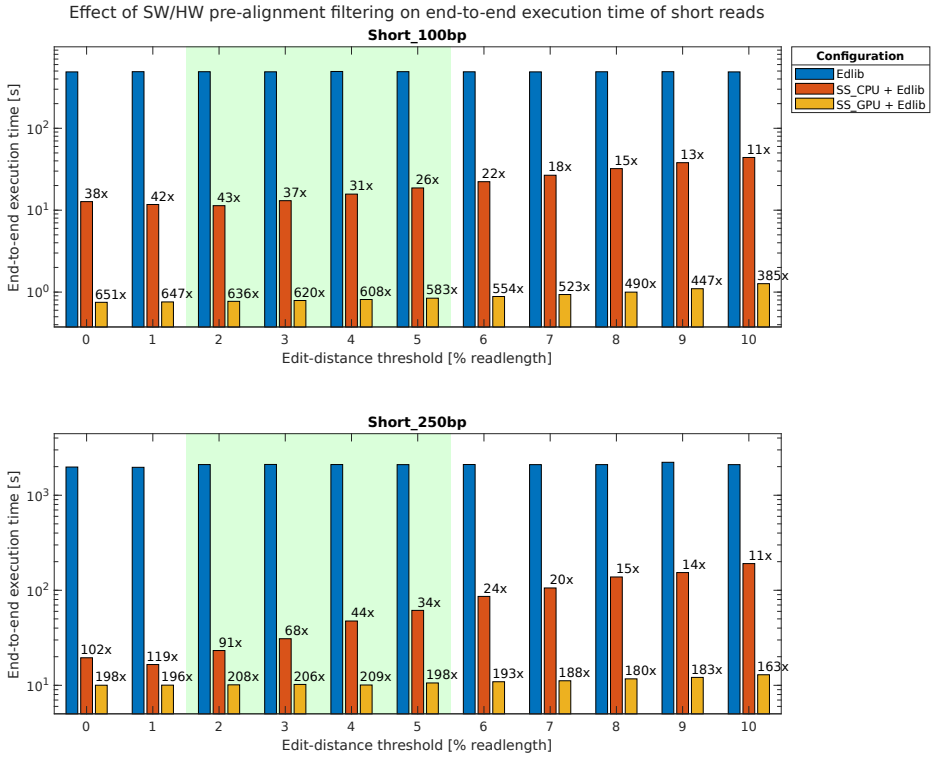


Figure 3.7: End-to-end execution time of the best-performing software filter and hardware accelerated filter compared to alignment without any pre-alignment filter. The numbers at each bar represent the reduction in end-to-end execution time as compared to alignment without filtering.

Here we find that SS-CPU provides an end-to-end improvement over just alignment of between 26x to 91x depending on the edit-distance threshold and data set. Using a GPU to accelerate pre-filtering shows an end-to-end improvement of 198x to 636x depending on the edit-distance threshold and data set. The graph shows that performance degrades when evaluating reads at a large edit-distance threshold for both the software and the hardware implementation of the algorithm.

The relative contribution of the filtering and alignment stages are measured and shown in Figure 3.8 for SS-GPU. Here we observe that the vast majority of the end-to-end execution time is spent on filtering rather than alignment. This shows that the bottleneck for the alignment of short-reads has shifted to the filtering stage, even when using the highest-performance filtering solution available.

Figure 3.8: The relative contribution of filtering and alignment using SS-GPU for short reads.

3.2.4. Data Movement Overhead

To find the limitations of the GPU-based solution, we use nvprof [93] to determine the time spent on data movement between host and device, and the GPU processing time. The results of this examination can be found in Figure 3.9. Here it can be seen that in the region of interest, the majority of the end-to-end execution time can be attributed to data movement. Furthermore, as we move to longer reads, data movement of the read and reference sequences makes up an increasingly large share of the overall execution time.

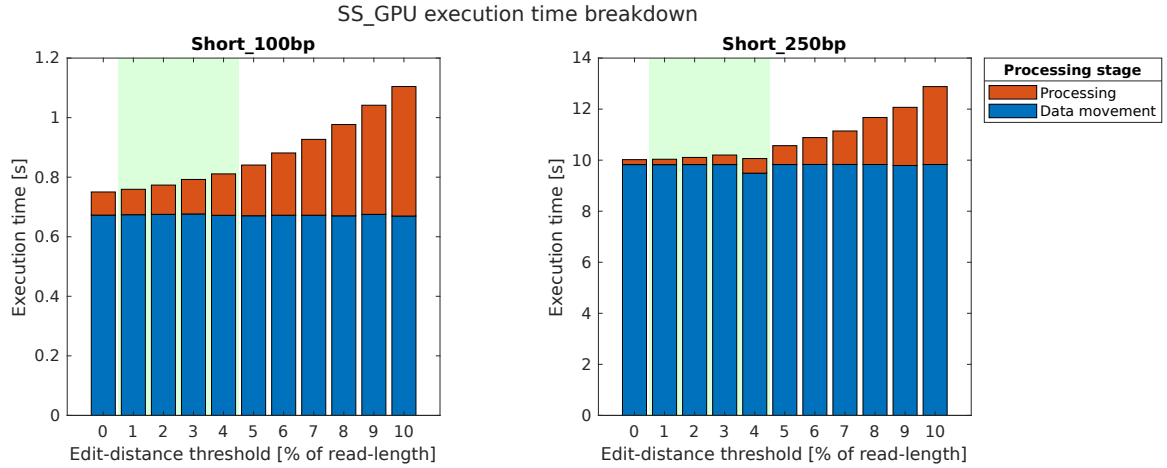


Figure 3.9: Contribution to the end-to-end execution time of data-movement and processing time for SS-GPU for short reads.

3.2.5. Long reads

Since SneakySnake is the only filtering algorithm compatible with long reads, there is no comparison to be made between pre-alignment filtering algorithms. Therefore, we refer to Appendix A for the filtering time and accuracy figures. The end-to-end execution time of SneakySnake is compared to the alignment time without any filter in Figure 3.10.

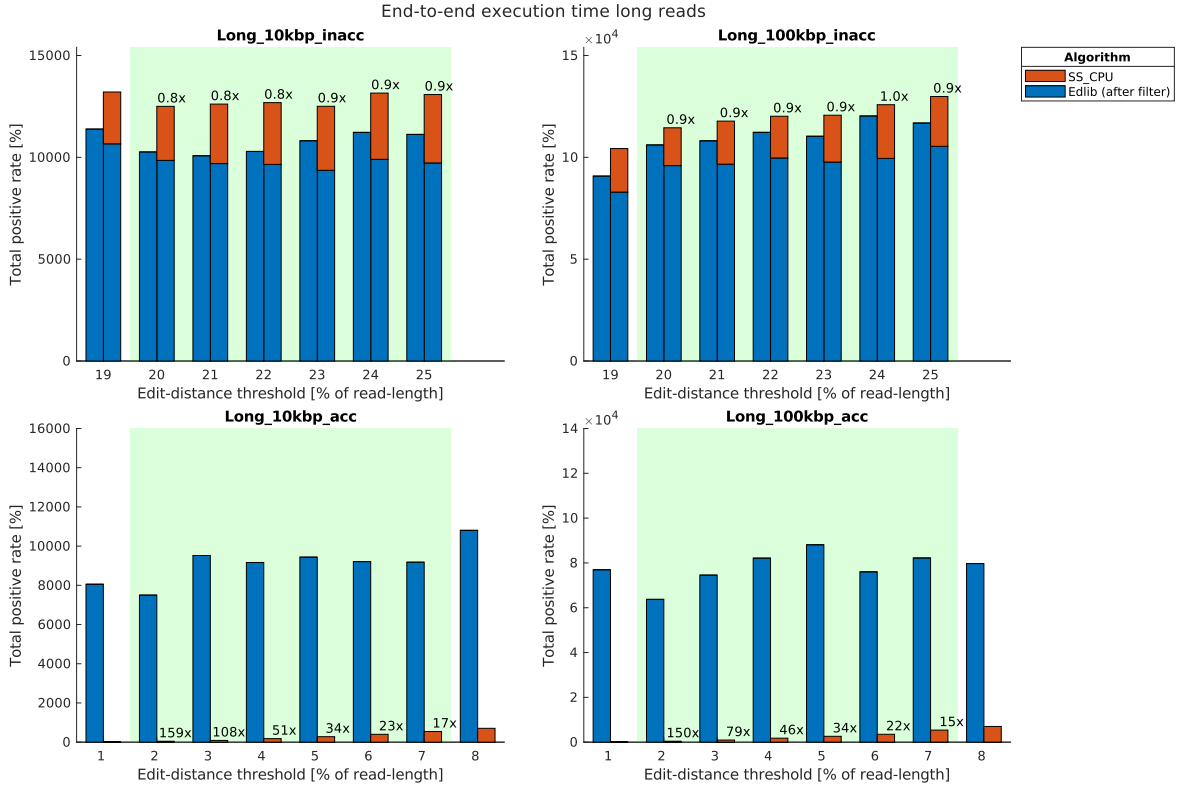


Figure 3.10: End-to-end execution time for long reads

Here we observe that, in the case of inaccurate reads, the end-to-end execution time is worse when introducing SneakySnake as a pre-alignment filter. For this type of read, the reduction in alignment time is not enough to compensate for the added filtering time. On the other hand, accurate reads do show significant improvement ranging from 15x to 159x. Here, the majority of the remaining execution time is due to the filter, rather than alignment.

3.3. Classification and Profiling

From the benchmarks, we find that for both short and long reads – in certain datasets – pre-alignment filtering is the dominant factor in the end-to-end execution time. While using hardware acceleration helps reduce this bottleneck, it is not completely removed. This motivates us to further improve the acceleration of pre-alignment filtering. To do so, we first have to determine what operations need to be implemented on the new architecture to support the filtering algorithms. In this section, we qualitatively examine the operations and determine what operations are commonly used by the algorithms. Additionally, we quantitatively measure the relative contributions of the operations to the overall execution time, where possible.

3.3.1. Common Operations

Firstly, we inspect the source code of the various algorithms, and we extract the underlying operations. These operations are indicated in Algorithms 1 to 6. An overview of this inspection can be found in Table 3.3.

Bitwise-XOR

The main underlying operation for comparison between reads and references for all algorithms is the XOR operation. In SHD, MAGNET, and Shouji, this is used to create a set of shifted-Hamming-masks or neighbor-

Table 3.3: Overview of common operations in pre-alignment filters.

Operation	SHD	MAGNET	Shouji	GRIM-filter	SneakySnake
Bitwise-XOR	X	X	X	X	X
Pairwise-OR	X	X	X		X
Short-pattern detection	X		X	X	
Longest zero-seq.		X			X
Bitwise-AND	X	X			
Multi-operand addition	X	X	X	X	X
Integer comparison	X	X	X	X	X

hood maps. In GRIM-filter, this operation is used to compare the existence-bit vector of the read to that of the reference. Lastly, the XOR operation is used in SneakySnake for detecting obstacles in the SNR problem. While performance optimizations remove this operation from the CPU implementation, it is still required for the FPGA implementation.

Pairwise-OR
To reduce data-movement overheads, the hardware-accelerated implementations of SHD, MAGNET, Shouji, and SneakySnake use encoding for the read and reference sequences. Generally, genomics data is stored as a series of character arrays, where the bases of the DNA are represented as 'A', 'C', 'G', or 'T'. Since we know that there are only 4 types of bases, we can represent them as 2-bit values. After performing the XOR operation, it needs to be determined whether any of these two bits differ between the two examined sequences. To do so, a pairwise-OR operation is required.

Longest Zero-sequence
MAGNET and SneakySnake both operate by finding the longest sequence of uninterrupted zeros. In the case of MAGNET, all shifted-Hamming masks are scanned for their longest sequence of uninterrupted zeros, and the longest sequence is propagated to the final bit vector. In SneakySnake, all rows of the chip maze are scanned, and the length of the longest sequence determines the starting point of the next iteration of the algorithm.

Short-pattern Detection
SHD, Shouji, and GRIM-filter make use of short-pattern detection. In the case of SHD and Shouji, the result of the pairwise-OR operation is examined using a sliding window, to detect series patterns of '0's and '1's. These then determine the value that needs to be stored in the final bit-vector. In GRIM-filter, the presence of k-mers in the read sequence needs to be determined for the creation of the existence-bit-vector before the xor-operation with the reference.

Bitwise-AND
In SHD the final bit-vector is the result of a bitwise-AND operation of all shifted-Hamming masks. In MAGNET, the final bit-vector is obtained iteratively, where each intermediate result is the bitwise-AND operation between the current result and that of the previous iteration. Note that this is only required for the FPGA implementation of MAGNET.

Multi-operand Addition
In all algorithms except SneakySnake, a final bit-vector is created in which '1's represent edits between the read and reference sequences. The number of edits is approximated by counting the number of '1's in a final bit-vector. In SneakySnake, the number of estimated edits is incremented after every obstacle is detected. For the FPGA implementation of SneakySnake, the number of errors found in every sub-problem is added to find the total number of estimated edits.

Integer Comparison
In all algorithms, the estimated edit-distance is compared to a pre-determined value, which is (based on) the edit-distance threshold. To determine whether the pairing should be accepted or rejected, the approximate edit-distance is compared to this threshold.

3.3.2. Profiling Results

Of the examined algorithms, SHD, MAGNET, Shouji, and SneakySnake are programmed such that the different operations are easily separable. For these algorithms, we quantitatively analyze the relative contribution of each operation in the algorithms by placing timer flags in the source code (for MATLAB implementations) or using Intel Vtune utility [94] (for C/C++ implementations). For SS-CPU, the longest-zero-sequence operation and the bitwise-XOR operation are implemented within the same line of code, and are therefore combined in the results. GRIM-filter is implemented with tight integration into the MrFAST read-mapper, and its operations are intertwined with the seed generation process. For this algorithm, we are therefore limited to qualitative analysis. The results of the quantitative analysis can be found in Figure 3.11. Here the 'other' category includes operations such as preparing the input data to have the right format for the algorithm, I/O calls, and measuring errors.

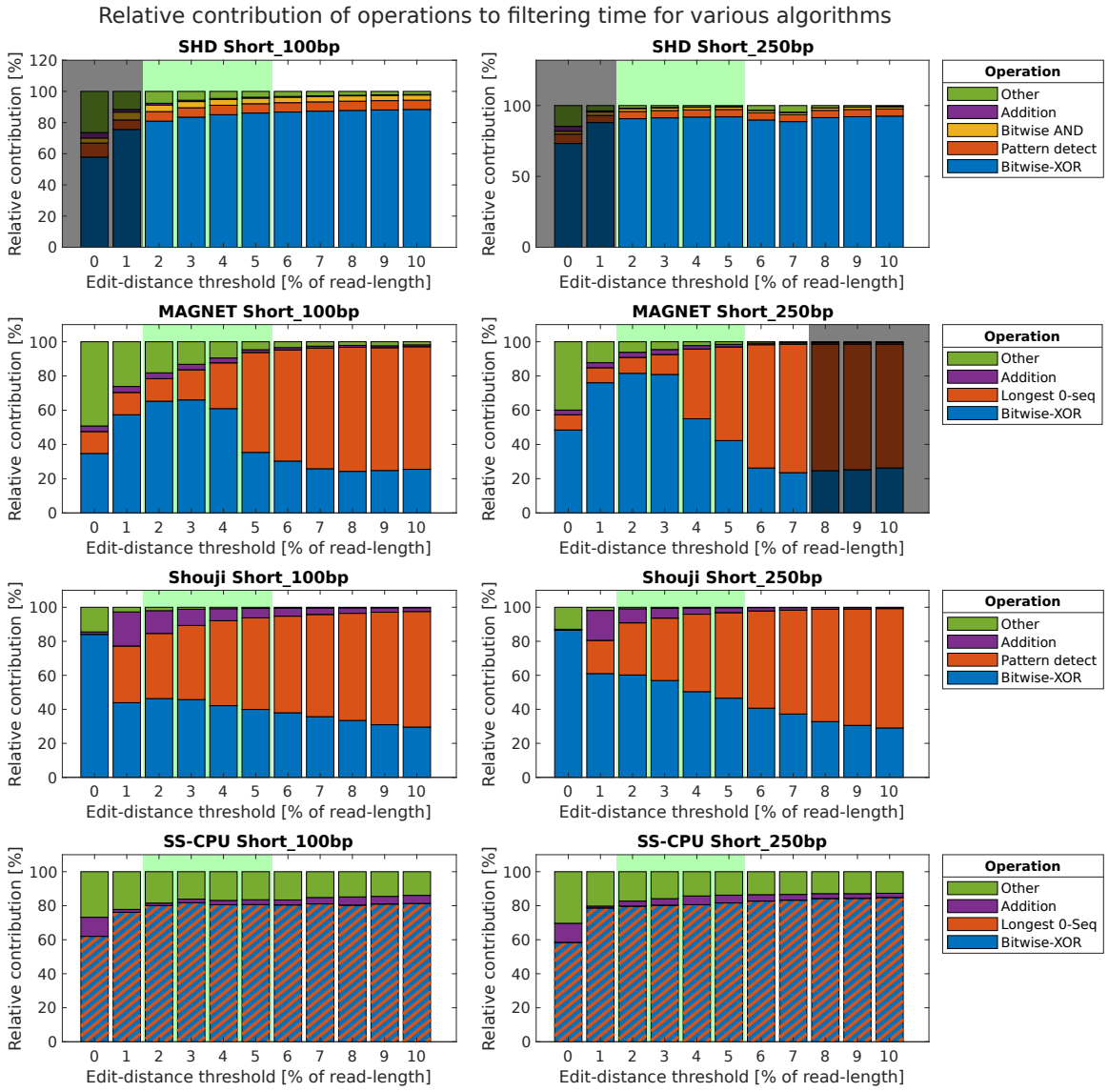


Figure 3.11: Quantitative profiling of pre-alignment filters. From top to bottom: SHD, MAGNET, Shouji, and SS-CPU.

The profiling of the C/C++-based algorithms is performed using the Intel Vtune utility [94], which introduces little to no profiling overhead. The MATLAB-based algorithms are measured by placing clock calls before and after each iteration of the algorithms and measuring the difference between the two measurements. As the clock itself introduces measuring overheads, we compare the execution time with and without clock calls. If this difference exceeds 25% when compared to the results achieved without clock calls, we discard the result

(marked in black in Figure 3.11. The discrepancy of SHD for small edit-distance thresholds can be explained by the relatively large contribution of clock calls to the overall execution time. It is however unclear why the discrepancy exists for large edit-distances for MAGNET performed on the 250 bp dataset. This investigation is left for future work.

From Figure 3.11, it can be seen that in the region of interest, all evaluated algorithms are for a large part limited by the bitwise-XOR operation. Furthermore, we see that in MAGNET, Shouji, and SneakySnake, a large part can also be attributed to finding the longest sequence of zeros. We observe that this share grows as the edit-distance threshold increases, as more iterations have to be performed on the same set of shifted-Hamming-masks. Lastly, a major contributor to execution time is the multi-operand addition. We see that for SHD, MAGNET, and Shouji, the contribution to the overall execution time shrinks for an increase in the edit-distance threshold. This can be attributed to the fact that the number of addition operands remains constant for all edit-distance thresholds. This trend is not present for SneakySnake, as the number of required additions is dependent on the edit-distance threshold.

We find that other operations only take up a small part of the execution time for Shouji and SneakySnake, which are implemented in C/C++, in contrast to SHD and MAGNET, which are implemented less efficiently in MATLAB. However, if bitwise-XOR and pattern detection are accelerated in a hardware implementation, these other operations might have a more significant share of the overall execution time in all algorithms.

3.3.3. Filter Characterization

From the inspection of the operations in the algorithms, we find that they can be classified into two main categories: local and global filters. We define local filters as those that find errors based on evaluating a set number of bits/base pairs of the read and reference sequences. These filters are characterized by the use of short-pattern detection. This approach to filtering has two main advantages. Firstly, since these algorithms consider only a small sub-sequence at a time, parallelism can be achieved by processing multiple non-overlapping sub-sequences at the same time. Secondly, because every sub-sequence has a fixed size, these types of filters can easily be scaled up in hardware accelerators. Additionally, because every sub-problem can be solved independently, these algorithms are suitable for time-multiplexing, such that the number of resources required in hardware can be limited.

On the other hand, global pre-alignment filters do not have a fixed sub-problem length, which is the case with MAGNET and SneakySnake. In these algorithms, the number of iterations required is not fixed for a certain read length and edit-distance threshold but is dependent on the nature of the input data. In Figure 3.3, we find that this type of filter tends to have greater accuracy as compared to their local counterparts. However, global filtering algorithms present some difficulties in hardware accelerators. For MAGNET, the hardware has to support the entire read length at once and requires additional masking to implement the subdivision into sub-problems. This approach makes it so that the area overhead is far greater than local filters [9]. SS-FPGA diminishes the area overhead by splitting up the SNR problem into multiple fixed-length sub-problems and combining the results. In essence, it turns the original global filter into a local filter. This, however, negatively impacts the accuracy of the algorithm as this implementation has a higher false-positive rate.

3.4. Conclusions

Based on the benchmarking results, the following key observations are made:

1. Most pairings generated by seed-and-extend-based read-mappers (>99.9%) turn out to be false pairings after alignment.

2. Out of all examined pre-filtering algorithms, SneakySnake has the highest accuracy for all tested data sets and edit-distances.

3. Out of all examined pre-filtering algorithms, SneakySnake has the lowest execution time for all tested datasets and edit-distances.

4. Using CPU-based filters, 26x-91x and 15x-159x improvement in end-to-end execution times can be obtained for short and accurate long reads, respectively.

5. All currently available pre-alignment filters are unsuitable for inaccurate long reads.

6. Using hardware accelerators, short read end-to-end execution time can be improved to 198x-636x.

7. For both short and accurate long reads, the largest contribution to end-to-end execution time is caused by pre-alignment filtering.

8. The performance of hardware accelerators in all prior works is limited by data movement from the host to the device.

From qualitative and quantitative profiling and characterization of the algorithms, we make the following observations:

1. All profiled algorithms can be broken down into a combination of seven elementary operations which are listed in Table 3.3.

2. In the quantitatively profiled algorithms, the majority of time is spent on bitwise-XOR operations, detection of short patterns, and finding long sequences of zeros.

3. The examined pre-alignment filters can be categorized into global and local filters. Global filters tend to have better accuracy but have a non-constant execution time and present challenges when implementing them on hardware. Local filters are hardware friendly but show worse accuracy than global filters.

4. SneakySnake demonstrates that global filters can be converted into local filters at the cost of reduced accuracy of the filter.

From benchmarking we find that the current filters with the best performance are SS-GPU and SS-CPU for short and long reads, respectively. From this, we conclude that any future filtering solution should aim to improve upon these works.

Furthermore, it can be concluded that to accelerate the alignment process using pre-alignment filtering, the focus must be on achieving a decrease in the execution time of the filtering stage. Additionally, in current hardware-accelerated filters, the execution time is limited by data movement, calling for the need for more data-efficient processing paradigms such as CNM or CIM.

To create a hardware accelerator that is suitable for multiple different algorithms, the architecture must support the seven operations listed in Section 3.3.1. Here, the focus must be on accelerating bitwise-XOR operation, short-pattern detection, and zero-sequence detection, as these operations have the greatest impact on execution time.

To obtain the best accuracy, the proposed architecture must have support for the SneakySnake algorithm. This algorithm is can be classified as a global filtering algorithm, for which a hardware implementation can be challenging. It is found that such global filtering algorithms can be converted into local algorithms by sacrificing accuracy. Because filtering has become the main bottleneck, this loss of accuracy can be justified provided that the decrease in filtering time is large enough to compensate for it.

Chapter 4

Algorithm Adjustment

In Chapter 3, we find that the performance of currently existing hardware accelerators of pre-alignment filters is limited by the rate at which data is supplied to the accelerator. This motivates us to explore the possibility of performing pre-alignment filtering using a CIM architecture. Through profiling, we identify a number of operations that create the backbone of existing pre-alignment filters, which this accelerator needs to support. In this work, we propose a hardware-software co-design that implements these operations and that supports multiple pre-alignment filtering algorithms on the same CIM architecture. In this chapter, we explore how these common operations can be mapped onto a CIM-enabled hardware architecture. For this, we first consider the existing CIM implementation styles in Section 4.1 and select the one that is most suitable for our design. We identify the limitations of this CIM implementation, and where necessary, we adjust the algorithms to account for them in Sections 4.2 and 4.3. We then examine the accuracy and we profile the adjusted algorithm in Sections 4.4 and 4.5. Finally, we present the main findings of the chapter in Section 4.6.

We choose to select two algorithms to map onto the architecture: SHD and SneakySnake. This choice is made for two main reasons. Firstly, from benchmarking it is found that SneakySnake is the highest-performance pre-alignment filter in terms of both execution time and accuracy. To show improvement over the state-of-the-art, we use the CIM implementation of this algorithm to compare to the current best filter. We select SHD as an additional algorithm because the combination of SHD and SneakySnake covers all of the most common operations found in pre-alignment algorithms (as determined in Section 3.3). Having coverage for all operations provides more flexibility in mapping future algorithms that might provide improvements over the current state-of-the-art. Furthermore, the selection of these algorithms provides coverage for both local and global filters.

4.1. CIM logic scheme

To determine how the algorithms need to be adjusted for a CIM implementation, we first select an underlying logic scheme. Here the choice is between CIM-A and CIM-P. We consider the advantages and disadvantages of each scheme and make our decision based on which architecture is most suitable for our design. The main basis for comparison is the bitwise-XOR operation, as from the profiling in Section 3.3 we find that one of the main bottlenecks in pre-alignment filters is the bitwise-XOR operation. Furthermore, it is the starting point of most algorithms, where the XOR operation is the only operation that uses raw read/reference sequence data as its operands. All other operations are performed on the results of the XOR operation.

CIM-P directly provides a way to perform an XOR operation within the crossbar in a single cycle, for example using Scouting logic. Using CIM-A methods, this operation requires multiple cycles. For example, MAGIC only supports NOR, OR, AND, NAND, and NOT operations. However, the XOR result can still be obtained by combining the outputs of multiple operations sequentially. One of these ways is demonstrated by Hoffer et Al. [95], which achieves correct circuit behavior of the XOR operation in two cycles using implication and an OR operation. One of the advantages of using this CIM-A approach is that the output of the operation is stored and can be accessed at a later stage. This could be useful in SneakySnake, for instance, where all Hamming masks that constitute the chip maze must be available at the same time.

However, the problem here is that SneakySnake traverses the chip maze in the horizontal direction, where it looks at bits in a single column of multiple Hamming masks. The values in a column cannot be accessed in parallel, meaning that the stateful properties of CIM-A cannot be taken advantage of. On the contrary, having

dedicated output-cell rows makes it so that the overall capacity to store operands (in this case read/reference sequences) is smaller when resources are held at a constant. Also, the iterative nature of the XOR operation in CIM-A requires multiple write operations to the same memory cell for initialization and the two computation stages. This can lead to more rapid degradation of the output cell's endurance.

CIM-P, on the other hand, can achieve an XOR operation in a single read cycle, which is beneficial to overall system latency. Also, it only requires a single write operation to load in the operands of the instruction, which improves endurance compared to CIM-A. The main disadvantages of CIM-P lie in that it cannot achieve the same level of parallelism as CIM-A. This is because the number of parallel logic operations that can be obtained in a cycle depends on the number of sense-amplifiers in the periphery of the cell. The number of sense amplifiers cannot be scaled indefinitely due to the fact that they constitute the majority of the tile's (crossbar and peripheral components) cell area and power consumption [14]. While more parallelism can be achieved by spreading the bit vector operands over multiple tiles, combining the results requires either inter-tile communication or segmentation of the algorithm. Segmentation of the algorithm poses no problem for local filter algorithms such as SHD but requires significant adjustment for global algorithms such as SneakySnake.

Overall, considering the advantages and disadvantages of the two logic styles, CIM-P comes out as a more compelling option, granted that SneakySnake can be adapted to have a segmented approach. In the following sections, we explore how this can be done and what effects this has on the accuracy of the algorithm.

4.2. Algorithm Adjustment

In SneakySnake, we see that to implement the algorithm on an FPGA, each read-reference pair is split up into multiple sub-problems which are solved independently. In effect, this changes the global algorithm into a local algorithm. However, this approach runs into several problems when implemented on a CIM architecture. Firstly, the SS-FPGA requires the computation of the entire chip-maze of the segment. The horizontal dimension of the chip-maze is dependent on the number of bases of each sub-problem, and the vertical dimension is dependent on the examined edit-distance. While the size of the maze is manageable for short reads, the resources that are required to store the chip-maze for long reads would be too large to implement for each tile. For example, for long reads that reach up to 100 kbps and have edit-distance thresholds of up to 25% (which is required for the processing of PacBio CLR reads), the chip maze would have to store up to 50 thousand rows of data per tile. Secondly, the algorithm takes several iterations to find the optimal path through the chip maze. This process is time-consuming and requires a relatively large number of lookup tables to implement.

We propose a new algorithm, SneakySnake-CIM, that draws inspiration from the segmentation approach of SS-FPGA, but does not require the generation of the chip maze. This algorithm also simplifies the traversal of the chip-maze which reduces the need for additional resources.

4.3. SneakySnake-CIM

SneakySnake-CIM (SS-CIM) sees many similarities to previous works. Like most existing works, it makes use of the following observations [6]:

1. If two strings differ by e edits, then all non-erroneous characters of the strings can be aligned in at most e shifts.

2. If two strings differ by e edits, then they share at most $e+1$ identical sections.

SS-CIM creates $2e+1$ shifted Hamming masks to account for e shifts to the left and right. Like SS-FPGA, the Hamming masks are subdivided into multiple segments, each with length T. SS-CIM differs from SS-FPGA in that it does not count the number of edits in each segment, but rather detects the presence of any edit. In doing so, the results of the two algorithms only differ if multiple errors occur in the same segment of the read.

To implement this idea, edits in the segments are detected using the following observation: if the section of the read that is processed in one sub-problem contains no edits, at least one of the Hamming masks of that segment must be free of errors. This means that we can check whether any of the Hamming masks belonging to the segment contains only '0's. This allows us to detect the number of segments without edits.

By subtracting this from the total number of segments, we find the number of segments that do contain errors. This process is explained in pseudo-code in Algorithm 7. Here T represents the segment size and E the absolute maximum edit-distance. The main advantage of this approach is that every Hamming mask corresponding to the different shifts can be processed independently. This removes the need to collect all Hamming masks for the creation of the chip-maze and removes the iterative nature of the chip-maze traversal step. This, along with the segmentation into sub-problems, makes this algorithm suitable for implementation on a CIM-P architecture.

Algorithm 7 SneakySnake-CIM

```
Input: Read, Ref, E, Length, T
Output: Accept
  N_seg ← ⌈Length/T⌉
  Matches ← 0
  for i ∈ {0 ... N_seg − 1} do
      Match ← 0
      for e ∈ {−E ... + E} do
          ReadSeg ← Read[Ti ... T(i + 1) − 1]
          RefSeg ← Ref[Ti + e ... T(i + 1) − 1 + e]
          if ReadSeg == RefSeg then
              Match ← 1
          end if
      end for
      Matches ← Matches + Match
  end for
  Accept ← (N_seg − Matches <= E)
  return Accept
```

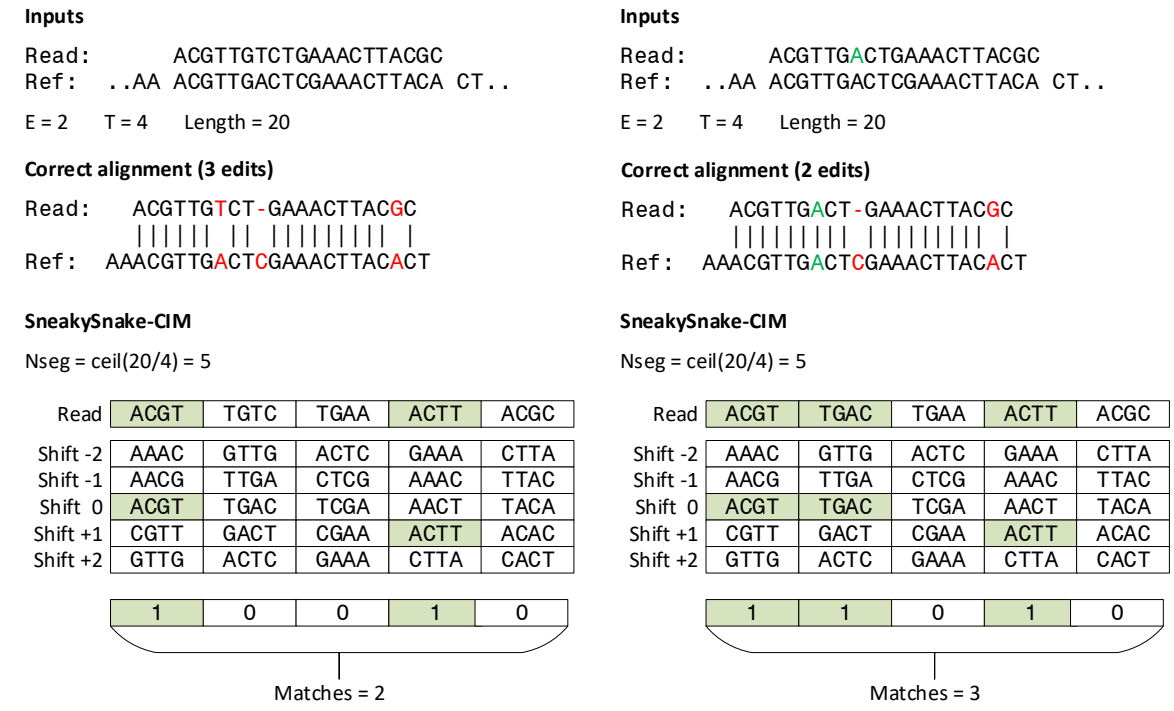
We present three examples of the algorithm in Figures 4.1(a)-(c), which show how the algorithm works on a true-negative, a true-positive, and a false-positive read-reference pair, respectively. In these examples, a read with a length of 20 bps is evaluated for an edit-distance of 2 using a segment length of 4 bps. The correct number of edits in the examples is obtained through DP-based alignment.

The first example in Figure 4.1(a) is of a true-negative mapping which contains 3 edits (2 substitutions and 1 deletion). Using SS-CIM, the read is divided into 5 segments of 4 bps, which are compared to references shifted by −2 to +2 ($-e$ to $+e$) base pairs (also divided into segments of 4 bps). Segments with exact matches with at least one of the reference segments are colored green and represented by a '1' in the final bit vector. In total, this pairing has two segments that are free of edits, meaning that $5-2=3$ segments do contain errors. This exceeds the edit-distance threshold of 2, meaning that the pairing is rejected by SS-CIM.

In Figure 4.1(b), the same read/reference pair is evaluated, except one of the substitutions is removed. In this example, only 2 edits are detected and therefore the pairing is correctly accepted. The last example in Figure 4.1(c) demonstrates a false-positive pairing. In this example, two substitutions occur in the same segment and are counted as a single edit by SS-CIM. Therefore, the algorithm finds only 2 edits, whereas the correct mapping contains three edits. It thus draws the incorrect conclusion that the pairing should be accepted, while it actually should be rejected.

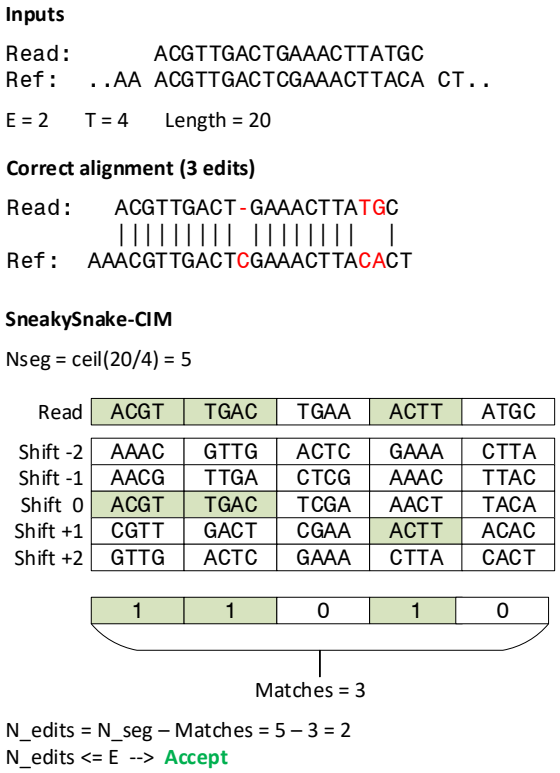
Abstracting from the number of reads in the segment thus ultimately leads to a decrease in filtering accuracy compared to SS-FPGA, as SS-CIM underestimates the number of errors when a single segment contains multiple edits. While this approach increases the false-positive rate of the algorithm compared to SS-FPGA, it does maintain a zero false-negative rate.

Furthermore, we observe that it is unlikely that two edits exist in the same segment when the segments are small and the read contains a small number of edits relative to its read length. This makes it so that the decrease in accuracy can be overseen as long as the filter is still able to distinguish true mappings (similar) from obviously false mappings (dissimilar) and enough speed-up is achieved using the CIM solution. To determine the maximum acceptable decrease in accuracy for which we the loss of accuracy can be compensated for, we linearly extrapolate the alignment time of SS-GPU to match its total end-to-end execution time. The number



(a) True negative

(b) True positive



(c) False positive

Figure 4.1: An example of a true negative mapping (a), a true-positive mapping (b), and a false-positive mapping (c) using SS-CIM with a read length of 20 bps and an edit threshold of 2.

of pairings that pass the filter is then scaled with the same factor. This gives an estimation of the minimum accuracy at which SS-CIM can provide end-to-end execution time improvement over SS-GPU, when fully exhausting Amdahl's law (zero filtering time). The maximum acceptable positive rate for short-read datasets and edit-distances can be found in Figure 4.2.

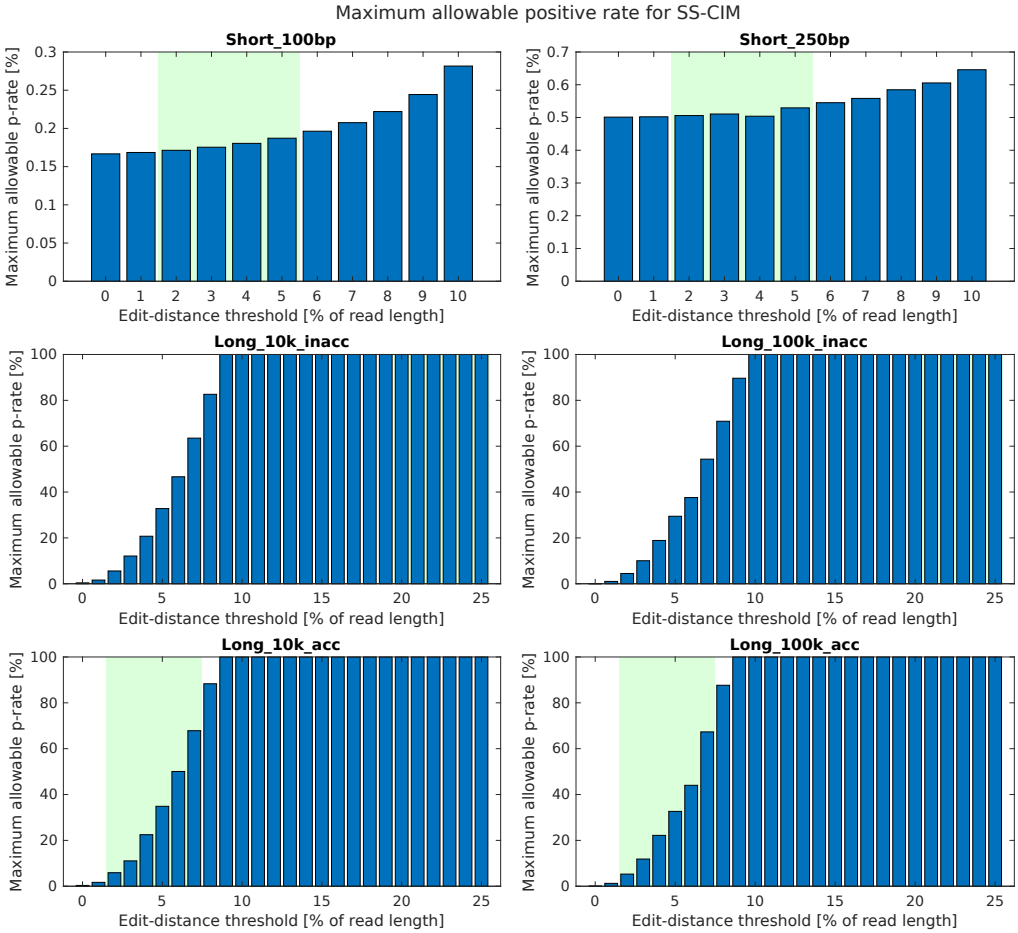


Figure 4.2: The maximum allowable total-positive rate for short reads. The accuracy of SS-CIM must not exceed this value to be able to achieve end-to-end improvement over the state-of-the-art.

Another limitation of the algorithm is that the number of edits that can be detected is limited to the number of segments, which makes it unsuitable for large edit-distances. However, as was found in Section 3.2, SneakySnake is mostly targeting accurate reads which require low edit-distance thresholds. In these types of read, the required number of edits can be detected if the segment size is chosen small enough.

4.4. Accuracy comparison

In this section, we measure the accuracy of SS-CIM for different segment sizes and compare it to the accuracy of SS-CPU, and show that the accuracy does not exceed the maximum allowable positive rate. To do so, first, the optimal segment sizes are found through experimentation. To decrease the likelihood of multiple edits occurring in the same segment, we would like to minimize the number of base pairs per segment. However, decreasing the segment size increases the probability that random matches occur. We define random matches as matches between bases that are not part of the actual alignment. These matches obscure would-be edits in the edit-distance estimation, leading to false positives. Assuming a random distribution of the four types of base pairs, the likelihood of random matches in between two sequences of length n can be described as $(\frac{1}{4})^n$. This causes a trade-off between minimizing the likelihood of multiple-edit segments and minimizing the likelihood of getting random matches. To determine the optimal segment size, we implement the algorithm in C++, and we compare the accuracy of the algorithm for different segment sizes ranging from 2 to 20 bps. This is done for all evaluated datasets and edit-distances. The result of this comparison for short reads

can be found in Figure 4.3 (note that only the segment sizes that show optimality in the tested edit-distance range are shown for clarity).

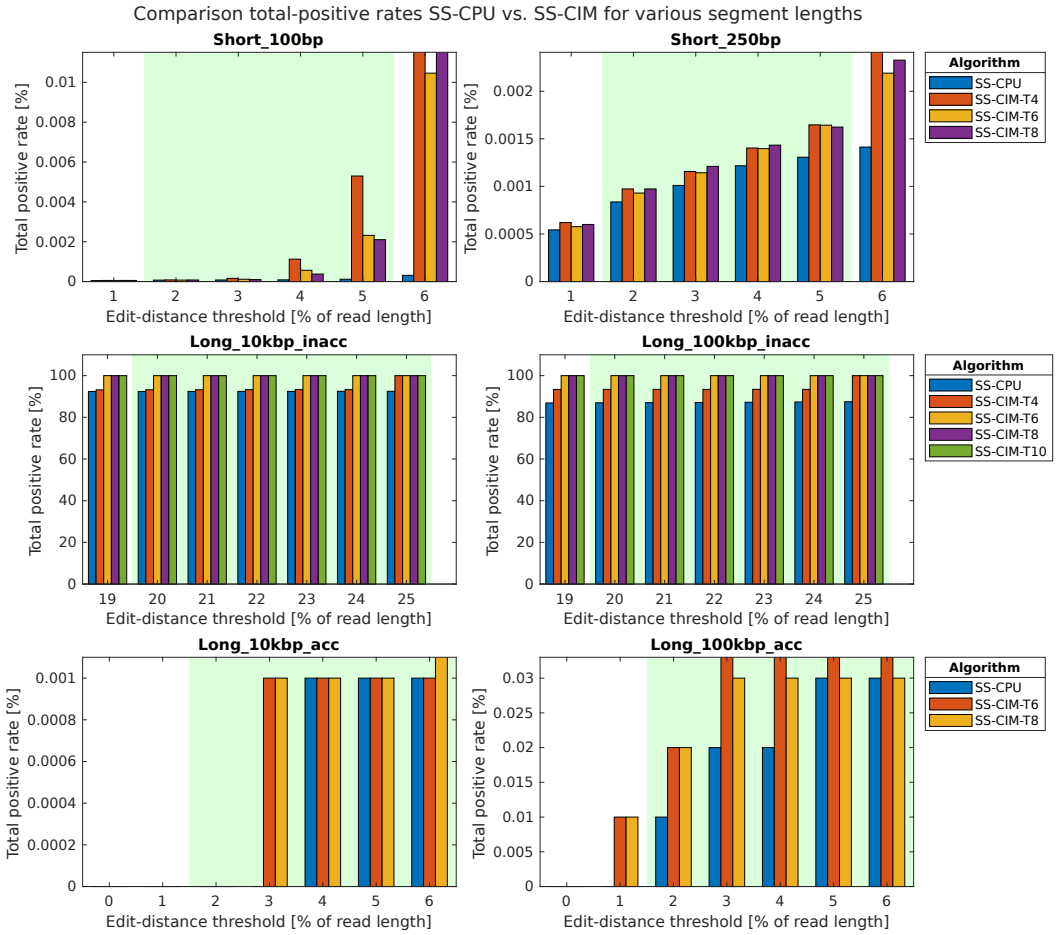


Figure 4.3: Comparison between SS-CPU and SS-CIM accuracies with different segment lengths.

Here, it is found that the best accuracy for short reads is achieved by using a segment length ranging from 4 to 8 bps, with 6 bps showing (close to) optimal performance in the entire region of interest. We find that for small edit-distances, the accuracy of SS-CIM tends to closely follow the accuracy of SS-CPU and that difference between total positive rate increases significantly at edit-distances over 4%. Furthermore, we observe that a smaller segment size (<4 bps) leads to an increase in false-positive rate, which can be attributed to a large number of random matches. Similarly, any larger segment size (>10 bps) also reduces accuracy, which can be explained by the increased likelihood of multiple edits occurring in the same segment.

A similar trend can be observed for long reads. For inaccurate long reads, the differences are small due to the fact that SS-CPU already has a large positive rate. In the region of interest, the optimal segment size is 4 bps, as this is the largest segment size that has enough segments to count the required number of edits. For accurate reads, the optimal segment size varies with the edit-distance, with a segment size of 8 bps being optimal in the region of interest.

In Figure 4.4, the P-rate of the best-performing segment size is compared to the maximum-allowable P-rate (FP-rate + TP-rate) as found in Figure 4.2. Here it can be seen that the positive rate of SS-CIM never exceeds the maximum allowable false positive rate in the region of interest for all datasets. This means that, if the filtering time is reduced by a large enough margin, the loss of accuracy can be compensated for. Additionally, we observe that for some edit-distances outside the region of interest, the P-rate does exceed the maximum-allowable P-rate. In these cases, SS-CIM is not accurate enough to provide speed-up over the baseline implementation. A possible to this degradation accuracy is to cascade SS-CIM with baseline SneakySnake, where

SS-CIM is used as a coarse filter, followed by the slower, but more accurate baseline solution.

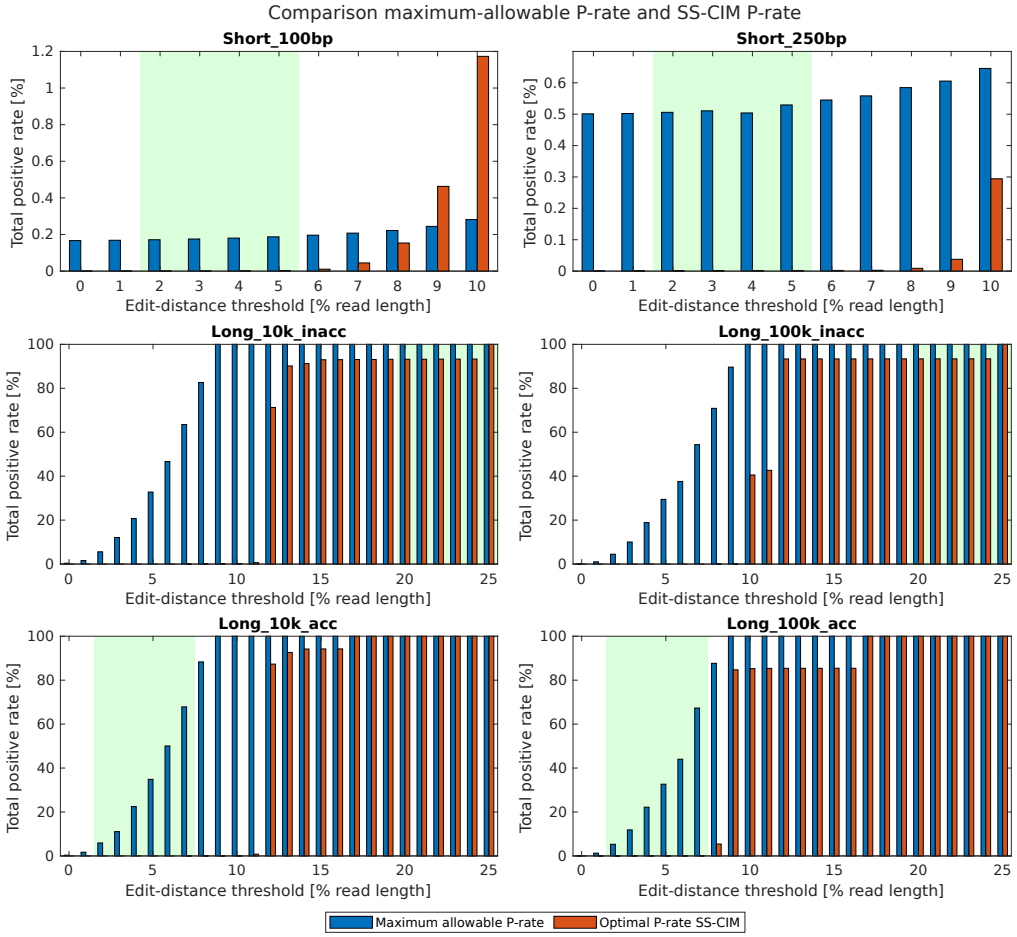


Figure 4.4: Comparison between the maximum-allowable P-rate and the P-rate achieved with the best-performing segment size of SS-CIM.

4.5. Profiling

The proposed algorithm is profiled with the same procedure as is used for previous algorithms in Chapter 3. The results of this profiling can be found in Figure 4.5. Here we observe the same trends as in Figure 3.11, where the majority of the execution time can be attributed to the bitwise-XOR and pattern detection operations. This suggests that SS-CIM can benefit from the same kind of architecture as the other algorithms.

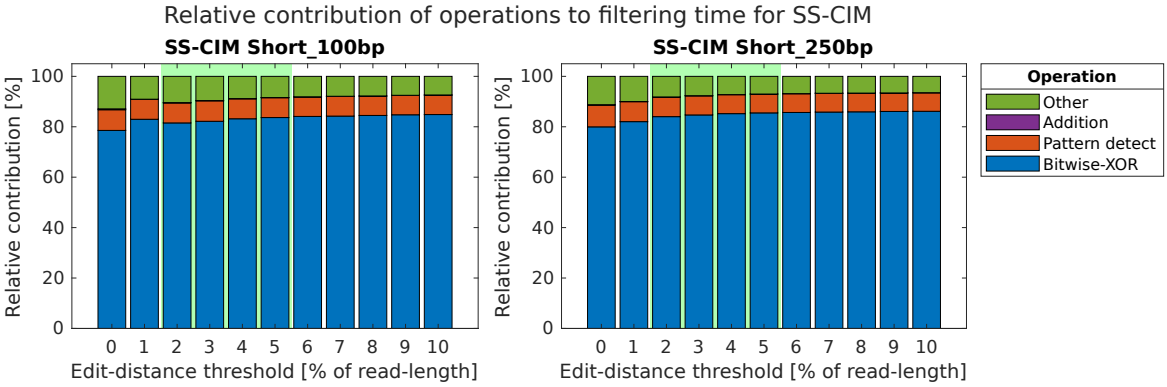


Figure 4.5: Relative contribution of various operations to the execution time of SS-CIM.

4.6. Conclusions

From the findings in this chapter, it can be concluded that it is possible to transform SneakySnake into a CIM-friendly algorithm: SneakySnake-CIM (SS-CIM). This local filtering algorithm divides the approximate string-matching problem into segments that can be evaluated independently of each other. The adjusted algorithm can be mapped onto a CIM-P architecture without the addition of large amounts of peripheral logic due to the simplification of the maze-traversal step. This algorithm maintains full sensitivity, as all true positives are maintained, and no false-positives are introduced.

SS-CIM does show a worse false-positive rate compared to baseline SneakySnake. However, we show that, in the region of interest, the increase in alignment time due to this worse accuracy can be compensated for if enough speed-up is achieved by the acceleration of the filtering stage. Furthermore, we find that the optimal segment size for SS-CIM varies for different edit-distance thresholds and datasets. The architecture that implements SS-CIM must therefore be reconfigurable to support various segment sizes ranging from 4 bps to 10 bps to obtain the best possible accuracy.

Chapter 5

Architecture

In Chapter 4, we propose an adjustment to the SneakySnake algorithm such that it can be accelerated efficiently using a CIM-architecture. Here we also establish that the loss in accuracy of this algorithm can be compensated for if the CIM-solution is sufficiently fast. In this chapter, we discuss the CIM-enabled hardware architecture used to accelerate SS-CIM. This architecture is designed around Scouting-logic-enhanced memristor-tiles to enable in-memory boolean operations and implements other operations of the pre-alignment filter using peripheral devices. It implements control logic for reading and writing data to the memory elements, as well as for the execution of the filtering algorithms. To achieve flexibility, the architecture is designed such that it can be programmed to support multiple different filtering algorithms and a wide range of datasets/edit-distance thresholds.

We provide a top-level overview of the system in its entirety in Section 5.1, followed by a detailed per-level description of each of its components in Sections 5.2 to 5.5. This is followed by a description of how the architecture should be programmed to map SHD and SS-CIM in Section 5.6. Furthermore, we discuss the modifications to the architecture required to support long reads in Section 5.7. Lastly, we summarize the main findings of the chapter in Section 5.8.

5.1. Top-level Overview

In this section, we provide an overview of the different components of the architecture along with their interconnections and data flow. An illustration of the structure of the architecture can be found in Figure 5.1. The overall structure is based on memory hierarchies found in many of today's DRAM-based memories and is augmented with peripheral devices to enable the operations required for pre-alignment filtering. It employs a multi-level hierarchy to improve resource utilization and consists of multiple bank-groups, banks, sub-arrays, and tiles (in order of decreasing levels of abstraction).

At the lowest level, the tile is composed of a memristor-array, which acts as the storage element of the design. The memristors are arranged in a 1T1R crossbar structure with multiple rows/columns in which the memristor devices act as interconnects. Its periphery has circuitry for the reading and writing of data to and from the crossbar, including multiple write-drivers, sense-amplifiers, row/column decoders, and multiplexers. Also, the sense-amplifiers are modified to support Scouting logic to enable CIM-P boolean operations.

Several of these tiles are grouped in sub-arrays, which contain logic for the execution of pre-alignment filtering algorithms including logic gates, counters, and TCAMs. At the sub-array level, the outputs of the tiles are combined to share sub-array-level devices, which improves resource utilization and reduces area overheads. These sub-arrays form the main computation units of the system and solve sub-problems independently of the rest of the system in a parallel fashion. They also contain input and output buffers required to reduce the stalling of the pipeline when the sub-array itself or its output-bus is occupied.

Multiple sub-arrays are grouped as banks, which in turn are grouped in bank-groups. Both of these levels implement an additional layer of input and output queues, to provide a low level of contention and to improve data flow. We speak of contention when a read-reference pair needs to be processed by the same sub-array while it is busy with the computation of another read-reference pair. The input/output queues also help reduce communication overheads between each level of the architecture.

At the highest level, the rank-level, inputs from the host device are processed in the correct format and sent to

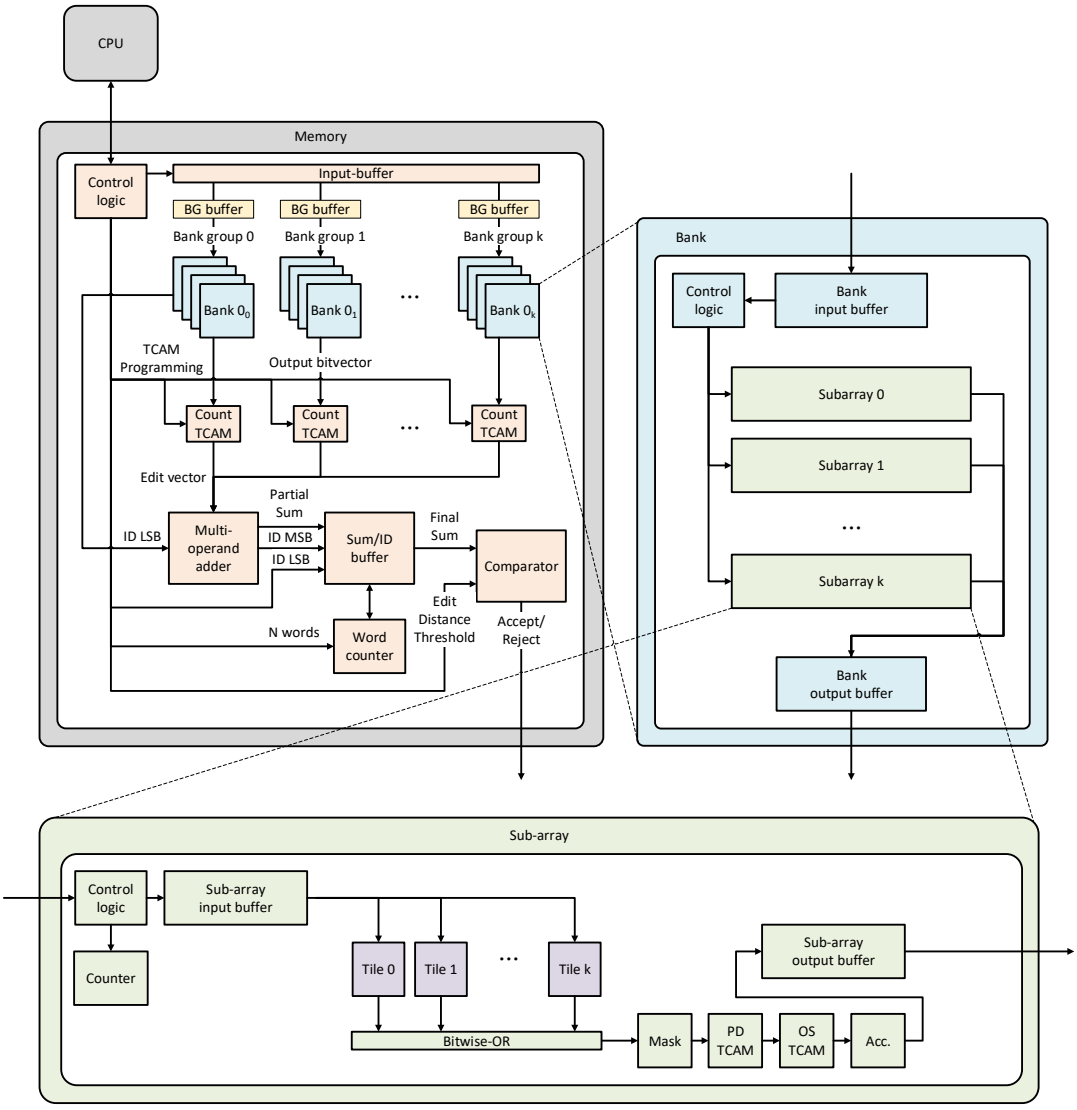


Figure 5.1: Top-level overview of the proposed CIM-architecture.

the appropriate memory locations. Also, the outputs of the bank-groups are combined, which are processed by several TCAMs, adders, and comparators to find the required output (i.e., the edit-distance estimation). Additionally, the buffering of inputs and the output are controlled, and the pipeline is stalled when necessary using this component.

In the remainder of this chapter, we go over each level of the architecture individually in more detail, and we explain the design choices made for each stage. This is done in order of increasing levels of abstraction.

5.2. Tile Architecture

The lowest level of the architecture is the tile level, which is discussed in this section. The tile is the main memory component and implements the bitwise-parallel XOR operation. The tile implements three main functions: write, read, and bitwise-parallel XOR. The read and write operations are required for the architecture to function as regular memory. For algorithm execution, the write operation is used to write the operands (read and reference sequences) to the appropriate memory locations. Here, the XOR-operation is performed between n-bits of the read and n-bits of the reference sequence, where n indicates the number of sense-amplifiers in the tile.

The underlying architecture of the tile can be found in Figure 5.2. Besides a clock and reset signal, the tile has

3 additional inputs which are provided by the sub-array controller. The 'data-in' signal contains the data that needs to be written to the crossbar and contains n binary bits. The instruction signal determines the behavior of the tile controller, selecting whether the tile should be idle, read, write, or perform an XOR operation. The address signal is used to index the correct rows and columns of the crossbar to/from which the data should be written/read. The tile has a single output signal, which is for output data. Like the input, this consists of n bits.

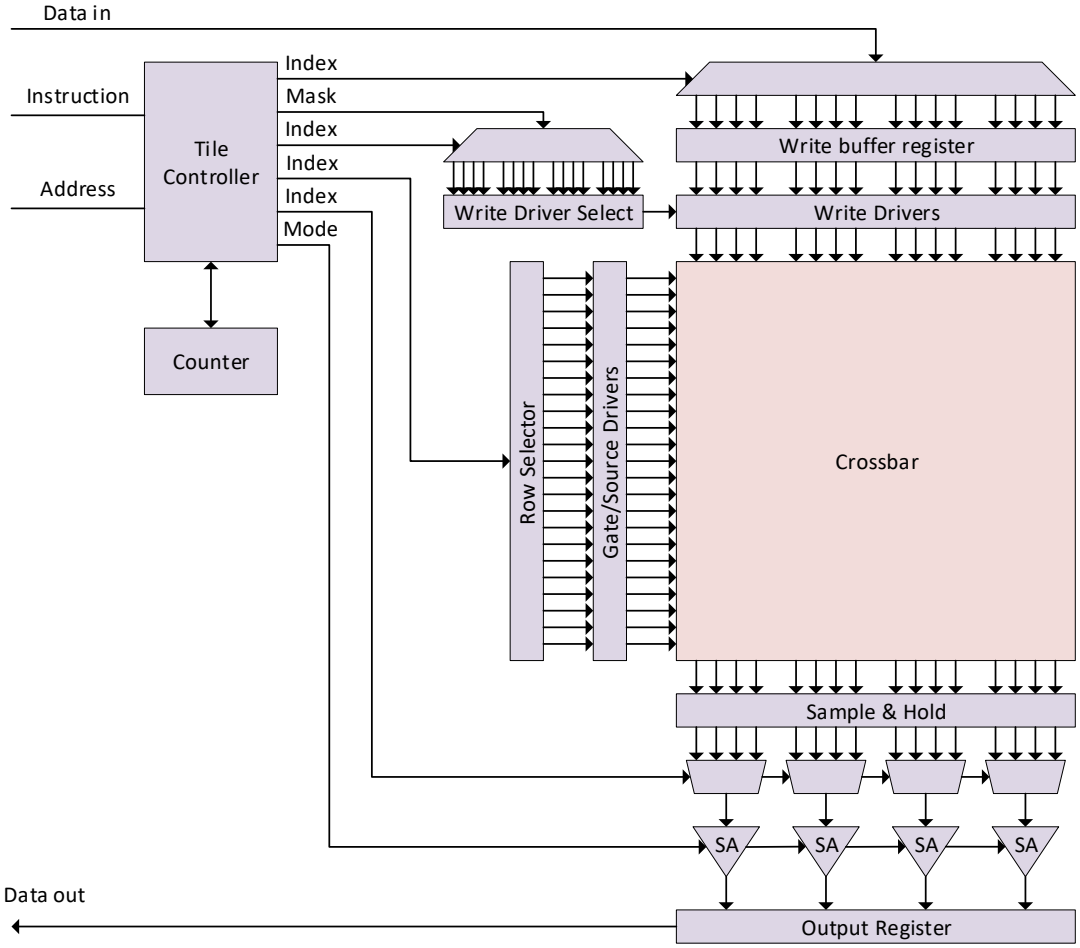


Figure 5.2: Tile-level architecture, consisting of the crossbar and its peripheral devices.

The main component of the tile is the crossbar, which consists of rows and columns of 1T1R memristors, as described in Scouting logic [14]. The crossbar is controlled by several peripheral components, which act as an interface between the digital architecture and the analog crossbar. These devices are also used as intermediate storage of input data to diminish overheads originating from the difference in timing between the read/write time of the crossbar (~ 10/100 ns) and the clock period of the digital components (~ 300 ps as demonstrated later on in this work).

In the remainder of this section, we discuss each of the operations that the tiles need to perform. Firstly, the writing process is discussed, which is done using the 'write' instruction. This operation is performed by first writing the input data to an array of write-buffer-registers, which together have the same width as the number of columns in the crossbar. An index is generated by the tile controller based on the input address, which is used to select the registers corresponding to the addressed columns using a de-multiplexer. During the write operation, the register values are written to the crossbar using write drivers, which apply the correct read-voltage to the bit-lines of the selected columns. Since not all columns are addressed in a single write cycle, a write-driver-select signal is required that indicates which write-drivers need to activate. This signal is again created using the address and a de-multiplexer. To select the correct row, a similar procedure is applied to generate a row-selector signal, which selects the gate/source drivers of the addressed row.

Secondly, to perform a 'read' instruction, a read-voltage is applied to the addressed row. The current that flows through the memristors as a result of Ohm's law is used to charge a sample-and-hold circuit at the crossbar output. The voltage captured using this circuit is used as an input to sense-amplifiers, which compare it to a reference voltage to determine the contents of the addressed memory cells. Since the sense-amplifiers are large compared to the 1T1R cells [14], the sense-amplifiers are shared amongst multiple columns through interleaving. Here the bits corresponding to a single write operation are stored in non-adjacent columns so that they can be read using different sense-amplifiers. In this architecture, interleaving is implemented within the column multiplexing logic. The correct memory rows are selected by indexing a series of de-multiplexers at the output of the sample-and-hold circuit. The digital output of the sense-amplifiers is placed in an output register. From these registers, the output can be accessed by the sub-array controller.

Lastly, during an XOR operation, the row selectors are indexed such that two rows are selected. The first operand of the operation is stored in the first row of the crossbar (referred to as the query row), and the second operand is the row that is indexed by the address decoder of the tile controller. The resulting current from both rows is combined through Kirchhoff's law and sensed by the same sample-and-hold circuit and sense-amplifiers. To differentiate between a regular read operation and an XOR operation, an additional control signal is provided by the tile controller to change the mode of the sense-amplifiers. This signal changes the (set of) reference signal(s) of the sense-amplifier according to what operation is required, as described in Scouting logic.

Since the analog read/write operations take multiple clock cycles, a counter is implemented to ensure that the read/write voltages are applied for the appropriate duration. This ensures that the memristor cells receive a correct resistance state during writes and that the sample-and-hold circuit is not over/undercharged during the read/XOR operation.

The low-level operations of the tile controller are performed based on an instruction signal generated by the sub-array. This instruction can have one of the five values listed in Table 5.1. Here, the expected input/output data is listed along with the expected behavior.

Table 5.1: Tile-level instructions

Instruction	Data in	Data out	Comments
Idle	-	-	Disables the tile when not in use.
Fill-WB	Write data	-	Fills the write buffer/mask to be programmed simultaneously with other tiles.
Write	Write data	-	Writes n-bits from the write-buffer inputs to the crossbar.
Read	-	Read data	Reads n-bits from the target row/column.
XOR	Query data	XOR (Ref, Query)	Performs XOR operations between n-bits of the query and n-bits of the reference.

5.3. Sub-array Architecture

The second lowest level of the design is the sub-array level. A single sub-array contains multiple tiles working in parallel to produce segments of the XOR result. The sub-array translates the bitwise XOR results from the tiles into base-pair results, which are used to perform the pattern detection and bitwise-AND operations found in the pre-alignment algorithms. Furthermore, the sub-array implements the logic that handles the masking of tile results that are not part of the read/reference pairings and keeps track of what read/reference pairing is being processed using an ID signal. An overview of the architecture that implements these functions is shown in Figure 5.3. To explain the functionality of each component, we go over the process of computing a single sub-problem of a filtering algorithm step-by-step.

To perform the computations, first, the input operands (read/reference sequences) are loaded using write operations. The length of the operands is determined by the minimum sub-problem size (segment size T in SS-CIM) and is equal to $2 * T$ to account for the 2-bit encoding scheme of the base pairs in Table 5.2. Because the number of sense-amplifiers in each tile is limited, the operands are split up and written to multiple tiles. This way, each tile can compute the XOR result in parallel in a single computation period, rather than

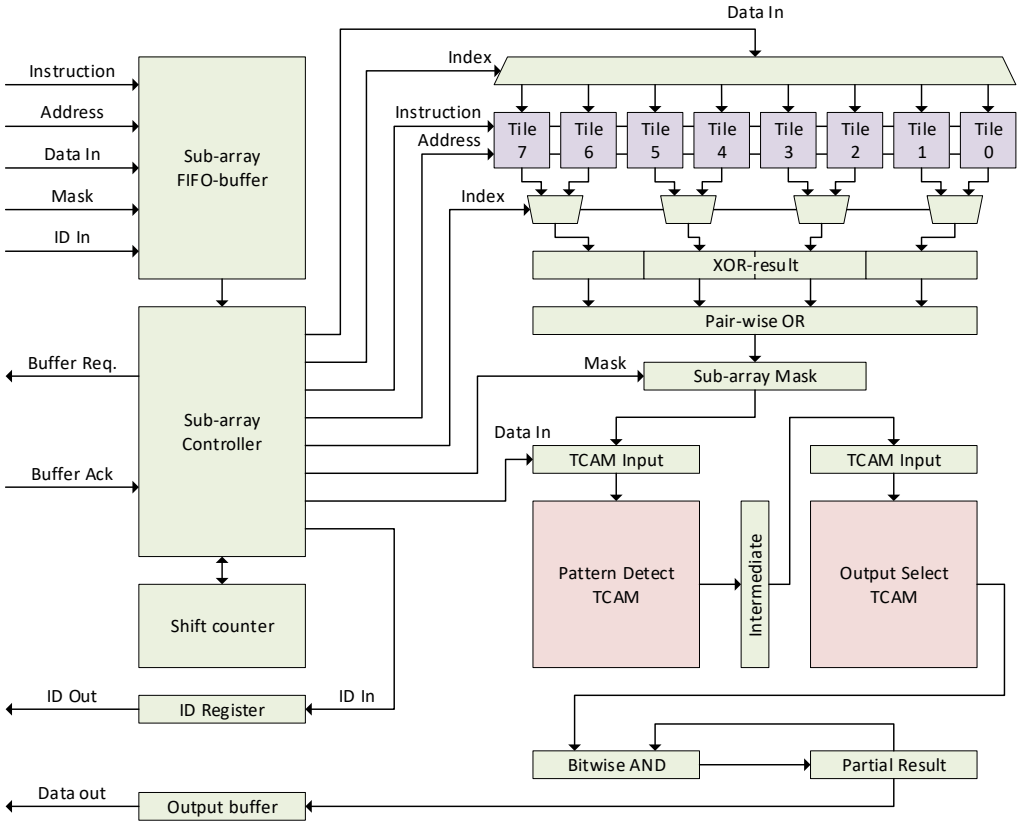


Figure 5.3: Overview of the sub-array level architecture.

sequentially over multiple iterations. This reduces computation latency and increases resource utilization. The partial operands are first written to the write buffer registers of each tile individually and are then programmed to the crossbar simultaneously. This makes it so that multiple tiles can use the same counter to keep track of the write period. The correct tiles are selected by the input address, which is converted into an index of the tile de-multiplexer by the sub-array controller.

Table 5.2: The 2-bit encoding scheme for all base-pairs found in the human genome.

Base pair	2-bit encoding
A	00
C	01
G	10
T	11

Once the operands are loaded into the tiles, the sub-array can perform the computation of the final bit vector in several steps. First, the XOR-operation is performed between the read sequence and one of the shifted reference sequences by giving an XOR instruction to the tiles. After the XOR-computation period, the results are retrieved from the output buffers of all selected tiles by indexing a series of multiplexers. These results are then combined into a single XOR-result register. From there, a series of OR-gates transforms the bit-level XOR results into a bit vector which represents the XOR operation between the base-pairs of the partial sequences.

Since not all bits of this OR-result are part of the actual read-reference pairing, they are masked off by performing a bitwise AND-operation with the sub-array mask. The masking procedure is explained in more detail in Section 5.6.

The masked OR-result is used as an input to the first of a pair of Ternary Content Addressable Memory (TCAM)s. This Pattern-Detect Ternary Content Addressable Memory (PD-TCAM) can be programmed to detect patterns of ‘1’s and ‘0’ in its input, depending on what is required by the algorithm. Each row is composed

of a series of '0's, '1's, and 'X's (don't-cares). If this pattern matches with the input of the TCAM, the output corresponding to this row is set to '1'. The results of each row are collected in the intermediate result buffer and represent each detected pattern. This is then used as input for the Output-Select Ternary Content Addressable Memory (OS-TCAM). This TCAM is responsible for constructing one partial result of the final bit-vector based on what patterns are detected by the PD-TCAM.

The output of the OS-TCAM is considered as one iteration of the partial final bit vector and is stored in the partial-result register. The same steps are repeated for all shifted reference sequences stored in the tiles. For each iteration, the shift counter is incremented by one, and the output of the OS-TCAM is accumulated by performing a bitwise AND-operation between the previous value of the partial result register and the OS-TCAM output of the current iteration. When the shift counter reaches a value of $2e+1$, all shifted references have been evaluated. The partial result is then presented at the sub-array input.

5.3.1. Sub-Array Queue

Since the computation process takes many clock cycles to complete, multiple sub-arrays operate in parallel to independently compute different read-reference pairings to improve system throughput. Because each sub-array stores a different part of the reference, the input dataset determines which sub-arrays are required. It is thus possible that a read-reference pairing requires a sub-array that is still busy computing a different pairing, which creates contention over the sub-array. Without an input queue, this would stall the system until the sub-array is freed up. To combat this, each sub-array contains a FIFO queue that stores all the input signals for the computation of a read-reference pairing until the sub-array finishes computing its current pairing. This eliminates the need to stall the rest of the pipeline until the FIFO queue is full.

Because of differing stalling patterns of sub-arrays due to the FIFO queue, it is possible that the order in which the computation of pairings finishes differs from how they are supplied to the system. To keep track of what pairings the results belong to, each pairing is assigned an ID, which is presented alongside the sub-array output. Another problem introduced by the input queue is contention over the output bus, which occurs when multiple sub-arrays finish their computation simultaneously. To resolve this contention, a request-acknowledgment scheme is implemented ensuring that outputs are read one after another without contention.

All of these processes are coordinated by the sub-array controller, which is implemented as a Finite-State Machine (FSM). The states of this controller are controlled by instructions supplied by the bank-level architecture. The 'idle', 'Fill WB', 'Write', and 'Read' operations essentially relay the inputs of the sub-array to the inputs of the tiles. Other sub-array-specific instructions are listed in Table 5.3. Various instructions are associated with programming peripheral devices in the sub-array to the values required by the algorithm. Firstly, 'Program Shifts' is used to set the number of iterations each sub-problem has to go through. 'Fill-TCAM-buffer' writes the input data to the TCAM input of the TCAM. This is done in several iterations to fill the entire width of the TCAM input. The input can then be programmed to the TCAM using 'Program TCAM 0s' or 'Program TCAM 1s'. As each cell in the TCAM has three possible values ('0', '1', or 'X'), the zeros and ones of the TCAM are programmed separately using the scheme in Table 5.4. For these instructions, the address is used to determine which one of the TCAMs is programmed. The way these TCAMs are programmed depends on the algorithm used and is explained in more detail in Section 5.6.

Table 5.3: Sub-array-level instructions

Instruction	**Data in**	**Data out**	**Comments**
Prog. Shifts	n_shifts	-	Program the total number of shifts for each sub-problem.
Fill-TCAM-Buffer	TCAM entry	-	Fill the TCAM input buffer with an entry.
Prog. TCAM 0s	-	-	Write the TCAM buffer to program the '0's of the TCAM rows.
Prog. TCAM 1s	-	-	Write the TCAM buffer to program the '1's of the TCAM rows.
Start Alg	Query data	Final BV	Writes the input data to the query row of the addressed tile and starts the algorithm.

Table 5.4: TCAM Programming scheme

Programmed value	Value during 'Program 0s'	Value during 'Program 1s'
'0'	'1'	'0'
'1'	'0'	'1'
'X'	'1'	'1'

5.4. Bank & Bank-group Architecture

The bank and bank-group levels are similar in functionality, as their main purpose is to increase the capacity of the memories while minimizing resource contention. The bank level combines multiple sub-arrays and adds an input and output queue. A bank-group has the same structure but instead groups multiple banks. This type of hierarchical structure is often found in current DRAM-based technologies [96] and we opt to adapt this structure for the proposed design for two main reasons. Firstly, splitting the two levels reduces the fan-out of the required busses of each stage, which reduces the minimum clock period. Secondly, the multi-level approach improves the contention over the lower-level resources without adding excessive amounts of buffer overheads. The underlying architectures of these two hierarchy levels can be found in Figure 5.4.

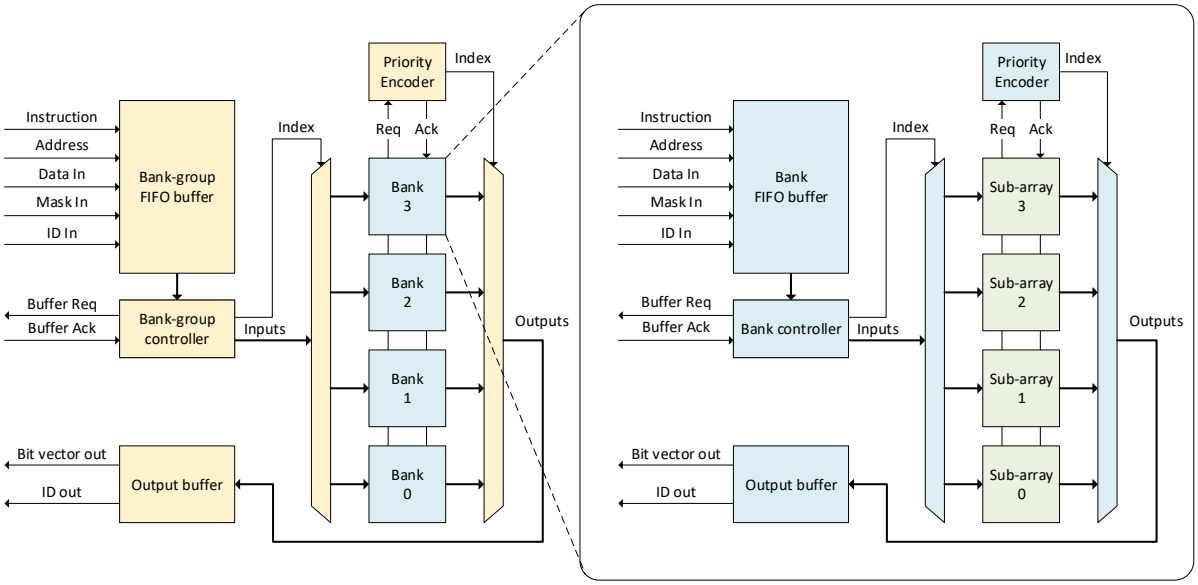
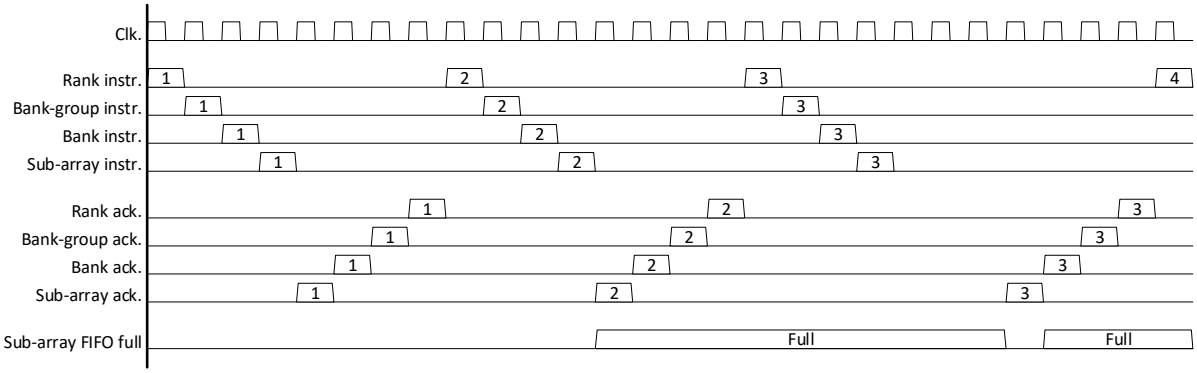


Figure 5.4: Overview of the bank-group and bank architectures.

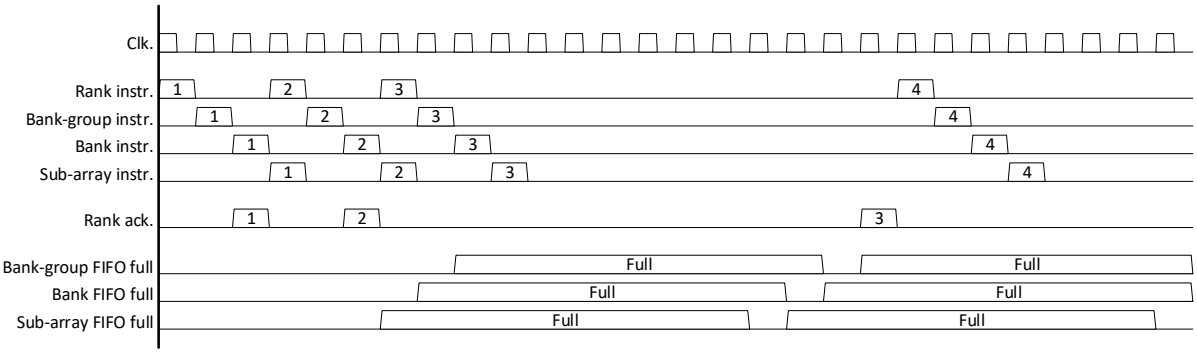
The bank-group receives its inputs (instruction, address, data, mask, and ID) from the rank level and relays these inputs to the targeted bank by indexing a de-multiplexer using the input address. The same process happens at the bank level to address sub-arrays.

For both levels of the architecture, a single-entry input queue is implemented. This decreases contention of the system by a small degree, but more importantly reduces the communication overhead between the different levels of the hierarchy. Since there is a possibility of contention occurring at the sub-array level, the top-level architecture must implement an acknowledgment scheme to determine whether or not the input should be stalled. Because the architecture consists of several layers, communication between the top level and the sub-array level happens over several clock cycles. The input queues reduce this latency by providing an acknowledgment signal after only a single clock cycle. Essentially, the 'FIFO full' signal acts as the complement of an acknowledgment signal. By implementing these queues, the latency of writing inputs to the sub-arrays is decreased when there is no contention over the addressed sub-array. This process is illustrated in Figure 5.5, where four instructions are relayed through the architecture without input queues (Figure 5.5(a)) and with input queues (Figure 5.5(b)).

As mentioned before, the multiple sub-arrays can provide outputs at the same time. To avoid contention



(a) Without FIFO queue.



(b) With FIFO queue.

Figure 5.5: Instructions and acknowledgments for all levels (a) without the queue, and (b) with the queue.

over the output bus, a request-acknowledgment scheme is implemented at the bank level. The bank receives output requests from the sub-arrays in the priority encoder and returns an acknowledgment to one of the sub-arrays after which the output of that sub-array is transferred to the bank. The output request of all other sub-arrays stays active until they have received an acknowledgment. The same structure is employed for the bank-group level to avoid contention by the bank outputs.

The bank and bank-group levels do not add any additional instructions and only relay the rank-level instructions to the sub-arrays.

5.5. Rank Architecture

Lastly, we discuss the rank-level architecture, which is the highest level of the architecture and is the interface between the host device and the CIM accelerator. It provides instructions, addresses, data, IDs, and masks to control the lower levels of the architecture, and collects the resulting outputs. Additionally, it keeps track of the read/reference pairings that are processed in the system and implements the edit-counting, summation, and comparison to the edit-distance threshold. An overview of the rank-level architecture is provided in Figure 5.6.

The inputs of the system are provided to the rank controller and include the rank-level instruction, address, input data, and pairing ID. The inputs serve different purposes depending on what instruction is given by the host device. Instructions are used to control the bank-groups, program the rank-level peripheral devices, and set algorithm parameters.

To explain the functionality of each component, we first go over the process of writing data to the tiles. The read/reference data is provided to the rank input controller over a data bus with a length equal to the product of the number of bank-groups and the number of sense-amplifiers per tile. One such data transfer is referred to as a 'word' and is divided over the bank-groups such that each part goes to a different tile, as is the case in DRAM-based memories [96]. This writing scheme makes it so that different parts of the read/reference are written to different sub-arrays. For the implementation of filtering algorithms, this has the consequence

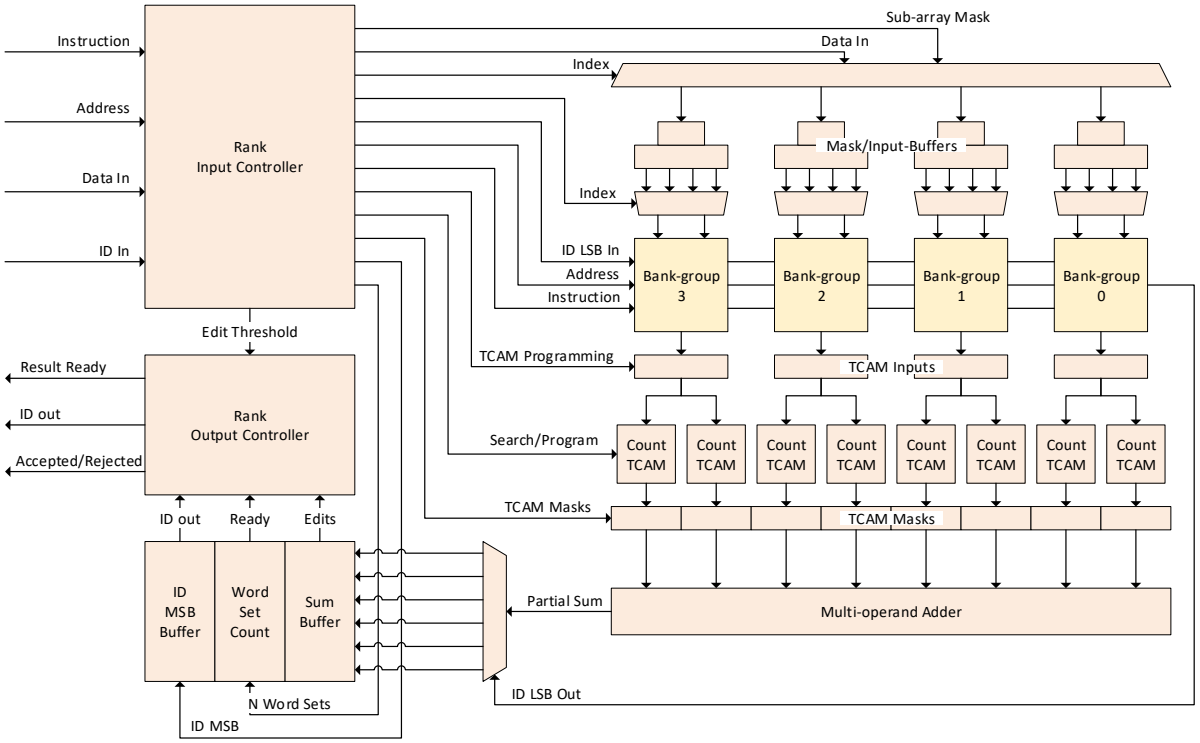


Figure 5.6: Overview of the rank-level architecture.

that each processing element only has access to a small part of the read/reference. Algorithms that require larger segments to be examined by a single processing element (e.g. SS-CIM for accurate long reads) therefore require multiple words to be written before starting the algorithm. This number of words required is called the Writes-Per-Bank (WPB), for further reference.

To support a WPB larger than 1, we implement a series of input buffers. These input-buffers are required for the host device to provide read/reference sequences in the correct order without the need for excessive pre-processing. To fill the buffers, consecutive words belonging to a read-reference pairing are loaded into the input buffer sequentially. Once the buffer is filled with one word-set, they are emptied in parallel, as is demonstrated in Figure 5.7. In this example, 4 words (1 word-set) are written to 4 bank-groups and each block represents a number of bits equal to the number of sense-amplifiers per tile. The number of required words/word-sets to write a single sequence of length l can be expressed using Equation 5.1, where 'W' stands for width in bits, and 'l' for the read length in bps.

$$\begin{aligned} W_{word} &= \text{SAs\_per\_tile} * \text{bankgroups\_per\_rank} \\ W_{wordset} &= \text{WPB} * W_{word} \\ n_{wordsets} &= \left\lfloor \frac{2 * l_{read}}{W_{wordset}} \right\rfloor + 1 \end{aligned} \tag{5.1}$$

Since in the CIM-design the first base pair of the read does not always coincide with the start of a word-set, the part that does not belong to the read-reference pairing is masked off. Therefore, a mask is passed to the sub-arrays alongside the read-sequence data. This mask is passed to the rank-level through the address bus, as this is not used during the filling of the input buffers. The masks are loaded using a process similar to the one in Figure 5.7. The exact implementation of this is discussed in Section 5.6. As the masking is based on base pairs, the mask only requires half of the bits as compared to the word-sets.

After all words of a word-set have been written to the appropriate tiles, the algorithm is started. Since the sub-arrays can operate independently, the rank-input-controller can load in a new word-set during the com-

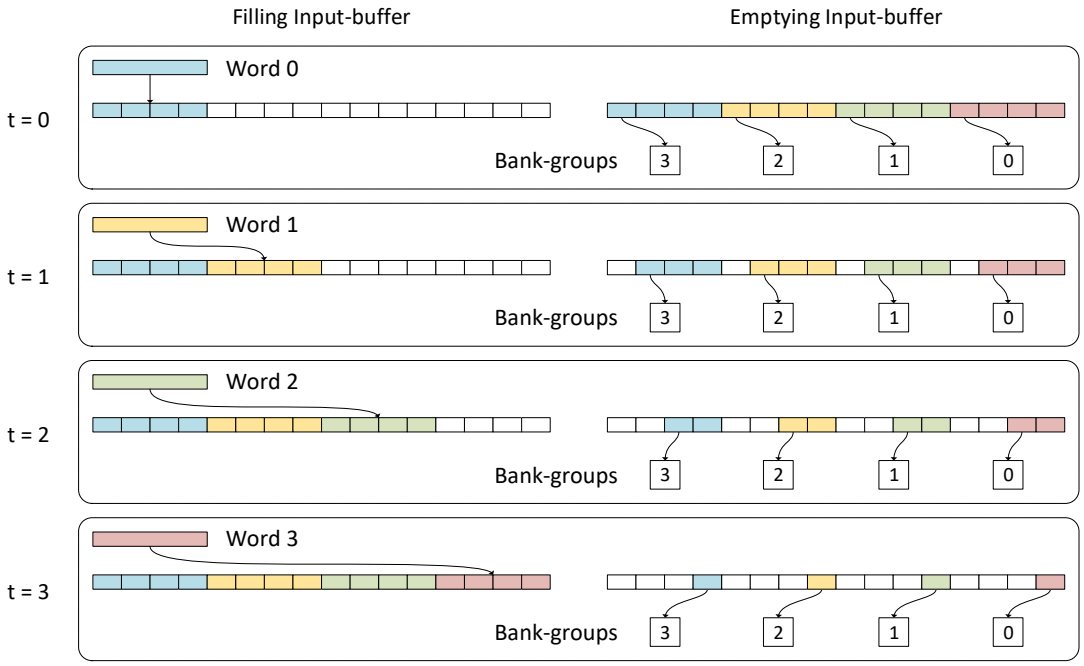


Figure 5.7: The process of filling (left) and emptying (right) the input-buffer.

putation of the previous word-set. Depending on the length of the read/reference pair, this can either be the next word-set belonging to the same pairing, or the first word of the next pairing. To be able to distinguish between different pairings, they are each given an ID number. The Least-Significant Bit (LSB)s of the ID are passed along with the input data to the sub-arrays, while the most significant bits are stored at the rank-level in the ID MSB buffer, which is indexed by the LSB of the ID. This is done to reduce the width of the ID busses. When the computation is completed, the results are returned, and the LSBs of the ID are matched up with the MSB again.

The outputs of the processing units contain a part of the final bit vector. Since different filtering algorithms have different ways of counting edits in the final bit-vector, the bit-vectors are passed through an array of TCAMs. These detect patterns in the final bit-vector, based on which it finds the number of edits in the partial final bit-vector. As different algorithms use different numbers of patterns, a mask is present to remove the outputs of the TCAM rows that are not in use. The masked TCAM outputs contain '1's in the bit vector that represent each edit. These are accumulated using a multi-operand adder. The output of this adder is the number of edits in only a single word-set of the read-reference pairing. Therefore, the partial sums are stored in a sum-buffer, and accumulated over all word-sets. The sum buffer is indexed by the output ID of the bank-group.

The word-set counter associated with the given ID is incremented by one for every processed word-set. If that number exceeds the number of required words, the total sum stored in the sum buffer is given as input to the rank-output-controller. This compares the counted number of edits to the edit-distance threshold, and based on the result it accepts/rejects the pairing. Then, the result-ready signal is activated, and the ID and the comparator result are presented in the system output.

The rank-level is controlled/programmed by several instructions given by the host device, which are listed in Table 5.5. The address that is provided to the rank-input-controller differs depending on the instruction. An overview of the addressing scheme is given in Figure 5.8. Here the unused part of the signal is blacked-out. During normal read/write operations, the signal indexes the targeted bank, sub-array, tile, row, and column of the memory elements. At the start of the algorithm, the same signal is applied, to indicate the comparison of the read (stored in the first row of the tiles) and the references (addressed row). To program the TCAMs, the targeted row is selected by the address. To distinguish between the PD-TCAM and OS-TCAM, an additional bit is used. Lastly, during the 'fill_input_buffer' instruction, the start and end bits of the mask for the given word-set are given alongside the index for the input-buffer.

Table 5.5: Rank-level instructions

Instruction	Input	Output	Comments
Idle	-	-	Deactivates the system when not in use.
Write	Word	-	Writes the input word to the addressed memory elements.
Read	-	Word	Reads a word from the addressed memory elements.
fill_input_buffer	Word	-	Fills one word of the rank-level input buffer.
write_input_buffer	-	-	Writes all input buffer entries to the addressed memory elements.
fill_SA_TCAM_buffer	TCAM entry	-	Writes the input to the addressed TCAM buffer of all sub-arrays.
program SA_TCAM_0s	-	-	Programs the TCAM buffer to the 0-cells of the addressed row of the addressed TCAM for all sub-arrays.
program SA_TCAM_1s	-	-	Programs the TCAM buffer to the 1-cells of the addressed row of the addressed TCAM for all sub-arrays.
fill_count TCAM_buffer	TCAM entry	-	Writes the input to TCAM buffer of the count-TCAMs at the rank-level.
program count_TCAM_0s	-	-	Programs the TCAM buffer to the 0-cells of the addressed row of the count-TCAMs.
program count_TCAM_1s	-	-	Programs the TCAM buffer to the 1-cells of the addressed row of the count-TCAMs.
program shifts	n_shifts	-	Programs the number of shifts to all sub-array controllers.
program edit_thresholds	e	-	Programs the edit-distance threshold of the rank-output controller.
program n_wordsets	n_word-set	-	Program the number of word-sets per read-reference pairing.
start_algorithm	-	-	Starts the algorithm in all addressed sub-arrays.

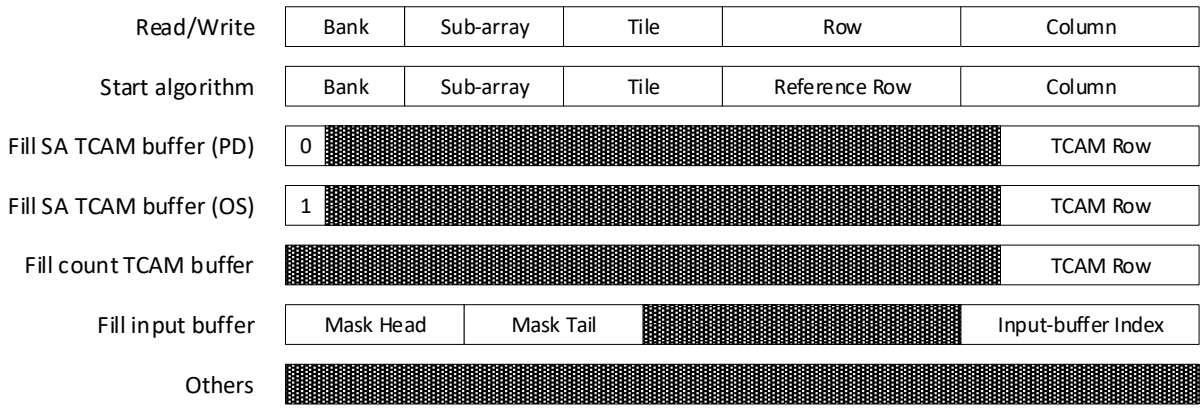


Figure 5.8: An overview of the addressing scheme for various rank instructions.

5.6. Algorithm Mapping

In this section, we explain how two pre-alignment filtering algorithms, SHD and SS-CIM, can be mapped onto the proposed architecture. The two algorithms can be mapped onto the same hardware architecture by programming the crossbars and TCAMs with the appropriate values. We provide an example of a read-reference pairing and walk through each of the steps required to find the approximate edit-distance between the sequences. Furthermore, we discuss how the hardware-implemented algorithms differ from their software counterparts, and the influence this has on their accuracies.

5.6.1. Read/Reference Mapping

The basic underlying principle for loading in sequence data is similar for both algorithms in that they both store the reference sequence in the memory as a pre-processing step and stream in the read sequences during runtime for evaluation of the algorithms. The reference genome is split up into word-sets which are written to the architecture consecutively as words. The contents of the input buffer are then divided over the bank-groups, which write the values to a set of tiles in a single sub-array/bank. Consecutive word-sets are written to different banks to avoid contention as much as possible. The sub-division of two consecutive word-sets of the reference is shown in Figure 5.9. In this example, we consider a hardware configuration with 32-bit words, 4 bank-groups, and tiles with 8 sense-amplifiers, and we assume a 4-WPB writing scheme.

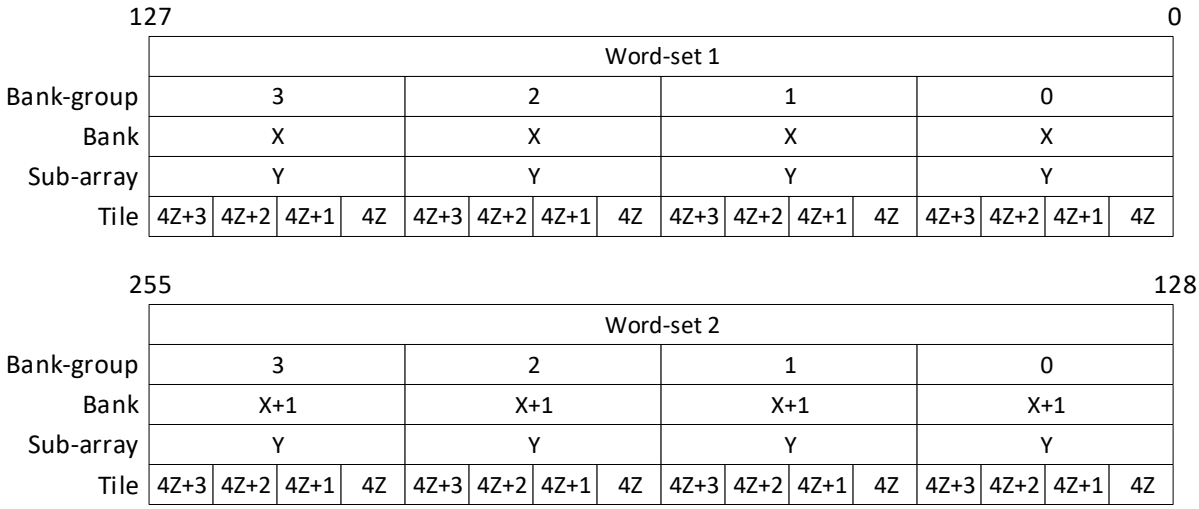


Figure 5.9: High-level writing scheme of two consecutive parts of the reference.

Within each tile, a part of a word-set is written to a single row in the memory, with shifted versions of the word-set in its adjacent crossbar rows. The first row of each tile is reserved for the read sequence, and is referred to as the 'query row'. This scheme is illustrated in Figure 5.10. This example pictures a small crossbar consisting of a 16×16 grid, containing parts of 6 word-sets which are evaluated for an edit-distance of 2. We highlight one word-set, which shows how the (shifted) references are stored. Note that this illustration does not account for interleaving for the sake of clarity. We adopt this writing scheme such that a single sub-array can generate XOR results for all shifted Hamming masks using only a single write of the read-sequence part.

The exact address for each word-set of the reference can be found using the expressions in Equation 5.2, where N_{bpbg} stands for bank-groups per bank, N_{spb} for sub-arrays per bank, N_{tps} for tiles per sub-array, and N_{sapt} for the number of sense-amplifiers per tile. Here the offset is defined as the position of the reference segment with respect to the first base pair in the evaluated reference genome. First, the word-set is determined by dividing the base-pair offset (as found by the seed-and-extend process) by the length of a word-set. This value is rounded down to find the word-set number. This is then used to determine what bank it should be written to by taking the modulus of the number of banks per bank-group. In doing so, consecutive parts of the reference are written to different banks so as to reduce contention. A similar method is used for the calculation of the sub-array number, where the word-set number is first divided by the number of banks per bank-group. Using this scheme, consecutive word-sets are first distributed over the banks, followed by different sub-arrays to reduce contention. At the tile level, multiple tiles are grouped together, such that partial words of each word-set are contained within a tile-group. The number of tiles in each tile-group is equal to

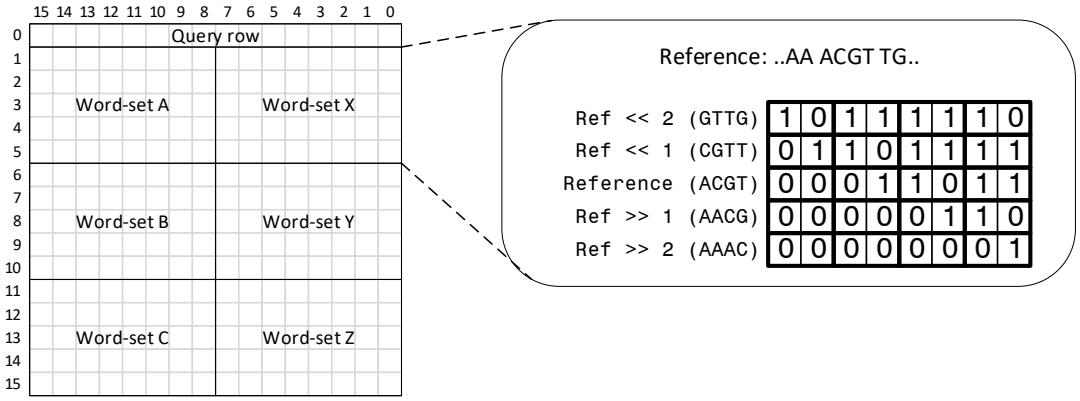


Figure 5.10: An example reference mapping on a small crossbar.

the WPB of the system. The rows of the tiles are grouped with the different shifts of the reference sequence, as demonstrated in Figure 5.10. Lastly, we iterate over the columns, with a similar approach as for the other levels.

$$
\begin{aligned}
\text{WS} &= \lfloor \text{Offset}/W_{wordset} \rfloor \\
\text{BankNum} &= \text{WS} \pmod{N_{bpbg}} \\
\text{SubarrayNum} &= \left\lfloor \frac{\text{WS}}{N_{bpbg}} \right\rfloor \pmod{N_{spb}} \\
\text{TilegroupNum} &= \left(\left\lfloor \frac{\text{WS}}{N_{bpbg} * N_{spb}} \right\rfloor \pmod{\frac{N_{tps}}{\text{WPB}}} \right) * \text{WPB} \\
\text{RowNum} &= \left(\left\lfloor \frac{\text{WS}}{N_{bpbg} * N_{spb} * (\frac{N_{tps}}{\text{WPB}})} \right\rfloor \pmod{\left\lfloor \frac{N_{rows}-1}{2e+1} \right\rfloor} \right) + \text{Shift} + e \\
\text{ColumnNum} &= \left(\left\lfloor \frac{\text{WS}}{N_{bpbg} * N_{spb} * (\frac{N_{tps}}{\text{WPB}}) * \lfloor \frac{N_{rows}-1}{2e+1} \rfloor} \right\rfloor \pmod{\left\lfloor \frac{N_{cols}}{N_{sapt}} \right\rfloor} \right) * N_{sapt}
\end{aligned}
\tag{5.2}
$$

In Figure 5.11, the subdivision of a read sequence into words and word-sets is illustrated alongside the masking process. In this example, we consider a 100 bps read encoded using a total of 200 bits. The read is divided over 2 word-sets of 128 bits. The address of the word-set containing the first bit of the read sequence is found using the statements in Equation 5.2. Since the start of the read does not necessarily coincide with the start of a word-set, the first part of the first word-set and the last part of the second word-set are masked off. The mask is determined by subtracting the offset of the first base pair of the word-set from the offset of the read (seeding location). This local offset is then used to find the length of the leading mask. The read is always written to the query row of the tiles such that the columns line up with the addressed part of the reference sequence.

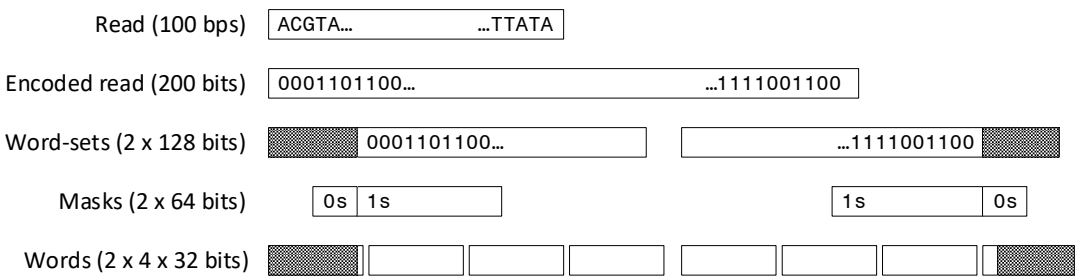


Figure 5.11: Overview of the subdivision of reads into words and word-sets, along with the masks.

5.6.2. Algorithm Execution

When the reference is loaded into the memory elements and the evaluated read-sequence is written to the query row of the mapping location found by seed generation, the algorithm is executed. The final bit vector is generated entirely in the addressed sub-arrays, using the process illustrated in Figure 5.12.

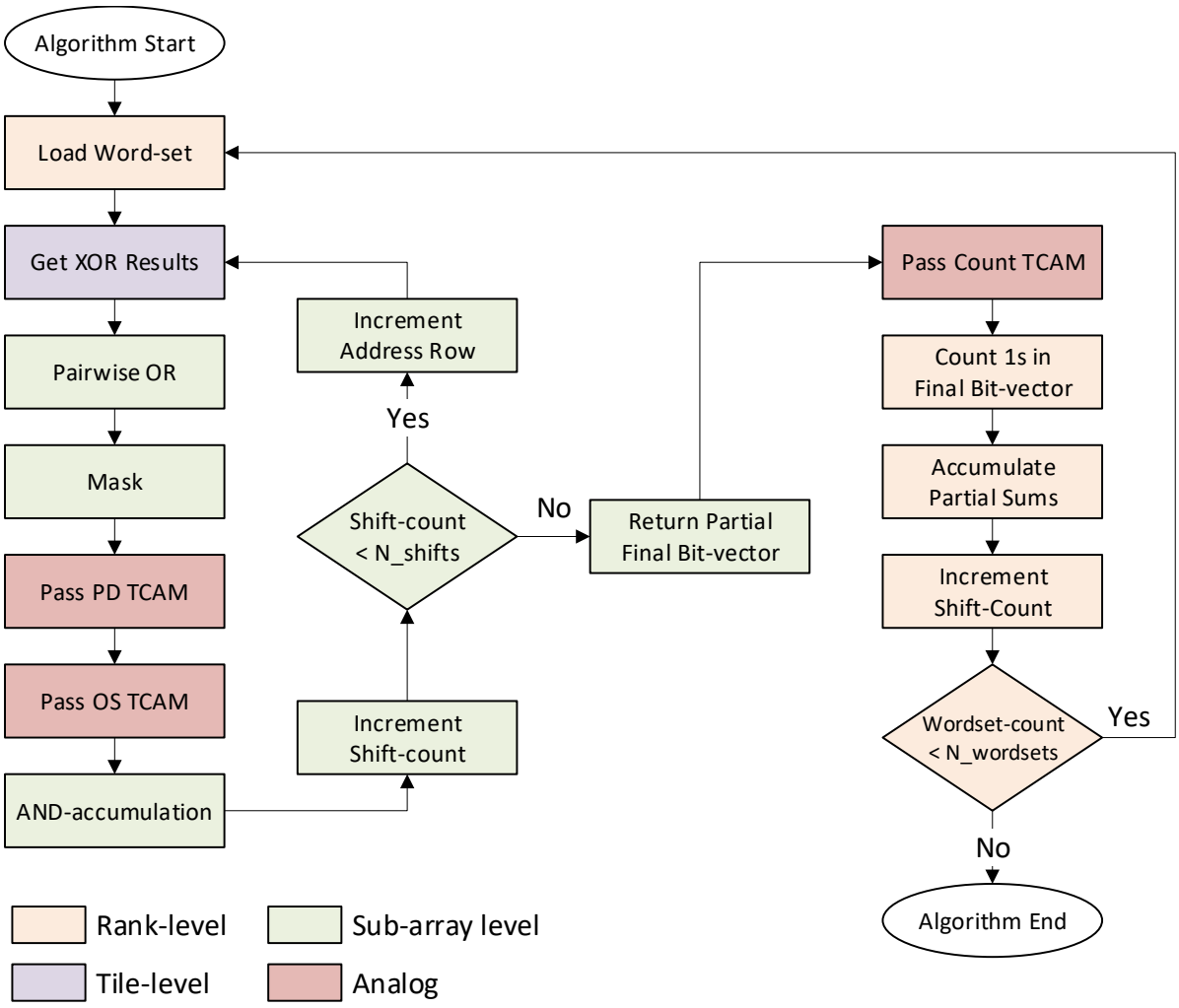


Figure 5.12: A flowchart demonstrating the algorithm process at the sub-array level.

To illustrate this procedure, we walk through the most important steps using the example input data in Table 5.6, which uses the last 4 segments of the example in Figure 4.1(a). For this example, we assume this pairing is part of a longer read. This evaluation is first performed using SHD, followed by SS-CIM.

Table 5.6: Example read-reference pairing.

Read Sequence	TGTC TGAA ACTT ACGC
Reference Sequence	..GTTGACTCGAAACTTACACT..
Correct Alignment	TGTC TG-AA ACTT ACGC || | || || |||| || | GT TGAC TCGAA ACTT ACAC T
Edit-distance	3
Edit-distance Threshold	2

Bitwise XOR

In this example, the read reference pairing is assumed to be written entirely on a single sub-array tile-group, with each segment of 4 base pairs (8 bits) written to a separate tile. The correct alignment of this pairing contains three edits. Two substitutions, colored red in Table 5.6, and a deletion colored in green. We focus on the left-most segment (mapped to tile 3), which contains a substitution error ('A' in the reference sequence is replaced by a 'T'), and the third segment (mapped to tile 1) which does not contain an edit. In Figure 5.13 we present the XOR-results of the second shift, which corresponds to 'Ref «1', as this iteration shows a mismatch in the first segment but a match in the third segment.

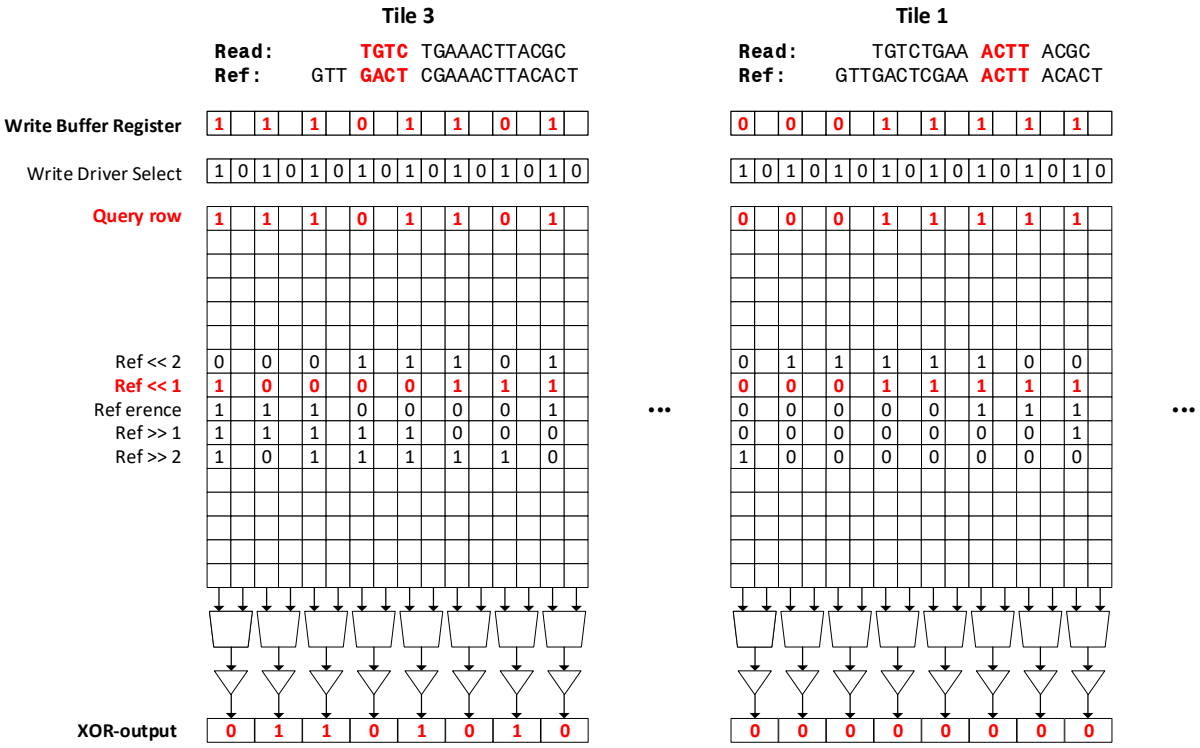


Figure 5.13: Example of the tile-level execution of the XOR result between the read-segment and the «1 shifted reference.

The XOR results of the tiles are combined and given as input to the pairwise-OR block. The output of this operation is put through the masking block, which in the given example does not mask off any part of the pairing. The result of this is then used as the input of the PD-TCAM. Figure 5.14 demonstrates this process on the XOR results obtained for the example pairing.

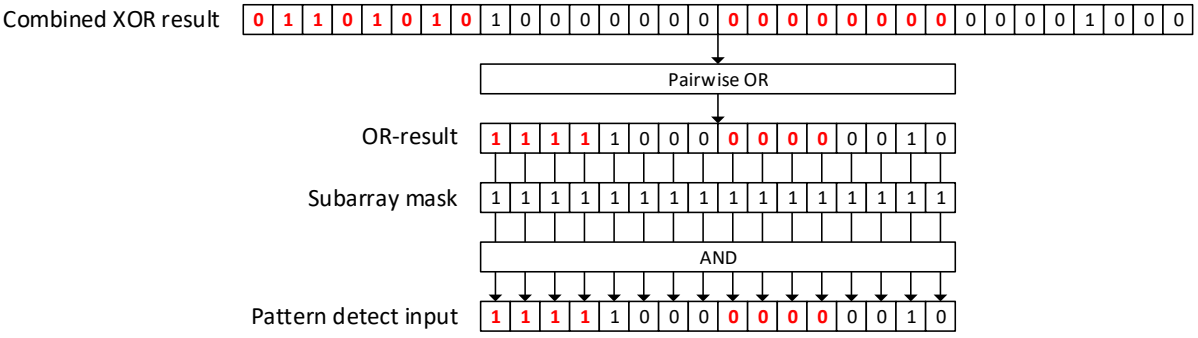
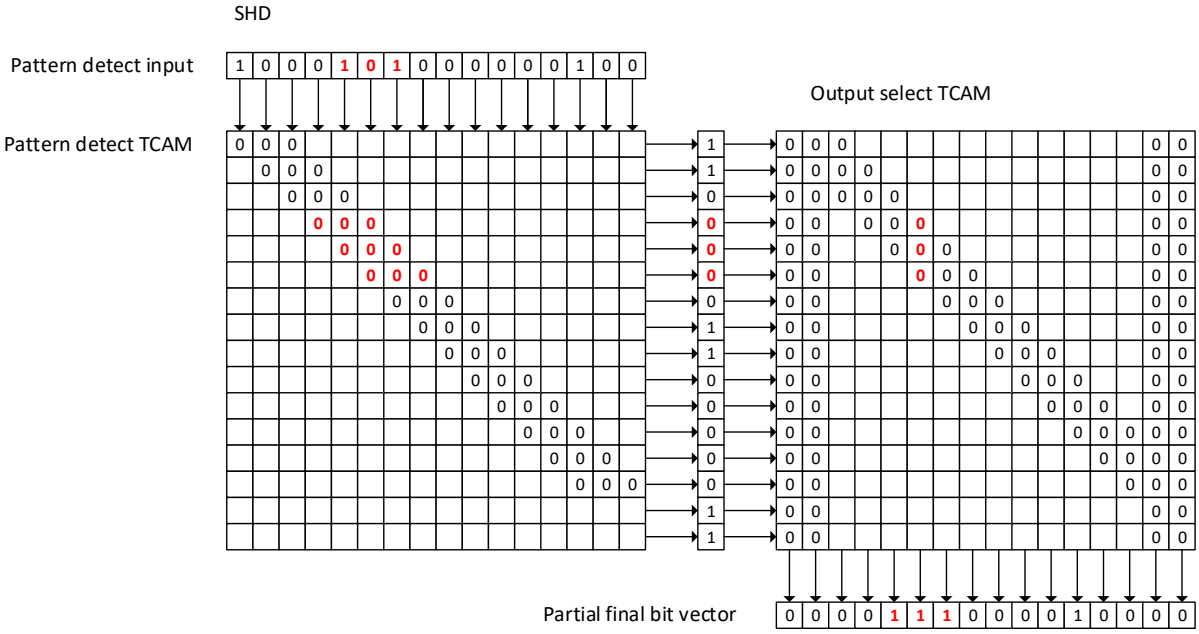


Figure 5.14: Example of the pairwise-OR operation and the masking process.

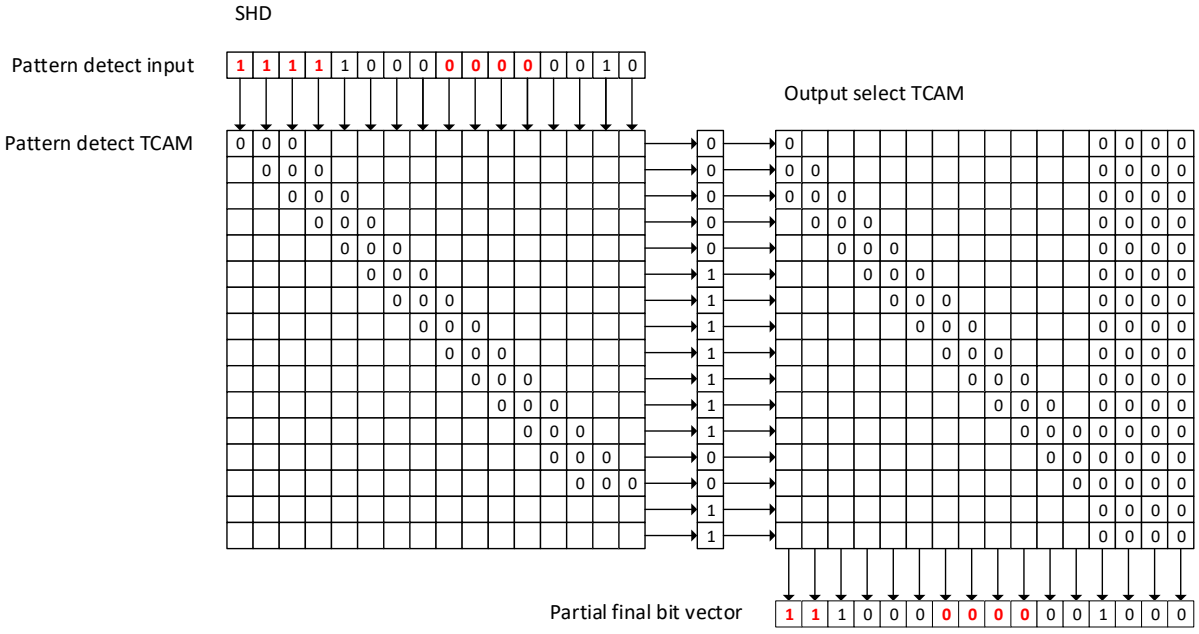
Pattern Detection

The XOR result is used as an input to the series of TCAMs. SS-CIM and SHD differ in how each algorithm handles the programming of the TCAMs. Here we first go over the programming as used by SHD, followed by SS-CIM.

SHD The goal of the TCAMs in SHD is to remove short sequences of zeros (<3 bps) from the input vector by turning them into ones. In the original software implementation of SHD, this is done by detecting sequences of '101' or '1001' in the shifted Hamming masks. The zeros in these short sequences are then amended to '1's in the final bit-vector. The algorithm searches for these patterns in all bit-positions sequentially. The proposed architecture implements the detection of these patterns using the PD-TCAM, which is programmed as shown on the left side in Figure 5.15(a). Here '1's and '0's for each cell are depicted, with blank cells for 'X' cells.



(a) SHD



(b) SHD optimized for count-TCAM utilization

Figure 5.15: Example of TCAM programming for (a) SHD, and (b) SHD optimized for count-TCAM utilization.

Rather than finding if a zero is part of a sequence of '101' or '1001', we instead detect the complement of this requirement, which is to detect if the bit is part of a sequence of 3 zeros or more. This allows for implementation with fewer TCAM entries. A zero can be part of a series of 3 zeros or more if it is part of a pattern of '00**0**', '0**0**0', or '**0**00', where the 0 in bold indicates the zero in the targeted bit-position. The detection of any of these

patterns is expressed as a '1' in the intermediate result. This is fed as an input to the OS-TCAM, which checks for the absence of any of these patterns for every bit-position. If so, the bit in the final bit-vector is set to '1'. In Figure 5.15(a), the OS-TCAM is transposed such that the patterns are vertical and the output is horizontal for clarity.

In the example in this figure (note: this is different from the output found in Figure 5.14), the '0' in the input vector highlighted in red should be amended to '1', as it is a short zero-sequence. The PD-TCAM detects that it is not part of any sequence of zeros larger than 3, as indicated by the red values in the intermediate results. Therefore, the OS-TCAM detects that none of the patterns is detected, and the bit in the final bit-vector is amended.

The left-most and right-most bits of the input cannot be amended as that would require the XOR results of the adjacent base pairs, which are generated in different sub-arrays. These bits, therefore, do not contain useful information for SHD. These parts of the shifted Hamming mask have to be processed in another sub-array, where the segments are overlapped with the preceding segments to not lose information.

To better align with the count-TCAM at the rank-level, the unused columns are therefore shifted, as demonstrated in Figure 5.15(b). Here we return to the original example in Table 5.6 and find that there are no amended zeros.

SS-CIM In SS-CIM the objective is to find if any shifted Hamming mask of a segment contains only zeros, which indicates an exact match of that segment. The pattern that needs to be detected by the PD-TCAM is therefore series of zeros starting and ending at the beginning and end of each segment, respectively. The programming of the PD-TCAM for a segment size of 4 bps is shown in Figure 5.16. A '1' in the output of the PD-TCAM indicates the existence of an exact match. For this algorithm, the final bit-vector should contain a '1' for every segment that has no exact matches for any of its shifted Hamming masks. The OS-TCAM essentially creates this by inverting the intermediate result. In the example, all segments contain errors except for the third segment. The partial final bit-vector thus contains 3 '1's to indicate the presence of at least 3 edits.

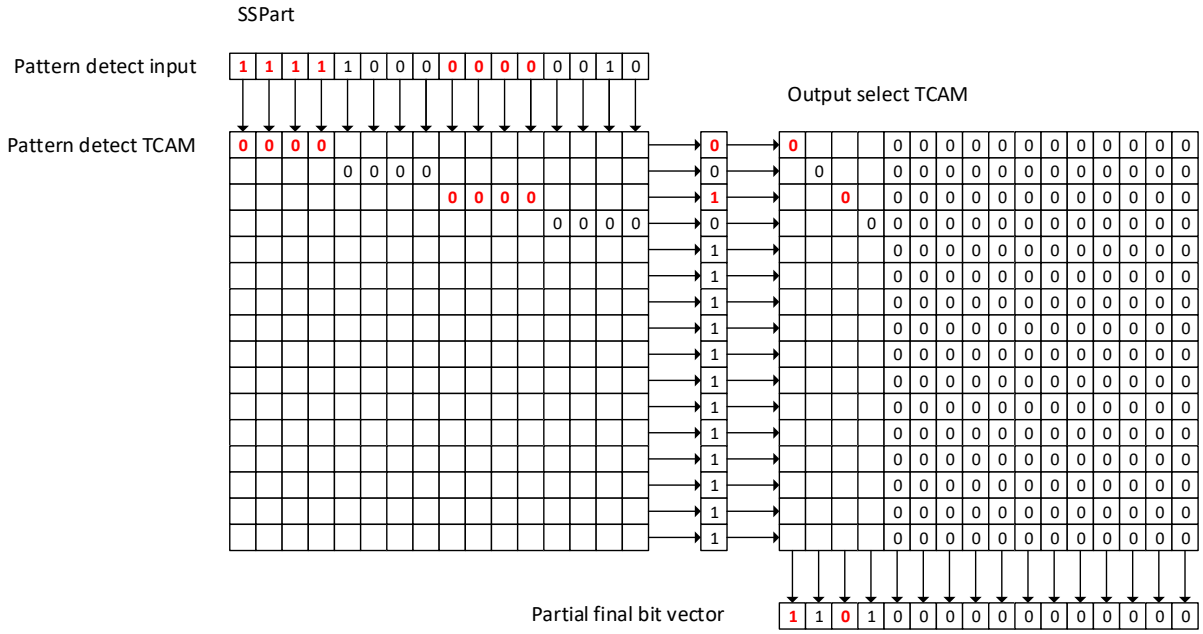


Figure 5.16: Example programming of the TCAMs for SS-CIM with a segment size of 4 bps.

Final Bit-vector Accumulation

The results of the count TCAM are collected in a register and accumulated over all iterations of the reference shifts. This intermediate-result register is initialized to all '1's, which is used as an operand for the AND-operation with the OS-TCAM output. The results of all iterations of the example pairing using SHD and SS-CIM are shown in Table 5.7 and 5.8, respectively. Here, the parts that do not contain useful information are

colored gray. When all iterations are have been evaluated, the intermediate result is returned to the rank-level for further processing, along with its corresponding ID.

Table 5.7: The evolution of the intermediate result register over the iterations of SHD.

Iteration	Hamming mask	OS-TCAM output	Intermediate Result
0	-	-	111111111111 1111
1	11 001100110111 11	111111111111 0000	111111111111 0000
2	11 111000000000 10	111000000000 0000	111000000000 0000
3	11 100110011011 11	111111111111 0000	111000000000 0000
4	01 111111011111 10	111111111111 0000	111000000000 0000
5	11 011111111111 11	111111111111 0000	111000000000 0000

Table 5.8: The evolution of the intermediate result register over the iterations of SS-CIM (segment size 4).

Iteration	Hamming mask	OS-TCAM output	Intermediate Result
0	-	-	1111 111111111111
1	1100 1100 1101 1111	1111 000000000000	1101 000000000000
2	1111 1000 0000 0010	1101 000000000000	1101 000000000000
3	1110 0110 0110 1111	1111 000000000000	1101 000000000000
4	0111 1111 0111 1110	1111 000000000000	1101 000000000000
5	1101 1111 1111 1111	1111 000000000000	1101 000000000000

Edit Counting

At the rank-level, the results of all bank-groups are collected to count the number of edits in a word-set of the read sequence. SHD and SS-CIM have different ways of counting edits in the final bit-vectors. Therefore, an additional set of TCAMs is required to detect patterns and assign a number of edits to the detected patterns. These 4-bit wide TCAMs are mainly required for SHD, where the final bit-vector is split up into segments of 4 bits. The sequence of '0's and '1's in these segments determines the number of counted edits. The programming of these TCAMs is shown in Figure 5.17(a) and 5.17(b), for SHD and SS-CIM, respectively.

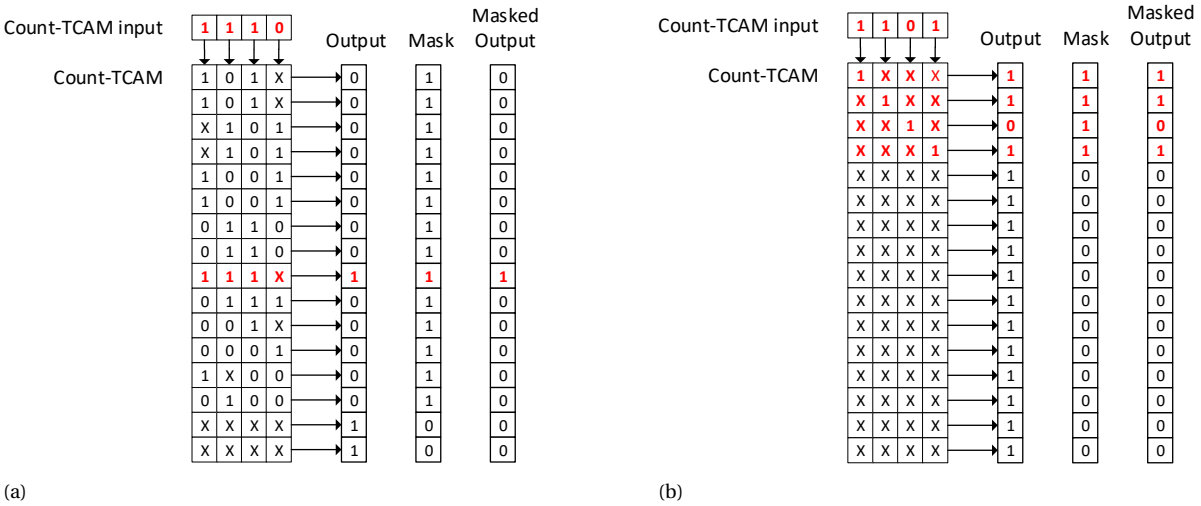


Figure 5.17: Example count-TCAM programming for (a) SHD, and (b) SS-CIM (segment size of 4).

For SHD, if the segment is "0000", no edits are counted, whereas for "0101", "0110", "1001", "1010", "1011" and, "1101", 2 edits are counted. All other cases lead to a single edit. These expressions can be compressed into 14 TCAM entries using the 'X' cells, where the patterns that count for 2 edits have double entries. For SS-CIM, the number of '1's in the output vector directly translates to the number of detected edits. Therefore, the inputs are relayed to the output without additional mutations. Since the algorithms require a different number of patterns, a mask is applied to the TCAM output to ignore unused entries.

For the evaluated example, it is found that SHD is able to find only 1 edit, while SS-CIM is able to find 3 edits. The results of each count-TCAM are added up by the multi-operand adder. The resulting sum in integer representation is stored in the sum buffer that is indexed by the ID. Once all word-sets of the read sequence have been processed, the total sum is compared to the edit-distance threshold. The result of this comparison determines whether the pairing is passed or blocked.

5.6.3. Differences Software vs. Hardware Implementation

Unlike the software implementation of the algorithms, where segments always start at the start of the read sequence, it is possible for the read to start in the middle of a segment in the hardware implementation. This is due to the fact that the references are stored in a fixed manner in the memory elements. The difference between the two implementations is demonstrated in Figure 5.18. This has two main consequences. Firstly, the software implementation always contains the minimum number of segments for a given reads sequence, while the hardware implementation can contain an additional segment. On one side, this allows the hardware implementation to detect more edits than its software counterpart. However, since the number of evaluated base pairs remains the same, the first/last segment are shorter than the intended segment length. This increases the likelihood of random matches occurring. The impact of the differences between the two implementations are verified experimentally in Section 6.2.

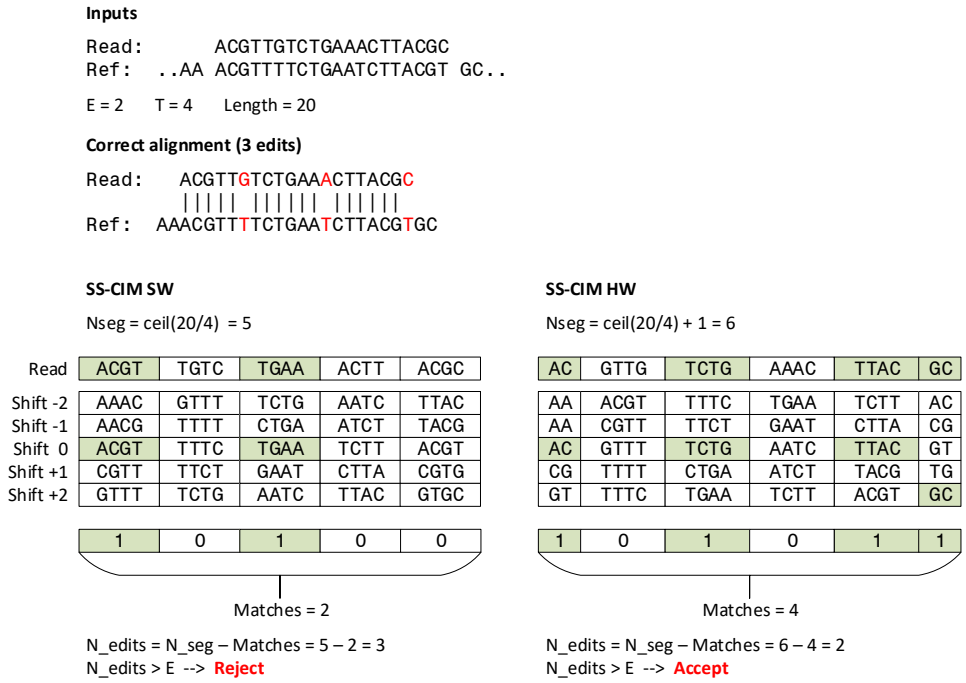


Figure 5.18: Difference between the software and hardware implementation of SS-CIM.

5.7. Long-Read Compatibility

The proposed architecture requires all shifted references to be written to the same tile. This approach has two limitations when considering the architecture for long reads. Firstly, the maximum edit-distance that the system can support is limited by the rows within each tile. Crossbars have limited dimensions due to factors such as increasing read/write current requirements and leakage currents as the dimensions are scaled-up. Because of this, the maximum edit-distance that can be evaluated is limited as well. Secondly, the number of shifted references required for long reads exceeds 50 thousand shifts in the case of inaccurate long reads. Storing all of these shifted references would require an enormous memory capacity.

In this section, we discuss adaptations to the proposed architecture that limit the number of resources required for long reads. This long-read architecture splits up the evaluation of the different shifts into parts, which are processed in different sub-arrays and over multiple iterations of the algorithm. These results are aggregated over multiple iterations at the rank-level before edits are counted. These adaptations are achieved by altering the hardware at the sub-array and rank levels, and the formatting of the system input.

The main observation that is made for the adaptation of the long read architecture is that, rather than comparing just comparing the read to numerous shifted references, reads can also be shifted against a fixed reference. Doing so results in the same result as the original implementation, provided that the output of the comparison is shifted back by the same amount. This has the advantage that only a single reference sequence needs to be stored, eliminating the need for excessive amounts of memory elements.

However, shifting the read sequence for every shift would require a write operation for every evaluated shift of the read-reference pairing. This has two distinct disadvantages. Firstly, the write operation requires control signals from the rank-level controller. As a single write operation only allows for the generation of a single shifted Hamming mask, this limits the amount of parallelism that can be achieved by running multiple sub-arrays simultaneously. Secondly, the large amount of write operations is detrimental to the endurance of the cells of the query row, as its values have to be overwritten constantly.

As a middle ground to these two extremes, we propose a trade-off between the required memory capacity and the performance/endurance of the system by storing a limited number of shifted references (N_{copies}). These adaptations allow for the parallel computation of Hamming masks over multiple sub-arrays while having a fixed memory requirement for all edit-distance thresholds.

The long-read architecture is implemented by not only splitting up read-sequences into segments of base pairs but also by splitting up the shifts, as is illustrated in Figure 5.19. The configuration in Figure 5.19(a) represents the original architecture, where the segments of the read sequence are compared to the entire set of shifted references in the same tile. In Figure 5.19(b), the read sequence is split up into segments, as well as in sections of 8 shifts, which are evaluated separately. We refer to these partitions as shift-sets. We make the observation that shift-sets that share a diagonal evaluate the same sections of the reference sequence. Therefore, in the final long read architecture in Figure 5.19(c), only one of each shift-set is stored, while the reads shifted with respect to the shift-sets.

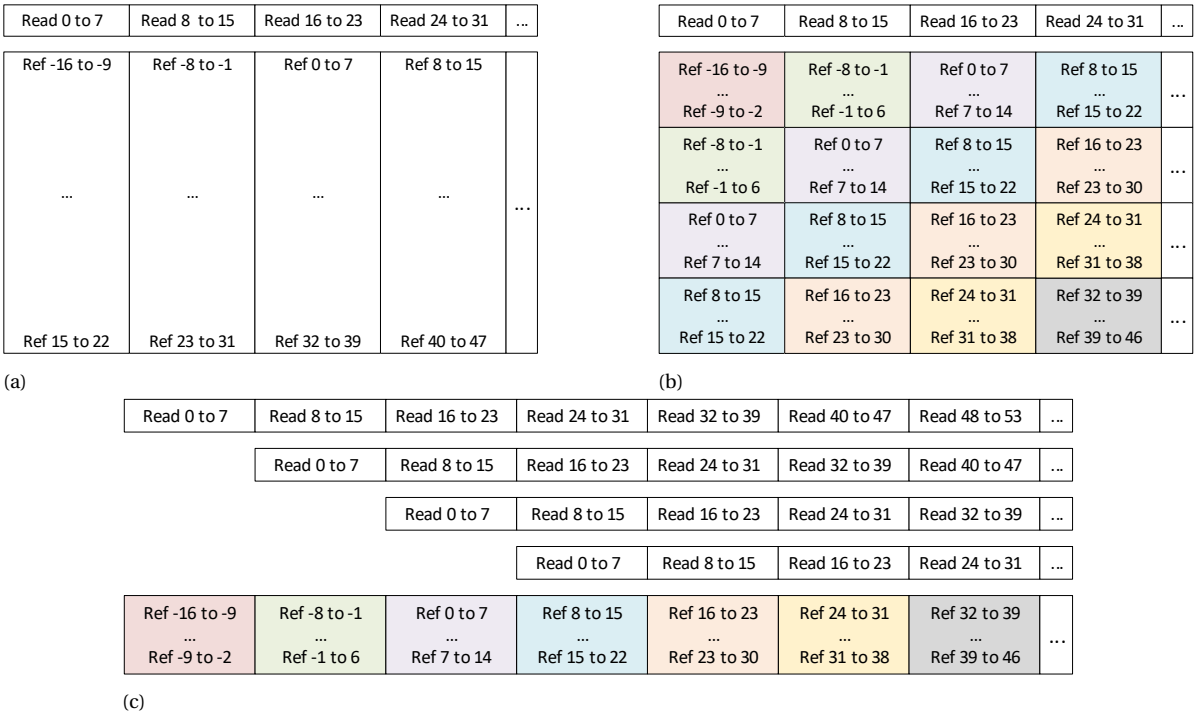


Figure 5.19: The underlying concept of the long-read architecture of (a) the original implementation, (b) the split into shift-sets, (c) and the memory-optimized configuration. Here, the numbers indicate the ranges of bits that are evaluated in each shift-set.

The shifting of the reads is handled by the host device. Each word-set generated by the host device corresponds to a number of horizontally adjacent shift-sets which are each handled by a separate bank-group. A

sub-array of each bank-group performs the calculations on the shift-set, after which the result is returned to the rank-level. This result is incomplete as it requires information from different shift-sets corresponding to the same read-segment. Therefore, the result is saved in an AND-buffer, which is placed alongside the sum-buffer of the short-read implementation. This buffer accumulates the partial bit-vector results of all shift-sets by performing a bitwise AND-operation between its stored value and the incoming result. When all shift-sets of the read-segment have been evaluated, the contents of the AND-buffer are used as input to the count-TCAM. The rest of the procedure is identical to that of the short-read architecture.

A problem with this approach is that, due to the read sequence being shifted with respect to the start of each word-set, the output of the sub-arrays is also shifted. This shift is corrected at the rank-level by shifting the result vector in the reverse direction. To provide the system with information on how much this shift should be, the shift amount is passed alongside the pairing ID.

Since the shift-set does not always have to be evaluated in its entirety (i.e., in the shift-set containing the »e reference), an end-point has to be indicated to the sub-arrays. This is done by addressing the last row of the shift-set at the start of the algorithm. This gives the ability to have different shift-counts for each sub-array.

A disadvantage of the long-read architecture is that it requires more write-operations to the query rows to implement the same read-reference pairing than the short-read architecture, which leads to worse endurance of the system. To mitigate this, we implement multiple query rows, each of which is used for a different shift-set.

The multiple-write operations also have the effect of reduced performance compared to the short-read-optimized architecture if the number of evaluated shifts exceeds N_{copies}. For this reason, we evaluate the trade-off between resource requirements and performance for short reads in Section 7.5.

5.8. Conclusions

In this chapter, we provide an overview of the proposed CIM-architecture capable of implementing the pre-alignment filters. We propose a hierarchical memory architecture akin to those found in existing DRAM memories. This memory is augmented with peripheral devices which are used to implement the pre-alignment filtering algorithms. We provide a per-hierarchy-level explanation of the functionality of each of its components and design considerations. Furthermore, we demonstrate the flexibility of the architecture using example mappings for two pre-alignment filtering algorithms: SHD and SS-CIM. Also, the data flow through the architecture is explained from the read-sequence input to the output of the system. Lastly, we identify possible shortcomings of the architecture for the mapping of long reads and provide an additional design as a solution. This leaves us with two architectures: one designed for short reads specifically, and one that supports both short and long reads at the cost of lower performance.

Chapter 6

Implementation

In Chapter 5, we propose two CIM-architectures for the acceleration of SHD and SS-CIM. In this chapter, we discuss the implementation, validation, and evaluation of these architectures. We first explain the methodology used for our implementations, and the various parameters used to configure them in Section 6.1. This is followed by the procedure for functional verification using real datasets in Section 6.2. Additionally, we present an analytical model which is used to calculate the resource requirements for a given dataset in Section 6.3. Furthermore, we devise a performance model in Section 6.4, which estimates the execution time of a combination of a given dataset and hardware configuration. These two models are combined in a design-space exploration in Section 6.5, which is used to determine the optimal hardware configurations for the proposed architectures. Finally, we summarize the main findings of the chapter in Section 6.6.

An overview of all steps discussed in this chapter is provided in Figure 6.1.

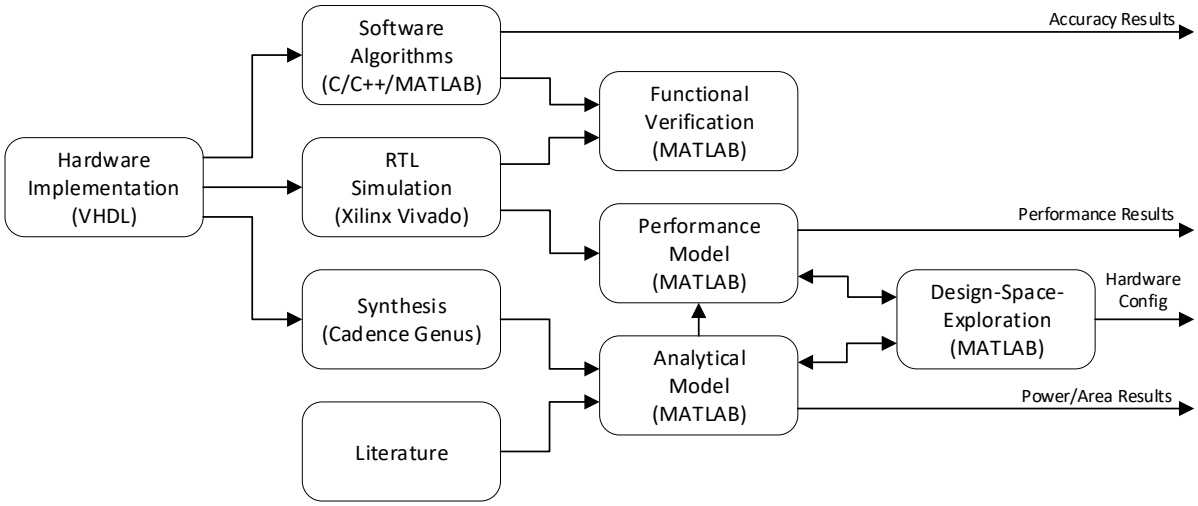


Figure 6.1: An overview of the evaluation process.

6.1. Hardware Implementation

To verify the functionality of the proposed designs, we implement the control logic of the digital components in synthesizable VHDL. We create a separate module for each of the levels of the hierarchy, which allows for the testing of each component separately. In this architecture, the analog components (i.e., the crossbars and TCAMs) are replaced by digital logic for functional verification. These components are removed before synthesis. Each component is parameterized such that the dimensions and timings of all components can be adjusted to support various hardware configurations.

6.1.1. Hardware Parameters

In this section, we discuss each of the variables and their effects on the overall design.

Tile Dimensions At the lowest level of the architecture, we are able to configure the dimensions of the tiles. By increasing the number of rows/columns of the tile, the total memory capacity is increased. The number

of cells per tile is determined by taking the product of the number of rows and columns. The size of the peripheral components is also dependent on this metric. The number of rows per tile determines the size of the row selector registers. The number of columns affects the number of registers required for the write driver select, write buffer registers, and MUX/DEMUX circuitry.

Sense-amplifiers per tile While most other peripheral devices of the crossbar are automatically adjusted to the tile dimensions, the number of SAs per tile can be configured manually. This parameter determines how many bits of data can be read from the tile in a single read operation. We limit the data interface width of the tile to the number of SAs such that both the read and the write operation can be done over the same data bus in a single cycle. This parameter, therefore, affects the throughput of the system.

Increasing the number of SAs per tile allows for more data to be written in a single cycle, which increases the throughput of the system. However, this comes at the cost of increased chip area, both directly by adding more components and indirectly by increasing the dimensions of the MUX/DEMUX components used for interleaving.

Sub-array/bank configuration The number of tiles per sub-array can be configured, which has two main effects on the system. Firstly, it impacts the total memory capacity of the system as a whole. Scaling up the number of tiles per sub-array increases memory capacity and the size of the control components of each sub-array. However, since these are shared amongst more memory elements, the overall chip area is decreased. Secondly, this parameter determines the amount of reference-sequence data stored in each sub-array. This has an effect on the average level of contention over the sub-arrays. The level of contention of the sub-arrays increases with the size of the sub-arrays, as the likelihood of two read-reference pairings requiring the same sub-array increases. Because of these effects, a trade-off exists between the chip area and the throughput of the system. To obtain higher capacities, while keeping contention low, multiple sub-arrays are grouped in banks. The number of sub-arrays per bank can be configured during design time.

Writes-per-bank To obtain the required amount of data per sub-array, multiple words are written to the sub-array before the computation is started. The number of words depends on the length of the patterns that need to be detected (determined by the algorithms). This parameter is calculated as demonstrated in Equation 6.1.

$$\text{WPB} = \frac{2^{\lceil \log_2(l_{max}) \rceil}}{2 * \text{interface\_width}} \tag{6.1}$$

Sub-array TCAM dimensions The dimensions of the PD-TCAM and OS-TCAM determine the length and the number of patterns that can be detected. The row width and column height of the PD-TCAM is equal to the column height and row width of the OS-TCAM, respectively. The row width of the PD-TCAM determines the maximum length of detectable patterns. This parameter is set to be equal to the product of the interface width of the tile and the number of writes-per-bank. The column height of the PD-TCAM determines how many patterns can be detected for a given input.

Bank-group configuration In the architecture, the input words originating from the host device are split up into segments with a length equal to the interface width, each of which is written to a different bank-group. The bank-group configuration determines how many bank-groups are present in each rank. This parameter needs to be adjusted to the required data bus width of the host device and the interface width of the memory tiles using Equation 6.2.

$$\text{BGs\_per\_rank} = \frac{W_{system\_data\_bus}}{\text{interface\_width}} \tag{6.2}$$

Input/Output queue lengths The length of the input queues of each hierarchy level can be controlled separately using this parameter. The input queues are used to reduce the effects of contention over the sub-arrays

to improve the throughput of the system. However, the addition of these queues comes at the cost of increased chip area. Because of this, there exists a trade-off between the area and performance, when adjusting this parameter.

Similarly, the output queues reduce contention over the output bus. The length of these queues can be adjusted for each level of the architecture. Here the same trade-off between area and performance exists as for the input queues.

Buffer entries This parameter determines the length of the sum-buffer, ID-LSB buffer, and the word-set-count buffer at the rank-level. The size of these buffers determines how many separate read-reference pairings can be processed by the system at the same time. If the buffers are full, the input of the system is stalled, which leads to a decrease in performance. However, the number of buffer entries should be limited to avoid excessive chip-area overheads. Again, this parameter introduces a trade-off between chip area and performance.

This parameter also affects the length of the required ID used to index the various buffers. As this ID is also passed to the lower-level memory elements, this influences the interfaces between the various levels.

Register & counter sizing The size of the registers containing programmed data can be controlled. For example, the size of the register in the sub-array controller that holds the number of required shifts can be adjusted using this parameter. This in turn influences the dimensions of the shift counter for instance. This is also done for the edit-distance threshold and max-word-set registers.

Analog component timings The number of write cycles, read cycles, and xor-compute cycles can be configured to account for different memristor technologies, which can each have different read/write times. These values are used by the counter at the tile level which allows the tiles and the rest of the system to operate with different frequencies. Similarly, the programming and search time of each of the TCAMs can be adjusted using this parameter.

Number of reference-copies This parameter is only relevant for the long-read architecture and determines how many shifted references are stored in the memory per word-set. This parameter affects how many operations each sub-array can perform on a shift-set. The higher the number of reference copies, the fewer shift-sets have to be evaluated. However, this comes at the cost of greater memory capacity requirements.

6.2. Functional Verification

To verify the functionality of the design, it is simulated on the RTL level using Xilinx Vivado on a per-coponent basis. For every level, all instructions are tested individually, and the outputs are read from the waveform to confirm the correct behavior. The complete architecture is verified using real genomic data. To produce the inputs of the system, a script is developed in MATLAB that generates the input values and instructions for the programming of software parameters and the values of the TCAMs. An additional file is generated which contains the inputs and instructions for writing the reference to the memory elements according to the writing scheme in Figure 5.9 and Figure 5.10, and the addressing scheme in Equation 5.2. These inputs are read from the file and used as inputs to the RTL simulation. The script also generates a file containing read-sequence data, which is generated using the seeds produced by MrFAST. These reads include the seeding locations, which are used by the script to produce the correct address and mask. Each read is also assigned an ID, which is used to identify the outputs of the system. The outputs of the system are collected in an output file, which contains the ID numbers of all accepted read sequences.

As explained in Section 5.6.3, the hardware implementation of the algorithms differs slightly from their software counterparts. For that reason, the behavior of the hardware architecture is modeled by making adjustments to the software algorithm, such that it accounts for the differing start/end segments. This model uses the seed location of the read sequence and a given hardware configuration to determine the size of the start/end segments. In doing so, it is able to generate outputs exactly as the hardware architecture would. The results of the model are compared to the baseline implementations of the algorithms to make sure that no false-negatives are introduced, and that the FP-rate of the hardware implementations is not significantly

impacted. The results of the software implementation of SHD and its hardware model are compared in Table 6.1. Here it can be seen that for the hardware implementation of SHD, the algorithm shows the same or a slight decrease in accuracy (at most 1.43%) as compared to the software version.

Table 6.1: Comparison between FPs of software and hardware implementations of SHD for (a) Short_100bps, and (b) Short_250bps.

(a) FPs of SHD for Short_100bps.

E[%]	Software	Hardware	Increase
0	0	0	0.00%
1	5	5	0.00%
2	9	9	0.00%
3	331	331	0.00%
4	5172	5255	1.60%
5	21446	21767	1.50%
6	112992	113263	0.24%
7	375716	379661	1.05%
8	1057011	1063899	0.65%
9	2276485	2301364	1.09%
10	4887592	4926583	0.80%

(b) FPs of SHD for Short_250bps.

E[%]	Software	Hardware	Increase
0	21	21	0.00%
1	55	55	0.00%
2	116	117	0.86%
3	158	158	0.00%
4	9555	9567	0.13%
5	101701	103261	1.53%
6	964399	977345	1.34%
7	2517317	2553325	1.43%
8	8626740	8737517	1.28%
9	15657142	15788363	0.84%
10	24247584	24437084	0.78%

In Table 6.2, it can be seen that for SS-CIM, the increase in FPs is between -12.50% and 2.08%, depending on the dataset/edit-distance threshold. From this, we conclude there is a minimal decrease in accuracy when mapping either of the algorithms onto the proposed architecture.

Table 6.2: Comparison between FPs of software and hardware implementations of SS-CIM for (a) Short_100bps, and (b) Short_250bps.

a FPs of SS-CIM for Short_100bps.

E[%]	Software	Hardware	Increase
0	0	0	0.00%
1	0	0	0.00%
2	0	0	0.00%
3	8	7	-12.50%
4	147	148	0.68%
5	1212	1220	0.66%
6	11395	11312	-0.73%
7	39063	38822	-0.62%
8	94647	94583	-0.07%
9	219129	219364	0.11%
10	687236	697091	1.43%

b FPs of SS-CIM for Short_250bps.

E[%]	Software	Hardware	Increase
0	0	0	0.00%
1	17	17	0.00%
2	48	49	2.08%
3	52	51	-1.92%
4	87	87	0.00%
5	209	210	0.48%
6	582	581	-0.17%
7	4574	4525	-1.07%
8	43518	43946	0.98%
9	135592	134989	-0.44%
10	675644	681959	0.93%

6.3. Analytical model

To evaluate the performance architecture, we create an analytical model that approximates the chip area, power consumption, and execution time for a given hardware configuration, data set, and edit-distance threshold. In this model, we split the evaluation of the digital components implemented with CMOS technology (control logic, counters, registers, etc.), and the analog components based on memristors (crossbars and TCAMs). To evaluate the performance figures of the digital components, the architecture is synthesized using Cadence Genus using the NanGate Open-Cell 15 nm library [97].

For each component, we perform a parameter sweep over a range of dimensions that are commonly found in other proposed memristor-based CIM architectures [15, 35, 83, 98–102] and conventional DRAM-based memory architectures [96, 103, 104]. The range of values used for each parameter can be found in Table 6.3. Here we evaluate all values between the listed minimum and maximum values by incrementing with powers of 2. To reduce the design space, we evaluate the area, power, and timings of each configuration of every

hierarchy level individually. The chip-area, power consumption, and timings of the crossbars and TCAMs are based on the numbers presented in Scouting logic [14] and the design presented in [83], respectively.

Table 6.3: Value ranges for the freely configurable parameters.

Hardware parameter	Abbreviation	Minimum value	Maximum value
Crossbar rows	N_{rows}	64	512
Crossbar columns	N_{cols}	64	512
Sense-amplifiers per tile	N_{SA}	8	16
Tiles per sub-array	N_{tpsa}	8	64
Words per bank	WPB	2	8
SA input queue length	Q_{isa}	1	16
SA output queue length	Q_{osa}	1	4
Sub-arrays per bank	N_{sapb}	8	64
Bank input queue length	Q_{ib}	1	1
Bank output queue length	Q_{ob}	1	4
Banks per bank-group	N_{bpbg}	4	16
Bank-group input queue length	Q_{ibg}	1	1
Bank-group output queue length	Q_{obg}	1	4
Bank-groups per rank	N_{bgpr}	4	8
Buffer entries	N_{be}	16	64
Reference copies	N_{copies}	8	32

6.3.1. Resource Requirements

We dimension the architecture such that it can store the entire human genome in the memory at once. For the short-read optimized design, the requirement is set that the architecture must support the highest edit-distance threshold in the region of interest of short reads. For the evaluated datasets, this is the Short_250bp dataset for an edit-distance threshold of 5% (12 bps). To evaluate this dataset, the short-read optimized architecture must be able to store 25 copies of the reference genome. The number of shifted copies stored by the long-read architecture depends on the configuration and ranges from 8 to 32 copies. A summary of the requirements imposed by the input data can be found in Table 6.4.

Table 6.4: Summary of software parameters.

Software parameter	Value
Maximum read length (Short_250bps)	250 bps
Maximum edit-distance threshold	5%
Reference genome length (GRCh38)	3.1 Gbp
Encoded reference size (1 shift)	0.77 GB

The memory capacity of the proposed architecture is calculated using the set of equations found in Equation 6.3, where c denotes the capacity of a given component/hierarchy level and l_{ref} is the length of the reference genome in base pairs. Using these equations, we are able to further reduce the design space by taking only the configurations that have no more than 2 times the required memory capacity. This can be done because, for every configuration that is larger than 2 times the required capacity, one of the parameters can be decreased by a single step to obtain a configuration with the same performance, but with a smaller area overhead.

$$\begin{aligned} c_{tile} &= N_{cols} * \left\lfloor \frac{N_{rows}-1}{N_{shifts}} \right\rfloor * N_{shifts} \\ c_{rank} &= N_{tpsa} * N_{sapb} * N_{bpbg} * N_{bgpr} * c_{bankgroup} \\ N_{ranks} &= \left\lceil \frac{2 * l_{ref}}{c_{rank}} \right\rceil \\ c_{total} &= N_{ranks} * c_{rank} \end{aligned} \tag{6.3}$$

6.3.2. Area & Power Estimations

To obtain an estimation of the power and area consumption of the digital components of the design, we sum the power and area figures for each level of the hierarchy based on the synthesis results. The analog components are based on the numbers presented in previous works, which are scaled to correspond to the dimensions required by the evaluated hardware configuration. However, since the design of Scouting logic is simulated using a 90 nm process, the power and area figures are scaled down further to account for the discrepancy between the different process nodes. Since chip power/area scaling shows a non-linear trend when shrinking down beyond 45 nm [105,106], we conservatively scale the power and area numbers to match that of a 45 nm design following Dennard's law [107]. This, therefore, provides a conservative estimate of the power and area of the overall architecture. A similar strategy is employed for the TCAM design proposed in [83], which is implemented on a 65 nm technology node. The power consumption, chip area, and timing numbers used for the model are listed in Table 6.5. These values are linearly scaled to match the required row/column dimensions.

Table 6.5: Summary of the power, area, and timing of the analog components.

Component	**Crossbar**	**Crossbar (scaled)**	**TCAM**	**TCAM (scaled)**
Dimensions	128×32	$N \times M$	64×256	$P \times Q$
Process node [nm]	90	45	65	45
Average power [μW]	15	3.6 *(N/128 * M/32)	4.8	2.4 * (P/64 * Q/256)
Chip area [μm^2]	410	105 *(N/128 * M/32)	9584	4792 * (P/64 * Q/256)

6.4. Performance Model

While the methods for shrinking the design space significantly reduce the number of configurations, it remains too large to perform RTL-level simulations for each configuration on a full dataset. Therefore, we instead develop a model in MATLAB that emulates the behavior of the architecture. Rather than evaluating the results of the algorithms, the model only accounts for the stalling behavior and the calculation time of the computation units. It keeps track of what components are busy, and how many cycles it takes for them to complete their computation steps. Furthermore, the input and output queues are simulated by keeping a record of how many instruction steps are present in each queue.

At the lowest level, we consider the sub-array elements, of which the computation and write time are tracked. For the calculation of the write-time, we assume a 100/10 ns write/read period of the memristors, as is often used in other memristor-based designs [102, 108, 109]. For the digital components, we use the timing results produced by synthesis by taking the longest path delay out of all components of a particular hardware configuration. The number of clock cycles required for a write operation can then be determined using Equation 6.4.

$$N_{c\_write} = \frac{T_{write}}{T_{clock}} \tag{6.4}$$

Similarly, the number of computation cycles for a single iteration of the algorithms can be obtained by summing the read time (10 ns [102, 108, 109]), TCAM search time (1 ns [83]), and the time required for switching FSM-states by the sub-array controller. This number is multiplied by the number of shifted Hamming masks that is computed in every sub-array, as shown in Equation 6.5.

$$N_{c\_comp} = \frac{N_{shifts} * (T_{read} + T_{control} + 2T_{search})}{T_{clock}} \tag{6.5}$$

The model assumes that at the start of the execution, all memory elements are in their idle state and that all input and output queues are empty. As input data, the model makes use of the same input file as the RTL simulation. Then, for every WPB-iterations, the address of a word-set is loaded from the input file. From this, the model calculates what sub-array is addressed by that word-set. If the sub-array is not already occupied, the cycle counter of that sub-array is set to $N_{c\_write} + N_{c\_comp}$. This value is decremented by 1 for every

iteration of the model, and once this value hits 0, the computation is complete. If during this processing period, another word is loaded that is addressed to the same sub-array, the queue count of that sub-array is incremented by 1. Once the cycle counter of the sub-array hits 0, its value is reset to $N_{c\_write} + N_{c\_comp}$, and the queue counter is decremented. If at any stage the addressed input queue is full, the system is stalled, meaning that no new word-sets are loaded in until the queue is free again.

Furthermore, the model accounts for communication delays between the levels of the architecture, such as loading the word-set into the rank-level or writing from the rank input-buffer to the lower levels of the hierarchy. The model assumes that stalling happens immediately when a sub-array queue is full. However, in reality, the instructions would first fill the bank-group and bank queues. This leads to the model stalling for longer than expected when compared to the timing results of the RTL simulation. This makes it so that the execution time estimate of the model is higher than those produced by the RTL simulation.

The model loads word-sets until the end of the input file is reached, after which it waits until all queues are empty and all sub-arrays are idle. During the execution, the model counts the number of iterations since the start of the execution. This is then multiplied by the clock period to obtain the approximate execution time for the given dataset.

To account for different hardware parameters, the performance model can be adjusted with the variables listed in Table 6.6.

Table 6.6: Input parameters of the performance model.

Parameter	**Description**
Hardware configuration	Values in Table 6.3
Clock period	Obtained through Genus synthesis.
Write time	Time to write from the write buffer registers to the crossbar.
Read time	Time to read from the crossbar to the output register.
Search time	Time to obtain the results from the TCAM.
Communication overhead	Time to load inputs from the rank input buffer to sub-array
Shifts	Number of shifts calculated in each sub-array.
Input dataset	Evaluated read-sequence data set.
Edit-distance threshold	Edit-distance threshold of the algorithm expressed in bps.

6.4.1. Performance Model Verification

To verify the accuracy of the performance model, we compare the timing results of the RTL simulation to those obtained using the performance model for ten randomly generated configurations. We evaluate the execution time of these configurations when evaluating the Short_100bps and Short_250bps datasets consecutively with an edit-distance of $e = 5$. The results of this comparison are shown in Figure 6.2, where the performance estimations of both methods are plotted against each other. Here we observe that the performance model consistently has a higher execution time estimation than the RTL simulation. However, the discrepancy between the two methods remains within 6%, which shows that the model is accurate enough for approximate performance estimations.

6.5. Design-space Exploration

To find the optimal hardware configuration for a given set of input data, we perform an exhaustive design-space exploration of the narrowed-down design space. The analytical model is used to find the memory requirements and their associated power/area/timing numbers. The performance of each configuration is determined using the performance model.

As input for the short-read architecture, we use 4 different real datasets (2x100 bps, 2x250 bps), ERR240726_1 [110], ERR240727_1 (Short_100bps), SRR826471_1 (Short_250bps), and SRR826471_2 [111]. These datasets are evaluated for all edit-distance thresholds in their respective regions of interest using both SHD and SS-CIM. For the long-read architecture, we additionally evaluate the 4 long-read datasets presented in Chapter 3 using only SS-CIM.

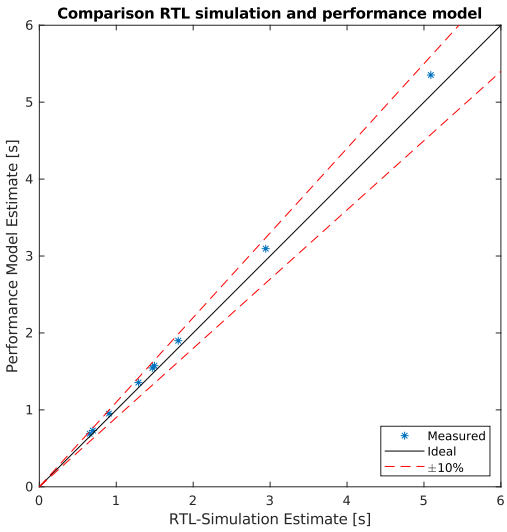


Figure 6.2: Comparison between the execution time estimates obtained using the performance model and the RTL-simulation.

From the results of the parameter sweep, we evaluate the results by selecting the Pareto-optimal configurations when evaluating performance per area and performance per watt. From these Pareto optimal configurations, the final configuration is selected manually.

6.6. Conclusions

In this chapter, we discuss the implementation details of the proposed architectures. We explain the methodology for verifying the designs using real datasets and explore the effects of mapping the algorithms onto the architecture in terms of accuracy. Additionally, we explain the process of obtaining performance metrics for various configurations of the designs. For this, we split the design into its digital and analog components, which are evaluated separately. The digital components are synthesized, which provides the area, power, and timing results for each hardware configuration. The area and power figures found through synthesis are combined with those of the analog components, which are based on literature. The overall area and power of the entire system are then calculated using an analytical model.

Furthermore, we put forward a performance model that simulates the stalling and queuing behavior of the designs. This model is used to evaluate the performance of the system, based on the timing numbers obtained through synthesis and the given hardware configuration. This model is verified to be accurate to within 6% of the execution time obtained using RTL-simulation. The model is then evaluated for several input datasets and edit-distance thresholds, and the results are averaged to obtain a performance number of the system.

To find the optimal hardware configuration, we combine the analytical model for area and power with the performance model. We perform an exhaustive sweep over all configurable hardware parameters and select the Pareto optimal configurations when optimizing for performance per area and performance per watt. From there, the final design parameters are selected manually.

Chapter 7

Results

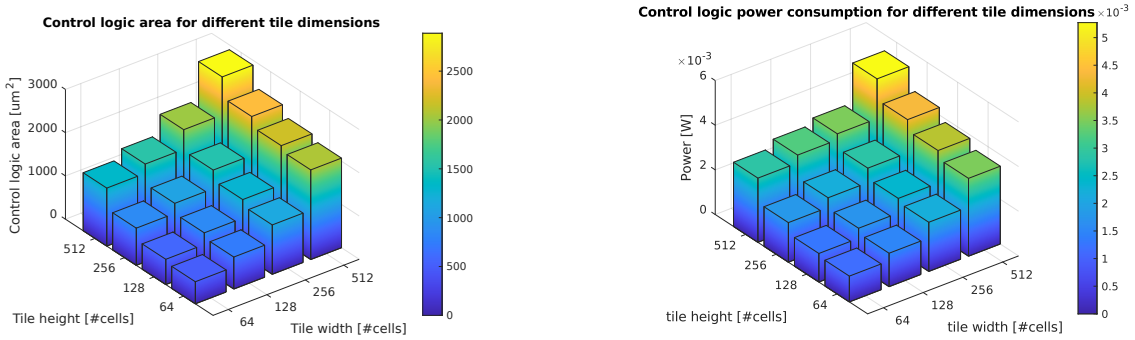
In Chapter 6, we describe the procedure used for the evaluation of the architectures proposed in Chapter 5. In this chapter, we present the results obtained through those methods. Firstly, we discuss the synthesis results and provide an explanation for trends found by changing hardware parameters in Section 7.1. Furthermore, we present the results of the design space exploration and the parameters used for the final optimized designs in Section 7.2. These designs are then evaluated based on their performance, accuracy, and end-to-end execution time compared to the state-of-the-art in Sections 7.3 and 7.4. Furthermore, we identify the proposed design's strengths and weaknesses and discuss solutions to further improve the end-to-end execution time through the use of cascading with other filters in Section 7.6. This is followed by an evaluation of the power and area evaluation of the proposed designs in Section 7.7. Lastly, the main findings of the chapter are summarized in Section 7.8.

7.1. Effect of Parameters

In this section, we discuss the synthesis results of each of the hierarchy levels. We examine the effects of varying input parameters while keeping the rest of the system constant. Furthermore, we provide explanations for why certain trends occur.

7.1.1. Tile Dimensions

We examine the effects of changing tile dimensions on the control logic area and power consumption of a tile. The results for the evaluated parameter ranges are shown in Figure 7.1(a) and 7.1(b).



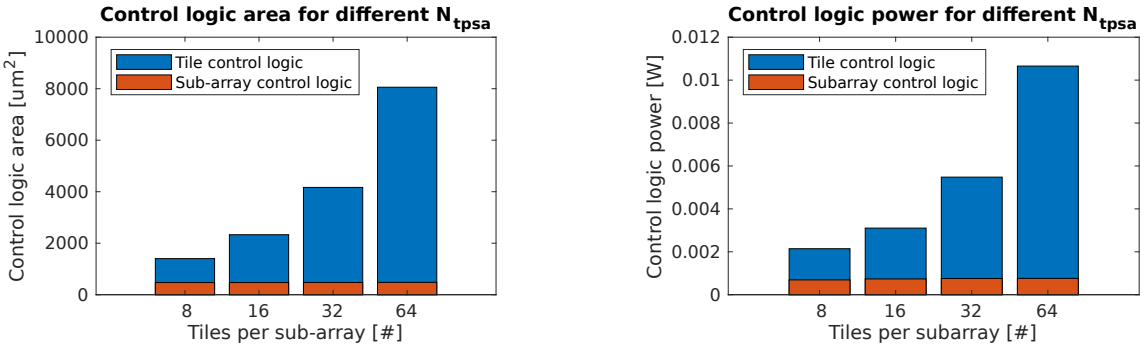
(a) (b)

Figure 7.1: An overview of the tile (a) control logic area and (b) control logic power for various crossbar dimensions.

Here it can be seen that area and power show similar trends in terms of scaling, as for both metrics, the control logic overhead increases for larger tile sizes. However, a doubling in the number of rows/columns translates to an area/power increase of less than 2x. This means that the overhead per cell decreases with an increase in crossbar size. Also, we observe that the tile width has a greater impact on area and power than the tile height. This can be explained by the fact that increasing the tile width increases the size of the write-buffer register, write driver select, and MUX/DEMUX logic. This is in contrast to the tile height, which only impacts the size of the row select registers. Furthermore, we observe that for a given memory capacity, the lowest overheads are achieved with a topology that is as close to square as possible.

7.1.2. Sub-array Configurations

We examine the impact of the number of tiles per sub-array (N_{tpsa}) on the overall control logic area/power in Figure 7.2(a) and 7.2(b).

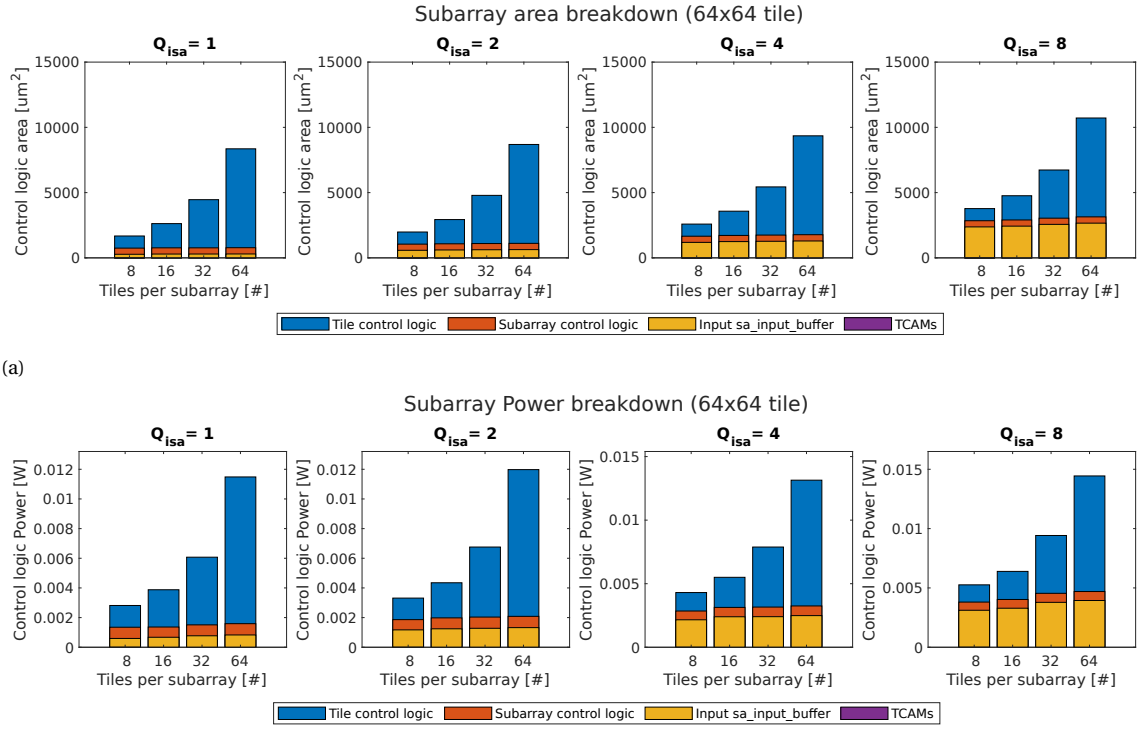


(a) (b)

Figure 7.2: A comparison between the sub-array and tile (a) control logic area and (b) control logic power for various numbers of tiles per sub-array.

Here we find that the size of the sub-array control logic area and power is not significantly impacted by an increase in N_{tpsa}. Moreover, for a large value of N_{tpsa}, the overhead added by the sub-array is overshadowed by the tile-level control logic. From a power/area perspective, it is therefore favored to have a large number of tiles per sub-array. However, as this increases the overall capacity of the sub-array, it also increases the likelihood of contention occurring.

In Figure 7.3, the results for the sub-array level area and power consumption can be found for different SA input queue lengths, the smallest tile size of 64x64, and 16x16 TCAMs.



(a)

(b)

Figure 7.3: Breakdown of the sub-array components' contribution to the overall (a) area and (b) power consumption for different input queue sizes.

In Figure 7.3(a), we find that for all configurations, the TCAM has an insignificant contribution to the overall area. Furthermore, we find that for small input queues, the overhead added by the queues is small compared to the overall area, especially for large numbers of tiles per sub-array. However, starting from $Q_{isa} = 4$, the queue starts to become a dominant factor in the overall area for certain configurations. We make the same observations when considering the power consumption of the control-logic in Figure 7.3(b). Here we find that the contribution of the queues is even larger than is the case for the area.

In Figure 7.4, we explore the effects of varying sub-array queue sizes on the performance of the system. We go over three different configurations, with different tile dimensions. This in effect changes the size of the sub-arrays, and therefore the degree of contention. For each of these configurations, we measure the effect of increasing the queue size on the average performance of the short-read datasets. Here, we find that for all tile configurations, we observe diminishing returns to execution time as we increase the queue length beyond 4 entries. Furthermore, we see that the effect of the queue size more heavily influences the performance for large sub-array sizes, as contention is more likely to occur in these configurations.

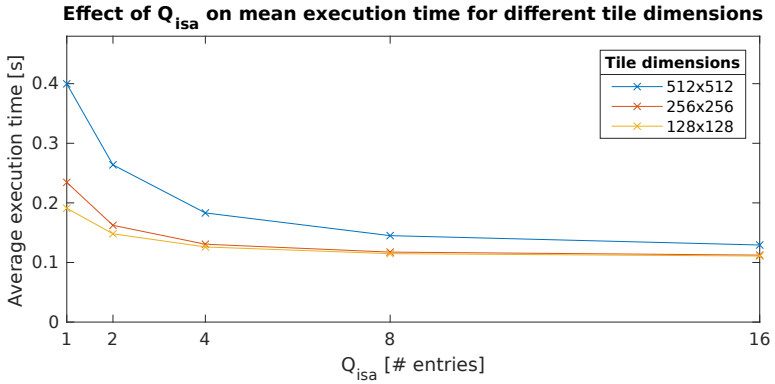


Figure 7.4: Effect of increasing the sub-array queue length on the mean execution time of short reads for different tile dimensions.

7.1.3. Bank, Bank-group, and Rank Configurations

We summarize the contribution of the higher-level components (bank, bank-group, and rank) in Figure 7.5. Here we plot the area and power breakdown of a configuration using the minimum values of the low-level (sub-array and tile) and high-level components (rank, bank-group, and bank). This is done to create a configuration with as much relative overhead from the high-level components. This is done while varying the number of buffer entries in the rank level. We observe that even when using this extreme hardware configuration, the overheads added by the bank, bank-group, and rank levels are insignificant in comparison with that of the lower-level components due to their lower quantity.

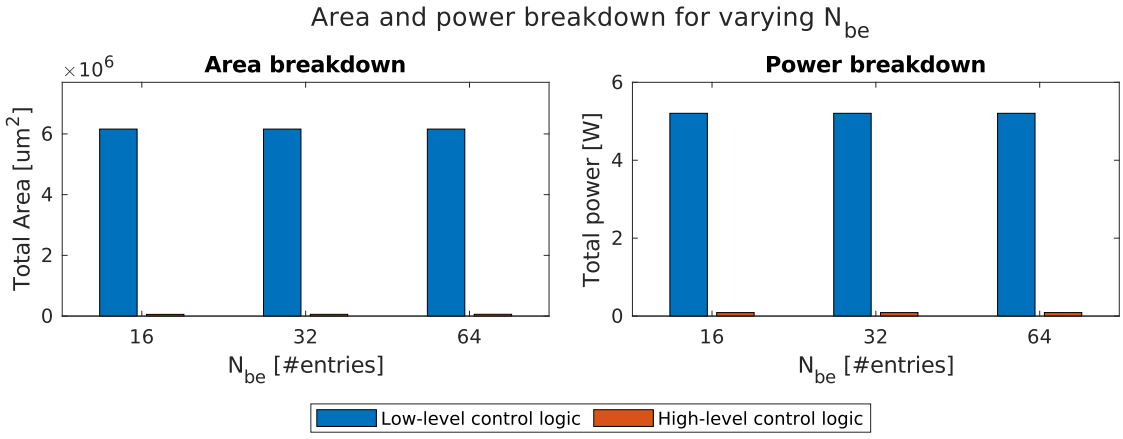
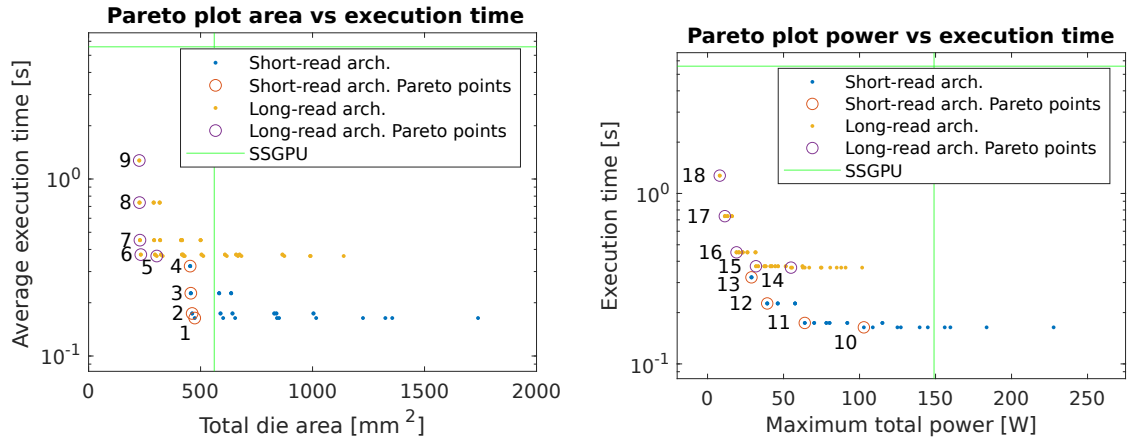


Figure 7.5: Control-logic area and power of high-level components (rank, bank-group, and bank) compared to low-level components (sub-array and tile).

7.2. Design Space Exploration

In this section, we discuss the results of the design space exploration. We perform an exhaustive parameter sweep of the hardware parameters listed in Table 6.3. The results of this evaluation are compared in terms of performance per area and performance per watt. The Pareto plot for the optimization of performance and area of short-reads can be found in Figure 7.6(a). Here the Pareto optimal configurations are labeled by numbers, of which the hardware configurations can be found in Table 7.1. We observe that the short-read optimized architecture generally has higher performance at the cost of increased chip area as compared to the long-read architecture. However, in both architectures, the Pareto optimal configurations are smaller than that of the Nvidia K80 GPU used for the evaluation of SS-GPU. Furthermore, we observe that for the area-optimized designs, the configurations with large tile dimensions are favored due to their smaller tile-control-logic area. From the Pareto optimal configurations for the minimization of area, we select configurations 3 and 7 as the best balance of power and area.



(a) (b)

Figure 7.6: Pareto plots for (a) performance per area and (b) performance per power.

Table 7.1: Pareto optimal configurations when optimizing for performance per area, using (top) the short-read architecture, and (bottom) the long-read architecture.

Design	**Conf.**	N_{rows}	N_{cols}	N_{tpsa}	N_{sapb}	N_{bpbg}	**P[W]**	**A[mm^2]**	**Mean exec. time**
Short	1	512	512	8	64	16	227.7	474	0.16
	2	512	512	16	64	8	115.0	462	0.17
	3	**512**	**512**	**32**	**64**	**4**	**57.6**	**456**	**0.23**
	4	512	512	64	32	4	28.9	453	0.32
Long	5	256	512	8	64	16	87.9	304	0.37
	6	512	512	8	64	8	62.5	233	0.37
	7	**512**	**512**	**16**	**64**	**4**	**31.5**	**229**	**0.45**
	8	512	512	32	32	4	16.0	227	0.74
	9	512	512	64	16	4	8.0	226	1.27

The Pareto plot for the optimization of performance and power of short-reads can be found in Figure 7.6(b). Again, it can be seen that the short-read architecture tends to have a larger total power consumption than the long-read architecture while providing better performance. Furthermore, we observe that for all Pareto-optimal configurations, the power consumption is lower than that of the examined GPU for SS-GPU. From the configuration specifications in Table 7.2, we observe that this optimization criterion favors configurations with large numbers of tiles per sub-array. This can be explained by the fact that these configurations have fewer active tiles at a given moment, as each sub-array only uses part of its available tiles at a time. From the Pareto optimal configurations for the minimization of power, we select configurations 12 and 16 as the best balance of power and area.

Out of all configurations, we find that the best configurations for both optimization strategies yield Pareto

Table 7.2: Pareto optimal configurations when optimizing for performance per watt, using (top) the short-read architecture, and (bottom) the long-read architecture.

Design	**Conf.**	N_{rows}	N_{cols}	N_{tpsa}	N_{sapb}	N_{bpbg}	**P[W]**	**A[mm^2]**	**Mean exec. time**
Short	10	128	256	64	64	16	102.8	1225	0.16
	11	256	256	64	64	8	63.9	829	0.17
	12	**256**	**512**	**64**	**64**	**4**	**39.2**	**583**	**0.23**
	13	512	512	64	32	4	28.9	453	0.32
Long	14	128	128	64	64	16	54.9	989	0.37
	15	128	256	64	64	8	31.8	609	0.37
	16	**256**	**256**	**64**	**64**	**4**	**19.0**	**413**	**0.45**
	17	256	512	64	32	4	11.4	291	0.74
	18	512	512	64	16	4	8.0	226	1.27

optimal configurations with the same performance figures, as this is only dependent on the size and quantity of sub-arrays in the system. The power/area balance of these configurations, however, differs per optimization strategy. We find that configurations 3 (area-optimized short-read) and 12 (power-optimized short-read) have the same average execution time. However, we find that configuration 12 has an area overhead greater than that of the GPU used for the evaluation of SS-GPU. For this reason, we select configuration 3 as the optimal configuration for the short-read architecture.

For the long-read architecture, we find that configuration 7 (area-optimized long-read) has a 66% greater power consumption than configuration 16 (power-optimized long-read). However, configuration 16 has an 80% greater chip area. Therefore, we select configuration 7 as the optimal configuration for the long-read architecture.

7.3. Performance Results

Using these optimal configurations, we measure the execution time of the datasets and edit-distance thresholds evaluated in Chapter 3. We observe that the filtering times of SS-CIM and SHD-CIM are identical. As we find that it provides better accuracy than SHD-CIM in all evaluated data sets and edit-distance thresholds, we focus on the performance of SS-CIM in the remainder of this chapter. The results of SHD-CIM can be found in Appendix B.5 to Appendix B.8. The short-read datasets are used to compare the performance of SS-GPU with the two CIM architectures, while the long-read datasets are used for comparison between SS-CPU and the long-read architecture only. The results of this comparison can be found in Figure 7.7 (note the use of the logarithmic scale in the long-read results), with a version zoomed in on the region of interest in Figure B.1. We observe a decrease in the filtering time for all evaluated datasets and edit-distance thresholds for both architectures. The speedup ranges in the region of interest for all datasets and architectures are listed in Table 7.3.

Table 7.3: Filtering time improvement ranges of SS-CIM over SS-GPU for short reads and over SS-CPU for long reads in the region of interest.

Dataset	Short-read architecture		Long-read architecture	
	Min	Max	Min	Max
Short_100bps	7.3x	7.9x	3.8x	7.3x
Short_250bps	92.3x	95.5x	14.9x	28.6x
Long_10k_inacc	-	-	206.2x	216.6x
Long_100k_inacc	-	-	146.3x	173.1x
Long_10k_acc	-	-	35.4x	120.3x
Long_100k_acc	-	-	32.8x	120.6x

We observe that the short-read architecture and long-read architecture perform identically when the absolute edit-distance threshold does not exceed the number of stored reference copies. The performance of the long-read architecture increases in a staggered fashion when moving to larger edit-distance thresholds due to the larger number of write operations required to write all shift-sets.

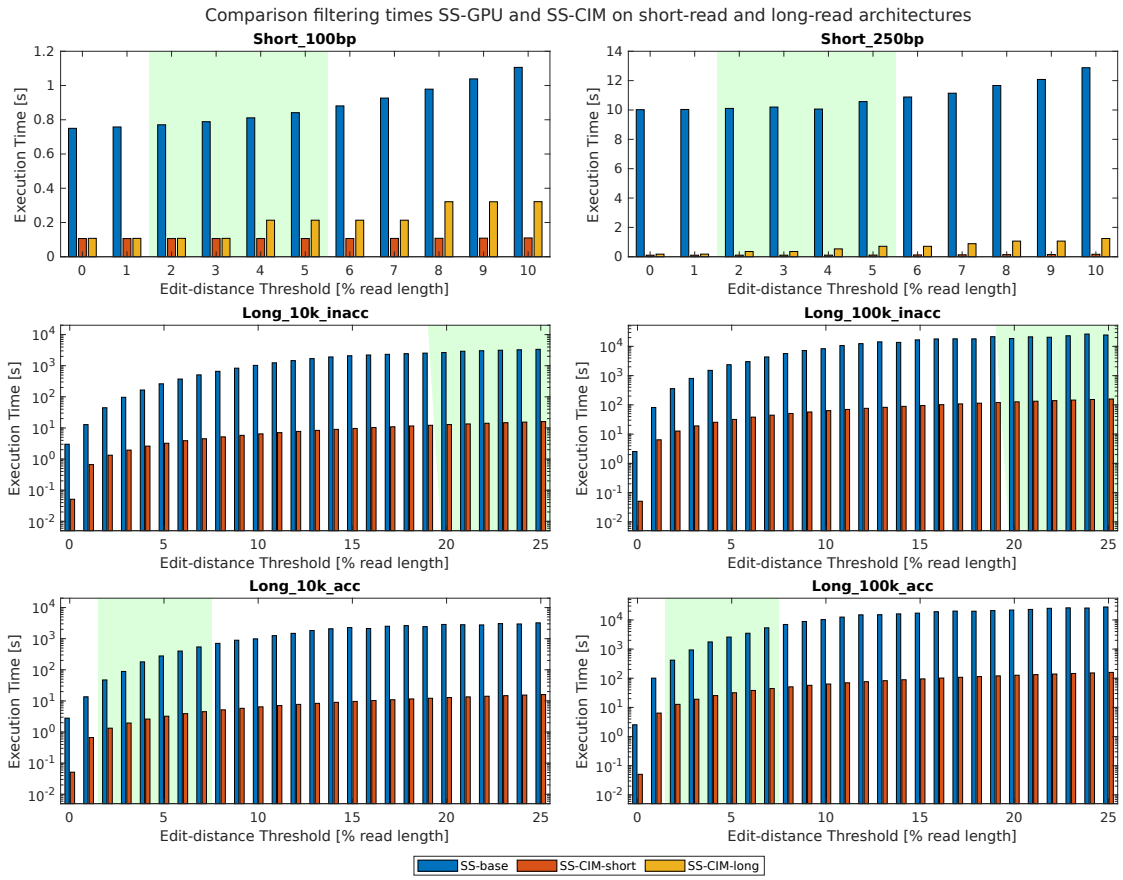


Figure 7.7: Comparison filtering times of SS-CIM to the state-of-the-art.

7.4. Accuracy Results

In this section, we discuss the accuracy of SS-CIM on the optimal design. Here we focus on the false-positive rate and the total positive rate.

7.4.1. False-positive Rate

The false positive rate of the optimal configuration of SS-CIM is shown in Figure 7.8. Here it is compared to the accuracy of the most accurate state-of-the-art filters (SS-GPU for short reads and SS-CPU for long reads). We use this metric to determine for which edit-distance thresholds the filters are most effective. For this, we examine the entire range of edit-distance thresholds. We observe that, in the case of short reads and accurate long reads, both SS-CIM and SS-GPU have a low false positive rate in the region of interest (<0.1%). Only for larger edit-distance thresholds outside the region of interest, the difference in accuracy becomes more apparent. For inaccurate long reads, the FP-rate in the region of interest is between 80 and 100 % for both algorithms, which indicates that both filtering solutions do not perform well for this type of read. For completeness, we provide a zoomed-in version of this figure in Appendix B.1, which better illustrates the difference between the accuracies of the implementations in the regions of interest. We discuss these results in more detail in the following section on the total-positive rate.

7.4.2. Total-positive Rate

To examine the total number of pairings that pass the filter, we examine the positive rate of the optimal configurations of SS-CIM, as this gives an indication of how many pairings need to be evaluated using exact alignment. The results of this comparison can be found in Figure 7.9, here we focus on the positive rate of the region of interest, as it needs to be confirmed that the increase in P-rate remains low in this region. The full results can be found in Appendix B.2. Here, the absolute increase in positive rate over the baseline SneakySnake implementation is given above each bar. Here it can be seen that the increase in positive rate is ranging from

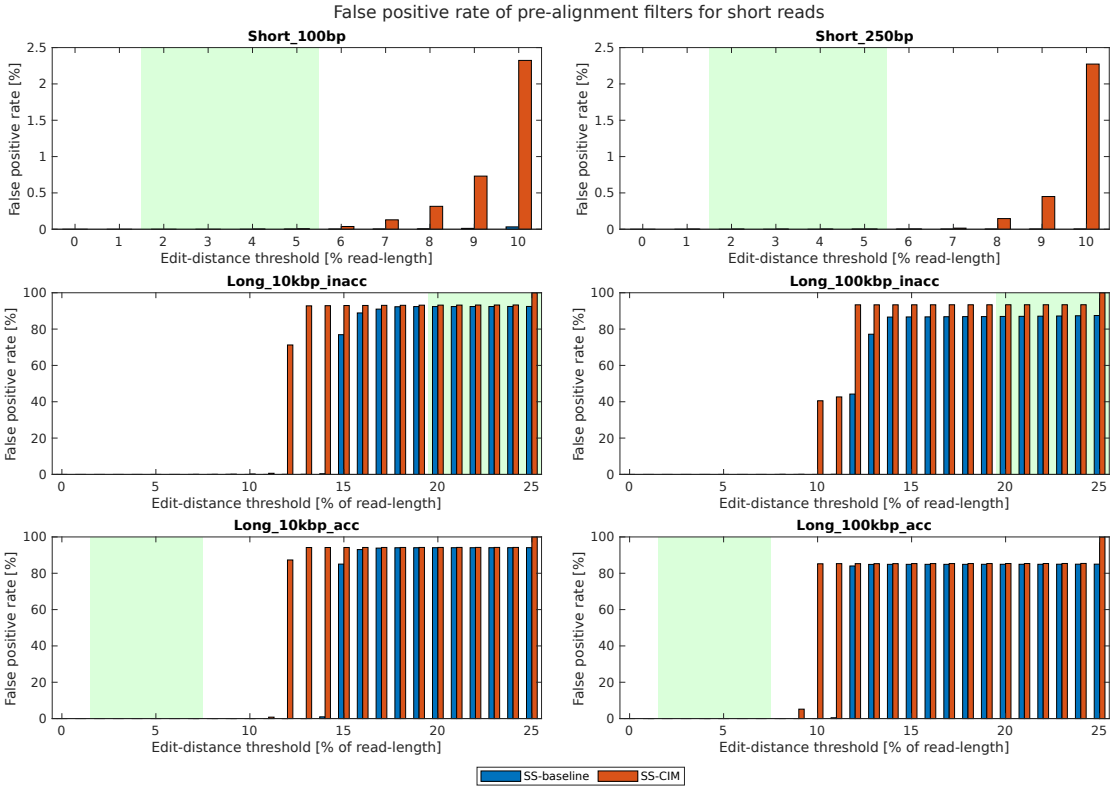


Figure 7.8: Comparison between the false-positive rate of SS-CIM and the state-of-the-art.

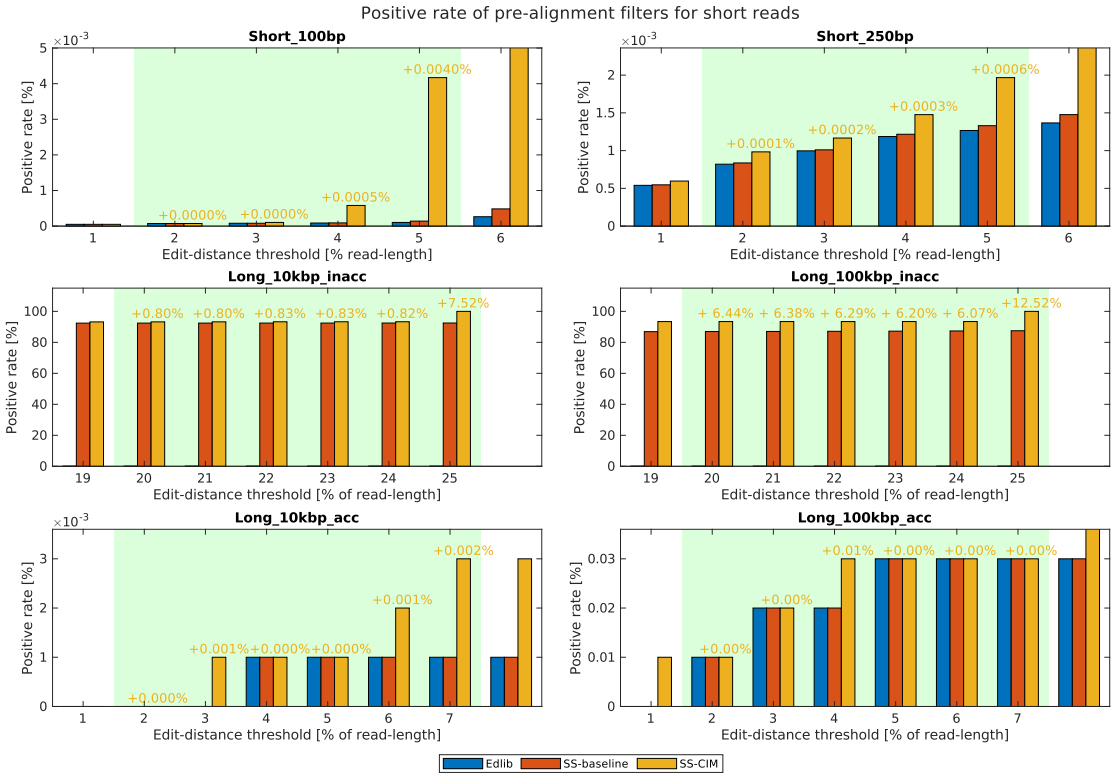


Figure 7.9: Comparison between the positive rate of SS-CIM and the state-of-the-art.

0% to 12.5% depending on the data set and edit-distance threshold. We find that, while the relative increase in positive rate is significant when compared to SS-GPU/SS-CPU, the absolute increase is low, and remains below the maximum allowable P-rate determined in Figure 4.2.

7.5. End-to-end Results

In this section, we discuss the end-to-end execution time of the pre-alignment filter combined with exact alignment with Edlib. The results obtained for the examined data sets and edit-distance thresholds can be found in Figure 7.10. Here it can be seen that for all data sets and edit-distance thresholds, both SS-CIM architectures show end-to-end improvement over the baseline solution (SS-GPU and SS-CPU for short and long reads, respectively) in the region of interest. This indicates that the increase in performance is large enough to compensate for the decrease in accuracy found in Figure 7.9.

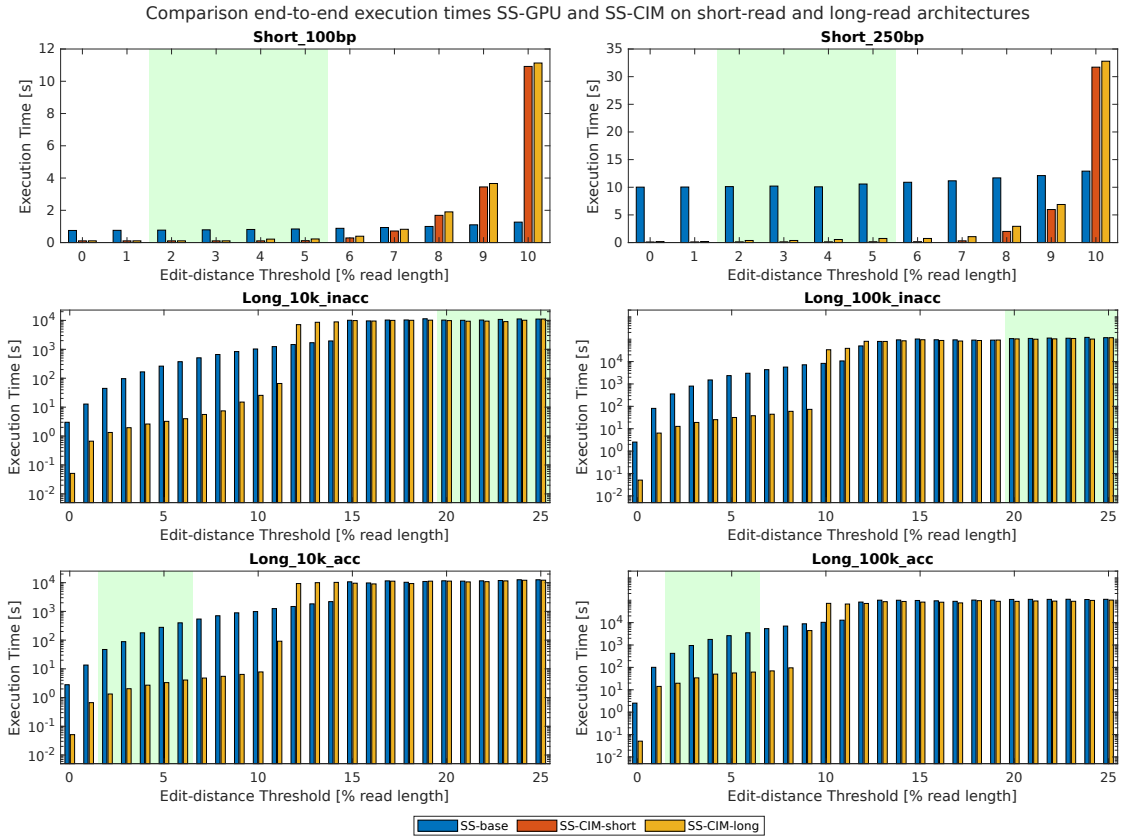


Figure 7.10: Comparison end-to-end execution times of SS-CIM to the state-of-the-art.

The end-to-end improvement ranges for all data sets and edit-distance thresholds in the region of interest can be found in Table 7.4. A per-edit-distance-threshold breakdown can be found in Appendix B.3.

Table 7.4: End-to-end execution time improvement ranges of SS-CIM over SS-GPU for short reads and over SS-CPU for long reads in the region of interest.

Dataset	Short-read architecture		Long-read architecture	
	Min	Max	Min	Max
Short_100bps	**7.2x**	**7.6x**	3.8x	7.4x
Short_250bps	**74.1x**	**80.5x**	14.3x	27.1x
Long_10k_inacc	-	-	**1.00x**	**1.19x**
Long_100k_inacc	-	-	**1.00x**	**1.19x**
Long_10k_acc	-	-	**35.4x**	**98.1x**
Long_100k_acc	-	-	**21.6x**	**57.0x**

Here we find that in all short-read configurations, using the filter shows a net benefit to the total execution time, ranging from 7.2x to 80.5x for the short-read architecture and from 3.8x to 27.1x for the long-read architecture. Here we find that the short-read architecture achieves up to 2x greater minimum improvements than the long-read architecture for 100 bps reads. This difference is even greater for 250 bps reads, where the difference is over 5x between the two designs. This can be attributed to the increase in the number of required write operations in the long-read architecture. Furthermore, we find that for inaccurate long reads, a modest increase in performance ranging from 1x to 1.19x can be achieved using SS-CIM on the long read architecture. The decrease in execution time is small due to the poor accuracy of the baseline algorithm for this type of read. For accurate long reads, on the other hand, increases in performance ranging from 21.6x to 98.1x can be achieved due to the high accuracy in the region of interest.

To determine whether the filter-bottleneck has been removed with the new architecture, we examine the distribution of filtering time and alignment time in Figure 7.11.

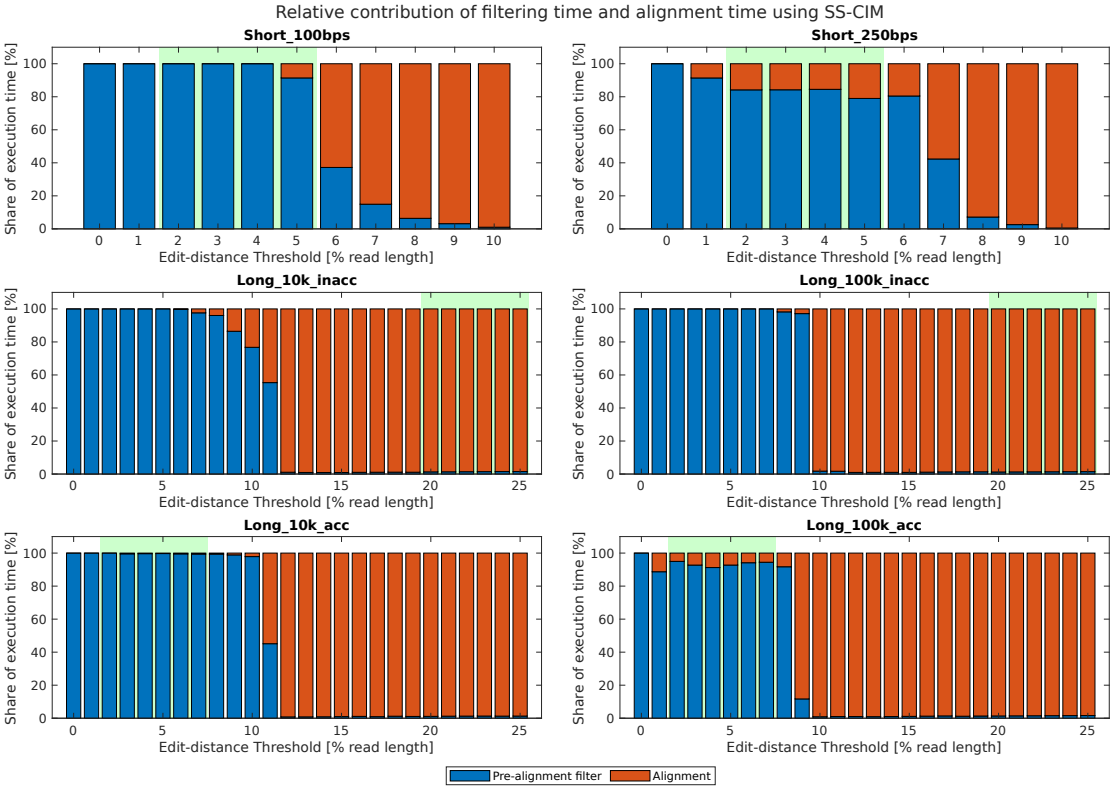


Figure 7.11: Relative contribution of pre-alignment filtering and alignment to the end-to-end execution time using SS-CIM.

Here we find that in the region of interest for short reads, most of the execution time can be attributed to pre-alignment filtering, though, in some cases, alignment takes up a considerable share (>20%) of the end-to-end execution time. A similar distribution can be found for accurate long reads, where alignment takes up to 25% of the end-to-end execution time. For inaccurate long reads, we find that the vast majority (>95%) of the time is spent on alignment, which can be attributed to the poor accuracy of the algorithm for this type of read. Furthermore, we observe that outside of the regions of interest, this distribution alignment can take up a majority of the end-to-end execution time for all data sets.

7.6. Filter Cascading

Since alignment has a considerable contribution to the end-to-end execution time for some data sets and edit-distances, better performance can be achieved by improving the accuracy of the filtering stage. Therefore, we propose a cascaded design, where the outputs of SS-CIM are used as input to SS-GPU/SS-CPU. In effect, SS-CIM is used as a fast but coarse filter, which is followed by a slower but more accurate filter. This has the potential to leverage both the fast execution time of SS-CIM and the high accuracy of SS-GPU/SS-CPU.

We compare the end-to-end execution time of this cascaded design with that of stand-alone SS-CIM in Figure 7.12. Here we plot the execution time of SS-CPU and the minimum execution time of stand-alone SS-CIM and the cascaded design. The edit-distances at which the cascaded design offers improvement are marked in blue. Additionally, a zoomed version of the same graph can be found in Appendix B.4, where we focus on regions of interest. We find that the cascaded design offers improvement when two conditions are met: 1) the disparity in accuracy between SS-CIM and SS-CPU is sufficiently large (as found in figure 7.8), and 2) SS-CPU shows improvement in end-to-end execution time over alignment without pre-alignment filtering (as found in Figure 3.10).

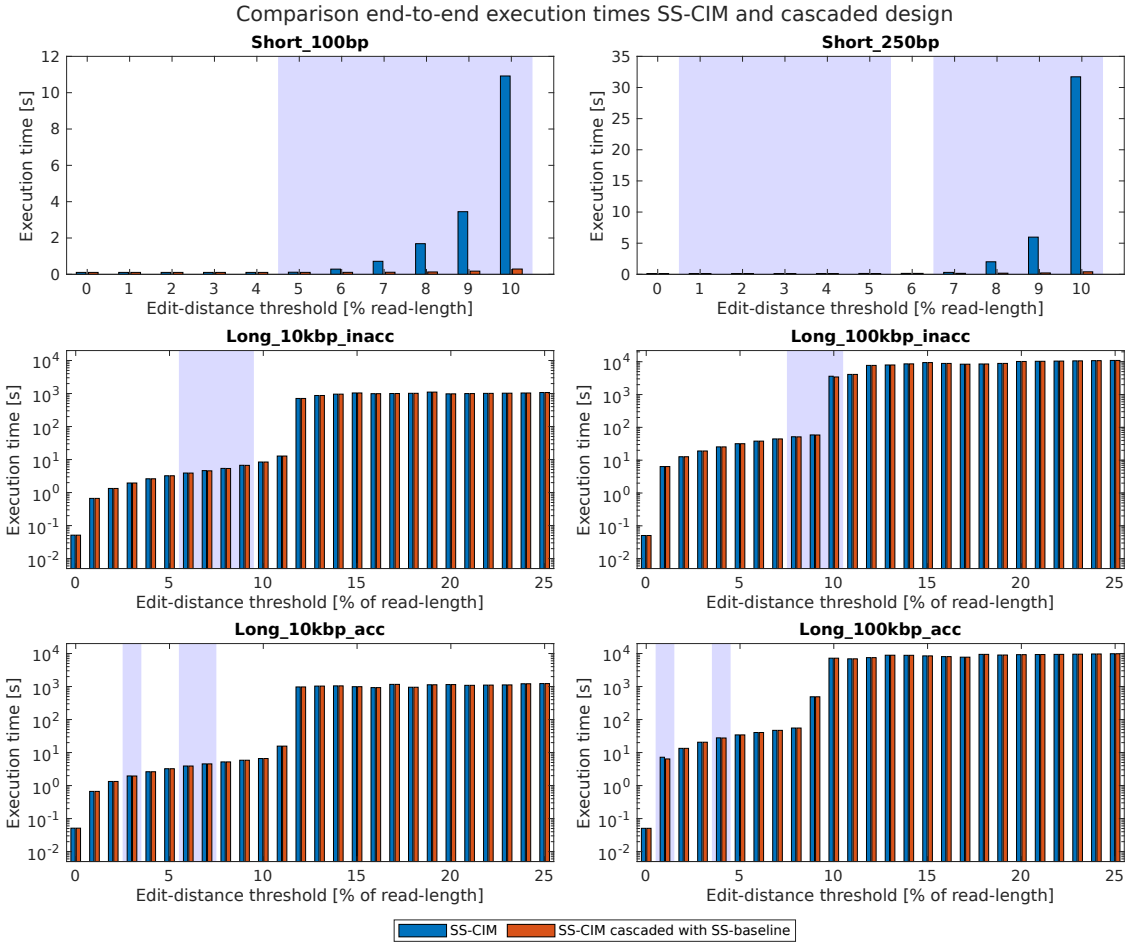


Figure 7.12: Comparison between stand-alone SS-CIM and SS-CIM cascaded with SS-GPU and SS-CPU for short and long reads, respectively.

The improvement range of the cascaded design in the region of interest can be found in Table 7.5, where it is compared to the short read architecture for short reads and the long read architecture for long reads. Here we find that a modest improvement of up to 1.09x can be achieved for some data sets and edit-distance

Table 7.5: Improvement range of the cascaded design of SS-CIM over stand-alone SS-CIM.

Data set	Min	Max
Short_100bps	1.00x	1.08x
Short_250bps	1.00x	1.07x
Long_10k_inacc	1.00x	1.00x
Long_100k_inacc	1.00x	1.00x
Long_10k_acc	1.00x	1.00x
Long_100k_acc	1.00x	1.01x

thresholds, with most improvement gained outside the region of interest. While the improvement is small for the examined data sets, this approach could be useful for other (future) types of reads which are not considered in this evaluation.

7.7. Power & Area Comparison

For the power and area comparison, we consider the optimal designs found using the design space exploration, the specifications of which can be found in Table 7.6. Here the queue lengths not listed in the table are set to 1, as they have no significant impact on performance.

Table 7.6: Final configurations for the short and long-read architectures.

Design	N_{rows}	N_{cols}	N_{tpsa}	N_{sapb}	N_{bpbg}	N_{bgpr}	WPB	Q_{isa}	N_{be}	N_{copies}
Short	512	512	32	64	4	8	4	4	64	$2E+1$
Long	512	512	16	64	4	8	4	4	64	8

We examine the chip area and power consumption of the optimal configurations of each of the proposed designs in Figure 7.13. Here we examine the relative contributions of the crossbar, control logic, and TCAMs to the chip area and power consumption of the entire design. Here we find that the majority of chip-area can be attributed to the crossbars and the tile-level control logic. The contributions of the higher-level components and TCAMs is negligible in comparison. For power, the largest contributor is the tile-level control logic. Where we again find that the TCAMs add insignificant power consumption overheads.

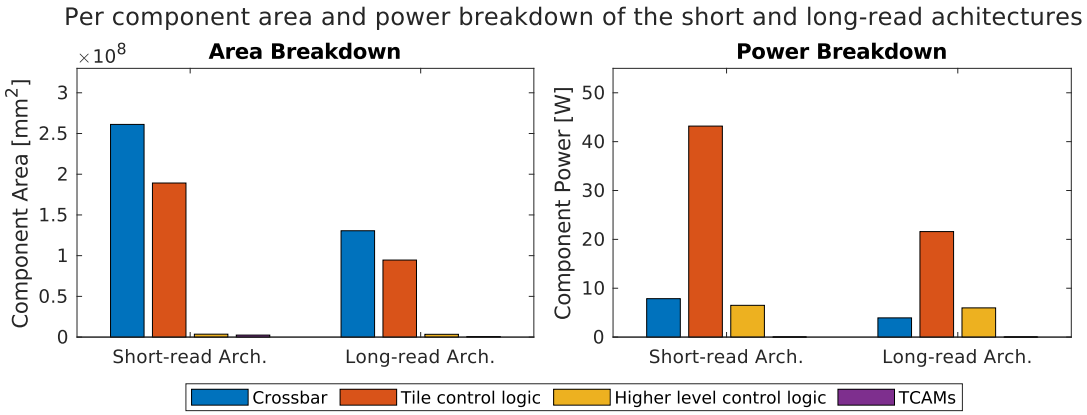


Figure 7.13: Per-component power and area breakdown of the optimal configurations of the short and long read architectures.

We compare the chip area and power consumption of the two proposed designs with the current state-of-the-art in Table 7.7, where the maximum GPU power is measured using `nvidia-smi` while running SS-GPU. We find that both designs have a smaller overall chip area than the NVIDIA Tesla K80 GPU used for the evaluation of SS-GPU. Furthermore, both designs show a lower maximum power consumption than the GPU implementation (-63.9% and -79.7% for the short and long-read architecture, respectively). Furthermore, it should be noted that the area and power estimations of the proposed architecture are scaled pessimistically, leading to conservative estimates.

Table 7.7: Comparison between the proposed architectures and the GPU used for the evaluation of SS-GPU.

Hardware	**Area** [mm^2]	**Power** [W]
NVIDIA Tesla K80 [112]	561	149
Short read architecture	456	57.6
	-18.7%	-63.9%
Long read architecture	229	31.5
	-59.2%	-79.7%

7.8. Conclusions

In this chapter, we discuss the result obtained with the process described in Chapter 6. We first discuss the synthesis results for the digital parts of the design. These are used in conjunction with the analytical model to perform a design space exploration based on parameters found in other memristor-based CIM designs. From the design space exploration, we find the optimal parameters for the two proposed architectures. We evaluate the performance of these optimal designs by evaluating their accuracy, filtering time, and end-to-end execution time.

We find that for all evaluated data sets and edit-distance thresholds, both proposed designs show improvement over the current state of the art in terms of filtering time. While the accuracy of SS-CIM is lower than that of existing implementations of pre-alignment filters, we find that the improvement in filtering time is enough to compensate for the resulting increase in alignment time. Here we find that the short-read architecture achieves 7.2x to 80.5x improvement in end-to-end execution time over the current state of the art. The long-read architecture achieves a smaller improvement ranging from 3.8x to 27.1x on the same datasets. Furthermore, the long read architecture provides a 1x to 1.19x improvement to the end-to-end execution time for inaccurate long reads, which is attributed to the poor accuracy of SS-CIM for this type of read. While this is only a minor improvement, we find that there is also no degradation in performance as is the case SS-CPU. For accurate long reads, a speedup of 21.6x to 98.1x is achieved.

Furthermore, we examine the relative contribution to the end-to-end execution time of the filtering and the alignment stage of SS-CIM. Here we find that for some data sets and edit-distance thresholds, alignment takes up the majority of the end-to-end execution time. As a further improvement to stand-alone SS-CIM, we propose a cascaded design, where the outputs of SS-CIM are used as input to SS-GPU/SS-CPU, creating a 2-step filter. Using this configuration, we achieve an additional 1.08x improvement over stand-alone SS-CIM in certain cases.

Lastly, we compare the chip area and power consumption of both designs to that of the GPU used for the evaluation of SS-GPU. Here we find an 18.7% and 59.2% reduction in chip area for the short and long-read architecture, respectively. The reduction in power consumption of the designs is between 63.9% and 79.7% respectively as compared to the average power consumption of the GPU.

Chapter 8

Conclusion

8.1. Summary

Due to the growing availability of DNA-sequence data and an increasing number of applications of DNA sequence processing, there has been an increase in demand for high-speed DNA-sequence analysis. One of the major bottlenecks of this is the alignment step, to which pre-alignment filtering has shown to be a promising solution. In this work, we explore current state-of-the-art pre-alignment filters and find that the hardware accelerators used to implement them are limited in throughput by data movement between the host and the accelerator. To alleviate this bottleneck we propose Computation In Memory (CIM) as a solution.

To do so, we first benchmark current pre-alignment filters and measure their accuracy and filtering times using a wide range of input datasets and edit-distance thresholds. From this, it is found that the best end-to-end performance is obtained using SneakySnake, which is implemented on GPU and CPU for short and long reads, respectively. This algorithm is therefore taken as a baseline for comparison. Alongside benchmarking, we profile multiple state-of-the-art algorithms, from which we find seven key operations that form the foundation of the algorithms, which need to be supported by the proposed CIM-architecture.

For the architecture, we opt for a hardware-software co-design. Firstly, we transform SneakySnake into a novel CIM-friendly algorithm: SneakySnake-CIM (SS-CIM). This algorithm splits up the SneakySnake algorithm into several sub-problems which can be solved individually using CIM-P operations. Secondly, we provide an overview of the proposed CIM-architecture capable of implementing the pre-alignment filters. This architecture has a structure akin to those found in existing DRAM memories, which is augmented with peripheral devices used to implement the pre-alignment filtering algorithms.

We demonstrate the architecture's flexibility using example mappings for two different pre-alignment filtering algorithms, SHD and SS-CIM, which together cover all seven commonly found operations. Furthermore, we identify possible shortcomings of the architecture for the mapping of long reads and provide an additional design as a solution. This leaves us with two architectures: one designed for short reads specifically, and one that supports both short and long reads, at the cost of lower performance.

These architectures are implemented in VHDL and their functionality is verified through RTL-simulation. Furthermore, we present an analytical model and a performance model based on synthesis results, which aid in obtaining the optimal configuration of the architectures.

Finally, we discuss the results obtained with this methodology. We discuss the synthesis results for the digital parts of the design and explain the effects of changing each parameter on the overall performance and area/power overheads. Furthermore, we perform a design space exploration to find the optimal parameters for both designs and we examine the performance of the proposed architectures in terms of accuracy and filtering speed.

Using these configurations, we find that the short-read architecture achieves 7.2x to 80.5x improvement in end-to-end execution time over the current state-of-the-art. The long read architecture achieves a smaller improvement ranging from 3.8x to 27.1x on the same datasets and a speedup of 21.6x to 98.1x for accurate long reads. For inaccurate long reads, the long read architecture provides a 1.0x to 1.19x improvement to the end-to-end execution time. While this is only a minor improvement, we find that there is no degradation in performance, which is the case when using SS-CPU. Additionally, as a further improvement over stand-alone

SS-CIM, we propose a cascaded design, where the outputs of SS-CIM are used as input to SS-GPU/SS-CPU, creating a 2-step filter. Using this configuration, we achieve an additional 1.08x improvement over stand-alone SS-CIM in certain data sets.

Lastly, we compare the chip area and power consumption of both designs to that of the GPU used for the evaluation of SS-GPU. Here we find an 18.7% and 59% reduction in chip area for the short and long-read architecture, respectively. In addition to that, we find a reduction in power consumption of 63.9% and 79.7% for the short and long-read architecture, respectively.

8.2. Main contributions

Returning to the problem statement and research goals proposed in Chapter 1, we find that all research goals have been fulfilled. The main contributions of this thesis are listed below.

- Creation of a fair comparison platform for existing pre-alignment algorithms, evaluating them in terms of accuracy, filtering time, and end-to-end execution time.
- Identification of the main bottlenecks of existing hardware-accelerated pre-alignment filters.
- Identification of common operations shared amongst pre-alignment filtering algorithms.
- Creation of a novel CIM-friendly pre-alignment algorithm based on the commonly found operations.
- Creation of a CIM-architecture to support the common operations, capable of implementing multiple pre-alignment filtering algorithms, including the proposed algorithm, for a wide range of edit-distance thresholds and data sets.
- Verification and evaluation of the proposed architecture and comparison with the state-of-the-art in terms of performance, power, and area.
- A conference paper ready to be submitted soon after the date of the thesis defense.

To answer the research question, we conclude that currently existing pre-alignment filters can be transformed into CIM-friendly algorithms using a combination of seven commonly occurring operations. These operations can be mapped onto the proposed CIM architectures, by leveraging the programmable nature of the memory elements and TCAMs. This approach achieves improvements in end-to-end performance up to 98.1x as compared to the current state-of-the-art. Additionally, the proposed architectures see up to a 59% reduction in chip area over a GPU implementation, while achieving an 89% reduction in power consumption.

8.3. Future work

Future work based on the work presented in this thesis includes the following three points. Firstly, the evaluation of the analog parts of this work is based on performance specifications presented in previous works, which are conservatively scaled to estimate the total chip area and power consumption of the proposed architectures. To obtain a more accurate estimation, future work can be aimed at obtaining these numbers through circuit-level simulations of the crossbars and TCAMs.

Secondly, we find that none of the currently existing pre-alignment filtering solutions provide significant acceleration of the alignment of inaccurate long reads due to high false-positive rates. To improve on this, future work can be aimed at devising new algorithms using the proposed architectures that provide better accuracy for this type of read.

Thirdly, future work can be aimed at implementing the proposed pre-alignment filter directly in existing seed-and-extend-based read mappers. In the current implementation, read mappers create a hash table containing the seeding locations in the form of an offset from the start of a chromosome, which is translated into a memory address. For more efficient processing, the hashing process can be re-programmed to directly output the memory address. Furthermore, programming the architecture requires a relatively deep level of understanding of the underlying structure. To aid adaptation, a higher-level compiler can be developed, which also simplifies the integration with read mappers.

References

[1] Oxford Nanopore Technologies, "Assessing the utility of long-read nanopore sequencing for rapid and efficient characterisation of mobile element insertions," 2020. [Online]. Available: https://nanoporetech.com/resource-centre/assessing-utility-long-read-nanopore-sequencing-rapid-and-efficient

[2] GenScript, "Advancing genomics, medicine and health together – by semiconductor DNA synthesis technology," 2022. [Online]. Available: https://www.genscript.com/advancing-genomics-medicine-and-health-together-by-semiconductor-dna-synthesis-technology-summary.html

[3] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto et al., "Real-time DNA sequencing from single polymerase molecules," Science, vol. 323, no. 5910, pp. 133–138, 2009.

[4] L. Aigrain, "What is Oxford Nanopore Technology (ONT) sequencing?" 2021. [Online]. Available: https://www.yourgenome.org/facts/what-is-oxford-nanopore-technology-ont-sequencing/ https://www.yourgenome.org/facts/what-is-oxford-nanopore-technology-ont-sequencing

[5] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern DNA read alignment algorithms," Proceedings - 2016 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2016, pp. 1421–1428, 2017.

[6] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan et al., "Shifted Hamming distance: A fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," Bioinformatics, vol. 31, no. 10, pp. 1553–1560, 2015.

[7] M. Alser and O. Mutlu, "MAGNET: understanding and improving the accuracy of genome pre-alignment filtering," Transactions on Internet Research, vol. 13, no. 2, pp. 1–10, 2017.

[8] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser et al., "GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," BMC Genomics, vol. 19, no. Suppl 2, 2018.

[9] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: A fast and efficient pre-alignment filter for sequence alignment," Bioinformatics, vol. 35, no. 21, pp. 4255–4263, 2019.

[10] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "SneakySnake: A fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs," Bioinformatics, vol. 36, no. 22-23, pp. 5282–5290, 2020.

[11] Wikipedia, "Von Neumann Architecture," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Von_Neumann_architecture

[12] H. A. Nguyen, J. Yu, L. Xie, M. Taouil, S. Hamdioui, and D. Fey, "Memristive devices for computing: Beyond CMOS and beyond von Neumann," IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC, no. November 2018, 2017.

[13] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman et al., "MAGIC—Memristor-Aided Logic," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 11, pp. 895–899, 11 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6895258/

[14] L. Xie, H. A. Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. Alfailakawi et al., "Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing," Proceedings of IEEE Computer Society An-

nual Symposium on VLSI, ISVLSI, vol. 2017-July, pp. 176–181, 2017.

[15] L. Zheng, S. Shin, S. Lloyd, M. Gokhale, K. Kim, and S. M. Kang, "RRAM-based TCAMs for pattern search," Proceedings - IEEE International Symposium on Circuits and Systems, vol. 2016-July, pp. 1382–1385, 2016.

[16] Y. Xie and Y. Zhao, "Emerging memory technologies," IEEE Micro, vol. 39, no. 1, pp. 6–7, 2019.

[17] F. E. Dewey, S. Pan, M. T. Wheeler, S. R. Quake, and E. A. Ashley, "DNA sequencing clinical applications of new DNA sequencing technologies," Circulation, vol. 125, no. 7, pp. 931–944, 2 2012.

[18] B. Bruijns, R. Tiggelaar, and H. Gardeniers, "Massively parallel sequencing techniques for forensics: A review," Electrophoresis, vol. 39, no. 21, pp. 2642–2654, 2018.

[19] H. Stranneheim, K. Lagerstedt-Robinson, M. Magnusson, M. Kvarnung, D. Nilsson, N. Lesko et al., "Integration of whole genome sequencing into a healthcare setting: high diagnostic rates across multiple clinical entities in 3219 rare disease patients," Genome Medicine, vol. 13, no. 1, pp. 1–15, 2021.

[20] L. G. Gordon, N. M. White, T. M. Elliott, K. Nones, A. G. Beckhouse, A. J. Rodriguez-Acevedo et al., "Estimating the costs of genomic sequencing in cancer control," BMC Health Services Research, vol. 20, no. 1, pp. 1–11, 2020.

[21] T. Magdy, H. Kuo, Hui–Hsuan, and P. W. Burridge, "Precise and Cost-Effective Nanopore Sequencing for Post-GWAS Fine-Mapping and Causal Variant Identification," iScience, vol. 23, no. 4, p. 100971, 2020. [Online]. Available: https://doi.org/10.1016/j.isci.2020.100971

[22] S. T. Park and J. Kim, "Trends in next-generation sequencing and a new era for whole genome sequencing," International Neurourology Journal, vol. 20, pp. 76–83, 2016.

[23] T. Kwon, W. G. Yoo, W. J. Lee, W. Kim, and D. W. Kim, "Next-generation sequencing data analysis on cloud computing," Genes and Genomics, vol. 37, no. 6, pp. 489–501, 2015. [Online]. Available: http://dx.doi.org/10.1007/s13258-015-0280-7

[24] B. Langmead and A. Nellore, "Cloud computing for genomic data analysis and collaboration," Nature Reviews Genetics, vol. 19, no. 4, pp. 208–219, 2018. [Online]. Available: http://dx.doi.org/10.1038/nrg.2017.113

[25] M. C. F. Chang, Y. T. Chen, J. Cong, P. T. Huang, C. L. Kuo, and C. H. Yu, "The SMEM Seeding Acceleration for DNA Sequence Alignment," Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, pp. 32–39, 2016.

[26] S. Angizi, J. Sun, W. Zhang, and D. Fan, "AlignS: A processing-in-memory accelerator for DNA short read alignment leveraging SOT-MRAM," Proceedings - Design Automation Conference, 2019.

[27] B. Berger and Y. W. Yu, "Navigating bottlenecks and trade-offs in genomic data analysis," Nature Reviews Genetics, 2022.

[28] 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.

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

[30] Y. Sun and J. Buhler, "Choosing the best heuristic for seeded alignment of DNA sequences," BMC Bioinformatics, vol. 7, pp. 1–12, 2006.

[31] N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars, "GPU acceleration of Darwin read overlapper for de novo assembly of long DNA reads," BMC Bioinformatics, vol. 21, no. Suppl 13, pp. 1–17, 2020. [Online]. Available: http://dx.doi.org/10.1186/s12859-020-03685-1

[32] T. B. Preußer, O. Knodel, and R. G. Spallek, "PoC-Align: An open-source alignment accelerator using FPGAs," 2014 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2014, 2014.

[33] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with FastHASH," BMC Genomics, vol. 14, no. Suppl 1, pp. 1–13, 2013.

[34] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping," Bioinformatics (Oxford, England), vol. 33, no. 21, pp. 3355–3363, 2017.

[35] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu et al., "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016, pp. 14–26, 2016.

[36] A. Ankit, I. El Hajj, S. Rahul Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams et al., "PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference," International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, no. Ml, pp. 715–731, 2019.

[37] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, "RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment," Proceedings of the International Symposium on Low Power Electronics and Design, vol. 2019-July, no. 4, 2019.

[38] A. K. Rajput and M. Pattanaik, "Implementation of Boolean and Arithmetic Functions with 8T SRAM Cell for In-Memory Computation," 2020 International Conference for Emerging Technology, INCET 2020, pp. 8–12, 2020.

[39] G. Singh, A. Wagle, S. Khatri, and S. Vrudhula, "CIDAN-XE: Computing in DRAM with Artificial Neurons," Frontiers in Electronics, vol. 3, no. February, pp. 1–17, 2022.

[40] L. O. Chua, "Memristor-the Missing Circuit Element," ieee transactions on circuit theory, vol. c, no. 5, pp. 507–519, 1971.

[41] A. Makosiej, O. Thomas, A. Amara, and A. V. Escu, "CMOS SRAM scaling limits under optimum stability constraints," Proceedings - IEEE International Symposium on Circuits and Systems, pp. 1460–1463, 2013.

[42] S. K. Park, "Technology Scaling Challenge and Future Prospects of DRAM and NAND Flash Memory," 2015 IEEE 7th International Memory Workshop, IMW 2015, pp. 18–21, 2015.

[43] Y. Chen, "ReRAM: History, Status, and Future," IEEE Transactions on Electron Devices, vol. 67, no. 4, pp. 1420–1433, 4 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8961211/

[44] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis et al., "Spin-transfer torque magnetic random access memory (STT-MRAM)," ACM Journal on Emerging Technologies in Computing Systems, vol. 9, no. 2, pp. 1–35, 2013.

[45] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson et al., "Phase change memory technology," Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena, vol. 28, no. 2, pp. 223–262, 2010.

[46] F. H. Crick, J. C. Wang, and W. R. Bauer, "Is DNA really a double helix?" Journal of Molecular Biology, vol. 129, no. 3, pp. 449–461, 1979.

[47] A. Ghosh and M. Bansal, "A glossary of DNA structures from A to Z," Acta Crystallographica - Section D Biological Crystallography, vol. 59, no. 4, pp. 620–626, 2003.

[48] A. J. Brookes and P. N. Robinson, "Human genotype-phenotype databases: Aims, challenges and opportunities," Nature Reviews Genetics, vol. 16, no. 12, pp. 702–715, 2015.

[49] P. de Knijff, "From next generation sequencing to now generation sequencing in forensics," Forensic Science International: Genetics, vol. 38, no. October 2018, pp. 175–180, 2019. [Online]. Available: https://doi.org/10.1016/j.fsigen.2018.10.017

[50] K. V. Voelkerding, S. A. Dames, and J. D. Durtschi, "Next-generation sequencing:from basic research to diagnostics," Clinical Chemistry, vol. 55, no. 4, pp. 641–658, 2009.

[51] J. D. Watson, "The Human Genome Project : Past , Present , Future," Science, vol. 248, no. 1978, 1987.

[52] R. A. Gibbs, "The Human Genome Project changed everything," Nature Reviews Genetics, vol. 21, no. 10, pp. 575–576, 2020. [Online]. Available: http://dx.doi.org/10.1038/s41576-020-0275-3

[53] M. A. Quail, I. Kozarewa, F. Smith, A. Scally, P. J. Stephens, R. Durbin et al., "A large genome center's improvements to the Illumina sequencing system," Nature Methods, vol. 5, no. 12, pp. 1005–1010, 2008.

[54] B. Segerman, "The Most Frequently Used Sequencing Technologies and Assembly Methods in Different Time Segments of the Bacterial Surveillance and RefSeq Genome Databases," Frontiers in Cellular and Infection Microbiology, vol. 10, no. October, pp. 1–7, 2020.

[55] E. J. Fox and K. S. Reid-Bayliss, "Accuracy of Next Generation Sequencing Platforms," Journal of Next Generation Sequencing & Applications, vol. 01, no. 01, 2014.

[56] T. Hu, N. Chitnis, D. Monos, and A. Dinh, "Next-generation sequencing technologies: An overview," Human Immunology, vol. 82, no. 11, pp. 801–811, 2021. [Online]. Available: https://doi.org/10.1016/j.humimm.2021.02.012

[57] A. Rhoads and K. F. Au, "PacBio Sequencing and Its Applications," Genomics, Proteomics and Bioinformatics, vol. 13, no. 5, pp. 278–289, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.gpb.2015.08.002

[58] Y. Ono, K. Asai, and M. Hamada, "PBSIM: PacBio reads simulator - Toward accurate genome assembly," Bioinformatics, vol. 29, no. 1, pp. 119–121, 2013.

[59] M. Jain, H. E. Olsen, B. Paten, and M. Akeson, "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community," Genome Biology, vol. 17, no. 1, pp. 1–11, 2016. [Online]. Available: http://dx.doi.org/10.1186/s13059-016-1103-0

[60] Z. Rabea, S. El-Metwally, S. Elmougy, and M. Zakaria, "A fast algorithm for constructing suffix arrays for DNA alphabets," Journal of King Saud University - Computer and Information Sciences, vol. 34, no. 7, pp. 4659–4668, 2022. [Online]. Available: https://doi.org/10.1016/j.jksuci.2022.04.015

[61] W. K. Hon, T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays," Algorithmica (New York), vol. 48, no. 1, pp. 23–36, 2007.

[62] J. Labeit, J. Shun, and G. E. Blelloch, "Parallel lightweight wavelet tree, suffix array and FM-index construction," Journal of Discrete Algorithms, vol. 43, pp. 2–17, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jda.2017.04.001

[63] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "An FPGA accelerator of the wavefront algorithm for genomics pairwise alignment," Proceedings - 2021 31st International Conference on Field-Programmable Logic and Applications, FPL 2021, pp. 151–159, 2021.

[64] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," BMC Bioinformatics, vol. 9, no. SUPPL. 2, pp. 1–9, 2008.

[65] J. Bu, X. Chi, and Z. Jin, "HSA: A Heuristic Splice Alignment Tool," BMC Systems Biology, vol. 7, no. Suppl 2, pp. 1–6, 2013.

[66] I. Corp, "perspective Moore's Law: past, present and future," IEEE Spectrum, 1997.

[67] P. Machanik, "Approaches to addressing the memory wall," Technical report, School of IT and Electrical Engineering, University of Queensland, no. May, pp. 1–22, 2002. [Online]. Available: https://pdfs.semanticscholar.org/75fd/4ecf8bef062fa1bfdbf87ab48e665c1d323c.pdf%0Ahttp://www.itee.uq.edu.au/~philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf

[68] S. Angizi, N. A. Fahmi, W. Zhang, and D. Fan, "PIM-Assembler: A processing-in-memory platform for genome assembly," Proceedings - Design Automation Conference, vol. 2020-July, 2020.

[69] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal et al., "Memristor based computation-in-memory architecture for data-intensive applications," Proceedings -Design, Automation and Test in Europe, DATE, vol. 2015-April, pp. 1718–1725, 2015.

[70] J. Von Neumann and M. D. Godfrey, "First Draft of a Report on the EDVAC," IEEE Annals of the History of Computing, vol. 15, no. 4, pp. 27–75, 1993.

[71] M. M. Shulaker, T. F. Wu, M. M. Sabry, H. Wei, H. S. Wong, and S. Mitra, "Monolithic 3D integration: A path from concept to reality," Proceedings -Design, Automation and Test in Europe, DATE, vol. 2015-April, pp. 1197–1202, 2015.

[72] D. Lavenier, C. Deltel, D. Furodet, J.-f. Roy, D. Lavenier, C. Deltel et al., "BLAST on UPMEM," HAL open science, 2016.

[73] M. B. G. Sulaiman, J. Y. Lin, J. B. Li, C. M. Shih, K. C. Juang, and C. C. Lu, "SRAM-Based CIM Architecture Design for Event Detection †," Sensors, vol. 22, no. 20, 2022.

[74] J. Lappas, C. Weis, M. H. Sadi, S. Member, M. Jung, A. Guntoro et al., "Signal DRAM-PIM Architecture for Deep Neural Network Inference," IEEE circuits and systems, vol. 12, no. 2, pp. 367–380, 2022.

[75] H. Choi, Y. Lee, J. J. Kim, and S. Yoo, "A Novel In-DRAM Accelerator Architecture for Binary Neural Network," IEEE Symposium on Low-Power and High-Speed Chips and Systems, COOL CHIPS 2020 - Proceedings, vol. 2, pp. 17–19, 2020.

[76] M. Horowitz, "Scaling, power, and the future of CMOS technology," Device Research Conference - Conference Digest, DRC, no. 650, pp. 7–8, 2008.

[77] Y. Chen, "Reshaping Future Computing Systems with Emerging Nonvolatile Memory Technologies," IEEE Micro, vol. 39, no. 1, pp. 54–57, 2019.

[78] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 10, pp. 2054–2066, 2014.

[79] S. Kvatinsky, S. Member, D. Belousov, S. Liman, G. Satat, S. Member et al., "MAGIC — Memristor-Aided Logic," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 11, pp. 895–899, 2014.

[80] G. Snider, "Computing with hysteretic resistor crossbars," Applied Physics A: Materials Science and Processing, vol. 80, no. 6, pp. 1165–1172, 2005.

[81] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable stateful logic operations via material implication," Nature, vol. 464, no. 7290, pp. 873–876, 2010.

[82] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," Proceedings - Design Automation Conference, vol. 05-09-June, 2016.

[83] C. C. Lin, J. Y. Hung, W. Z. Lin, C. P. Lo, Y. N. Chiang, H. J. Tsai et al., "A 256b-wordlength ReRAM-based TCAM with 1ns search-time and 14× improvement in wordlength-energyefficiency-density product using 2.5T1R cell," Digest of Technical Papers - IEEE International Solid-State Circuits Conference, vol. 59, pp. 136–137, 2016.

[84] J. Li, R. K. Montoye, M. Ishii, K. G. Stawiasz, T. Nishida, K. Maloney et al., "1Mb 0.41 μm2 2T-2R cell nonvolatile TCAM with two-bit encoding and clocked self-referenced sensing," 2013 Symposium on VLSI Circuits, pp. C104–C105, 2013.

[85] L. Y. Huang, M. F. Chang, C. H. Chuang, C. C. Kuo, C. F. Chen, G. H. Yang et al., "ReRAM-based 4T2R nonvolatile TCAM with 7x NVM-stress reduction, and 4x improvement in speed-wordlength-capacity for normally-off instant-on filter-based search engines used in big-data processing," IEEE Symposium on VLSI Circuits, Digest of Technical Papers, pp. 9–10, 2014.

[86] S. Matsunaga, S. Miura, H. Honjou, K. Kinoshita, S. Ikeda, T. Endoh et al., "A 3.14 um 2 4T-2MTJ-cell fully parallel TCAM based on nonvolatile logic-in-memory architecture," IEEE Symposium on VLSI Circuits, Digest of Technical Papers, pp. 44–45, 2012.

[87] National Center for Genome Analysis, "Human Genome Assembly GRCh38," 2013. [Online]. Available: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.26/

[88] European Nucleotide Archive, "ERR240727_1," 2017. [Online]. Available: https://www.ebi.ac.uk/ena/browser/view/ERR240727

[89] ——, "SRR826471_1," 2018. [Online]. Available: https://www.ebi.ac.uk/ena/browser/view/SRR826471?show=reads

[90] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari et al., "Personalized copy number and segmental duplication maps using next-generation sequencing," Nature Genetics, vol. 41, no. 10, pp. 1061–1067, 2009. [Online]. Available: https://doi.org/10.1038/ng.437

[91] P. Danecek, J. K. Bonfield, J. Liddle, J. Marshall, V. Ohan, M. O. Pollard et al., "Twelve years of SAMtools and BCFtools," GigaScience, vol. 10, no. 2, pp. 1–4, 2021.

[92] M. Šošić and M. Šikić, "Edlib: A C/C ++ library for fast, exact sequence alignment using edit distance," Bioinformatics, vol. 33, no. 9, pp. 1394–1395, 2017.

[93] Nvidia Corporation & Affiliates, "nvprof Profilers' User Guide," 2023. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof

[94] Intel Corporation, "Intel VTune Profiler User Guide," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/overview.html

[95] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky, "Comments on 'Experimental Demonstration of Memristor-Aided Logic (MAGIC) Using Valence Change Memory (VCM)'," IEEE Transactions on Electron Devices, vol. 68, no. 11, p. 5925, 2021.

[96] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," Proceedings - International Symposium on Computer Architecture, pp. 368–379, 2012.

[97] I. Silicon Integration Initiative, "15Nm Open-Cell Library and 45Nm Freepdk," 2022. [Online]. Available: https://si2.org/open-cell-library/

[98] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui et al., "19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology," in 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014, pp. 338–339.

[99] M.-F. Chang, J.-J. Wu, T.-F. Chien, Y.-C. Liu, T.-C. Yang, W.-C. Shen et al., "19.4 embedded 1Mb ReRAM in 28nm CMOS with 0.27-to-1V read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme," in 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014, pp. 332–333.

[100] C.-F. Lee, H.-J. Lin, C.-W. Lien, Y.-D. Chih, and J. Chang, "A 1.4Mb 40-nm embedded ReRAM macro with 0.07um2 bit cell, 2.7mA/100MHz low-power read and hybrid write verify for high endurance applica-

tion," in 2017 IEEE Asian Solid-State Circuits Conference (A-SSCC), 2017, pp. 9–12.

[101] Y. Zhang, D. Feng, J. Liu, W. Tong, B. Wu, and C. Fang, "A Novel ReRAM-Based Main Memory Structure for Optimizing Access Latency and Reliability," in Proceedings of the 54th Annual Design Automation Conference 2017, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3061639.3062191

[102] M. Zahedi, R. Van Duijnen, S. Wong, and S. Hamdioui, "Tile Architecture and Hardware Implementation for Computation-in-Memory," Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI, vol. 2021-July, pp. 108–113, 2021.

[103] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yağlıkçı, L. Orosa et al., "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 666–679.

[104] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving DRAM latency with dynamic asymmetric subarray," in 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015, pp. 255–266.

[105] G. G. Shahidi, "Chip Power Scaling in Recent CMOS Technology Nodes," IEEE Access, vol. 7, pp. 851–856, 2019.

[106] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with Design Constraints: Predicting the Future of Big Chips," IEEE Micro, vol. 31, no. 4, pp. 16–29, 2011.

[107] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of Si MOSFETs and their application dependencies," Proceedings of the IEEE, vol. 89, no. 3, pp. 259–288, 2001.

[108] A. Hardtdegen, C. La Torre, F. Cuppers, S. Menzel, R. Waser, and S. Hoffmann-Eifert, "Improved switching stability and the effect of an internal series resistor in HfO2/TiOx Bilayer ReRAM Cells," IEEE Transactions on Electron Devices, vol. 65, no. 8, pp. 3229–3236, 2018.

[109] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, "Compressed Sensing with Approximate Message Passing Using In-Memory Computing," IEEE Transactions on Electron Devices, vol. 65, no. 10, pp. 4304–4312, 2018.

[110] European Nucleotide Archive, "ERR240726_1," 2017. [Online]. Available: https://www.ebi.ac.uk/ena/browser/view/ERR240726

[111] ——, "SRR826471_2," 2018. [Online]. Available: https://www.ebi.ac.uk/ena/browser/view/SRR826471

[112] TechPowerUp, "NVIDIA Tesla K80," 2014. [Online]. Available: https://www.techpowerup.com/gpu-specs/tesla-k80.c2616

Appendix A

Long-read Benchmarking Results

A.1. Long-read P-rate

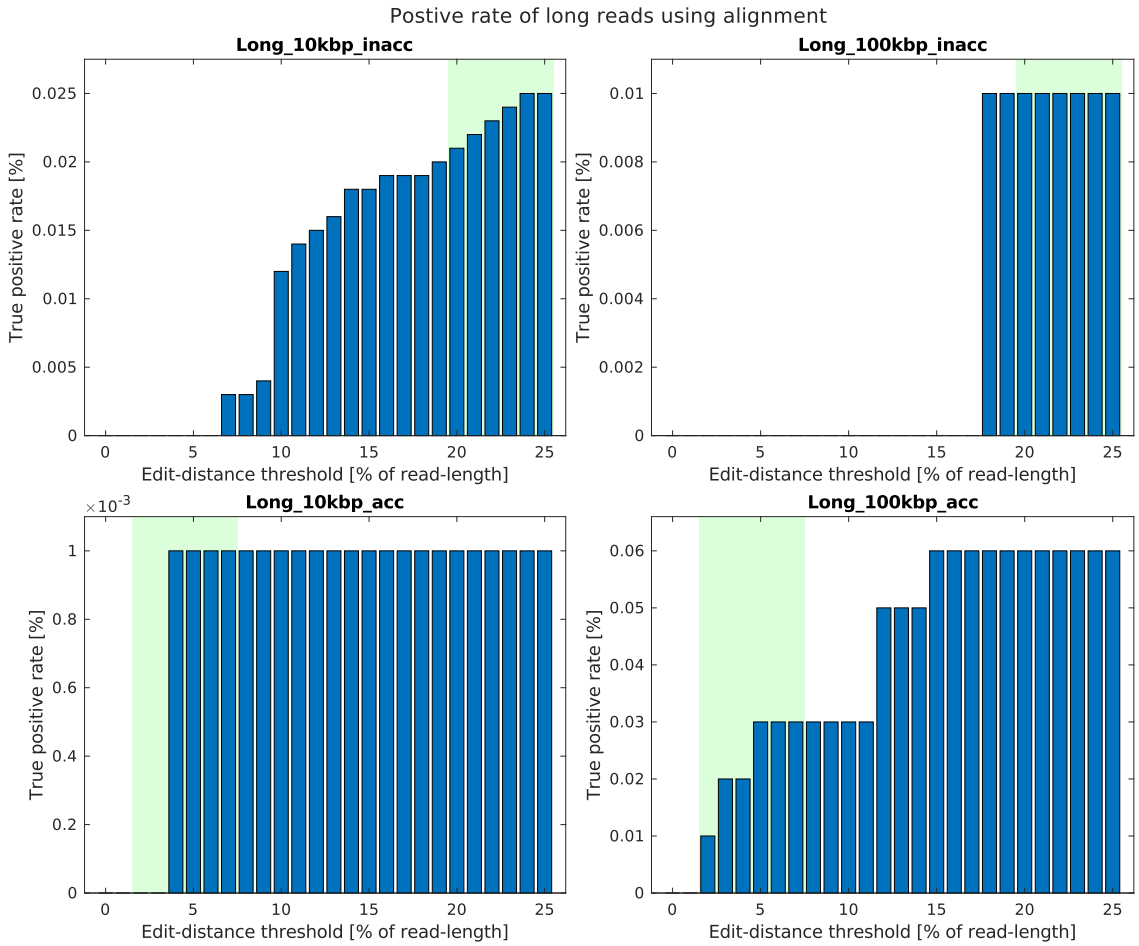


Figure A.1: True positive rate of the long-read datasets, as determined by alignment with Edlib.

A.2. Long-read FP-rate

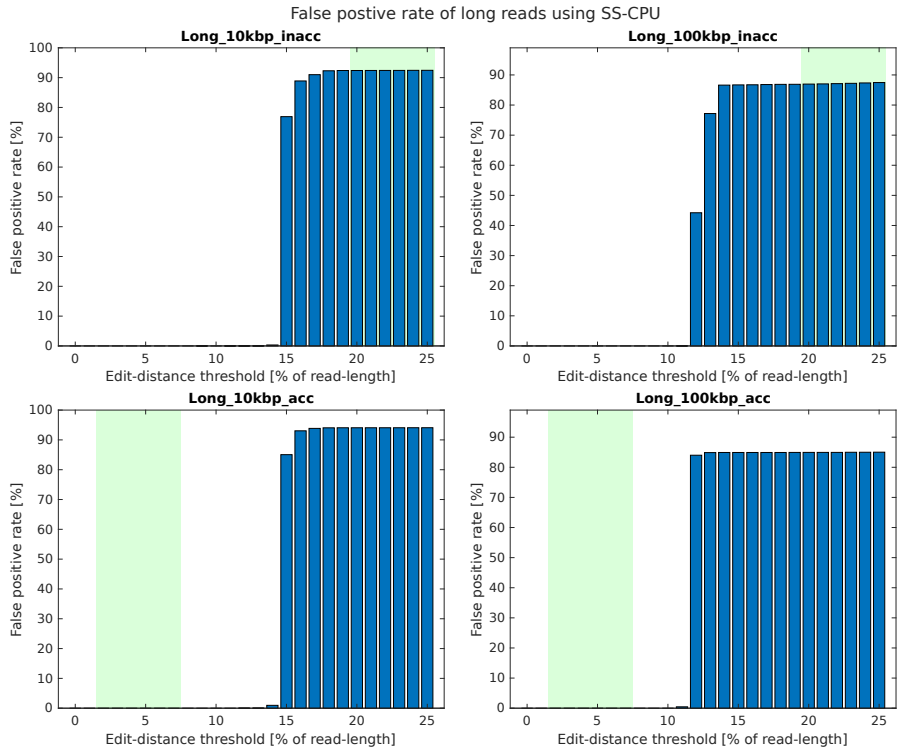


Figure A.2: False-positive rate of the only currently existing long-read capable pre-alignment filter: SS-CPU.

A.3. Long-read Filtering Time

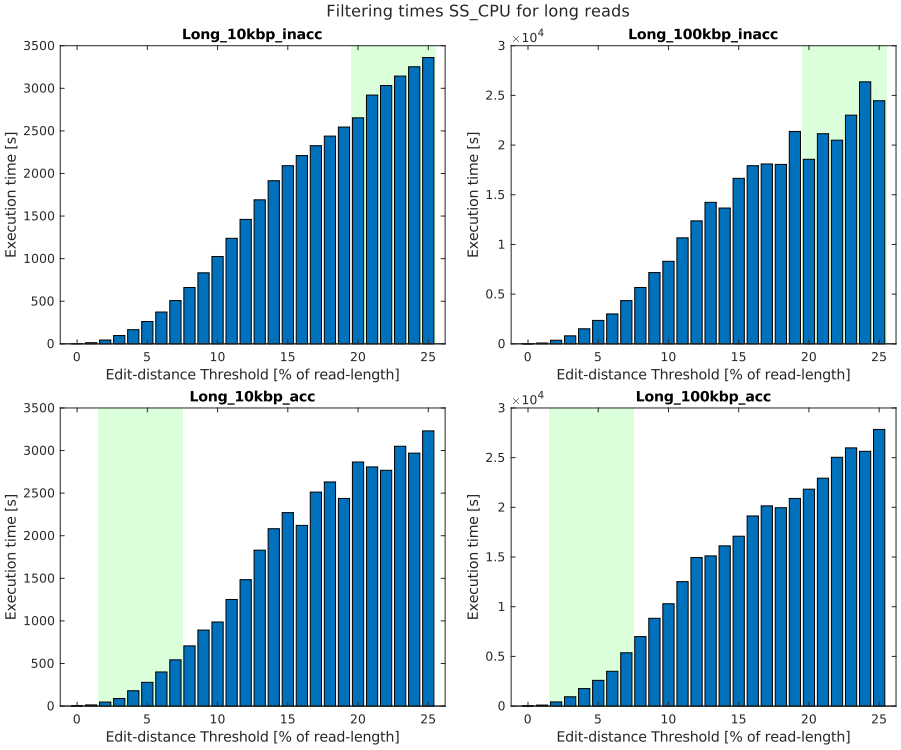


Figure A.3: Filtering times of the only currently existing long-read capable pre-alignment filter: SS-CPU.

Appendix B

Additional Results

B.1. SS-CIM FP-rate Zoomed In

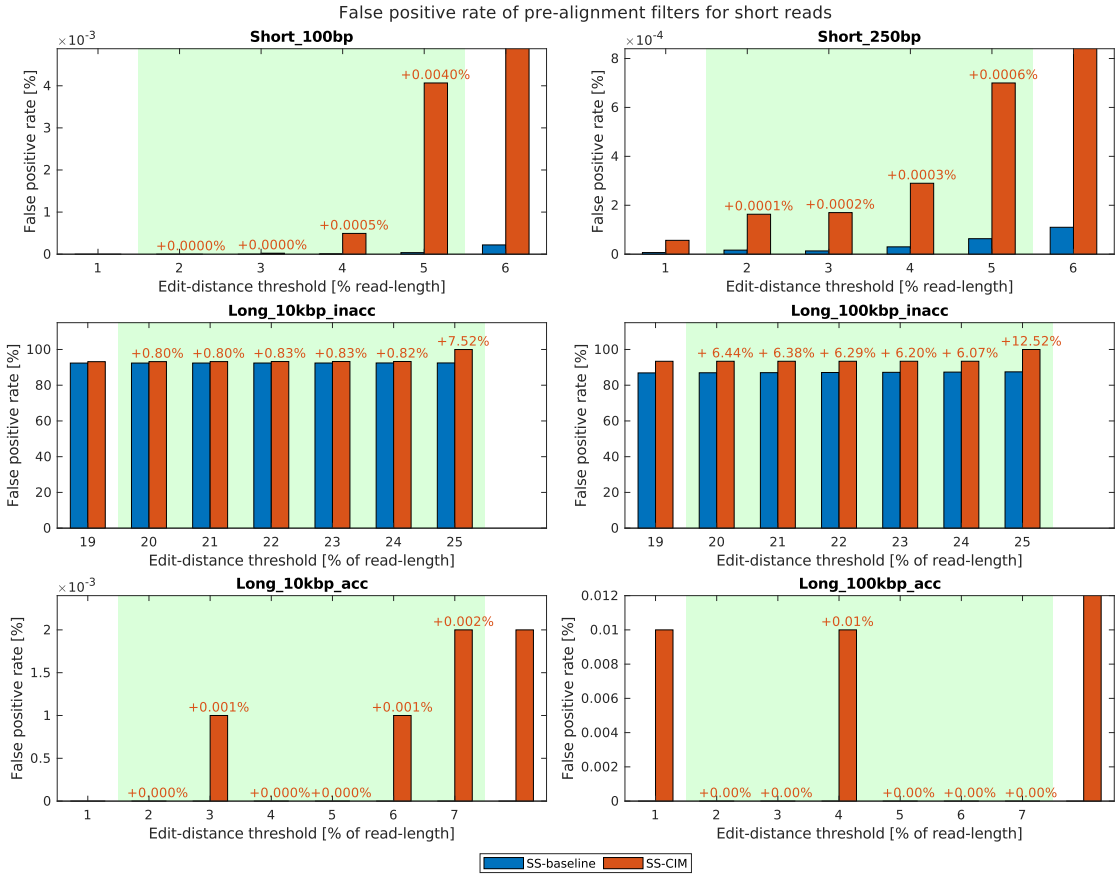


Figure B.1: Zoomed in version of Figure 7.8, focusing on the region of interest to better illustrate the difference between the two algorithms.

B.2. SS-CIM P-rate Full Range

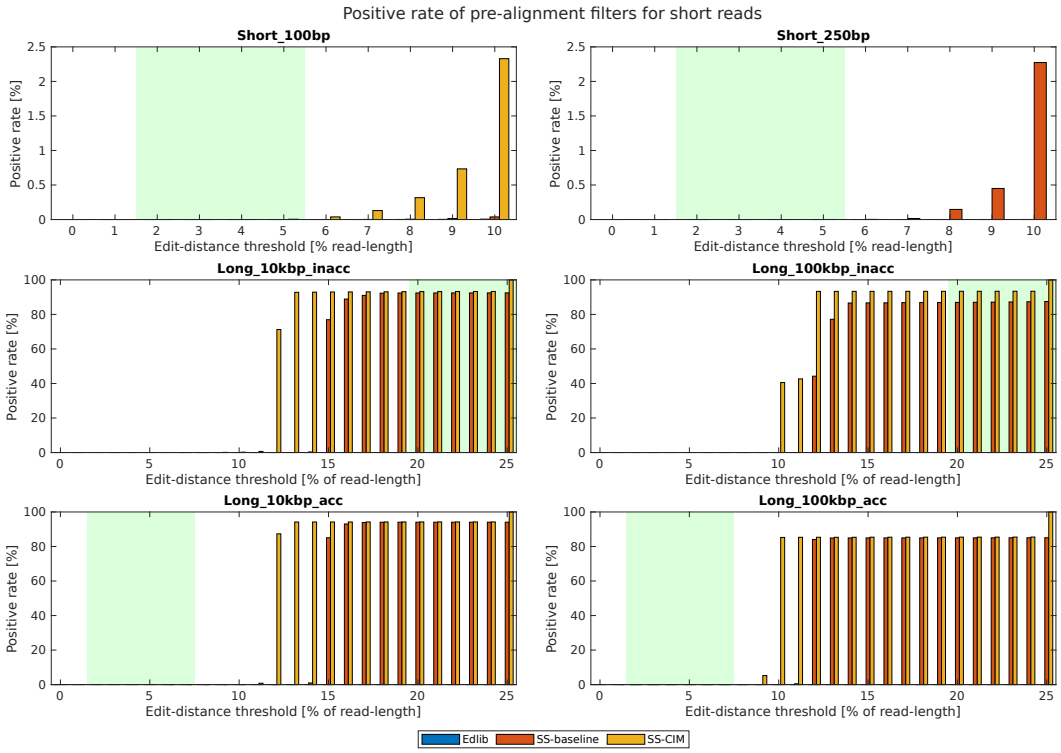


Figure B.2: Full range version of Figure 7.9, focusing on the region of interest to better illustrate the difference between the two algorithms.

B.3. SS-CIM End-to-End Execution Time Zoomed In

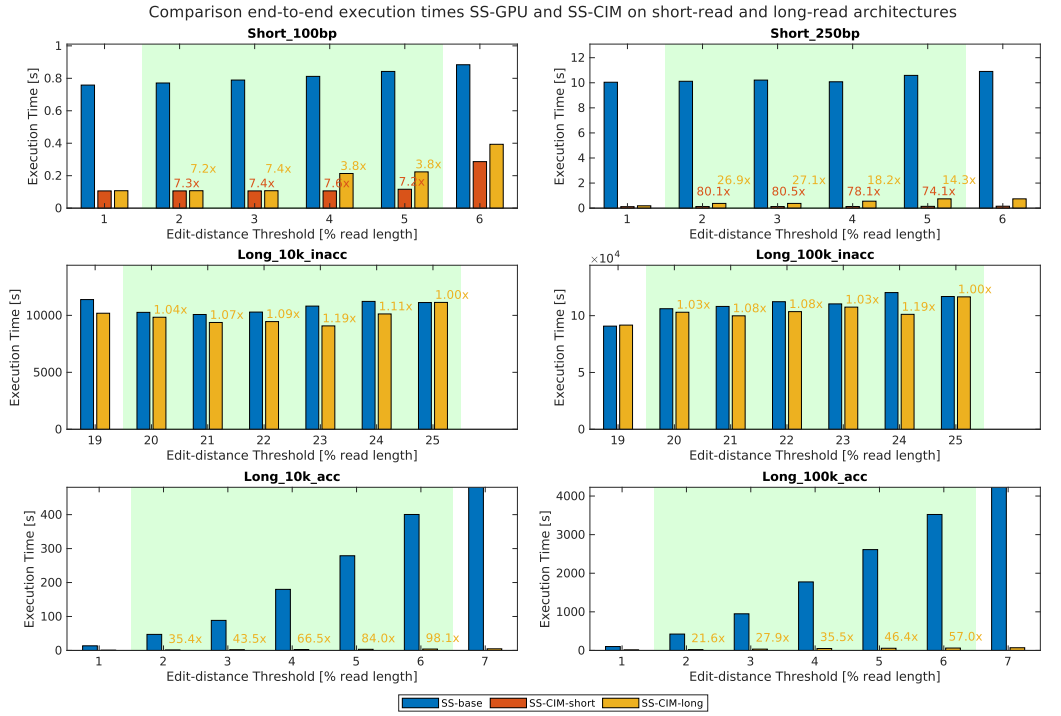


Figure B.3: Zoomed out version of Figure 7.10, focusing on the region of interest to better illustrate the difference between the two algorithms.

B.4. SS-CIM Cascaded End-to-End Execution Time Zoomed In

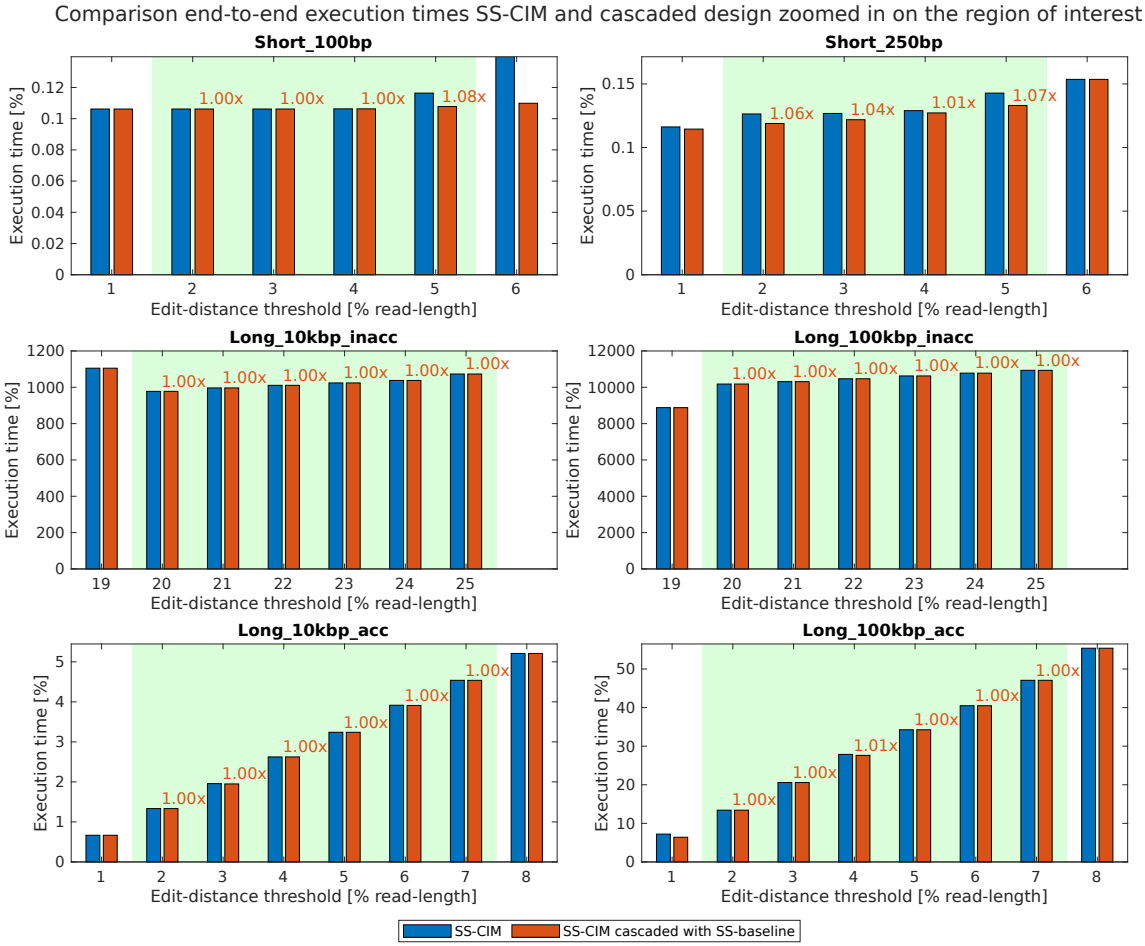


Figure B.4: Zoomed out version of Figure 7.12, focusing on the region of interest to better illustrate the difference between the two algorithms.

B.5. SHD-CIM FP-rate

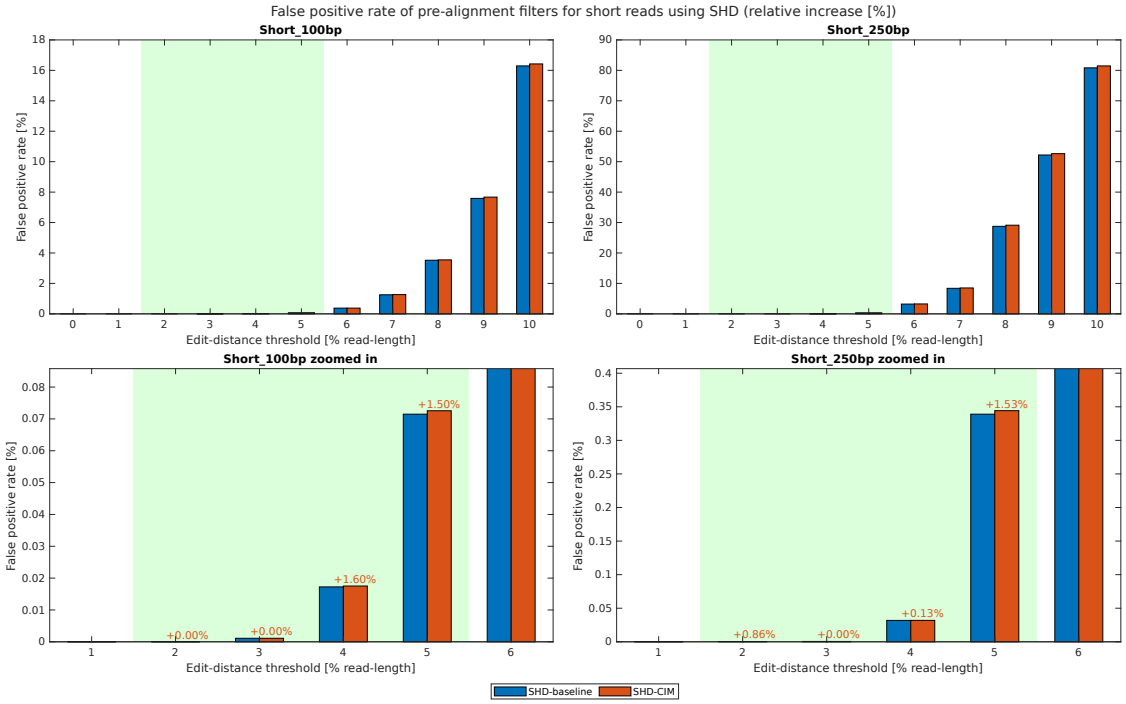


Figure B.5: False-positive rate of SHD-CIM compared to baseline SHD (top), with a version that is zoomed in on the region of interest (bottom). The numbers indicate the relative increase in false-positive rate of SHD-CIM over the baseline implementation.

B.6. SHD-CIM P-rate

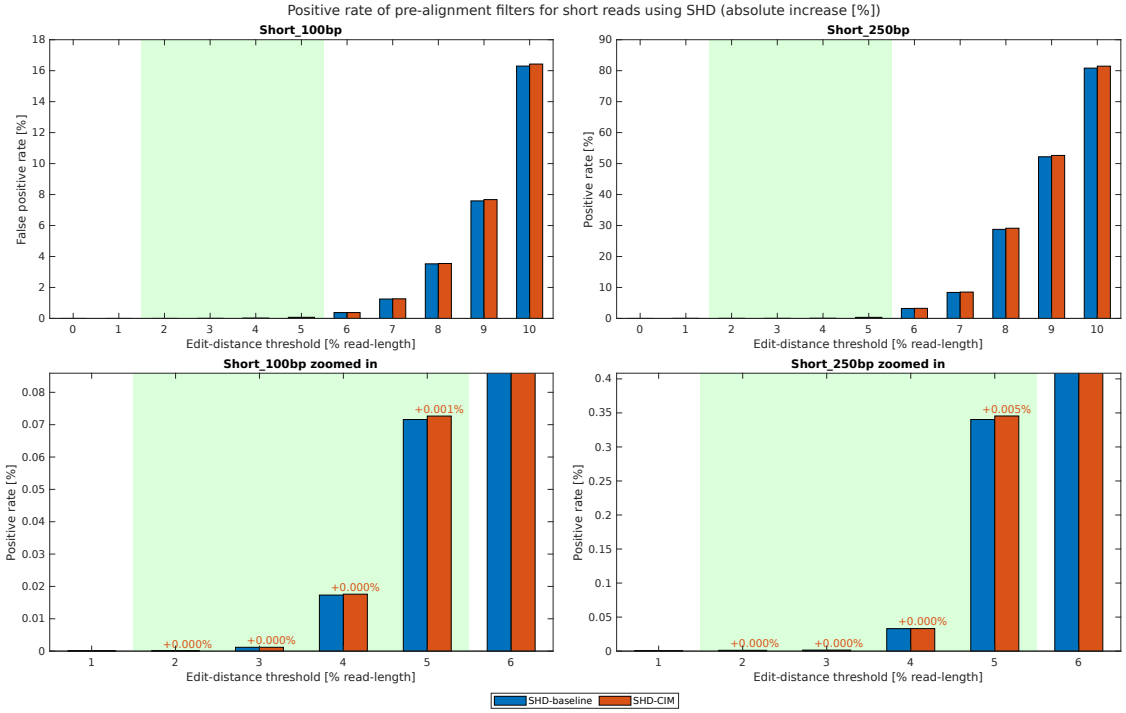


Figure B.6: Total positive rate of SHD-CIM compared to baseline SHD (top), with a version that is zoomed in on the region of interest (bottom). The numbers indicate the absolute increase in total positive rate of SHD-CIM over the baseline implementation.

B.7. SHD-CIM Filtering Time

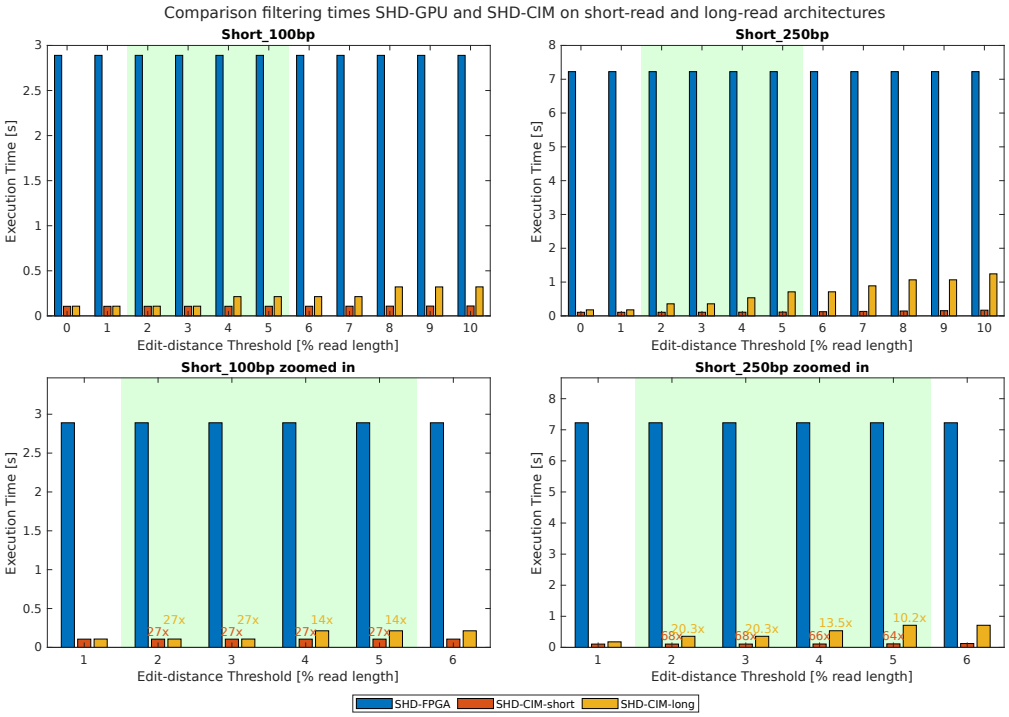


Figure B.7: Filtering time of SHD-CIM compared to baseline SHD-FPGA (top), with a version that is zoomed in on the region of interest (bottom). The numbers indicate the increase in performance of SHD-CIM over the baseline implementation.

B.8. SHD-CIM End-to-end Execution Time

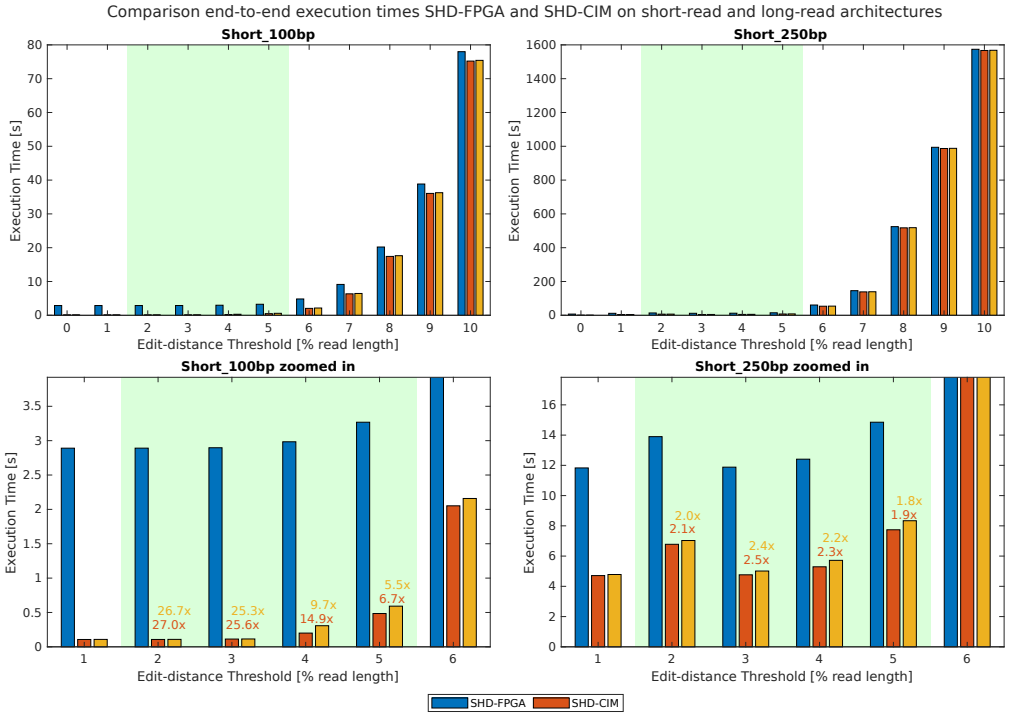


Figure B.8: End-to-end execution time of SHD-CIM compared to baseline SHD-FPGA (top), with a version that is zoomed in on the region of interest (bottom). The numbers indicate the increase in performance of SHD-CIM over the baseline implementation.