Non-volatile In-memory Computing for Large Scale Data-Intensive Workloads: Challenges and Opportunities

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zamshed Iqbal Chowdhury

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Professor Ulya R. Karpuzcu

December, 2021

Acknowledgements

As I am writing my acknowledgments, I am overwhelmed by the flurry of emotions and memories from these past years of my PhD study. I would like to begin by expressing my utmost gratitude toward my advisor, Professor Ulya R. Karpuzcu. I feel really fortunate to have decided to pursue my PhD under her supervision. As an international student, coming to a foreign land, away from the loved ones, can be a daunting experience with the constant pressure in a graduate student's life. Professor Karpuzcu always helped to manage my workload so that I don't get succumbed to that pressure; I cannot be more thankful for that. During my training with Professor Karpuzcu, she always encouraged me to share my ideas (however unrealistic those were), listened to me patiently, and guided me to validate and pursue those ideas. Her encouragement helped me a lot in thinking out of the box and grow to become a better researcher. The honest reviews and feedback, from Professor Karpuzcu, on my research has been a cornerstone of my PhD training. Professor Karpuzcu and her husband, Brian Greskamp, have always been kind enough to take an interest in my progress toward my future career, and provided valuable advice whenever I needed. Not only that Prof. Karpuzcu has been a kind and patient mentor to me, she was always concerned about my well-being all these years too. I am really thankful to them for this wonderful experience.

I would like to thank the rest of my thesis committee: Professor Sachin Sapatnekar, Professor Jian-Ping Wang, and Professor Abhishek Chandra. I have had great feedback on my research from Professor Sachin Sapatnekar and Professor Jian-Ping Wang all these years, which helped me to shape my research for better. I would also like to thank Professor Meisam Razaviyayn (University of Southern California) for his help with the problem formulation in the genomics accelerator projects.

I am glad to have worked with so many brilliant fellow graduate students during my

PhD. Salonik Resch, Masoud Zabihi, Zhengyang Zhao, Thomas Peterson, and I spent a lot of time together discussing our research, asking questions, and presenting domain-specific knowledge to broader audiences. S. Karen Khatamifard was my first lab mate and although we did not get to spend much time with each other, we eventually worked closely on a few successful projects. Husrev Cilasun quickly became an integral part of our lab, and we spent long hours talking about our research and beyond.

It would have been difficult to survive these years without friends who provided me with the much needed release. I would like to thank Ibrahim Ahmed, Farhana Sharmin Snigdha, AHM Mahfuzur Rahman, Arshia Zernab Hassan, Muntaser Mehmud Khan, and Farhana Sharmin Moon for the countless weekends of fun– talking and watching movies.

Finally, I would like to thank my family for being there for me all these years. My parents made sure that I get the best possible education while I was growing up. My brother helped keeping the family together and took care of my parents while I was away from home. My sister has always cared for me during our toughest times. My in-laws always kept in touch with us the whole duration of my PhD. I am especially thankful to my wife, Tanjila Nawshin, without whom I could not have possibly survived this journey. She sacrificed a lot in her career for my PhD and I am really grateful that she is in my life.

Dedication

This thesis is dedicated to both my parents. My father, A T M Zulfiqur Haidar Chowdhury, has always supported my endeavours in academia and inspired me to pursue my PhD. Begum Shahin Ara, my loving and wonderful mother, has been the constant source of warmth and care, and my life-long supporter. They made countless sacrifices for me and my siblings to provide a good life and education for us. It would have been impossible to make it this far without their support.

Abstract

The application(domain)s that depend on the large amount of data for solving problems, e.g., genome sequence analysis, graph analytics, machine learning etc., suffer from growing overhead of data communication between physically separate logic (i.e., compute) and memory elements in conventional von Neumann computing. The recent progress in processing(/computing)-in-memory (PIM/CIM) or simply, in-memory computing addresses data communication overhead in these applications by fusing compute capability with memory where the data reside– thereby achieving reduced energy consumption, and higher application throughput due to access to the higher internal bandwidth of the memory substrate as compared to the off-chip bandwidth.

In this thesis, we focus on the architecture- and application-level characterizations of PIM architecture, Computational RAM (CRAM) in particular, for large scale data-intensive workloads–in terms of opportunities and challenges. We demonstrate the efficacy of CRAM in reducing the communication bottleneck of *genomic sequence analysis*, as a representative application domain due to its importance and inherent characteristics that are suitable for PIM-based implementation, by designing various CRAM-based Hardware (HW) accelerators. The designs cover all architectural aspects such as data layout, spatio-temporal scheduling of compute, system integration etc.

First, we introduce an in-memory accelerator architecture, BWA-CRAM, for DNA sequence alignment by direct mapping of state-of-the-art Burrows–Wheeler Aligner algorithm on CRAM. This architecture outperforms corresponding software implementation in terms of throughput and energy efficiency, even under conservative assumptions.

Next, we improve the performance of DNA sequence (pre-)alignment (and other similar, generic pattern matching applications) through HW/SW co-design and introduce SpinPM, a novel high-density, reconfigurable spintronic in-memory pattern matching substrate based on CRAM with Spin-Orbit-Torque (SOT)– specifically Spin-Hall-Effect (SHE) MTJ devices; and demonstrate the performance benefit SpinPM can achieve over conventional and near-memory processing systems.

Subsequently, we present CRAM-Seq, an accelerator for RNA-Seq abundance quantification based on CRAM. Through HW/SW co-design, we demonstrate that CRAM-Seq outperforms a commonly used state-of-the-art software abundance quantification algorithm, Kallisto, in terms of throughput and energy efficiency.

We introduce Content Addressable Memory or CAM, which is very efficient in large scale pattern matching, functionality in CRAM, next. We present CAMeleon- a novel compute substrate that leverages the high energy efficiency benefit of CRAM, and is capable of satisfying very stringent hardware resource (area) budget in embedded/edge computing applications, e.g., handheld sequencing device. CAMeleon performs CAM operations more energy-efficiently while consuming less/similar area, and supports logic and memory functions beyond CAM operations on demand through reconfiguration, as compared to conventional CAM-only designs based on SRAM and emerging memory technologies (such as STT-MTJ, ReRAM and PCM).

Finally, we study the impact on applications' reliability due to mapping on a PIM substrate, focusing on PIM architectures that perform logic operations directly within memory arrays, *in-situ*, obviating any need for data transfers (even to and from the array periphery), e.g., CRAM. Here we (i) quantitatively characterize *gate–flip errors*, an acute class of functional errors specific to such PIM systems, where, due to parametric variations, a logic gate can behave as another; and (ii) analyze to what extent algorithmic noise tolerance can mask gate-flips.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are living in an era that is experiencing an unprecedented proliferation of data generated from most (if not all) facets of our lives and consumed by different analytical tasks across different application domains. As [4] shows, we have already entered the era of EB (ExaBytes = 10^{18} Bytes) data afterward of 2011. For instance:

- In 2013 alone, 153 EB of data were produced from health care industry [5].
- According to [6], the Large Hadron Collider (LHC) experiments generated PB (PetaBytes = 10^{15} Bytes) of data per second during the discovery of the elusive Higgs boson particle back in 2012.
- Modern social media platforms also contribute by generating ∼PB of data each day [7].

The data volume available for different applications are expected to grow in the coming years. [8] compared the growth of genome sequence (i.e., DNA) data with Moore's law [9], and showed that by 2010, sequence data volume was already winning; by 2025, genome sequence data is expected to outweigh Moore's law by at least 3 orders of magnitude [10].

Many application(domain)s, including a significantly high number of emerging applications, thrive on such abundance in data volume to provide high quality analytical output. For example, image classification, with real-time constraints, on High-Definition video streams requires millions of example images and videos to train a classification

model [11]. RNA-Seq abundance quantification, an important tool for various genomic studies (e.g., analysis of functionally similar genes in a biological sample) including medicine research, significantly benefits from high volume of RNA-Seq data generated by next generation sequencing (NGS) platforms.

The conventional computing model, i.e., von Neumann model, is built on separate processing elements, e.g., logic and memory units, as shown in Fig. 1.1a. The communication link that connects the processing and memory units is limited in bandwidth, and sets up a maximum performance point beyond which the compute system becomes memory-bound for an application. Such model, although simple, served quite well in terms of performance for a long time, until the surge in data volume occurred and a large portion of the workloads for such computing systems became data-intensive.

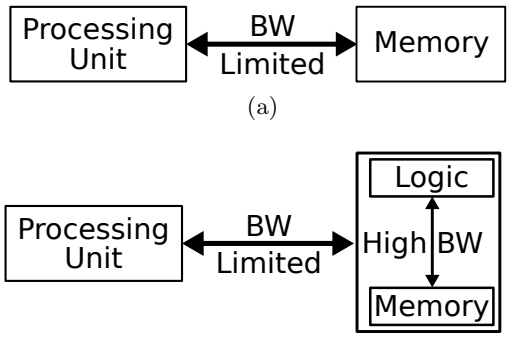


Figure 1.1: (a) von Neumann and (b) Processing-in-Memory (PIM) models of computing.

Workloads, that utilize a massive amount of data, require the data to be communicated between the processing and memory units (i.e., reads/writes from/to memory) during computation. Such requirement puts pressure on the limited bandwidth between logic and memory units, making the workloads suffer from waiting for the data transaction requests to complete– thereby, reducing performance of the applications. This *memory wall* problem gets worse with technology scaling: compared to the energy consumption of a full-precision fused-multiply-add (FMA), an off-chip (64-bit) SRAM load consumes >50X more energy at 40 nm [12].

Processing-in-Memory (PIM), also known as Computing-in-Memory (CIM) or simply, in-memory computing is an elegant solution to this data communication bottleneck– illustrated in Fig. 1.1b. PIM fuses compute capability with the memory unit, thereby

offering a feature of *in-situ* processing of the data stored in the memory itself. Such approach has the benefits of (i) reduction of data communication overhead to/from processing unit(s) due to *in-place* processing of data, (ii) high internal bandwidth available for use during *in-situ* data processing, and (iii) high parallelism of the operations within memory units. The advantages of such unconventional, i.e., non-von Neumann, style of computing are: reduced energy consumption for applications, particularly for those with high data usage, and higher application throughput vs. conventional computing.

The idea of PIM has been here from the later part of last century. Due to continuous benefit from aggressive technology scaling, and lack of compatibility between proposed PIM architectures and then existing semiconductor based design environments led to such unconventional model of computing to be mostly ignored, until recently. The wide-spread availability of big data, increasing pressure exerted on the conventional computing system by the corresponding data communication overhead and progress made in the technology compatibility fronts resulted in renewed interest in PIM-based computing solutions. This not only has opened up a new frontier for general purpose computing, but also for application(domain)-specific computer, e.g., accelerator, designs as well.

Reduction in amount of required data movement during execution of data-intensive workloads through PIM-based implementation presents unique challenges for the people at the different levels of the systems stack, e.g., architects, application developers etc. One of the key challenges in adopting PIM at wide-scale is to identify the opportunities in an application to be mapped to a PIM substrate–in order to get the desired performance boost. It is a crucial decision to be taken whether it is the part(s) or the full application that is going to be mapped to PIM, depending on architectural specifications of PIM (type of logic operations that can be performed *in-situ*), target performance values for the application and whether the application suffer from memory bottleneck issue in conventional computers [13]. Having said that, this is a complex design-space dependent on architectural fine-tuning and HW/SW co-design approach to make a data-intensive application suited for mapping on a given PIM architecture–with the objective of meeting strict performance goals. At the same time, it is also desired that the application features are intact during HW/SW co-design in order for the application to not loose its commodity value.

In this thesis, we explore and evaluate the opportunities for mapping data-intensive workloads on PIM architectures–particularly, spintronics (e.g., STT-MTJ, SOT- or SHE-MTJ) based non-volatile PIM. We select computational RAM (CRAM) as a high-density reconfigurable spintronics-based platform providing true PIM semantics, as opposed to most CMOS-based solutions which only deliver processing-near-memory, leaving the highly demanded potential in energy-efficiency untapped [1]. We identify potential data-intensive workloads and the kernel(s) with the applications to offload to PIM, keeping in mind the performance improvement goals expected from such mapping. We focus on architecture- and application-level characterizations of PIM architectures based on CRAM, in terms of opportunities and challenges. The designs cover various architectural aspects such as data layout, spatio-temporal scheduling of compute, system integration, memory footprint optimizations etc. During application mapping, the design challenges experienced–given the limitations of a PIM architecture, are solved using custom circuits design and architectural optimizations, as required. As a representative application domain, *genome sequence analysis*, is selected which is thriving in recent times due to abundance of genome sequence data from the advent of Next Generation Sequencing (NGS) systems, and are essentially pattern matching workloads.

We begin our exploration by introducing BWA-CRAM for accelerating DNA sequence alignment at scale. DNA sequence alignment is a powerful bioinformatics tool that is used in many important applications such as identifying similarity in genome sequence, comparative modeling of protein etc. BWA-CRAM maps a popular state-of-the-art sequence alignment tool, Burrows–Wheeler Aligner (BWA), directly (without no modification to the BWA algorithm itself) to CRAM in order to reduce the costly data communication overhead during alignment. In effect, such mapping improves the throughput and energy efficiency of alignment as compared to the software implementation.

Next, we continue our exploration with the DNA sequence (pre-)alignment application and other widely used similar pattern matching applications that are key computational blocks in many large scale data analytics algorithms, with HW/SW co-design approach– in order to better utilize the performance benefits offered by CRAM. To this end, we introduce SpinPM, a PIM substrate based on CRAM and demonstrate the performance benefit SpinPM can achieve over conventional and near-memory processing

systems.

We further investigate another *genome sequence analysis* application: RNA Sequence (RNA-Seq) abundance quantification, which is an important application in different fields of genomic studies, e.g., analysis of functionally similar genes in a biological sample. Presenting CRAM-Seq, an accelerator architecture for RNA-Seq abundance quantification algorithm based on CRAM and resulting from HW/SW co-design, we analyze the impact of such accelerator on the performance of RNA-Seq abundance quantification.

As pattern matching covers a high number of emerging applications, and alternative computing models such as content addressable memory or CAM presents itself as a very efficient way of searching large-scale database with high efficiency, we next introduce CAM functionality to CRAM. To this end, we introduce CAMeleon that addresses the very stringent hardware resource (area) budget and a need for extreme energy efficiency in embedded/edge computing applications, e.g., handheld sequencing device, by enabling repurposing (i.e., reconfiguring) HW on demand where the overhead of reconfiguration itself is subject to the very same tight budgets in area and energy efficiency. CAMeleon has a similar level of latency to conventional CAM designs based on SRAM and emerging memory technologies (such as STT-MTJ, ReRAM and PCM), however, performs CAM operations more energy-efficiently (owing to energy efficient logic operations in CRAM), consumes less area, and can support traditional logic and memory functions beyond CAM operations on demand, unlike other CAM designs.

Finally, we study the errors in CRAM to understand the impact of mapping applications on such a novel substrate. Accordingly, we (i) quantitatively characterize *gate-flip errors*, an acute class of functional errors specific to PIM systems similar to CRAM, where, due to parametric variations, a logic gate can behave as another; and (ii) analyze to what extent algorithmic noise tolerance can mask gate-flips.

This thesis is organized as follows:

- Chapter 2 briefly presents a high-level comparison between available PIM architectures in literature, explains the reasoning behind selection of CRAM as the PIM architecture for this thesis, and the low-level working principle of CRAM.

- Chapter 3 discusses the details of directly mapping a popular DNA sequence

alignment application– Burrows–Wheeler Aligner (BWA) on CRAM.

- In Chapter 4, the detailed HW/SW co-design of an accelerator for pattern matching using spintronic CRAM is presented.
- Chapter 5 further explores the HW/SW co-design opportunity with an accelerator for RNA-Seq abundance quantification application.
- Chapter 6 extends the CRAM architecture to introduce reconfigurable Content Addressable Memory (CAM) functionalities in CRAM for energy-efficient operations at low area and reconfiguration overheads.
- Chapter 7 presents an error model, specific to CRAM and similar PIM architectures, to capture the impact on reliability of the applications mapped to CRAM, due to low-level device and circuits variations.
- Finally, Chapter 8 concludes this thesis and provides a high-level discussion on future research directions.

Chapter 2

PIM Basics

2.1 All the PIM Architectures!

Since the resurgence of PIM as a viable solution to the memory bottleneck for data-intensive applications, there have a plethora of architectures proposed– ranging from logic die with 3-D stacked DRAM to more unconventional true in-memory computing that performs logic with direct involvement of memory cells. Before we can select a PIM architecture for this thesis, it is mandatory to have an overall insight about the key architectures and what they bring to the table if adopted for an application. All available PIM architectures can be grouped into three categories:

- Processing-near-memory (PNM): This refers to designs that are actually variants of von Neumann computing architectures. These systems have reduced latency and higher bandwidth to/from memory due to closely placed logic and memory units. Although this is an improvement, it still suffers from high data movement overhead and limited bandwidth to access the memory.

- Processing-in-periphery (PIP): This refers to architectures which use peripheral circuits within the memory arrays, including sense amplifiers (SA), to perform computation of stored data. While such designs actually improve the data movement problem significantly, use of (CMOS) peripheral and dedicated logic hardware to accomplish logic operations leaves a lot of energy reduction on the plate, on top of data movement incurred within the array during complex computation

(i.e., sequence of logic operations on data).

- Processing-in-memory (PIM): The true PIM, or simply PIM refers to the architectures that perform computation using memory cells, and have much lower data movement overhead when performing complex computations required for an application, as compared to PNM or even PIP.

Table 2.1 lists some significantly studies recent architectures, into the categories explained above. The relative values are provided with H (=HIGH) or L (=LOW), or ±. These designs are evaluated, qualitatively, in terms of the following:

- Row(/column)-level parallelism refers to the ability of the architecture to support logic operations along multiple rows/columns simultaneously. The higher, the better.

- CMOS logic refers to dedicated logic inside the memory array or off-chip to perform or assist in the computation. It has associated area and energy overheads.

- The designs that use SA to perform computation require change in the design of SA, as compared to a regular memory array of the same technology. This entails writing back the data from the output of SA to memory, incurring data movement overhead.

- Data movement refers to the need to move the data within the array/system to perform basic logic/arithmetic operations (e.g., 1-bit adder) in an application mapped to an architecture. For example, NOR operations on a N-bit input and subsequent NAND operations on the outputs of the previous NOR operations would require the output of the NOR (first logic op.) to be written back to the array in an architecture that uses SA based computation. Most real-life applications, if not all, involve a sequence of logic operations where one logic operation would operate on the result of the previous logic operations. Therefore, "data movement overhead' actually refers to the overhead of data movement during the execution of an application mapped to a specific architecture. The lower, the better.

- #Basic op refers to the range of basic logic operations supported by an architecture (e.g., AND, OR, NAND, NOR, NOT). The higher, the better.

Table 2.1: Comparison between in-memory computing (i.e., PIM) architectures.

Tech.	Ref.	Class	Parallelism	CMOS Logic	SA Change	Data Mov.	#Basic Op.
SRAM	[15]	PIP	H	+	+	H	L
DRAM	[16]	PIP	H	+	+	H	L
	[2]	PIP	H	-	+	H	L
	[17]	PNM	–	+	-	H	H
	[18]	PNM	–	+	-	H	H
	[19]	PNM	H	+	-	H	H
ReRAM	[20]	PIP	L	-	-	H	L
	[21]	PIP	L	+	+	H	L
	[22]	PIM	L	-	-	L	H
	[23]	PIM	L	-	-	H	H
	[24]	PIM	L	-	-	L	H
FeFET	[25]	PIP	L	+	+	H	L
MRAM	[26]	PIP	H	+	+	H	L
	[27]	PIP	H	+	+	H	L
	[28]	PIP	H	+	+	H	L
	[14]	PIM	H	-	-	L	H

Considering all relative values, Computational RAM (CRAM) [14] is the obvious choice as a true PIM architecture that offers high row/column level parallelism, requires no CMOS peripheral logic or specially designed SA to perform computation, and consequently, has very low intra-array data movement during computation. It also supports a wide-range of logic gates, including MAJ(ority) gates on top of basic and universal gates–rendering it Boolean complete.

2.2 CRAM Basics

2.2.1 Fusing Computation and Memory

In its most basic form, a CRAM array is essentially a 2D arrangement of magneto-resistive RAM (MRAM) cells. Here we assume 2T(ransistor)1M(TJ) cells with Spin-Orbit Torque (SOT)- or Spin-Hall Effect (SHE)-MTJ devices, without the loss of generality[1]. When compared to the standard MRAM cell, however, the array features an additional *Logic Line (LL)* per cell – as indicated in Fig.2.1(a) and Fig.2.1(b) – to connect cells to perform logic operations. A CRAM cell can operate as a regular MRAM memory cell or serve as an input/output to a logic gate.

Each MTJ consists of two layers of ferromagnets, termed as pinned and free layers, separated by a thin insulator. The magnetic spin orientation of the pinned layer is fixed;

[1]We direct interested readers to literature that use CRAM architectures with 2- and 3-terminal spintronic devices [1, 29], for more insight on portability of CRAM-Seq to other cell technologies.

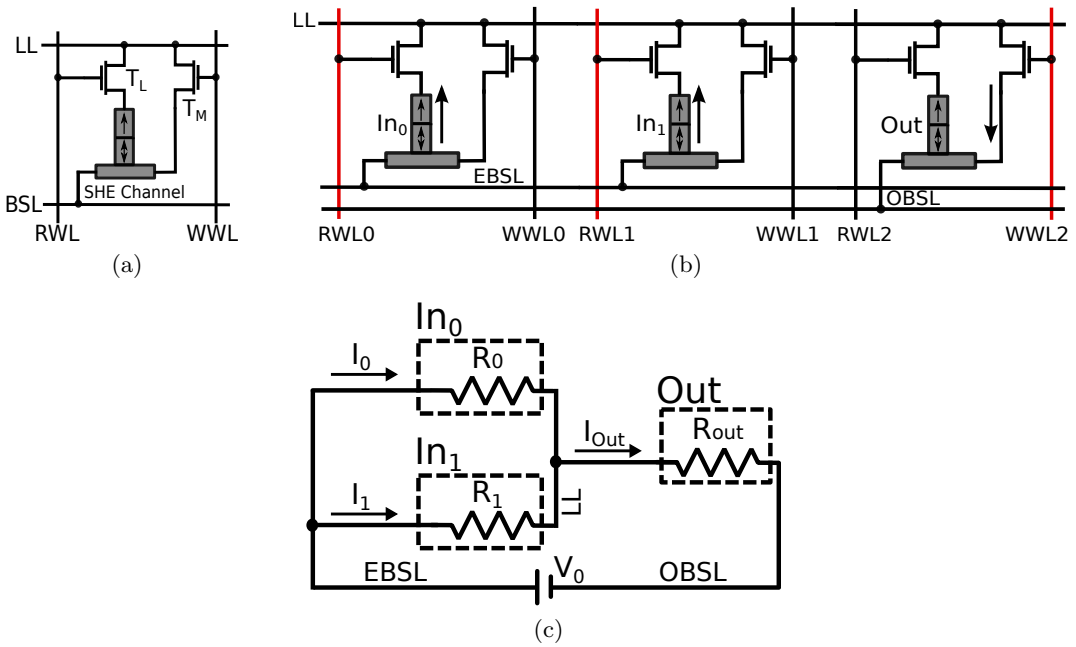


Figure 2.1: (a) CRAM cell; (b) 2-input gate formation in the array; (c) 2-input NOR gate circuit equivalent.

of the free layer, controllable. A heavy metal layer (i.e., the SHE channel) is juxtaposed with the free layer in SHE-MTJs– in order to separate read and write paths for more effective optimizations. The SHE channel has a high resistance that brings down the write current, leading to lower energy consumption. Changing the spin orientation of the free layer entails passing a (polarized) current through the SHE channel, where the current direction sets the free layer orientation. The relative orientation of the free layer with respect to the pinned layer, i.e., anti-parallel (AP) or parallel (P), gives rise to two distinct MTJ resistance levels, i.e., R_{high} and R_{low}, which encode logic 1 and 0, respectively.

Memory Configuration: When the array is configured as a memory, the *Logic Line (LL)* is active, i.e., connected to a voltage source. In the following, we detail the configuration for *Read Word Line (RWL)* and *Write Word Line (WWL)*, considering various memory operations:

- Data retention: The *RWL and WWL* are set to 0 to isolate the cells and to prevent current flow through the MTJ and the SHE channel (which we refer to together as

the SHE-MTJ).

- Read: *RWL* is set to 1, to connect each SHE-MTJ to its *LL*. *WWL* is set to 0. A small voltage pulse applied between *LL* and *Bit Select Line (BSL)* induces a current through the SHE-MTJ, which is a function of the resistance level (i.e., logic state), and which in turn a sense amplifier attached to *BSL* captures.

- Write: *WWL* is set to 1 and transistor T_M is switched ON, to connect the SHE channel to its *LL*. *RWL* is set to 0. A large enough voltage pulse (in the order of the supply voltage) is applied between *LL* and *BSL* to induce a large enough current through LL and the SHE channel to change the spin orientation of the free layer.

Logic Configuration: In logic mode, *LL* establishes the connection between inputs and outputs of logic gates. We also distinguish between two sets of *BSL*: even *BSL* (*EBSL*) and odd *BSL* (*OBSL*). These are utilized to connect all cells participating in computation, on a per column basis, as input and output cells. Such cells may act as logic gate inputs or outputs, where the only restriction is having all inputs connected on the same set of *BSL*, and the output, on the different set. For each CRAM input cell participating in computation, *RWL* is set to 1 to connect its MTJ to *LL*. On the other hand, for each CRAM output cell participating in computation, *WWL* is set to 1 to turn the switch T_M on, which in turn connects the corresponding SHE channel to the *LL*. A voltage pulse applied between *EBSL* and *OBSL* induces a current, dependent on the resistance levels of input cells, through the SHE channel of the output cell. As an example, Fig.2.1(b) demonstrates the formation of a two input logic gate in the array, where cells labeled by "0", "1", and "2" correspond to the inputs In_0, In_1, and the output *Out*, respectively. The output cell is preset to a known logic value ("0" or "1"), depending on the type of logic operation to perform, by a standard write. Fig.2.1(c) depicts the equivalent circuit: *OBSL* is grounded; while *EBSL* is set to voltage V_0. The value of V_0 determines the current through the input MTJs, I_0 and I_1, as a function of their resistance values R_0 and R_1. Each input resistance captures the resistance of the corresponding SHE-MTJ, whereas the output resistance R_{Out} is only the SHE channel resistance of the output cell. Input and output cells are connected to *LL* by setting the respective *RWL* and *WWL* to 1 (colored red in Fig. 2.1(b)). $I_{Out} = I_0 + I_1$ flows through R_{Out}. If I_{Out} is higher than the critical switching current I_{crit}, it will change

In_0	In_1	Out	$I_{Out} = I_0 + I_1$	
0 (R_{low})	0 (R_{low})	1	I_{00}	$> I_{crit}$
0 (R_{low})	1 (R_{high})	0	I_{01}	$< I_{crit}$
1 (R_{high})	0 (R_{low})	0	$I_{10} = I_{01}$	$< I_{crit}$
1 (R_{high})	1 (R_{high})	0	I_{11}	$< I_{crit}$

Table 2.2: 2-input NOR truth table (Out preset = 0).

the free layer orientation of Out's MTJ, and thereby, the logic state of Out. Otherwise, Out will keep its previous (preset) state.

We can easily expand this example to more than two inputs. The key observation is that we can change the logic state of the output as a *function* of the logic states of the inputs, within the array. And voltages applied between *BSL*s of the participating cells dictate how such *function*s look like.

Continuing with the example from Fig.2.1(b)/(c), let us try to implement an universal, 2-input NOR gate. Table 2.2 provides the truth table. Out would be 0 in this case for all input combinations but $In_0 = 0$, $In_1 = 0$, which incurs the lowest R_0 and R_1, and hence, the highest $I_{Out} = I_0 + I_1$. Let us refer to this value of I_{Out} as I_{00}, following Table 2.2. Accordingly, if we preset Out to 0 (before computation starts), and determine V_0 such that I_{00} does exceed I_{crit}, while both I_{11} and $I_{01} = I_{10}$ do not, Out would not switch from (its preset value) 0 to 1, for all input combinations but $In_0 = 0$, $In_1 = 0$.

As Boolean gates of practical importance (such as NOR) are commutative, a single voltage level at the *BSL*s of the inputs suffices to define a specific logic function. Each voltage level can serve as a signature for a specific logic gate, along with its preset value. In the following, we will refer to such as V_{gate}. In the example above, $V_{gate} = V_{NOR}$. While NOR gate is universal, we can implement different types of logic gates following a similar methodology for mapping the corresponding truth tables to the CRAM array.

2.2.2 Basic Computational Blocks

We will next introduce basic CRAM computational blocks required in different applications, including inverters (INV), buffers (COPY), 3-input and 5-input majority (MAJ) gates, and 1-bit full adders.

INV: INV is a single-input gate. Still, we can follow a similar methodology to the

NOR implementation (Table 2.2): preset output to 0, and define V_{INV} in a way such that I_0 (I_1), i.e., the current if the input is 0 (1), is higher (lower) than I_{crit} such that the output does (not) switch from the preset 0 to 1. By definition, $I_1 < I_0$ applies, as $R_1 > R_0$. With SOT/SHE-MTJ cells, INV is a non-destructive gate, i.e., input data does not change due to INV operation, since the required switching current depends only on SHE channel which is lower than the critical current of MTJ device. However, in case of STT-MTJ cells, the input cell will inevitably switch due to same current flowing through the input and output cells. INV therefore is a destructive gate in this case, which still can be very useful if the input data is not needed in the subsequent steps of computation. If non-destructive INV is required, INV can be be converted to a 2-input (1 output) INV gate where the additional input is fixed to a logic value, e.g., 0.

COPY: For 1-bit copy, two back-to-back invocations of INV can suffice. A more time and energy efficient implementation, however, can perform the same function in one step as follows: Preset output to 1, and define V_{COPY} in a way such that I_0 (I_1), i.e., the current if the input is 0 (1), is higher (lower) than I_{crit} such that the output does (not) switch from the preset 1 to 0. By definition, $I_1 < I_0$ applies, as $R_1 > R_0$. The discussion about destructive INV gate also applies here.

MAJ: MAJ gates accept an odd number of inputs, and assign the majority (logic) state across all inputs to the output. The structure for a 3-input MAJ3 or 5-input MAJ5 gate is not any different from the circuit structure in Fig. 2.1(c) except the higher number of inputs. As an example, I_{Out} of the MAJ3 gate assumes its highest value for the 000 assignment of the three inputs – as the resistances of the three inputs, R_0, R_1, and R_2, assume their lowest value for 000.

Any input assignment having at least one 1, gives rise to a lower I_{Out} than I_{000}; and having at least two 1s, to an even lower I_{Out}. Finally, I_{Out} reaches its minimum for the input assignment 111, for which the inputs assume their highest resistance. Accordingly, we can preset the output to 1, and define V_{MAJ3} in a way such that I_{Out} remains higher than I_{crit} if the three inputs have less than two 1s, such that Out switches from the preset 1 to 0, to match the input majority. We can symmetrically define V_{MAJ5}, assuming a preset of 1.

XOR: XOR is an especially useful gate for comparison, however, a single-gate CRAM

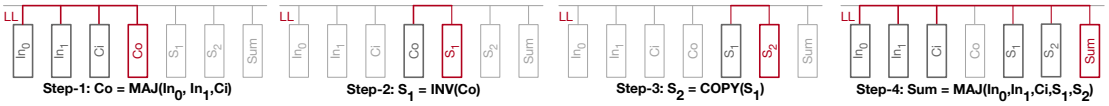


Figure 2.2: Full adder implementation [1]. Output of each gate is depicted in red.

implementation is not possible: In this case we need Out (not) to switch for 00 and 11, but not for 01 and 10, if the preset is 1 (0). However, due to $I_{00} > I_{01} = I_{10} > I_{11}$, and assuming a preset of 1, we cannot let both I_{00} and I_{11} remain higher than I_{crit} (such that Out switches), while $I_{01} = I_{10}$ remain lower than I_{crit} (such that Out does not switch). The same observation holds for a preset of 0, as well.

In_0	In_0	S_1= NOR(In_0,In_1)	S_2= COPY(S_1)	Out= TH(In_0,In_1,S_1,S_2)
0	0	1	1	0
0	1	0	0	1
1	0	0	0	1
1	1	0	0	0

Table 2.3: XOR implementation.

We can implement XOR using a combination of universal CRAM gates such as NOR. Thereby each XOR takes at least 4 steps (i.e., logic evaluations). For pattern matching, we will rely on a more efficient 3-step implementation (Table 2.3):

In *Step-1*, we compute S_1=NOR(In_0,In_1). In *Step-2*, we perform S_2=COPY(S_1). In the final *Step-3*, we invoke a 4-input thresholding (TH) function, which renders a 1 only if its inputs contain more than two zeros: Out = TH(In_0,In_1,S_1,S_2). TH has a preset of 0, and the operating principle is very similar to the majority gates except that TH accepts an even number of inputs. We can further optimize this implementation, and fuse *Step-1* and *Step-2* by implementing NOR as a two-output gate.

Full Adder: A full adder has three inputs: In_0, In_1, and carry-in C_i. The two outputs are Sum and the carry-out C_o. Like other logic functions, we can implement this adder using NOR gates. However, an implementation based on a pair of MAJ gates reduces the required number of steps significantly [30]. Fig.2.2 provides a step-by-step overview:

Step-1: C_o = MAJ(In_0,In_1,C_i)

Step-2: S_1 = INV(C_o)

Step-3: S_2 = COPY(S_1)

Step-4: Sum = MAJ(In_0,In_1,C_i,S_1,S_2)

Step-2 and *Step-3* can be done in one step if we construct a 2-output INV gate, which is fairly straight-forward–thereby reducing the 1-bit addition to a 3-step logic operation.

2.2.3 Row/Column-level Parallelism

Depending on specific memory device technology and CRAM architecture, CRAM supports either row or column-level parallelism. Without the loss of generality, here we assume column parallelism; the row parallelism also works in the similar way with each *LL* spanning along a row.

CRAM can perform only one type of logic function in a column at a time. This is because there is only one *LL* that spans the entire column, and any cell within the column to participate in computation gets directly connected to this *LL* (Section 2.2.1).

On the other hand, the voltage levels on *BSL*s determine the type of the logic function, where each *BSL* spans an entire column. Furthermore, in each row, each *WWL* and *RWL* – which connect cells participating in computation to *LL* – span an entire row. Therefore, all columns can perform the very same logic function in parallel, on the same set of rows. In other words, CRAM supports a special form of *SIMD (single instruction multiple data)* parallelism, where *instruction* translates into *logic gate/operation*; and *data*, into *input cells in each column, across all columns, which span the very same rows.*

Multi-step operations are carried out in each column independently, one step at a time, while all columns operate in parallel. The output from each logic step performed within a column stays in that column, and can serve as input to subsequent logic steps (performed in the same column). All columns follow the same sequence of operations at the same time. In case of a multi-bit full adder, as an example, the carry and sum bits are generated in the same column as the input bits, which are used in subsequent 1-bit additions in the very same column.

To summarize, CRAM can have part or all columns computing in parallel, or the entire array serving as memory. Regular memory reads and writes cannot proceed simultaneously with computation.

Chapter 3

DNA Sequence Alignment Acceleration

3.1 Introduction

The evolution in different domains of science and technology, from social networking to astronomical observation, have brought us into the age of large scale data where abundance of data are available for analysis to aid in the investigation of wide range of problems. This presents itself with a new kind of challenge for the conventional computing since the applications that use large scale data do not scale up well with such model of computing. Such applications suffer from increased energy consumption and latency due to high overhead from data movement between physically separate compute logic and memory. Processing-in-memory (PIM), also known as compute-in-memory or CiM, is an exciting solution to this issue that fuse the capability to perform logic operations with the standard memory functionalities – effectively reducing the separation between compute logic and memory. Moreover, high degree of parallelism, in form of column or row parallelism in 2-D array of memory cells, can boost the throughput of *in-array* or *in-situ* logic operations. PIM presents opportunity to achieve better performance, in terms of latency and energy consumption, in comparison to conventional computing substrates such as CPU or GPU.

The technology of genome sequencing has improved rapidly over the last decade, generating abundance of data for analysis in different domains of research and precision

medicine, e.g., finding causes for diseases such as cancer and Alzhimer's. However, applications that utilize these large scale data suffer from low scalibility associated with classical Von-Neumann systems due to excessive data movement overhead. Presence of high number of memory accesses and high degree of parallelism available in these applications makes them good candidates for PIM based acceleration.

In this chapter we propose an end-to-end PIM accelerator for genomics, BWA-CRAM, which is based on the spintronic CRAM [14] as the PIM substrate. BWA-CRAM represents a PIM accelerator for Burrows-Wheeler Transform (BWT) based DNA short read alignment (BWA). BWT based DNA sequence alignment has become a standard critical tool for sifting through abundance of sequence data, e.g., DNA, available from next generation genome sequencing platforms. We provide the architectural details of BWA-CRAM and characterize the performance in terms of throughput and energy efficiency of alignment. We also compare the performance against a state-of-the-art software implementation of BWA and a PIM based DNA sequence alignment accelerator that uses similar memory technology as BWA-CRAM. In a nutshell, our contributions in this chapter are:

1. We present an accelerator architecture for an emerging and critical genomic application, which by itself represents a key first computational step for many different types of genomic analysis.

2. In doing so, we significanlty reduce the total required memory footprint utilizing PIM features.

3. We showcase that even without altering an algorithm designed for a conventional system (as it is the case for BWA) to better exploit the underlying PIM substrate, significant performance benefits can be achieved.

The rest of the chapter is organized as follows: Section 3.2 discusses the basics of the BWA sequence alignment algorithm, with the architectural details of mapping of BWA in CRAM explained in Section 3.3. Section 3.4 and Section 3.5 report the evaluation setup and outcome of the performance characterization of the accelerator. Finally, Section 3.6 qualitatively compares other similar approaches available in literature and Section 3.7 concludes the chapter.

3.2 Basics

3.2.1 DNA Sequence Alignment

A DNA sequence is a collection of 4 nucleotide base pairs (bp)– (A)denine, (C)ytosine, (G)uanine and (T)hymine. The sequences of bases are recognized by a sequencing machine that encodes the sequences using a string of characters from the alphabet {A,C,G,T}. DNA Sequence alignment is the problem of finding out the location of highest degree of similarity between a very long sequence– a reference, and an orders of magnitude shorter sequence– a read. Typical length of a short read is 100bp while a reference sequence can be in the order of billions of bp. Figure 3.1 shows two reads aligned with an example reference sequence at two locations. The first read aligned at i^{th} location of the reference is an exact alignment since all bps in the reference and the read are same. This is not the case for the second read, since there are some mismatches between the read and the reference (when aligned at the j^{th} location of the reference), known as inexact alignment. Finding inexact alignment is similar, in the context of algorithm, to finding exact alignment, however more expensive in terms of required computation.

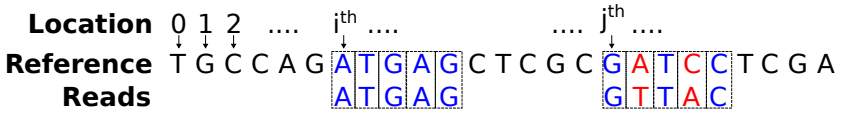


Figure 3.1: DNA sequence alignment.

3.2.2 BWT based Alignment (BWA)

Burrows–Wheeler Transform or BWT is a data compression technique [31], later adopted for fast alignment of short DNA reads [32] due to efficient search (alignment) of sub-strings, i.e., reads, within a very long reference sequence[1]. The first step in BWA is to generate a BWT of the reference, which we will refer to as BWT throughout rest of this chapter. Figure 3.2 shows the generation of the BWT of an example string. The string, i.e., reference sequence in this case, is assumed to be terminated with character "$" where no character in the reference is lexicographically smaller than "$". As shown

[1]We refer the interested reader to numerous BWA references from the literature for more detailed algorithmic discussion.

in Figure 3.2, all possible cyclic rotations of the reference sequence is created and then lexicographically sorted based on the first character– residing in the column labeled by *F(irst)* in the figure. Since each permutation is of same length, it creates a square matrix of characters, where the *L(ast)* column is the BWT of the reference sequence, and stored as the reference– residing in the column labeled by *L(ast)* in the figure. Only BWT is stored in memory.

To perform alignment of short reads, i.e., read sequences, some additional data structures are required. These include the Occurrence Table (*Occ*) which records, at each index in BWT, the number of occurrences of each character in the alphabet of the reference until that index. Figure 3.2 illustrates how *Occ* keeps track of all occurrences of each character in the alphabet. The second data structure is relatively much smaller than *Occ*: *Count*, that stores the number of lexicographically smaller characters in the *F(irst)* column, for each character in the alphabet. Finally, the *Suffix Array (SA)* stores the occurrence (index) of suffixes of all characters in the order they appear in the *F(irst)* column. Typically, a SA is constructed by generating all suffixes of a sequence, i.e., DNA reference, before sorting it lexicographically and storing the sorted suffix indices. As a result, an index in F refers to the same index in SA. The suffixes of the reference in Figure 3.2 are [0] "ATCGAT", [1] "TCGAT", [2] "CGAT", [3] "GAT", [4] "AT", [5] "T" and [6]"$". Sorting the suffixes in lexicographical order of the first characters also sorts the corresponding indexes: {6,4,0,2,3,5,1} which is the suffix array of the reference sequence.

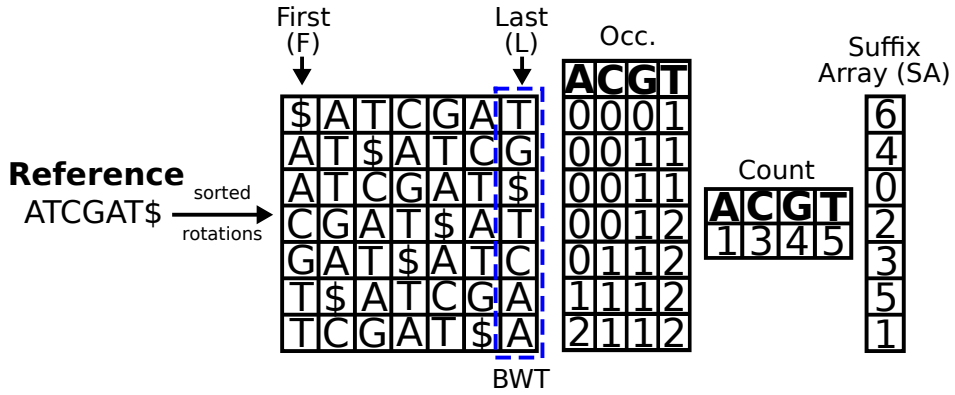


Figure 3.2: Burrows–Wheeler Transform (BWT) of a reference DNA sequence (the length is not up-to-scale to ease illustration).

Figure 3.3 shows an example of how BWA works. BWA searches whether a DNA read is present in the original reference, by considering one character of the read at a time in reverse order. In the example, we start with the last character, A, of the read CGA, and try to find it in BWT which contains two As. The next step is finding the indices of these two As in F. To this end, we consult the corresponding entries of *Occ* and *Count* tables. From the definition of these tables and BWT, it follows that the index in F simply is the sum of the entry (for each A in BWT, as read-out) from *Occ* and the corresponding entry (for A) from *Count*. This renders 1+1=2 for the first A; and 2+1=3, for the second A, respectively. As we start indices of F from 0, we then subtract a 1 from these values, arriving at 1 for the first A; 2, for the second A. Then, we repeat this procedure for the next character in sequence, i.e., G, but this time we limit our search to the BWT entries which reside at F indices 1 and 2. Recall that as F and BWT form the first and last columns of the same square matrix, F indices demarcate row indices in BWT, where we perform the character matching at each step. BWT entries at row indices 1 and 2 are G and *$*. Hence, this time there is only one match in BWT, G – with *Occ* entry 1 and *Count* entry 4, which renders an F index of 1+4-1=4 for the next search. Finally, we move on to the last character of the read in reverse order, C, and confine our search to the BWT row corresponding to F index 4, i.e., BWT row index 4, where a C resides. This C indicates a match and resides at index (row) 1+3-1=3 of F, following the same procedure as before. From the definition of F, BWT, and SA, it follows that the row of SA demarcated by this last index of F we identified (i.e., 3) holds the starting location (index) of the read within the original reference, i.e., 2 – which, as depicted in Figure 3.3 was indeed the case.

To summarize, the alignment of a character is dependent on *L(ast)*, i.e., BWT, to *F(irst)* mapping. A character being aligned is first searched in BWT, which identifies a range for that character in BWT. This range (bound by low and high indices) corresponds to ranks, i.e., order, of that character which is same in both BWT and *F(irst)* – e.g., in Figure 3.3, second A in the BWT and the second A in *F(irst)* correspond to the same location in the reference sequence. Once the ranks are known, the corresponding indices in *F(irst)* are computed. Each such range in *F(irst)* refers to a range of indices in BWT for the alignment of next character (in reverse order) in the read.

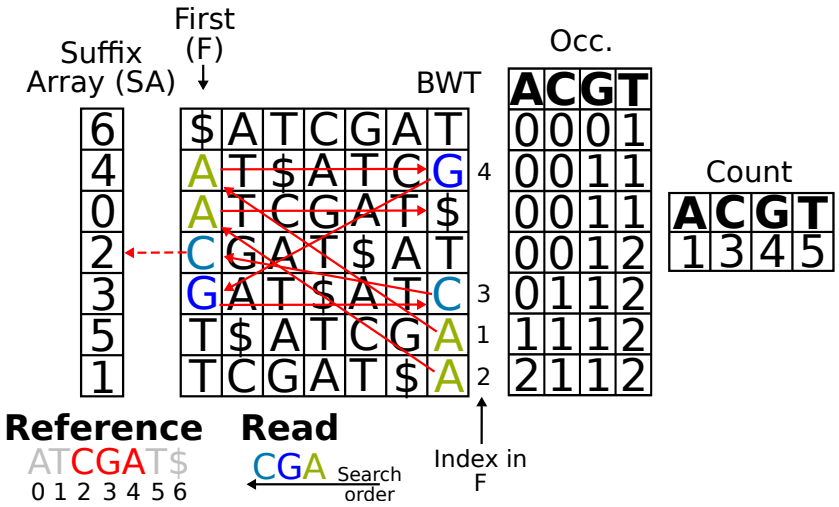


Figure 3.3: BWA reverse alignment of DNA read.

3.2.3 Memory Requirement

Among the data structures required to be stored, in addition to the BWT, *Occ* table is the largest with the number of entries equal to the length of BWT and each entry storing 4 multi-bit numbers. To reduce the required memory size, instead of the entire data structure, we can keep a reduced table where every i^{th} entry is stored – reducing the required size by a factor of i. This does not compromise the correctness: Since not all *Occ* entries are stored, the rank computation during search process can be performed with the help of additional character matching operations, trading storage for computation. Figure 3.4 shows an example for i=5, where the rank of a character A is needed, but the corresponding *Occ* entry is not stored (marked in red). In this case, we can sum the number of As, between that index and the index corresponding to the nearest stored *Occ* entry (i.e., 2) together with the respective stored *Occ* entry (i.e., 253) to derive the rank of that particular A. To reduce computational complexity, the values of *Count* table entries are added to sampled *Occ* table entries – which just needs to be performed one time for the whole reference.

3.2.4 BWA Algorithm: Putting It All Together

BWA is comprised of two stages: i) alignment of a read and ii) accessing SA when the alignment is done. In the following, we provide the pseudo-code which formally summarizes the procedure we outlined by the example in the previous section, and which,

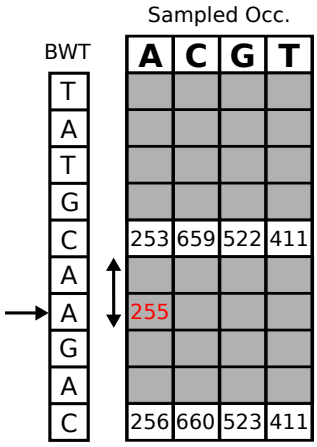


Figure 3.4: Rank calculation with sampled *Occ*.

at the same time, suits well to the column-parallel CRAM implementation (detailed in Section 3.3). Algorithm 1 captures the steps involved in the first stage. For all characters in a read, in reverse order, the characters are aligned and a set of high (idx_h) and low (idx_l) indices are computed (by *interval* function) for each character. This procedure continues until all characters are aligned. No alignment is found if low index becomes greater than the high index.

Algorithm 1 Read alignment

1: $idx\_l = 0$, $idx\_h = len(BWT) - 1$
2: **for** all characters Ch in the read **do**
3: $idx\_l = interval(Occ[idx\_l/i], Ch, idx\_l)$
4: $idx\_h = interval(Occ[idx\_h/i], Ch, idx\_h)$
5: **if** $idx\_l > idx\_h$ **then**
6: No alignment found
7: **end if**
8: **end for**

Algorithm 2 covers the *interval* function, which has three inputs: nearest sampled *Occ* table entry, character to be aligned, and the BWT index (low/high) over which the character is to be aligned. The rank of input character is computed- as explained in the previous section, by counting the number of times that input character is found between the input BWT index and the BWT index corresponding to the input *Occ* entry. The *COMPARISON* operation (line 3) performs character comparison and outputs 1 upon a match. *interval()* returns the computed index (through addition of number of character matches with the corresponding *Occ* table entry) which is used as (low/high) index for

alignment of the next character in the respective read. Recall that, *Occ* is augmented with *Count* entries– no further addition is needed. Here, i is the sampling factor for the reduced *Occ* table. *COMPARISON* is essentially a bit-wise operation (as each character is encoded in multiple bits) which makes the ADD (line 7) a bit-wise operation, as well.

Algorithm 2 *interval* function

1: num_match=0
2: **for** all characters *char* in $BWT[idx, idx/i]$ **do**
3: **if** $COMPARISON(char, Ch) == 1$ **then**
4: $num\_match+ = 1$
5: **end if**
6: **end for**
7: return $ADD(num\_match + Occ[idx/i])$

In the second stage of the algorithm, after all characters from a read are aligned through *interval* function ($idx_l = idx_h$), the location of alignment for the respective read is found by accessing idx_l entry in SA.

3.3 High Level Architecture

From a high-level of abstraction, BWA-CRAM is comprised of a number of functional blocks. Figure 3.5(a) illustrates the overview of the architecture. The core task of aligning reads through BWT is performed by processing elements (PE), whereas SA vectors and Sampled SA are used to find the location of alignment once the alignment is complete. A global controller schedules and orchestrates all operations in BWA-CRAM. All data necessary for alignment, e.g., BWT, *Occ* and *SSA* are generated one time for a given reference database on a conventional Von-Neumann system and stored in the BWA-CRAM. The cost of such is amortized over alignment of millions of reads against that reference database.

3.3.1 Processing Element (PE)

Each PE stores part of the BWT and corresponding entries in the sampled *Occ*, in column major order. PEs are designed from a collection of CRAM tiles, i.e., collection of CRAM cells arranged in 2D with local controller and peripheral to perform memory

and logic operations. Majority of the tiles store BWT, while others store the sampled *Occ* entries. Adjacent tiles are column-wise connected through transistors to perform computation across tiles. Each PE has its own controller that controls the tiles. Figure 3.5(b) shows the organization of tiles in a PE. The layout of the BWT in a PE is captured in Figure 3.5(c). Characters in BWT are represented using 2 bits each. Each tile stores part of the BWT stored by the entire PE with a few rows dedicated to storing of the alphabet {A,C,G,T}. There are a number of rows allocated, in each tile, for use in computation: *Score* (to store the number of character matches) and scratch (for intermediate steps of computation). The purpose of dividing the stored BWT, in a PE, over a number of tiles is to utilize the tile-level parallelism as these tiles can perform computation in parallel. Figure 3.5(c) illustrates how a BWT is segmented and stored in such a tile where (over)underline indicates segments stored in different columns.

PE executes the *interval* function in the Algorithm 2, that is dependent on character comparison and addition, both bit-wise operations. During the first stage of alignment, character comparison is performed between a character from the stored alphabet (which corresponds to the queried character from the read) and all BWT characters (by bit-wise XOR) in column(s). The result of the comparison is stored in *Score* designated rows in respective column(s). Figure 3.6 illustrates the bit-wise matching of the characters. The similarity string, in a column, capture the bits in *Score* designated rows which indicate how many character matches are there in that particular column for a specific character being aligned. Next, bit-wise additions over the similarity string bits are performed in respective column(s) to compute the binary representation of the number of matches, i.e., population count. In this example illustration, a character C is being aligned, therefore C from the alphabet is compared against all BWT characters in column(s). The output of the comparison is stored and bit-wise added to produce the number of matches in respective columns. The output of this addition is the rank for the character being aligned.

The tiles storing the sampled *Occ* in a PE correspond to the BWT stored in that particular PE. Specifically, all 4 numbers in a column (in Figure 3.5(d), for 4 characters in the alphabet, each 32-bit) corresponds an entry of the sampled *Occ* associated with the index of the first BWT character stored in that particular column. Figure 3.5(d) shows such a tile storing the entries of a sampled *Occ* where the full *Occ* is sampled at every

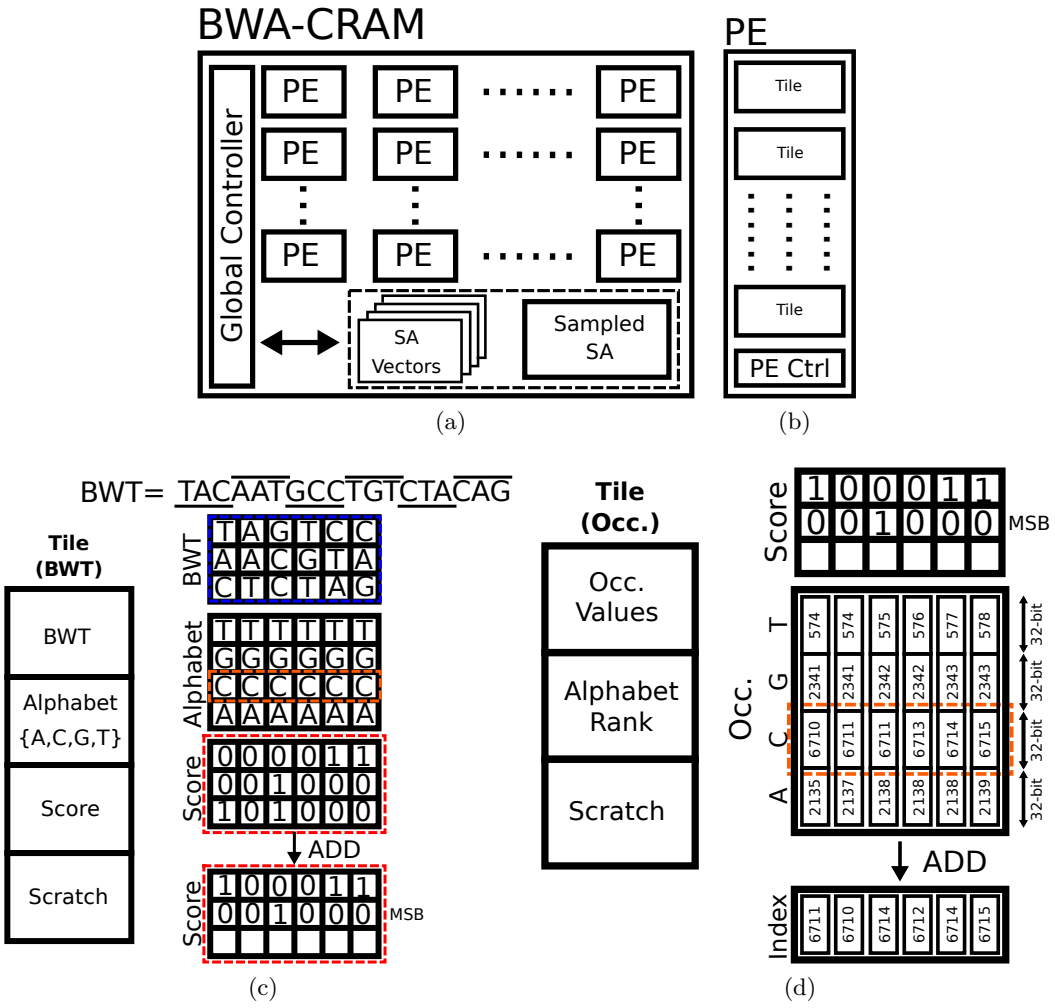


Figure 3.5: High level architecture of BWA-CRAM: (a) Functional blocks, (b) Architecture of PE, (c) Data layout and BWT search in PE, (d) Rank calculation in PE.

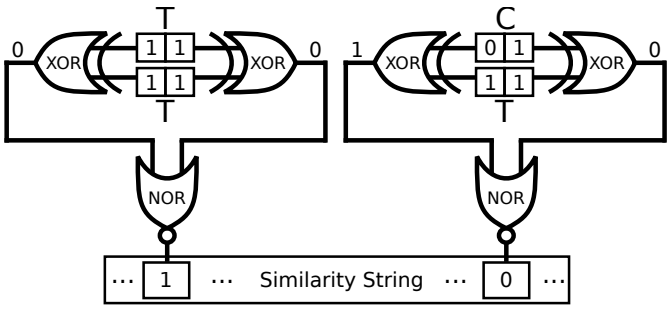


Figure 3.6: Bit-wise character comparison.

3^{rd} BWT character, in this example. Now, to compute the index for the character being aligned – *C*, the number stored in *Score*, i.e., rank and the *Occ* number corresponding to character *C* are bit-wise added which outputs the *index* for that character.

When a character is scheduled for alignment to a PE, the input (high or low) index value to *interval* function is converted to a column index for all tiles in that PE. That is, only one column performs the logic operation in each PE during alignment. However, the columns in all tiles are capable of performing logic operations in parallel as well– hence, the optimization opportunity is there to schedule multiple index computations to a single PE simultaneously.

3.3.2 Accessing SA

Storing the entire SA requires memory that is proportional to the size of the reference. For a large reference, it becomes a problem in the context of practical system design. Here, we show that memory requirement of storing the SA can be reduced by sampling the SA– sampled SA or *SSA*, at regular intervals along the original reference. Although, SA sampling is not a new idea, the contribution lies in accessing the SSA since not all entries in the SA are stored. Let's take a look at the example illustrated in Figure 3.7. It shows an example BWT and the corresponding *F* column along with the original reference. The suffixes are sampled at every even location {0,2,...} of the original reference. Now, the length of the SSA and *F* are different, i.e., a suffix corresponding to an index on *F* might not be stored in SSA. To be able to access such suffixes, a Suffix Vector (SV) is stored, equal in length to the BWT (or *F*), where each bit indicates whether the suffix value at a particular location along SA is sampled in SSA. In case of a SSA

access for a suffix that is not stored (corresponding to the second G on BWT, which is the same as the second G in F), as shown in Figure 3.7, the suffix can be computed by executing the *interval* function in iteration. At each execution of *interval* function (in PE), it returns the F index of the character that precedes it in the original reference. Since the suffix array is sampled at every k^{th} location of the original reference, eventually *interval* function will return a F index whose corresponding suffix value is stored in SSA, with at most k-1 iterations. In this example, after executing *interval* function 1 time, SV indicates that the corresponding suffix is stored in SSA. The suffix value corresponding to the *origin* index is computed by the bit-wise addition between the number of times *interval* function is executed (=1) and the stored suffix value (=2).

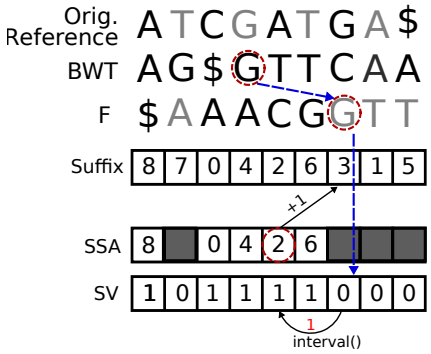


Figure 3.7: Accessing suffix for an index in SSA.

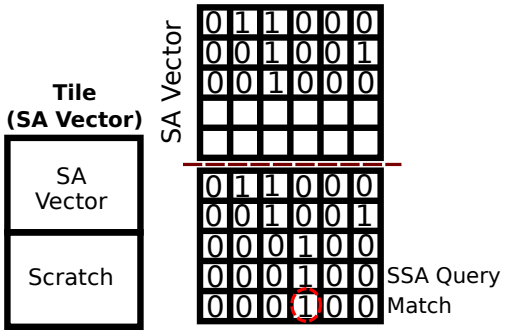


Figure 3.8: Search for a F index in SSA.

The SV is stored in a collection of tiles. Each such tile stores a part of the entire SV, with two rows allocated for performing the check, i.e., bit-wise AND operation, whether the suffix corresponding to a F index is stored in the SSA. The first allocated row is for storing index and the second one stores the result of the bit-wise comparison between the first row and another row with SV. Figure 3.8 demonstrates that a query F index

is written as logic 1 in the *SSA Query* row. After the AND operation, the result is stored in the second allocated row which is read out through standard read mechanism of CRAM. Similar to the tiles in PE, tiles storing SV are also capable of performing column parallel logic operations, which is utilized to perform multiple index checks at the same time. This optimization, as well, directly follows from the basic definitions and mathematical characteristics of the algorithm.

3.3.3 Global Controller

The global controller is responsible for i) scheduling the characters to PE for alignment and ii) performing the index check in SV and access SA– once the alignment is complete.
Runtime scheduling: BWA performs alignment through backward search, i.e., in the reverse order of characters in a read, one character at a time. There are two invocations of *interval* computation for each character during alignment which are scheduled to at most two PEs at the same time- leaving a high number of PEs remaining idle even though each PE can run in parallel. This hints at opportunity to schedule multiple characters to PEs, i.e., multiple *interval* computation at the same time. For this purpose, global controller hosts a runtime scheduler which schedules multiple characters to PEs simultaneously. Since alignment of each read is sequential, i.e., one character after another, this translates into scheduling multiple characters from multiple reads. This involves storing multiple reads concurrently, and dynamically evaluating which characters can be scheduled together. A straightforward implementation would be to store a large number of reads, so that the probability of finding and scheduling a minimum number of characters at the same time is high enough. This comes with a memory overhead since some additional information, for each read, is required to be stored, e.g., next character to schedule, high and low indices from last *interval* computation.
Index comparison in SV: Although intermediate *interval* computations during index comparison in SV are scheduled to PEs, this phase of BWA does not overlap with the alignment stage, for simplicity of design. A more optimized controller can interleave *interval* computations from both stages. For each SSA access, a *count* value is required to be updated each time an *interval* computation is scheduled to a PE. This count operation is also executed using a CRAM tile that supports multiple count operations at the same time through addition of a 32-bit value with logic 1 that indicates one

iteration of *interval* computation.

3.3.4 System Interface

The interface between BWA-CRAM and the host is modelled after SpinPM [33]. The programming interface supports both memory and logic operations, while providing abstraction between host and alignment steps in BWA-CRAM. Being an accelerator, BWA-CRAM does not have access to the virtual memory space. The interface is similar to the loosely-coupled CPU-GPU interface of modern platforms.

3.4 Evaluation Setup

Simulation: Without loss of generality, we consider CRAM with SHE-MTJ for BWA-CRAM. We evaluate the design by an in-house CRAM simulator that incorporates all low level details of the system, e.g., energy and latency values of the logic operations in BWA-CRAM, spatio-temporal scheduling. This tool simulates the design at logic operation granularity– in order to capture the throughput performance and energy consumption of BWA-CRAM with the technology parameters listed in Table 3.1. I_{crit} refers to the threshold current, through SHE channel, for the MTJ to switch resistance state. The peripheral overheads are modeled with NVSIM [34] to extract the row decoder, mux, precharge, and sense amplifier induced energy and latency overheads. Parasitics such as temperature effects on wire resistance are incorporated in this model. All peripheral overheads and the access transistors in each memory cell are modeled at 22-nm (HP) PTM [35].

Dataset: Real human genome [36] is used as the reference and a set of 10 million reads [37] is used as the DNA reads for BWA-CRAM. The reference genome is 3×10^9 bp in length while each read is 100 bp long.

PE modelling: The tiles in a PE are assumed to be 128x128 in dimensions. An array of transistors between adjacent tiles connect the corresponding columns during computation, when required. The overhead for these transistors are considered in the simulation. Each PE stores 512x128 characters of BWT reference in 16 tiles. There are 2 additional tiles in each PE for storing the sampled *Occ* that is sampled at every 512 BWT characters.

SA storage: The entire SA is sampled every 32 locations from the beginning of the reference (*SSA*), and stored in the memory. The resultant memory size is ∼358 MB. To store the bit vector that represents the presence of a particular suffix in the SSA, a collection of 128x128 tiles are used. Each tile stores 126 vectors, each 128-bit long. The remaining rows in each tile are used for bit-wise *AND* operation to check whether a particular suffix is stored.

Runtime scheduler: The scheduler assumes a default dispatch rate of 1000 characters , i.e., 1000 read sequences, simultaneously. This is a conservative assumption in that it represents ∼4.37% utilization of the available PEs.

Design sizing: The entire design, with the selected dataset, requires 45777 PE. The SA is sampled at every 32 suffix position, starting from the beginning of the reference genome. Considering all memory overheads, i.e., all tiles in all PEs and storage for SSA, the total memory footprint reaches ∼2.3 GB.

Baselines for comparison: For the purpose of performance comparison, a BWA based DNA sequence alignment software, soap3-dp [38], is considered. soap3-dp represents the state-of-the-art highly optimized GPU implementation. We also use this software to validate BWA-CRAM's correctness. A Tesla K40 GPU is used for running the soap3-dp without, for a fair comparison, allowing any mismatch during alignment, i.e., exact alignment with seeding. The seeding algorithm helps pruning the alignment space and favor the GPU baseline. To demonstrate the performance improvement achieved by BWA-CRAM, in terms of throughput and energy efficiency, we also consider AlignS [39]- an PIM BWA DNA sequence alignment accelerator that use Spin Orbit Torque (SOT)-MRAM as the memory cell technology. However, alignment in AlignS is performed using dedicated logic circuits, including sense amplifiers and counter circuits, and has intermediate data movement overhead during alignment- unlike BWA-CRAM. Both baselines represent the fastest known solutions from the literature.

3.5 Evaluation

The characterization is in terms of two metrics: i) throughput, in K(ilo)Reads/sec, which represents the rate at which the DNA reads are aligned, and ii) energy efficiency, in KReads/sec/W, which represents how many DNA reads are aligned by BWA-CRAM

Table 3.1: Technology parameters.

Parameters	Value
MTJ Type	Interfacial PMTJ
MTJ Diameter (nm)	10
TMR (%)	100
RA Product ($\Omega\mu m^2$)	20
Critical Current I_{crit} (μA)	3.0 (SHE Channel)
Switching Latency (ns)	1
R_P ($K\Omega$)	253.97
R_{AP} ($K\Omega$)	507.94
R_{SHE} ($K\Omega$)	64
$R_{Trans.}$ ($K\Omega$)	1
V_{INV} (V)	1.07-1.83
V_{COPY} (V)	1.07-1.83
V_{NOR} (V)	0.64-0.77
V_{AND} (V)	0.77–1.02
V_{MAJ3} (V)	0.55-0.62
V_{MAJ5} (V)	0.42-0.45
V_{THR} (V)	0.44-0.47

per unit of energy. The baselines for comparison are also evaluated in terms of these metrics.

3.5.1 Performance

The throughput performance of BWA-CRAM is shown in Figure 3.9. Although GPU, running soap3-dp, has a fairly high alignment throughput, it suffers from large data movement overhead between GPU cores and the global DRAM, which is evident in the throughput value that is much lower than the PIM architectures- AlignS and BWA-CRAM. Being the true PIM that eliminates intermediate data movement during in-memory computation, BWA-CRAM outperforms both GPU and AlignS baselines by 49.17X and 1.68X respectively. The relatively modest improvement in comparison to AlignS is due to the fact that BWA-CRAM performs more in-memory computations than AlignS. For instance, AlignS uses CMOS peripheral circuitry for some operations, e.g., counting the number of character match, which is performed in-memory by BWA-CRAM. Also, there is a computational overhead to search through the SSA for an index, which contributes toward this relatively smaller throughput improvement.

The energy efficiency of the baselines and BWA-CRAM is illustrated in Figure 3.10. It is no surprise that BWA-CRAM outweighs the GPU baseline by 18025.2X due to elimination of a significant number of energy hungry and long latency memory accesses.

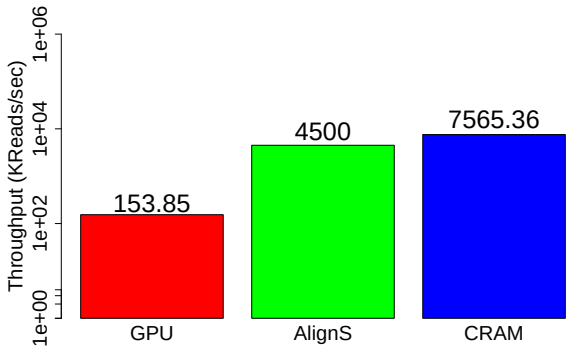


Figure 3.9: Throughput comparison of BWA-CRAM (log scale).

Interestingly, as compared to AlignS, BWA-CRAM is ∼327X more energy efficient although both AlignS and BWA-CRAM use the similar spintronic memory cell technologies. This improvement in energy efficiency is the result of not using the peripheral circuitry to perform in-memory computing, unlike AlignS.

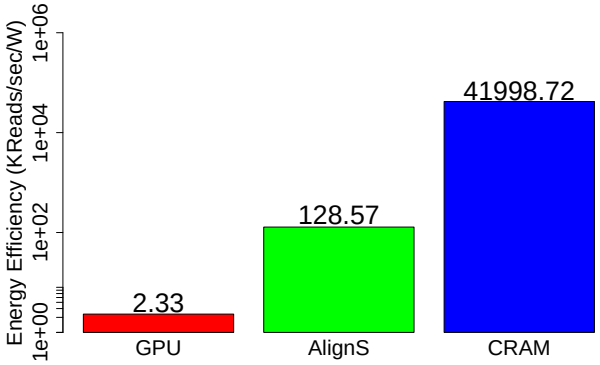


Figure 3.10: Energy efficiency of BWA-CRAM (log scale).

Figure 3.11 shows the latency and energy breakdowns of the design. On the latency front, character comparison operations consume the most latency while addition operations consume around half of that. Controller consumes least amount of latency. SSA access latency, including all intermediate computations, takes ∼25% of the total latency of the design. Due to serialized access to SSA memory, it can become a bottleneck with very large number of reads scheduled to BWA-CRAM simultaneously. Reducing this bottleneck is possible by careful spatio-temporal scheduling of *interval* computation during SA access so that it overlaps with the alignment of next batch of reads, which is left as a future work. On the energy side as well, a similar pattern holds. The majority

of the energy is consumed by the character comparison operations. The next highest energy consuming component is the controller- taking up ∼20% of the energy due to CMOS based implementation. Both latency and energy components corresponding to reading out of index from PE, after *interval* computation is complete, are less than 2%.

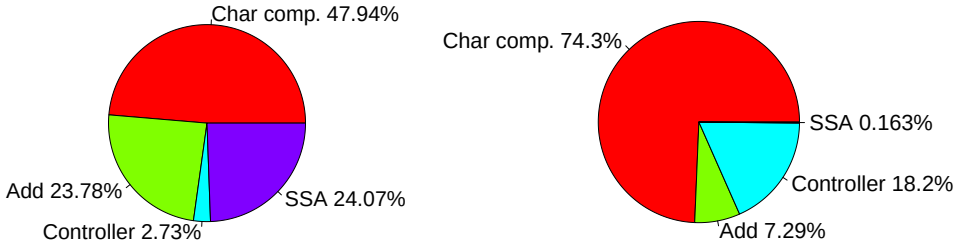


Figure 3.11: Breakdown of (a) latency and (b) energy.

3.5.2 Impact of Runtime Scheduler

The throughput performance reported here corresponds to a specific number of reads, conservatively assumed, scheduled to BWA-CRAM by the runtime scheduler. A more optimized scheduler would increase the throughput performance manyfold. Figure 3.12 captures the impact on performance as more characters (reads) are scheduled simultaneously to BWA-CRAM (X-axis captures increasing number of characters scheduled to BWA-CRAM). As the intuition suggests, the throughput performance increases almost linearly as more characters are scheduled simultaneously. As for energy efficiency, it tends to drop a little due to sequential SA access by BWA-CRAM posing as a serial bottleneck.

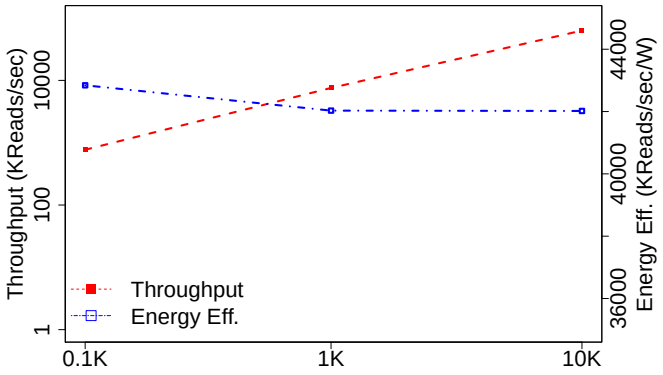


Figure 3.12: Impact of simultaneous scheduling of characters.

However, more reads handled by the scheduler translates into higher memory requirement for context storage. Context is the additional information related to a stored read, e.g., the last character from the read to be scheduled to BWA-CRAM, the current low and high index values for a character. Figure 3.13 shows how the context memory requirement scales as more characters are scheduled to BWA-CRAM. Here, we conservatively assume that during runtime, only 10% of the stored reads will have one character each of which can be scheduled to BWA-CRAM simultaneously. The corresponding storage requirement increases with that, although, with up to 10K characters scheduled, i.e., 100K read contexts stored, the context storage requirement is <1MB.

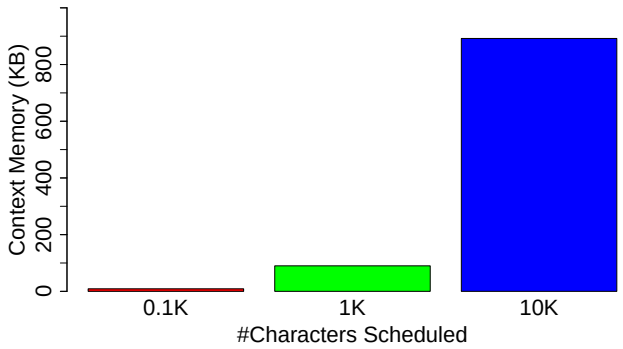


Figure 3.13: Trade-off between BWA-CRAM throughput and required memory for context storage.

3.5.3 Design Size

The memory footprint used by BWA-CRAM is ∼2.3 GB, roughly 25% of that by AlignS [39], largely due to sampling of SA. Figure 3.14 illustrates the distribution of the memory requirements for BWA-CRAM. Unsurprisingly, the majority of the memory is used to store the BWT of the reference DNA sequence and the corresponding overhead for in-memory computation, i.e., PE. The rest of the memory stores the SSA and the SA vector. In comparison to the SA, this represents a reduction of ∼93%, at the expense of additional computation in BWA-CRAM. The size of the SSA can be reduced further, with more computation during the SA access stage of the alignment.

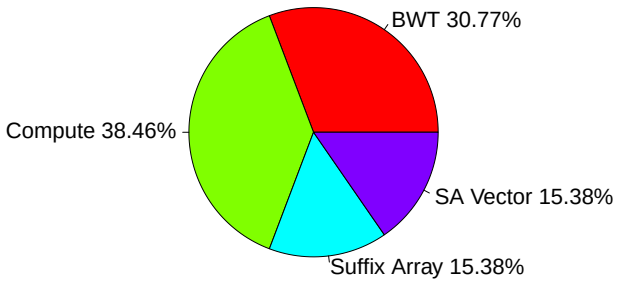


Figure 3.14: Memory footprint breakdown of BWA-CRAM.

3.5.4 Inexact Alignment

Although, the current design of BWA-CRAM supports only exact alignment, it can be extended to include inexact alignment as well- at the expense of more computation. Specifically, it involves executing *interval* function recursively to generate SA intervals that matches a given read with no more than allowed number of mismatches or gaps. The changes required in the design is, mainly, on the scheduling side.

As an example, we implemented such an BWA-CRAM design that allows up to 2 mismatches during alignment. This design exhibits >7X improvement in alignment throughput over SOAP3-dp while maintaining 1820.5X better energy efficiency. Recall that BWA-CRAM does not feature seeding, as opposed to the GPU baseline. If we augmented BWA-CRAM with a seeding algorithm to prune the alignment space, throughput improvement would be even higher.

3.6 Related Work

There is a significant amount of research in DNA sequence alignment due to its criticality and importance in genomics. BWT represents the current state-of-the-art, and has been adopted by a number of software based solutions such as Bowtie(2) [40, 41] and BWA [32, 42]. Apart from the software solutions intended for CPU and GPU platforms, different hardware accelerator designs exist, as well. Most relevant hardware solutions to BWA-CRAM rely on reconfigurable or exotic computing substrates, including near-memory and PIM. MEDAL [43], a near-memory processing architecture based on mature DRAM technology, accelerates DNA seeding algorithm that performs FM-index based

exact match between the read and the reference. FPGA based implementations of high-throughput DNA sequence alignment [44, 45] typically incur orders of magnitude higher power consumption than BWA-CRAM, where FPGA based accelerators at cloud scale are reported to achieve significant speed-up [46]. The exotic race logic based dynamic programming accelerator [47] and resistive Content-Addressable-Memory (CAM) based design [48] both target the Smith-Waterman algorithm, to identify similarity between strings of comparable length. [49], a 3D-ReRAM based alignment accelerator, targets another popular alignment algorithm – Basic Local Alignment Search Tool (BLAST). The resistive memory (ReRAM) based PIM that accelerates BWA-based alignment [50], on the other hand, is significantly slower than BWA-CRAM. Finally, [33] covers a basic CRAM design for large scale string matching, but does not provide an end-to-end solution to the DNA sequence alignment problem.

3.7 Conclusion

DNA sequence alignment has become an integral part of genomics in recent years owing to the availability of high volume of DNA sequence data. Analyzing such large scale data in conventional computing systems is not efficient due to data movement overhead between compute logic and memory. PIM has emerged as an alternative computing paradigm for such applications. In this chapter, we map BWT based DNA sequence alignment (BWA) to SHE-MTJ based Computational RAM (CRAM) substrate to accelerate DNA read alignment with low energy consumption. We show that the CRAM based architecture, BWA-CRAM, outperforms the GPU and PIM baselines, in terms of alignment throughput and energy efficiency, even under conservative assumptions. Furthermore, we show that reduction of large data structures (required for BWA) is also possible using in-memory computing features of CRAM.

Chapter 4

Spintronic Pattern Matching

4.1 Introduction

Emerging spintronic technologies show remarkable versatility for the tight integration of logic and memory. This chapter introduces a high-density, reconfigurable spintronic in-memory compute substrate for pattern matching, SpinPM, which fuses computation and memory by using Spin-Orbit-Torque (SOT) – specifically, Spin-Hall-Effect (SHE) – as the switching principle to perform in-situ computation in spintronic memory arrays. The basic idea is adapting Computational RAM (CRAM) [14] to add compute capability to the magnetic tunnel junction (MTJ) based memory cell [51, 52], without breaking the array regularity. Thereby each memory cell can participate in gate-level computation as an input or as an output. Computation is not disruptive, i.e., memory cells acting as gate inputs do not loose their stored values.

SpinPM can implement different types of basic Boolean gates to form a functionally complete set, therefore there is no fundamental limit to the types of computation. Each column in an SpinPM array can have only one active gate at a time, however, computation in all columns can proceed in parallel. SpinPM provides *true* in-memory computing by reconfiguring cells within the memory array to implement logic functions. As all cells in the array are identical, inputs and outputs to logic gates do not need to be confined to a specific physical location in the array. In other words, SpinPM can initiate computation at any location in the memory array.

Pattern matching is at the core of many important large-scale data analytics applications, ranging from bioinformatics to cryptography. The most prevalent form is string matching via repetitive search over very large reference databases residing in memory. Therefore, compute substrates such as SpinPM, that collocate logic and memory to prevent slow and energy-hungry data transfers at scale, have great potential. In this case, each step of computation attempts to map a short character string to (the most similar substring of) an orders-of-magnitude-longer character string, and repeats this process for a very large number of short strings, where the longer string is fixed and acts as a reference.

In this chapter we detail the end-to-end design of the SpinPM accelerator for large-scale string matching. The design covers device, circuit and SpinPM architecture level details, including the programming interface. We evaluate SpinPM using representative benchmarks from emerging application domains to pinpoint its potential and opportunities for optimization Specifically, Section 4.2 introduces a SpinPM implementation for pattern (string) matching; Sections 4.3 and Section 4.4 provide the evaluation; Section 4.5 compares and contrasts SpinPM to related work; and Section 4.6 concludes the chapter.

4.2 Spintronic Pattern Matching

Pattern matching is a key computational step in large-scale data analytics. The most common form by far is character string matching, which involves repetitive search over very large databases residing in memory. Therefore, compute substrates such as SpinPM, that collocate logic and memory to avoid the latency and energy overhead of expensive data transfers, have great potential. Moreover, comparison operations dominate the computation, which represent excellent acceleration targets for SpinPM. As a representative and important large-scale string matching problem, in the following, we will use DeoxyriboNucleic Acid (DNA) sequence pre-alignment [53] as a running example without loss of generality, and expand SpinPM's evaluation to other string matching benchmarks in Section 4.4.

At each step, DNA sequence pre-alignment tries to map a short character string to (the most similar substring of) an orders-of-magnitude-longer character string, and

repeats this process for a very large number of short strings, where the longer string is fixed and acts as a reference. For each string, the characters come from the alphabet A(denine), C(ytosine), G(uanine), and T(hymine). The long string represents a complete genome; short strings, short DNA sequences (from the same species). The goal is to extract the region of the reference genome to which the short DNA sequences correspond to. We will refer to each short DNA sequence as a *pattern*, and the longer reference genome as *reference*.

Aligning each pattern to the most similar substring of the reference usually involves character by character comparisons to derive a *similarity score*, which captures the number of character matches between the pattern and the (aligned substring of the) reference. Improving the throughput performance in terms of *number of patterns processed per second* in an energy-efficient manner is especially challenging, considering that a representative reference (i.e., the human genome) can be around 10^9 characters long, that at least 2 bits are necessary to encode each character, and that a typical pattern dataset can have hundreds of millions patterns to match [54], where SpinPM can help due to reduced data transfer overhead and (column) parallel comparison/similarity score computations.

By effectively pruning the search space, DNA pre-alignment can significantly accelerate DNA sequence alignment – which, besides complex pattern matching in the presence of errors, include pre- and post-processing steps typically spanning (input) data transformation for more efficient processing, search space compaction, or (output) data re-formatting. The execution time share of pattern matching (accounting for possible complex errors in patterns and the reference) can easily reach 88% in highly optimized GPU implementations of popular alignment algorithms [55] [1]. In the following, we will only cover basic pattern matching (which can still account for basic error manifestations in the patterns and the reference) within the scope of pre-alignment.

Mapping any computational task to the SpinPM array translates into co-optimizing

[1]For this implementation of the common Burrows-Wheeler-Aligner (BWA) algorithm, the time share of the pattern matching kernel, inexact_match_caller, increases from 46% to 88%, as the number of base mismatches allowed (an input parameter to the algorithm) is varied from one to four (both representing typical values).

the data layout, data representation, and the spatio-temporal schedule of logic operations, to make the best use of SpinPM's column-level parallelism. This entails distribution of the data to be processed, i.e., the reference and the patterns, in a way such that each column can perform independent computations.

The data representation itself, i.e., how we encode each character of the pattern and the reference strings, has a big impact on both the storage and the computational complexity. Specifically, data representation dictates not only the type, but also the spatio-temporal schedule of (bit-wise) logic operations. Spatio-temporal scheduling, on the other hand, should take intermediate results during computation into account, which may or may not be discarded (i.e., overwritten), and which may or may not overwrite existing data, as a function of the algorithm or array size limitations.

4.2.1 Data Layout & Data Representation

We use the data layout captured by Fig. 4.1, by folding the long reference over multiple SpinPM columns. Each column has four dedicated compartments to accommodate a fragment of the folded reference; one pattern; the similarity score (for the pattern when aligned to the corresponding fragment of the reference); and intermediate data (which we will refer to as *scratch*). The same format applies to each column, for efficient column-parallel processing. Each column contains a different fragment of the reference.

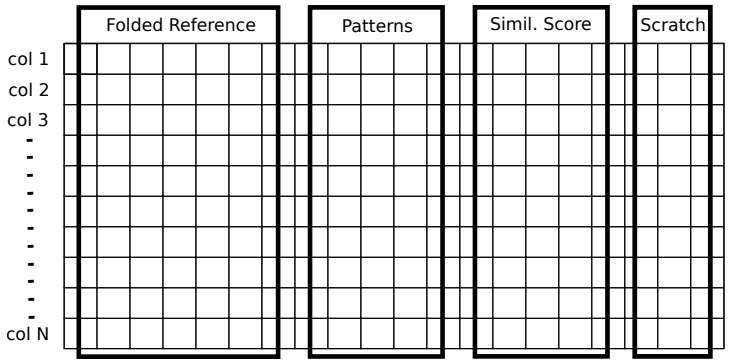


Figure 4.1: Data layout per SpinPM array.

We determine the number of rows allocated for each of the four compartments, as follows: In the DNA pre-alignment problem, the reference corresponds to a genome, therefore, can be very long. The species determines the length. As a case study for large-scale pattern matching, in this chapter we will use approx. 3×10^9 character-long human

genome. Each pattern, on the other hand, represents the output from a DNA sequencing platform, which biochemically extracts the location of the four characters (i.e., bases) in a given (short) DNA strand. Hence, the sequencing technology determines the maximum length per pattern, and around 100 characters is typical for modern platforms processing short DNA strands [56]. The size of the similarity score compartment, to keep the character-by-character comparison results, is a function of the pattern length. Finally, the size of the scratch compartment depends on both the reference fragment and pattern length.

While the reference length and the pattern length are problem specific constants, the (reference) fragment length (as determined by the folding factor), is a SpinPM design parameter. By construction, each fragment should be at least as long as each pattern. The maximum fragment length, on the other hand, is limited by the maximum possible SpinPM column height, considering the maximum affordable capacitive load (hence, RC delay) on column-wide control lines such as *BSL* and *LL*. However, column-level parallelism favors shorter fragments (for the same reference length). The shorter the fragments, the more columns would the reference occupy, and the more columns, hence regions of the reference, would be "pattern-matched" simultaneously.

For data representation, we simply use 2-bits to encode the four (base) characters, hence, each character-level comparison entails two bit-level comparisons.

4.2.2 Proof-Of-Concept SpinPM Design

At the column–level, Algorithm 3 captures two computational phases in SpinPM: *match*, i.e., *aligned bit-wise comparison* and *similarity score computation*. As each column performs the very same computation in parallel, in the following, we will detail column-level operations.

In Algorithm 3, $len(fragment)$ and $len(pattern)$ represent the (character) length of the reference fragment and the pattern, respectively; and *loc*, the index of the fragment string where we align the pattern for comparison. The computation in each column starts with aligning the fragment and the pattern string, from the first character location of the fragment onward. For each alignment, a bit-wise comparison of the fragment and pattern characters comes next. The outcome is a $len(pattern)$ bits long string, where a 1 (0) indicates a character-wise (mis)match. We will refer to this string as the *match*

Algorithm 3 2-phase pattern matching at column-level

```
loc = 0
while loc < len(fragment)−len(pattern) do
  Phase-1: Match (Aligned Comparison)
  align pattern to location loc of reference fragment;
  (bit-wise) compare aligned pattern to fragment
  Phase-2: Similarity Score Computation
  count the number of character-wise matches;
  derive similarity score from count
  loc + +
end while
```

string. Hence, the number of 1s in the match string acts as a measure for how similar the fragment and the pattern are, when aligned at that particular character location (*loc* per Algorithm 3).

A reduction tree of 1-bit adders counts the number of 1s in the match string to derive the similarity score. Once the similarity score is ready, next iteration starts. This process continues until the last character of the pattern reaches the last character of the fragment, when aligned.

Phase-1 (Match, i.e., Aligned Comparison): Each aligned character-wise comparison gives rise to two bit-wise comparisons, each performed by an 2-input XOR gate. Fig.4.2a provides an example, where we compare the base character 'A' (encoded by '00') of the fragment with the base character 'A' (i), and 'T' (encoded by '10') (ii), of the pattern. A 2-input NOR gate converts the 2-bit comparison outcome to a single bit, which renders a 1 (0) for a character-wise (mis)match. Recall that a NOR gate outputs a 1 only if both of its inputs are 0, and that an XOR gate generates a 0 only if both of its inputs are equal. The implementation of these gates follows Section 2.2.2.

SpinPM can only have one gate active per column at a time (Section 2.2.3). Therefore, for each alignment (i.e., for each *loc* or iteration of Algorithm 3), such a 2-bit comparison takes place *len*(*pattern*) times in each column, one after another. Thereby we compare all characters of the aligned pattern to all characters of the fragment, before moving to the next alignment (at the next location *loc* per Algorithm 3). That said, each such 2-bit comparison takes place in parallel over all columns, where the very same rows participate in computation.

Phase-2 (Similarity Score Computation): For each alignment (i.e., iteration of

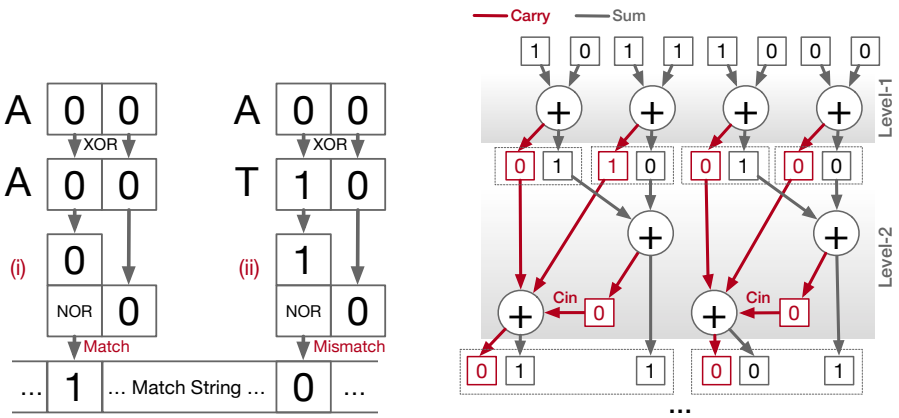


Figure 4.2: Aligned bit-wise comparison (a), and adder reduction tree used for similarity score computation (b).

Algorithm 3), once all bits of the match string are ready – i.e., the character-wise comparison of the fragment and the aligned pattern string is complete for all characters, we count the number of 1s in the match string to calculate the similarity score. A reduction tree of 1-bit adders performs the counting, as captured by Fig.4.2b, with the carry and sum paths shown explicitly for the first two levels. The top row corresponds to the contents of the match string; and each $\oplus$, to a 1-bit adder from Section 2.2.2. $len(pattern)$, the pattern length in characters, is equal to the match string length in bits. Hence, the number of bits required to hold the final bit-count (i.e., the similarity score) is $N = \lfloor log_2[len(pattern)] \rfloor + 1$. A naive implementation for the addition of $len(pattern)$ number of bits requires $len(pattern)$ steps, with each step using an N-bit adder, to generate an N-bit partial sum towards the N-bit end result. For a typical pattern length of around 100 [56], this translates into approx. 100 steps, with each step performing a $N = 7$ bit addition. Instead, to reduce both the number of steps and the operand width per step, we adopt the reduction tree of 1-bit adders from Fig.4.2b. Each level adds bits in groups of two, using 1-bit adders. For a typical pattern length of around 100 [56], we thereby reduce the complexity to 188 1-bit additions in total.

Alignment under basic error manifestations in the pattern and the reference is also straight-forward in this case. For DNA sequence alignment, the most common errors take the form of substitutions (due to sequencing technology imperfections and genetic

mutations), where a character value assumes a different value than actual [57, 58, 59, 60]. We can set a tolerance value t (in terms of number of mismatched characters) based on expected error rates and pass an alignment as a "match" if less than t characters mismatch.

Assignment of Patterns to Columns: In each SpinPM array we can process a given pattern dataset in different ways. We can assign a different pattern to each column, where a different fragment of the reference resides, or distribute the very same pattern across all columns. Either option works as long as we do not miss the comparison of a given pattern to all fragments of the reference. In the following, we will stick to the second option, without loss of generality. This option eases capturing alignments scattered across columns (i.e., where two consecutive columns partially carry the most similar region of the reference to the given pattern). A large reference can also occupy multiple arrays and give rise to scattered alignments at array boundaries, which column replication at array boundaries can address.

Many pattern matching algorithms rely on different forms of search space pruning to prevent unnecessary brute-force search across all possibilities. At the core of such pruning techniques lies indexing the reference, which is known ahead of time, in order to direct detailed search for any given pattern to the most relevant portion of the reference (i.e., the portion that most likely incorporates the best match). The result is pattern matching at a much higher throughput. SpinPM, as well, features search space pruning for efficient pattern matching. The idea is chunking each pattern and the reference into substrings of known length, and creating a hash (bit) for each substring. Thereby, both the pattern and the reference become bit-vectors, of much shorter length than their actual representations. Search space pruning in SpinPM simply translates into bit-wise comparison of each pattern bit-vector to the (longer) reference bit-vector, within the memory array, in a similar fashion to the actual full-fledged pattern mapping algorithm, considering all possible alignments exploiting SpinPM's massive parallelism at the column level. Thereby we eliminate unnecessary attempts for full-fledged matching (using actual data representation and not hashes).

4.3 Evaluation Setup

Simulation Infrastructure: Without loss of generality, we consider CRAM with SHE-MTJ for SpinPM. We developed a step-accurate simulator to capture the throughput performance and energy consumption of SpinPM based pattern matching as a function of the technology parameters. We model the peripheral circuitry using NVSIM [34] to extract the row decoder, mux, precharge, and sense amplifier induced energy and latency overheads at 22nm. NVSIM captures parasitic effects such as the temperature impact on wire resistance and contribution of drain-to-channel capacitance in the drain capacitance.

Technology Parameters: Table 4.1 provides technology parameters for a representative SHE-MTJ and a more conventional Spin-Torque-Transfer (STT)-MTJ (near-term and projected long-term) based implementations. The critical current I_{crit} refers to an MTJ switching probability of 50%, which would incur a high write error rate (WER). To compensate, when deriving gate latency and energy values, we conservatively assume a 2× (5×) larger I_{crit} for the near (long) term STT-MTJ technology. The corresponding value for SHE-MTJ is by definition lower due to Spin-Hall-Effect based switching. The long-term STT specification is based on projections from literature [61]. SHE-MTJ specification comes from [62]. We model access transistors after 22nm (HP) PTM [35].

Array Size & Organization: It is evident that, depending on the pattern matching problem at hand, we might need SpinPM arrays ranging from modest to very large in size. The thought provoking issue here is how to deal with sufficiently large arrays as it might restrict the design space, considering fabrication and circuit-level-design related limitations. As an example, the proof-of-concept implementation requires 300 arrays of 10K columns and around 2K rows each for the string matching case study from genomics. This renders a total size of roughly 24Mb per array, which is not excessively large. Still, the fabrication technology might not be mature enough to synthesize such an array. Commercial MRAM manufacturers address this challenge by banking. For example, EverSpin [66] uses 8 banks in its 256 Mb (32Mb ×8) MRAM product. Distributing array capacity to banks helps satisfy the latency and energy requirement per access, as well. For SpinPM based pattern matching, we too are inclined to use a hierarchy of banks, to enhance scalability. While a clever data layout, operation scheduling and

Table 4.1: Technology parameters.

	SHE	**STT Near-term**	**STT Long-term**
MTJ Type	Interfacial PMTJ		
MTJ Diameter (nm)	10	45	10
TMR (%)	100	133 [63]	500
RA Product ($\Omega\mu m^2$)	20	5	1 [64]
Critical Current I_{crit} (μA)	3.0 (SHE) 3.9 (MTJ)	100	3.95
Switching Latency (ns)	1	3 [65]	1 [63]
R_P ($K\Omega$)	253.97	3.15	12.7
R_{AP} ($K\Omega$)	507.94	7.34	76.39
R_{SHE} ($K\Omega$)	64	–	–
Write Latency (ns)	1.72	3.65	1.72
Read Latency (ns	1.24	1.21	1.24
Write Energy (fJ)	0.4	12.41	2.62
Read Energy (fJ)	0.29	0.29	0.29
V_{INV} (V)	1.05-1.81	0.84–1.3	0.23–0.48
V_{COPY} (V)	1.05-1.81	0.84–1.3	0.23–0.48
V_{NOR} (V)	0.62-0.75	0.68–0.74	0.20–0.22
V_{MAJ3} (V)	0.53-0.61	0.65–0.69	0.20–0.21
V_{MAJ5} (V)	0.40-0.43	0.61–0.62	0.19–0.20
V_{TH} (V)	0.43-0.46	0.62–0.63	0.19–0.20

parallel activation of banks can mask the time overhead, the energy and area overhead would be largely due to replication of control hardware across banks. The most straightforward option for banked SpinPM would be to treat each bank simply as an individual array which would map even shorter fragments of the reference to patterns from the input pattern dataset.

Search Space Pruning: Without loss of generality, we use GRIM filter [67] to convert the patterns and the reference to bit-vectors. Except bit-vector generation, all operations (including bit-vector mapping) are implemented entirely in SpinPM arrays. We assume dedicated logic for bit-vector generation and synthesize it in 22nm to extract the energy and time overheads. We account for the overhead of search space pruning throughout the evaluation, which spans bit-vector generation and matching in the SpinPM arrays.

Benchmarks: We evaluate SpinPM using four pattern matching applications (which also include common computational kernels for pattern matching such as bit count), besides the running example of DNA sequence pre-alignment throughout the chapter. Table 4.2 tabulates these applications along with the corresponding problem sizes.

Table 4.2: Benchmark applications.

Benchmark	Problem Size	Pattern Length	Sub-Array Size
DNA	3G char.	100 char.	512×512
Bit count	1000000 32-bit vectors	1-bit	512×512
String Matching	10396542 words	10 char. string	512×512
Rivest Cipher 4	10396542 words	248 bit	512×512
Word count	1471016 words	32 bits	512×512

DNA sequence pre-alignment (DNA) is our running case study throughout the chapter.We use a real human genome, NCBI36.54, from the 1000 genomes project [36] as the reference, and 3M 100-base character long real patterns from SRR1153470 [68].
Bit count (BC) [69] counts the number of ones in a set of vectors of fixed length. This includes addition of bits in the vectors and the subsequent addition of all individual counts. The input vectors are mapped to the columns of SpinPM to exploit parallelism.
String Match (SM) [70] matches a search string with a pre-stored reference string to identify the part of the reference of highest or lowest similarity. Space separated string segments and the search substring (which forms the pattern) itself are mapped to SpinPM columns such that all searches are performed in parallel.
Rivest Cipher 4 (RC4) is a popular stream cipher. Upon generating a cipher key, i.e., a string, it performs bitwise XOR on the cipher key and the text to cipher. The same key is used to decipher the text, as well. Segments of input text and the cipher key are mapped to SpinPM columns.
Word Count (WC) [70] counts the number of occurrence of specific words in an input text file, through word matching. The words are mapped to SpinPM columns along with search words, and the word matching in each column is executed concurrently.

Baselines for comparison: For evaluation, we use *Near-Memory-Processing (NMP)* and SpinPM-STT baselines. Since both NMP and STT baselines would outperform a GPU baseline comfortably, GPU is not considered as a baseline for evaluation.
NMP Baseline: For NMP based pattern matching, we use an HMC model based on published data [71], which covers memory and logic layers, and communication links. To favor the NMP baseline, we ignore the power required to navigate the global wires between the memory controller and the logic layer, and intermediate routing elements. For logic layer, we consider single issue in-order cores, modeled after ARM Cortex A5 [72]

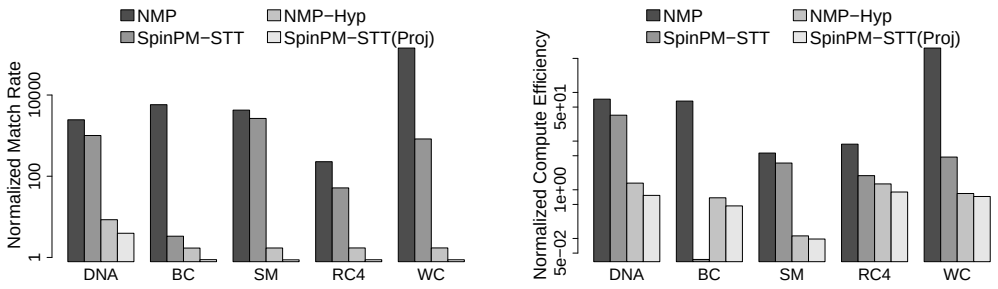
with 1GHz clock and 32KB instruction and data caches. We first consider a total of 64 cores to provide parallel processing. For communication, we assume an HMC-like configuration with four communication links operating at their peak frequency of 160 GB/s. To derive the throughput performance, we use the same reference and input patterns to profile each benchmark. We then use the instruction and memory traces to calculate the throughput. We validated this model through CasHMC [73] simulations. For reference, we also include a hypothetical NMP variant which includes 128 cores in the logic layer, and incurs zero memory overhead.

SpinPM-STT Baseline: To demonstrate the effect of different cell technologies, we also use an STT-MRAM based SpinPMimplementation as a baseline for comparison. STT-MTJ based SpinPM(SpinPM-STT) performs logic operations following the same principle as the SHE-based (SpinPM-SHE) implementation, however, transposed [1].

4.4 Evaluation

We next characterize benchmark applications, in terms of match rate and compute efficiency, when mapped to SHE-based SpinPM (SpinPM-SHE). We use match rate (in terms of number of patterns processed per second) for throughput; match rate per milliwatt, for compute efficiency. Fig.4.3a depicts the match rates for SpinPM-SHE normalized to four baselines: NMP, a hypothetical variant of NMP with no memory overhead (NMP-Hyp), and two variants of STT-based SpinPM (near term SpinPM-STT and projected SpinPM-STT) respectively. Overall, we observe that, SpinPM-SHE outperforms NMP baselines in all benchmark applications. Moreover, in comparison to near-term SpinPM-STT, SpinPM-SHE shows a significant improvement in throughput performance and compute efficiency. All applications have smaller improvement w.r.t. NMP-Hyp as compared to NMP, since NMP-Hyp has no memory overhead and hence has a much higher match rate than NMP to start with.

Fig.4.3b depicts the outcome for compute efficiency. Generally we observe a similar trend to match rate, with all benchmarks (but *BC*) featuring $\geq 2\times$ improvement even w.r.t. the ideal baseline NMP-Hyp. Overall, *BC* shows the least benefit in compute efficiency w.r.t. NMP-Hyp, since *BC* has a lower compute to memory access ratio and eliminating memory overhead greatly improves the NMP-Hyp throughput and compute



(a) Match rate (patt/sec) (b) Compute Eff. (patt/sec/mW)

Figure 4.3: Throughput and energy efficiency of SpinPM-SHE.

efficiency.

SpinPM-SHE performs significantly better, in terms of match rate and compute efficiency, than near term SpinPM-STT due to smaller switching latency and energy consumption. Moreover, the match rate and compute efficiency of SpinPM-SHE are quite close to that of projected long-term STT-MTJ based implementations.

4.4.1 Impact of Process Variation

We conclude the evaluation with a discussion on the impact of process variation, which, due to imperfections in manufacturing technology, may result in significant deviation in device parameters from their expected values. Both access transistors and the SHE-MTJ in an SpinPM cell are subject to process variation. Since access transistors are fabricated using the relatively more mature CMOS technology, the effect of process variation is far less dominating than in its initial years. Being a relatively new technology, MTJ devices are more susceptible to process variation, which directly affects critical parameters such as switching current and switching latency. However, as MTJ technology matures, it is likely that it too will be able to reduce the impact of process variation.

One concern is variation in critical switching current, which can directly translate into variation in bias voltages on bit select lines, i.e., V_{gate}, which determines the gate type. However, different SpinPM gates featuring close V_{gate} values (and hence may be subject to this type of variation) are usually distinguished either by a different value of the preset or a different number of inputs, which makes it unlikely that the gate functions would overlap with each other as a result of variation. We validated this observation assuming a variation in switching current by $\pm 5\%$,$\pm 10\%$ and $\pm 20\%$, respectively, for

all evaluated gates implemented in the SpinPM array.

4.4.2 Gate-level Characterization

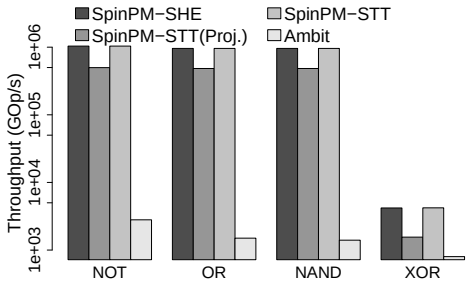


Figure 4.4: Throughput w.r.t. Ambit [2].

We next compare the performance of SpinPM with Ambit [2] and Pinatubo [26], in terms of throughput. Ambit reports a comparative bulk throughput with respect to CPU and GPU baselines, in executing basic logic operations on fixed sized vectors of one-bit operands. Pinatubo reports bit-wise throughput of OR operation only, on a 2^{20} bit long vector. We considered the highest throughput (for a 128-row operation) reported by Pinatubo. To conduct a fair comparison, we assume the same vector size of 32MB used in Ambit. Fig.4.4 captures the outcome, w.r.t. Ambit, in terms of Giga operations per second (GOPs), for NOT, OR, NAND, and XOR implementations. We observe a higher throughput for SpinPM-SHE across all of these bitwise operations.

The high degree of parallelism and lack of actual data transfer within the array – which is not the case for Ambit per Section4.5 – are the main reasons behind such improvement. For the more complex logic operation XOR, the throughput improvement for SpinPM-SHE and projected SpinPM-STT are both $\approx 4\times$ over Ambit; whereas for near-term SpinPM-STT, only $1.34\times$. In this case, SpinPM-SHE performs slightly worse, although very insignificant – around 0.3% less, than projected SpinPM-STT. In comparison to OR throughput of Pinatubo, SpinPM-SHE has similar improvement as projected SpinPM-STT ($12\times$). For this comparison, we do not optimize data layout or operation scheduling for SpinPM. That said, Ambit is based on a mature (DRAM) technology, and therefore more versatile for integration in conventional systems.

4.5 Related Work

Without loss of generality, we base SpinPM on the spintronic PIM substrate CRAM which was briefly presented in [1] and evaluated for a single-neuron digit recognizer along with a small scale 2D convolution in [61]. CRAM is unique in combining multi-grain (possibly dynamic) reconfigurability with true processing *in* memory semantics. The resistive *Associative Processor* [74] and DRAM-based *DRAF* [75] on the other hand, rely on look-up-tables to support reconfigurable fabrics like FPGA.

The conventional bitline computing substrates employ sense amplifier based logic operations and can not truly eliminate data movement overhead within the array boundary. The SRAM-based *Compute Cache* [15] can carry out different vector operations in the cache, but SpinPM supports a wider range of computations on much larger data than could fit in cache. Maintaining data coherence among cores which constitute near-memory logic is also an issue [76, 77] which is not the case for SpinPM due to the absence of dedicated cores (with full-fledged memory hierarchies) to implement logic operations. Recent proposals for bit-wise in memory computing include Ambit [2], DRISA [16], Pinatubo [26] and STT-CiM [27]. DRAM-based solutions such as Ambit or DRISA use modified sense amplifier based designs. These designs support bitwise operations in DRAM, but can only perform computation on a designated set of rows. Thus, to compute on an arbitrary row, the row must first be copied to these dedicated compute rows and then copied back once the computation is complete. While both designs feature high degrees of (column) parallelism, they suffer from data movement overheads within the array boundary. Pinatubo [26] on the other hand, can perform bitwise operations on data residing in multiple rows, using a specialized sense amplifier with variable reference voltage, which increases the susceptibility to variation.

4.6 Conclusion

This chapter introduces SpinPM, a novel, reconfigurable spintronic pattern matching substrate for true in-memory pattern matching, which represents a key computational step in large-scale data analytics. When configured as memory, SpinPM is not any different than an MRAM array. Each MRAM cell, however, can act as an input or output to a logic gate, on demand. Therefore, reconfigurability does not compromise

memory density. Each column can have only one logic gate active at a time, but the very same logic operation can proceed in all columns in parallel. We implement a proof-of-concept SpinPM array with SHE-MTJ technology for large-scale character string matching to pinpoint design bottlenecks and aspects subject to optimization. The encouraging results from Section 4.4 indicate a great potential for throughput performance and compute efficiency.

4.7 Supplementary

4.7.1 Reconfigurability

Invoking a logic gate within the SpinPM array translates into pre-setting the output, connecting all cells participating in computation to LL by setting the corresponding WLs, and setting a voltage between BSLs of the inputs and output cells that equals to V_{gate}, which depends on the type of the logic gate. Therefore, modulo output pre-set, the complexity of reconfiguration is very similar to the complexity of addressing in the memory array. SpinPM is reconfigurable along two dimensions:

- Each cell can serve as an input or as an output for a logic gate depending on the computational demands of the workload within the course of execution.
- For a fixed input-output assignment, the logic function itself is reprogrammable. For example, we can reconfigure the gate from Fig.2.1(b)/(c) to implement another function than NOR by simply changing V_{gate}, to, e.g., V_{NAND} (and applying a different output pre-set, as need be).

By default, SpinPM acts as an MRAM array. A dedicated architecturally visible set of registers keeps the configuration bits to program SpinPM cells as logic gate input/outputs. These configuration bits capture not only the physical location in the array, but also whether the cell represents an input or an output, the pre-set value for the output, and V_{gate}. A fixed or floating portion of the SpinPM array can keep these configuration bits as part of the machine state, as well.

4.7.2 Spatio-Temporal Scheduling

The goal of classic memory data layout optimizations is to perform as many computations as possible per unit data delivered from the memory to the processor, as the data communication between the processor and the memory represents the bottleneck. SpinPM, on the other hand, brings compute capability to the data to be processed. The goal becomes *minimizing the direct physical distance between the cells participating in computation*. Considering that an output cell can serve as an input cell in subsequent steps of computation, the physical location of the cells carrying the input data for subsequent steps can dynamically change as computation proceeds.

This optimization problem gives rise to two strongly correlated sub-problems: the layout of data to be processed in the memory array, and the spatio-temporal scheduling of computations within the array. In this regard, the optimization problem has many analogies to floor-planning and placement algorithms deployed in the computer aided design of digital systems, which aim to minimize the "distance" (in terms of wire length) between interconnected circuit blocks. In SpinPM context, "interconnected blocks" translate into interconnected cells (over LL) participating in computation (Section 2.2.1). We will look closer into this effect in Section 4.7.3.

SpinPM hence features a unique trade-off between data replication and parallelism: Due to the internal array structure, (unless replicated), the same cell can only participate in one computational step at a time, which may impair further opportunities for parallel execution. Data replication can unlock more parallelism in such cases, at the expense of a larger memory footprint.

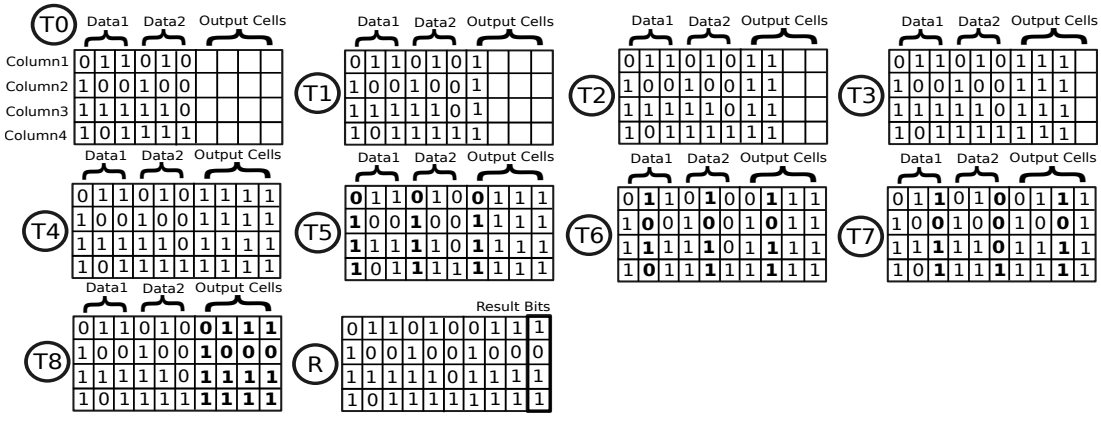


Figure 4.5: Timing diagram of example logic execution.

As an example, Figure 4.5 illustrates a time lapse of logic operations on two datasets, each 3-bit in length, in a SpinPM array of four columns (rotated in horizontal direction for ease of illustration). Time *T0* captures the initial state. *T1-T4* is spent on preset operations of output cells in all four rows, conservatively through standard writes of one row at a time, before a sequence of bitwise ORs and MAJ3s take place (on each column in parallel). Time T5 shows the OR operation on the first bits of the datasets, performed in each column at the same time (bold bit values are involved in computation). *T6* and *T7* do the same for next two bits of the datasets. The last stage of computation, T8, performs MAJ3 on the three bit result of the previous sequence of ORs. Last step, *R*, highlights the final result bits in each column.

4.7.3 Practical Considerations

Array Size: The maximum column height (i.e., the maximum number of rows) per SpinPM array depends on the gate voltage V_{gate} (Section 2.2.1), the interconnect material for LL and BSL (which connects the input and output cells together in forming a gate), as well as the technology node. We conduct the following experiment to determine the maximum column height: We consider a two-input, one output SpinPM gate which has the input cells and the output cell located in adjacent rows. In each experiment, we shift the output cell further away from the input cells, by one cell at a time. The process continues until we reach the terminating condition, which is when the current through the output cell falls below the required critical switching current for the most conservative input cell resistance states.

Assuming copper interconnect segments of 160nm for LL, for representative SpinPM gates used in pattern matching, this analysis renders approximately 2K cells per column at 22nm, where the latency overhead induced by this maximum distance computation barely reaches $< 1\%$ of the switching time of the SHE-MTJ as detailed in Section 4.3.

The feasibility of array dimensions is contingent upon the correct functionality of the array at subarray granularity. Our circuit-level analysis reveals a maximum subarray size of 512×512 bits, without sacrificing the reliability of array functionality.

Since SpinPM cells use two transistors, the area of each cell is dominated by the transistors. The area of each SpinPM cell (at 22nm) is roughly $100F^2$ considering the current density requirement of MTJ devices (MTJs are placed on top of transistors and

roughly consume $\sim 5\%$ of the transistor area).

Array Periphery: Peripheral overheads, mainly induced by addressing and control operations, can play a vital role in determining the pattern matching throughput. Accordingly, throughout the evaluation, we consider the time and energy overheads of peripheral circuitry including row and column decoders, multiplexers, and sense amplifiers. For memory read and write operations a SpinPM array is not quite different than a standard STT-MRAM array, hence we model periphery after the standard STT-MRAM. During computation, however, as all columns operate in parallel, column decoder overhead does not apply (which we conservatively keep). The periphery during computation rather becomes similar to the periphery of Pinatubo [26], an alternative PIM substrate (although SpinPM computation relies on a different mechanism, totally excluding sense amplifier involvement during computation contrary to Pinatubo). Even during computation where all columns are active, the current draw in an SpinPM array remains relatively modest. For example, for SHE-MTJ (as detailed in Section 4.3), a 128MB array would still consume considerably less current than a DDR3 SDRAM write operation [78].

Preset Overhead: Each logic operation requires the output to be (pre)set to a predefined value. Computation is column parallel, i.e., in all columns, the output cell resides in the very same row. Accordingly, before firing column-parallel computation, the corresponding row where the output cells reside should be preset. To this end, we can use a "gang" preset, which presets all cells in the output row(s) simultaneously. The alternative is relying on the standard write operation, which can preset (columns in) one row at a time. The gang preset is equivalent to a parallel COPY operation – where all columns compute in parallel and where the output cells are all in the respective rows subject to gang preset. Hence, the discussion about the periphery overhead during column-parallel computation directly applies here, and the current draw remains modest.

Read Disturbance: Read disturbance is an issue that arises when read current and write current become similar due to non-linear technology scaling of different electrical components of an array. SHE-MTJ devices have separate read and write paths through the device, eliminating read disturbance effects.

4.7.4 System Interface

SpinPM can serve as a stand-alone compute engine or a co-processor attached to a host processor. Following the near-memory processing taxonomy from [79], due to the reconfigurability (Section 4.7.1), both SpinPM design points still fall into the "programmable" class. A classic system has to specify how to offload both *computation and data* to the co-processor, and how to get the results back from the co-processor. For a SpinPM co-processor, we do not need to communicate data values – instead, the SpinPM array requires (ranges of) data addresses to identify the data to process, and the specification for computation, i.e., which function to perform on the corresponding data.
We will next cover the SpinPM system stack to support in-memory execution semantics for pattern matching.

SpinPM Instructions: In addition to conventional memory read and write, SpinPM instructions cover computational building blocks for in-memory pattern matching. Instructions in SpinPM hence form two classes: data transfer (read, write) and computational (arithmetic/logic). By construction, computational SpinPM instructions are *block* instructions: two dimensional vector instructions, which operate *on all columns* and *on a subset of rows* of an SpinPM array at a time. Hence, key operands for any computational SpinPM instruction are the row numbers of the source(s) (i.e., input(s) to computation) and destination(s) (i.e., output(s) to computation). Depending on the size of the pattern matching problem, multiple SpinPM arrays may be deployed in parallel. Therefore, the computational subset of SpinPM instructions facilitates gang-execution on all SpinPM arrays, as well. In the following, we will generically use the term SpinPM *substrate* to refer to all arrays participating in computation. We also make the distinction between *macro-* and *micro*-instructions. The set of micro-instructions covers actual bit-level operations performed in the SpinPM substrate, while the set of macro-instructions forms the high-level programming interface.

Programming Interface: To match SpinPM's column-level parallelism, memory allocation and declaration of variables (which represent inputs and outputs to computation) happen at column granularity. Depending on the problem, a variable may cover the entire column or only a portion. The following code snippet provides an example, where an integer variable `x` gets written (assigned) to row `r` and column `c` in a SpinPM array

(line 5):

```
1  int x = ...;
2  ...
3  int y;
4  preset(r, ncell, val);
5  int_pm x_pm = write_pm(x, r, c, sizeof(x));
6  y = readdir_pm(x_pm);
```

In this case, besides `x` and `y`, `ncell`, `val`, `c` and `r` represent (already defined) integer values. The SpinPM-specific (composite) data type $\texttt{int}_{\texttt{pm}}$ captures row and column coordinates for each variable stored in the array. $\texttt{x}_{\texttt{pm}}$ in line 5 keeps this information for variable `x`, after it gets written to column `c`, from row `r` onwards, by the $\texttt{write}_{\texttt{pm}}$ function. The subsequent read in line 6, conducted by the $\texttt{readdir}_{\texttt{pm}}$, directly assigns the value of `x` to `y`. SpinPM also features a read function, $\texttt{read}_{\texttt{pm}}$, which has a similar interface to $\texttt{write}_{\texttt{pm}}$ with explicit row and column specification. We consider each such function as a macro-instruction.

The `preset` function in line 4 presets `ncell` number of (consecutive) cells, starting from row `r`, each to value `val`. SpinPM features different variants of this function, including one to gang-preset the entire scratch area (Fig. 4.1), and another where `val` is interpreted as a bitmask (of `ncell` bits) rather than a single-bit preset value which applies over the entire range of the specification.

Each pattern matching problem to be mapped to SpinPM features three basic stages:

(i) Allocating and initializing the reference, pattern, and scratch regions in each array (Fig. 4.1);

(ii) Computation;

(iii) Collecting the pattern matching outcome.

Variants of `preset` and $\texttt{write}_{\texttt{pm}}$ functions cover stage (i); and stage (iii) by variants of $\texttt{read(dir)}_{\texttt{pm}}$. Stage (ii) can take different forms depending on the encoding of pattern and reference characters. Primitives such as $\texttt{add}_{\texttt{pm}}(\texttt{int start}, \texttt{int end}, \texttt{int}_{\texttt{pm}}\ \texttt{result})$ apply, which sums all cell contents between rows `start` and `end`, on a per column basis, and writes the result back where `result` points. $\texttt{add}_{\texttt{pm}}$ macro-instruction can directly

implement Phase-2 from Algorithm3 to calculate the bit-count on the match string (Section 4.2.2).

Code Generation: Code generation simply entails translating a sequence of macro-instructions to a sequence of micro-instructions for the SpinPM memory controller (SMC) to drive in-array computation. Micro-instructions specify the type of operation and the rows to connect as inputs and outputs. For example, $\texttt{nand}(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j}, \texttt{r}_\texttt{k})$ specifies row $\texttt{r}_\texttt{i}$ as the output and row $\texttt{r}_\texttt{j}$ and $\texttt{r}_\texttt{k}$ as inputs to form a NAND gate in the SpinPM array. The macro-instruction $\texttt{nand}_\texttt{pm}$, on the other hand, performs the very same operation on multi-bit operands (of width `ncell`): $\texttt{nand}_\texttt{pm}(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j}, \texttt{r}_\texttt{k}, \texttt{ncell})$. In this case, $\texttt{r}_\texttt{i}$, $\texttt{r}_\texttt{j}$, and $\texttt{r}_\texttt{k}$ still demarcate the starting rows for the source and destination (of `ncell` bit) operands. $\texttt{nand}_\texttt{pm}$ hence translates into a sequence of `ncell` number of `nand` micro-instructions. For $\texttt{add}_\texttt{pm}$ type of macro-instructions, on the other hand, a spatio-temporal scheduling pass (Section 4.7.2) determines the corresponding composition of micro-instructions. The goal is to maximize the throughput performance for the given data layout. This usually translates into masking the overhead of presets or other types of writes (per row) by coalescing when possible.

SpinPM Memory Controller (SMC): SMC orchestrates computation in the SpinPM substrate. SpinPM features an internal clock. During computation, SMC allocates each micro-instruction a specific number of cycles to finish depending on the operation and operand widths. This time window includes peripheral overheads and the scheduling overhead due to SMC, besides computation. After the allocated time elapses (and unless an exception is the case), SMC fetches the next set of micro-instructions. SMC features an instruction cache where micro-instructions reside until they are issued to the SpinPM substrate. Before issue, SMC decodes the micro-instructions using a look-up table to initiate preset, and subsequently, to set the appropriate voltage level on input and output BSL (as a function of the operation, as explained in Section 2.2.2), before activating the corresponding rows in the specified arrays for computation. The look-up table keeps the voltage level and the preset value for each bit-level operation from Section 2.2.2, which forms a SpinPM micro-instruction. No look-up table access is necessary for read and write operations.

4.7.5 A Closer Look into Performance and Energy

We will next provide a detailed throughput performance and energy characterization, along with a sensitivity analysis, using *DNA* as a case study. SHE-MTJ is considered as the cell technology used in SpinPM. To characterize SpinPM based DNA sequence pre-alignment, we use an NVIDIA Tesla K20X GPU based implementation of the common Burrows-Wheeler Aligner (BWA) algorithm [80] as a baseline. We deploy the very same reference and input pattern pools for the GPU and SpinPM. In order for the comparison to be fair, we only take the basic pattern matching portion of the GPU baseline into consideration (Section 4.2).

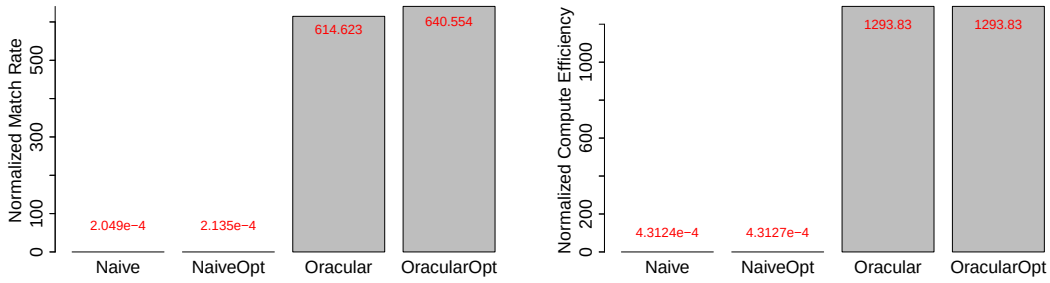
We consider two design points, which differ in how the patterns (from the input pattern pool) get assigned to columns for matching. In other words, how patterns are *scheduled* for computation in the SpinPM array: The first one is a *Naive* implementation, where we take one pattern and blindly copy it to every column of all arrays to perform similarity search. The second implementation, on the other hand, features *Oracular* pattern scheduling, which can avoid assigning a pattern to a column where a too dissimilar (reference) fragment resides. *Oracular* is straight-forward to implement by adding a search-space pruning step before full-fledged mapping takes place, as explained in Section 4.2.2, by e.g., using hash-based filtering [67]. We will leave exploration of this rich design space to future work, but cover the overhead of a representative practical implementation in the following. Any practical SpinPM implementation would fall somewhere in the spectrum between these two extremes.

Naive Design (*Naive*): The caveat here is the very high overhead of redundant computation, due to processing one pattern at a time and mapping each such pattern naively to all reference fragments. As a single pattern is matched to the entire reference, across all arrays, at a time, the apparent serialization hurts the throughput, in terms of the number of patterns matched per second, i.e., the *match rate*.

Oracular Pattern Scheduling (*Oracular*): The oracular scheduler resides between the input pattern pool and SpinPM, and controls to which column in which array each pattern goes. *Oracular* may still feed a given pattern to multiple columns, in multiple arrays, however, does not consider columns which carry a too dissimilar (reference) fragment. In other words, *Oracular* directs patterns to columns and arrays in a way

such that achieving a high similarity score becomes more likely. While *Oracular* bases its pattern scheduling decisions on perfect information, a practical implementation of this idea would incur the overhead of gathering this information, i.e., extracting a schedule to keep pattern matching confined to columns where a high similarity score is more likely. In any case such smart scheduling of patterns benefits the throughput performance by reducing redundant computation which eats from the energy budget.

However, since all columns in a SpinPM-SHE array perform pattern matching (in lock-step but) in parallel, before computation begins, we require that all columns have their patterns ready. Scheduling patterns takes time, which might further affect the throughput performance of SpinPM, if we let the array sit idly, waiting for scheduling decisions to take place. We can mask this overhead, as drawing pattern scheduling decisions for all the columns in an array takes less time than writing patterns in the columns of that array. This, in effect, would not introduce any timing overhead towards the system throughput, although there is an energy overhead.

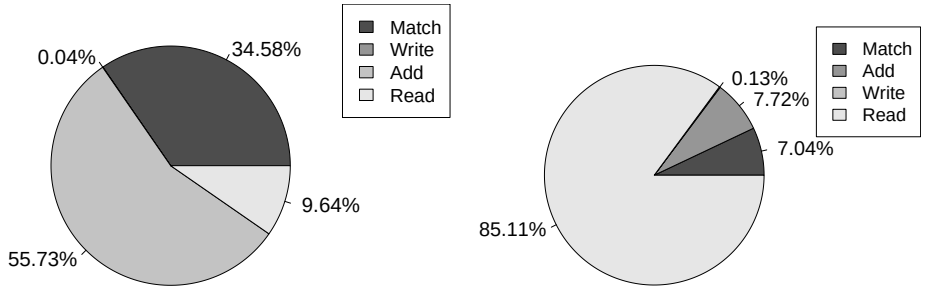


(a) Match Rate (patt/sec) (b) Compute Eff. (patt/sec/mW)

Figure 4.6: Performance and energy characterization.

Fig.4.6 shows the throughput performance and compute efficiency, normalized to GPU baseline, for *Naive* and *Oracular*, when processing a pool of 3M patterns. We use match rate (in terms of number of patterns processed per second) for throughput; match rate per milliwatt, for compute efficiency. *Naive* yields very low throughput – by mapping each pattern to every column of each array at a time, and thereby increasing the total execution time significantly. *Oracular* pattern scheduling is very effective in eliminating this inefficiency: we observe that the throughput performance w.r.t. *Naive* increases by orders of magnitude in this case. The fundamental limitation for *Naive* is

the redundancy in computation.



(a) Energy (b) Latency

Figure 4.7: Breakdown of energy and latency in computation.

We next identify the individual contributions of actual computation stages. Fig.4.7 shows the distribution of energy and latency components. The preset overheads are 67.67% and 70.7% in energy and latency, respectively, where the bit-line (BL) driver energy and latency overheads are $< 1\%$. The breakdowns in Fig.4.7 do not contain preset and BL driver related overheads. Apart from these, we observe that the majority of the energy (Fig.4.7a) is consumed by the match operations and additions during similarity score computations. However, in case of latency (Fig.4.7b), the dominant components change to read-outs of similarity scores while the match and additions have similar shares. In case of both energy and latency, writes consume $< 1\%$ of the share.

This breakdown clearly identifies preset overhead as the essential bottleneck. Also, although the time required by the match and similarity score compute phases are not drastically different, the energy required by the similarity score compute phase is around $1.5\times$ of that of match phase. Accordingly, we next look into preset and similarity score computation operations for optimization opportunities.

Optimized Designs (*NaiveOpt*, *OracularOpt*): As the reduction tree for addition (Fig.4.2b), which is at the core of similarity score computations, already represents an efficient design, we focus on optimizations to reduce the preset overhead. Since presets are inevitable for logic operations, it is not possible to entirely get rid of them. However, we can still hide preset latency through careful scheduling of presets.

As presets do not correspond to actual computation, *Naive* and *Oracular* simply perform them in between computation. The challenge comes from successive steps in computation using the very same set of cells to implement logic functions. Instead

of interrupting computation to preset these cells every time a few computation steps are completed, we can distribute such consecutive steps to different cells, using the scratch area from Fig.4.1, and preset them at once, before computation starts. We call the resulting designs *NaiveOpt* and *OracularOpt*, respectively. The *NaiveOpt* and *OracularOpt* bars in Figure 4.6a and Figure 4.6b capture the resulting energy and throughput performance. We observe that, for each design option, energy consumption of the optimized case is unchanged. This is because the optimization only changes the scheduling of presets, where the total number of presets performed still remains the same. The throughput performance, on the other hand, skyrockets in both cases thanks to gang presets (Section 4.7.3).

Practical Search Space Pruning: The throughput for *Oracular* represents the theoretically achievable maximum. We next consider a practical implementation, as detailed in Section 4.2.2. For the GRIM filter based implementation, we observe that the filtering overhead, as compared to the actual pattern matching overhead, is very insignificant and therefore has effectively no impact, even considering very high sub-string lengths (used for chunking the patterns and the reference in converting them to bit-vectors). However, the accuracy of the filtering still has an impact, as captured by Fig. 4.8. This figure shows, without loss of generality, how the throughput and compute efficiency (both normalized to NMP baseline) of DNA changes when the number of locations (column indices) in an array for possible matches increases (which would be the case under heavy aliasing during hashing). The decrease in performance numbers is intuitive since more match locations refer to more iterations of the same set of search patterns through SpinPM arrays. Luckily, even for a high degree of filter inaccuracy, SpinPM-SHE can perform better than the baseline.

How close a practical implementation can come to *Oracular* strongly depends on the actual values of the patterns, as well, which may or may not ease scheduling decisions. Since each array keeps consecutive fragments of the reference, it is always possible that patterns directed into a particular array do not have any matches in any of the columns. We may not always be able to eliminate such ill-schedules, depending on the pattern values, where the incurred redundant computation would degrade performance. The feasibility of any pattern scheduler is contingent upon the distribution of the patterns, in terms of the columns in the arrays where the most similar fragments reside.

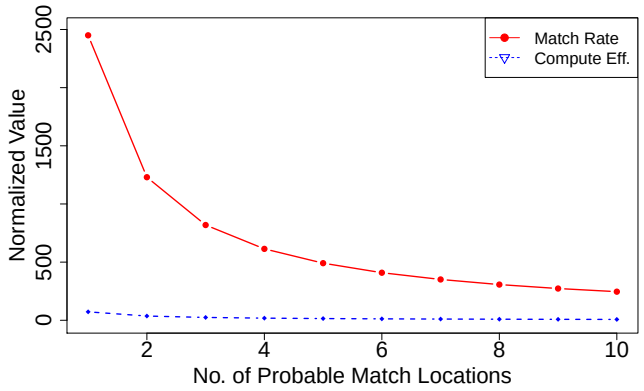
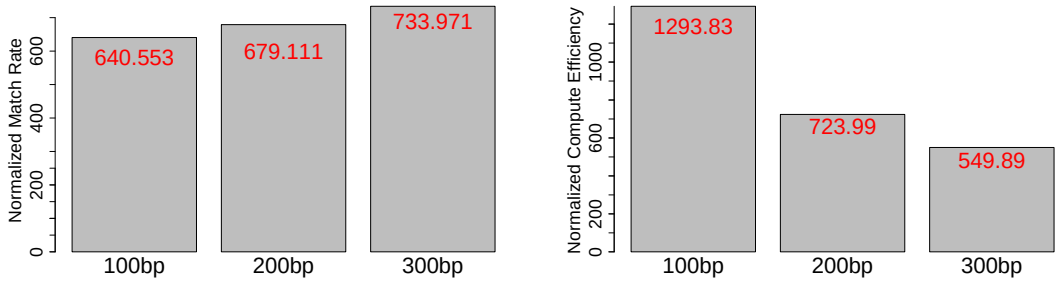


Figure 4.8: Impact of filtering inaccuracy on throughput.

Sensitivity Analysis: Up until now, we have used a pattern length of 100 characters. We will next examine the impact of pattern length on energy and throughput characteristics. Without loss of generality, we confine the analysis to *OracularOpt*. For the purpose of design space exploration, we experiment with pattern lengths of 200 and 300 characters, which are representative values for the alignment of short DNA sequences [56]. We keep the array structure the same, while the reference length remains fixed by construction.



(a) Match rate (patt/sec)

(b) Compute Eff. (patt/sec/mW)

Figure 4.9: Sensitivity to pattern length for *OracularOpt*.

Fig.4.9 summarizes the outcome. Understandably, with the pattern length increasing, more computation becomes necessary to generate the similarity scores in each row. However, this effect does not directly translate into degraded performance: The throughput for increasing pattern lengths remains close to the baseline throughput for 100-character patterns. This is because the preset optimization is scalable. Irrespective

of the application domain, the maximum pattern length is actually limited by technology constraints, since the required number of cells per column also increases with increasing pattern length. We further observe that the compute efficiency (i.e., the match rate per mW) decreases due to increases in computation per alignment, which is congruent with the intuition.

Chapter 5

Acceleration of RNA-Seq Abundance Quantification

5.1 Introduction

Given a biological molecule, *sequencing* is the process of constructing its genomic composition in terms of the basic building blocks –i.e., nucleotide bases A(denine), T(hymine) or U(racil), C(ytosine), G(uanine)– where each base is represented by a character. The output hence is a character string comprised of these bases, termed *read*. Next Generation Sequencing (NGS) machines can typically generate very high volumes of sequence data per run, easily reaching hundreds of Giga (10^9) bases that translates into millions of fixed length strings of base characters called *reads*. Fig.5.1 shows sequencing throughput over time as a proxy for the volume of sequencing data produced by NGS platforms [10]. This trend is expected to hold and result in a steady increase in the volume of sequence data available for genomic analysis, enabling unprecedented advances in bioinformatics and medical research.

RNA molecules, as well, represent chains of nucleotide bases A, U, C and G. *RNA-Seq(uencing) Abundance Quantification* is an important emerging application that particularly benefits from the growth in sequence data volume as more data translates into higher computational accuracy. The goal is to estimate the relative distribution of a given set of RNA sequences in a biological sample. Hence, RNA-Seq in a sense "learns"

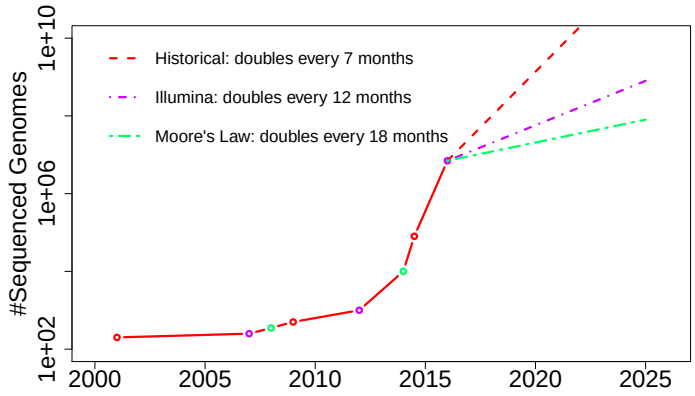


Figure 5.1: Evolution of sequencing throughput over time.

from data. Important use cases for RNA-Seq include novel gene identification, gene expression quantification, mutation analysis, protein synthesis, precision and personalized medicine research (to identify active genes in cells, e.g.) to name a few.

Each RNA-Seq *read* usually is a smaller sub-sequence of a longer RNA sequence called an RNA *transcript*. Such reads are typically sequenced from a biological sample such as a single cell. The set of transcripts that is fixed and already known for a given sample is called RNA *transcriptome*, and uniquely characterizes that sample. The reads sequenced from a sample come from different transcripts in that sample, where the transcripts vary in length and are more likely to generate a read if longer. Therefore, to get a representative RNA-Seq read dataset where the relative number of reads from each transcript follow a similar distribution to the transcripts in that sample, a very large number of RNA-Seq reads are required.

Technically, the *abundance* of the RNA-Seq reads refers to the relative distribution of the transcripts in a sample. Quantifying this distribution requires *alignment* between a large number of RNA-Seq reads and the transcriptome. (Exact) Alignment entails finding out where in each transcript (from that transcriptome) an RNA-Seq read matches the most– through base by base comparison. Classic (exact) alignment algorithms such as TopHat2 [81] or Cufflinks [82], however, require extensive computational resources due to exact alignment that relies on base by base comparison between RNA-Seq reads and the transcriptome. This results in a large number of slow and energy-hungry data transfers between the compute and memory elements (in a traditional setting), which

inevitably degrades the performance. Luckily, for abundance quantification, the actual location of alignment (i.e., exact alignment) is not required. Commonly used algorithms such as Sailfish [83] and Kallisto [84] hence avoid the costly exact alignment process by exploiting the presence of common sub-sequences (called *k-mers*) in both RNA-Seq reads and transcripts, while maintaining a comparable accuracy to exact-alignment based methods. Such *pseudo-alignment* approximation significantly reduces the volume of data transfers during quantification, although for representative problem sizes it still remains forbidding.

A processing-in-memory (PIM) solution such as Computational RAM (CRAM) [14] can effectively address this performance bottleneck by fusing memory and compute elements together. In this chapter, we introduce and detail the HS/SW co-design of CRAM-Seq, a CRAM-based accelerator for RNA-Seq abundance quantification. We demonstrate that CRAM-Seq can achieve accuracy similar to state-of-the-art software solutions such as Kallisto [84], while operating faster and more energy-efficiently. The key contributions of this chapter are as follows:

- We show that *only* presence (i.e., not order) of unique k-mers suffices to perform RNA-Seq abundance quantification.
- We present an end-to-end PIM-based accelerator architecture to achieve higher quantification throughput with lower energy consumption and comparable accuracy when compared to a commonly used high throughput abundance quantification algorithm, even in the presence of noise in the sequence data.

The rest of this chapter is organized as follows: Section 5.2 discusses the core concepts related to RNA-Seq, transcripts and abundance quantification. Section 5.3 illustrates the architecture and discusses different design aspects. Section 5.4 and Section 5.5 present the evaluation setup and experimental findings. Related work is discussed in Section 5.6 with Section 5.7 concluding this chapter.

5.2 Background

5.2.1 RNA-Seq

Typically, both transcripts and RNA-Seq reads don't contain U(racil) as they are sequenced from complementary DNA molecules composed of {A, T, C, G} where each character represents a base pair (bp). Transcripts are longer than fixed length reads, and read length depends on the sequencing technology. Reads from Illumina [85] platforms, e.g., usually are ≈100 bp long.

5.2.2 RNA-Seq Abundance Quantification

Given a set of transcripts, abundance quantification determines the distribution of each transcript in a biological sample when a large number of RNA-Seq reads are sequenced from that sample. If $T = \{t_0, t_1, t_2, ...t_{N_T-1}\}$ is the set of transcripts, the quantified abundance of a transcript is: $Q(t_i) = C(t_i) / \sum_{j=0}^{N_T-1} C(t_j)$ where $C(n)$ is the number of RNA-Seq reads mapped to transcript n.

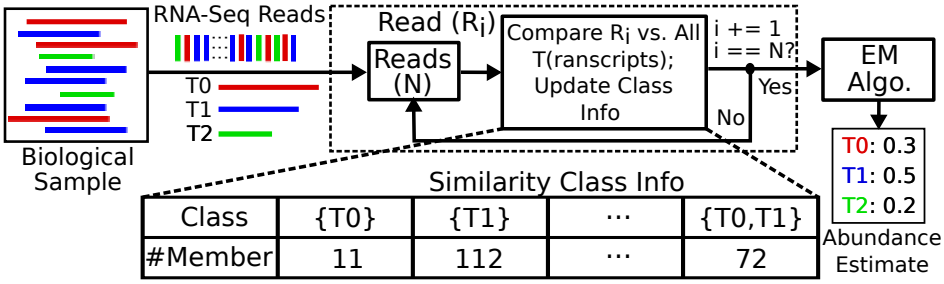


Figure 5.2: Quantification overview.

Fig. 5.2 shows a typical abundance quantification pipeline: A biological sample (containing varying length transcripts *T0*, *T1* and *T2* in different quantities) produces a large (N) number of fixed length RNA-Seq reads. The first step involves, for each RNA-Seq read, identifying the transcripts that have the highest degree of *similarity* (e.g., the total number of locations where both the transcript and RNA-Seq read have a common sub-sequence of a given length in bp) with that read. Be it exact or approximate, this alignment step (as enclosed in the dashed box) helps to identify the (group of) transcript(s) that more likely originated that particular read. Each unique group of

transcripts denotes a *similarity class*. As the read dataset is exhausted, upon the processing of each read, a collection of such similarity classes with corresponding counts of RNA-Seq reads (i.e., members) that map to them gets updated. The similarity class of a read can have one transcript (which is uniquely identified as the source of that read), or multiple transcripts (where maximum similarity applies to all transcripts in the class). In the final step, an iterative clustering algorithm –typically, Expectation Maximization (EM) [86]– distributes read counts to the cluster points, i.e., the transcripts, utilizing all similarity class information, to maximize the abundance of each transcript.

The most compute-resource-heavy part of this problem is the alignment step. As an example, even pseudo alignment (which is typically faster than exact alignment) can consume >85% of total runtime in Kallisto [84] or Sailfish [83]. Therefore, accelerating this step is of particular importance to improve the overall quantification throughput as shown in Fig. 5.3. Being independent from (and significantly faster than) the alignment step, EM algorithm for one set of reads can overlap with the alignment step of the next set of reads. The alignment step determines the rate of quantification as the slower step of computation, i.e., accelerating alignment is critical in improving the overall rate of quantification.

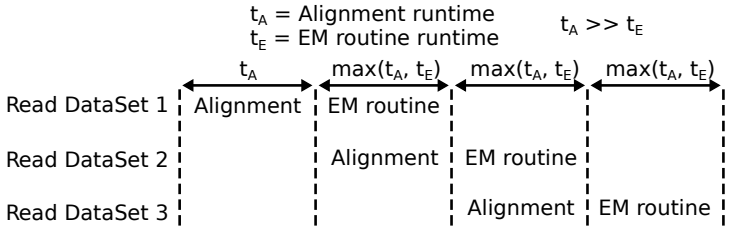


Figure 5.3: Timing diagram of abundance quantification.

5.2.3 Errors in RNA-Seq Reads

Due to imperfection in sequencing technology, RNA-Seq reads can have errors at random locations along the sequence, which typically take the form of insertion, deletion or replacement of one or more bps. The correctness of the abundance algorithm depends on the ability of quantification algorithm to tolerate such noise. Transcripts are considered to be free from noise since these are verified across multiple iterations of sequencing.

5.3 Design

5.3.1 Problem Statement

k-mers are sub-sequences of length k (e.g., a RNA-Seq "TCGAC" has three 3-mers: TCG, CGA and GAC). *The presence, and not the order, of distinct k-mers in a RNA-Seq read sequenced from a transcript matters.* Abundance quantification only requires information regarding the presence of different k-mers in transcripts and corresponding RNA-Seq reads to produce an estimate with an accuracy comparable to commonly used quantification algorithms. Accordingly, we convert a (read/transcript) sequence to a bit-vector (of length determined by k), using the sequence's k-mers. Since k-mers are overlapping in a sequence, the maximum number of k-mers in a sequence of length N is equal to $N - k + 1$. Each unique k-mer generates a unique numerical value when passed through a hash function (due to position and value of each base character in that k-mer). For example, with a hash function $\sum_{i=0}^{k-1} 4^i \times \{A, T, C, G\}$ – where {A=0, C=1, G=2, T=3} and i demarcates the location in the k-mer – a 5-mer "CTCGA" gets the value of 157, as shown in Fig.5.4a. The maximum number of unique hash values from a sequence depends on k, e.g., hash values in Fig.5.4a have a range between 0 and $4^k - 1$. By using hash values to demarcate individual bit locations on a bit-vector and setting those bits, the presence of unique k-mers can be marked in the bit-vector. In a nutshell, a bit-vector records all unique k-mers in a (read/transcript) sequence, but the order of 1's in the bit-vector does not reflect the order of k-mers in the corresponding sequence.

A longer k-mer yields longer bit-vectors that are more likely to be unique, hence such sparser bit-vectors can determine similarity between a RNA-Seq and a transcript more accurately. A small k-mer, on the other hand, can save hardware resources due to smaller (but denser) bit-vectors. Considering noise in sequence data, however, large k may not always deliver higher accuracy. The choice of k depends on the trade-off between available hardware resources and the desired quantification accuracy. Ideally, a bit-vector should be sparse enough such that it uniquely represents a sequence to distinguish it from other sequences.

A set of very long sequences, e.g., transcripts, might have all unique k-mers (for a given k) in each of them, resulting in identical bit-vectors. This loss of information

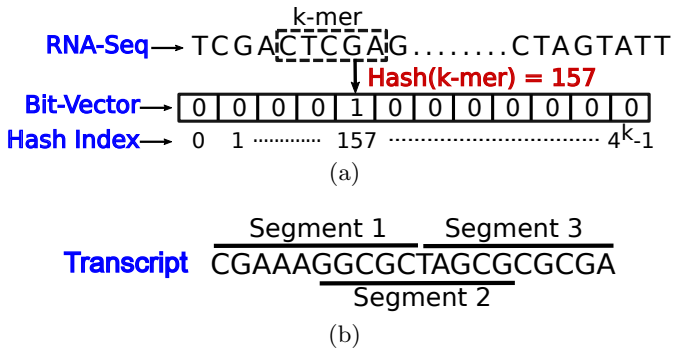


Figure 5.4: (a) k-mer in RNA-Seq; (b) Segmentation of transcript.

(i.e., distinction between different sequences) is very likely if the sequence length $L >>$ #uniquely identifiable k-mers. To handle this issue, we break the transcripts down into overlapping segments (as shown in Fig.5.4b) of constant maximum length, and generate for each segment the corresponding bit-vector. These bit vectors are more likely to uniquely represent the corresponding transcript. The maximum segment length (> read length) is selected to be neither much greater than read length nor too small to produce too many segments. Without overlapping between segments, a RNA-Seq read that spans across two consecutive segments of a transcript could have low similarity with both segments even though that RNA-Seq read comes from that particular transcript.

5.3.2 Algorithm Design

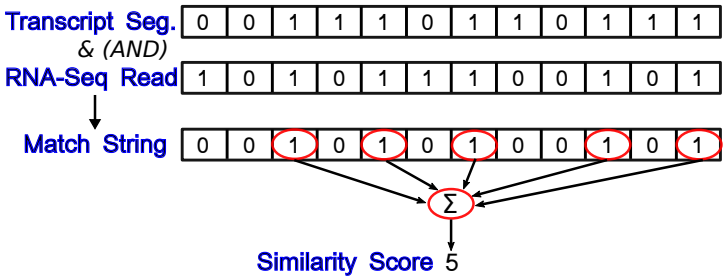


Figure 5.5: Similarity computation.

A set of RNA-Seq reads, $\{RS_1,RS_2,...,RS_N\}$ and a set of transcripts, $\{TS_1,TS_2,...,TS_M\}$ are given. Reads are transformed into the corresponding bit-vectors $\{R_1,R_2,...,R_N\}$. Each transcript is transformed into a number of bit-vectors (T_{10},T_{11},T_{12},T_{20},T_{21}, ..., T_{M0},T_{M1}) since each transcript is also divided into smaller fixed-size segments. In the

first stage of the algorithm, each RNA-Seq read bit-vector is bit-wise compared with all transcript segment bit-vectors to compute the similarity between each {RNA-Seq read, transcript segment} pair. The comparison operation (bit-wise AND), in Fig.5.5, generates a string of 1s and 0s, the *Match String*, where logic 1 marks presence of a unique k-mer in both RNA-Seq read and transcript segment. The number of logic 1's in the *Match String* is a proxy of similarity between that {RNA-Seq read, transcript segment} pair.

Once a RNA-Seq read is evaluated against all transcript segments, transcript segments with maximum similarity are identified, and member counts of corresponding classes are updated accordingly. Once all reads are processed, member counts suffice to compute abundance.

Algorithm 4 CRAM-Seq algorithm

```
1: // Number of transcripts: T
2: // Number of transcript segment bit-vectors: S (S >> T)
3: // Number of RNA-Seq read bit-vectors: N (for N reads)
4: Initialize entries of Similarity Class Table SCT to 0
5: Initialize entries of count array (of size S) to 0
6: for all R in N do
7:     for all t in S do
8:         MS = R & t          // MS: Match String
9:         count[t] = pop_count(MS)
10:    end for
11:    sclass = SimClass(count)
12:    if !(sclass in SCT) then
13:        Create class entry in SCT and return sclass
14:    end if
15:    SCT[sclass] += 1
16: end for
```

Algorithm 5 SimClass(count)

```
1: Initialize transcript_indices array
2: index_list = MAX(count) // List of all segment indices with maximum score
3: for all E in index_list do
4:     transcript_indices.append(LUT[E]);
5: end for
6: return transcript_indices
```

Algorithm 4 summarizes CRAM-Seq's processing steps. For each read, the *for* loop (lines 7-10) counts the number of common k-mers with each transcript segment by performing a sequence of bit-wise additions (i.e., pop_count). Then, for each read, *SimClass* function (Algorithm 5) returns the transcript indices with maximum similarity score (line 11). The list of indices (stored in *sclass*) serves as the ID of the similarity

class of the read currently being processed. If this class hasn't been encountered before, it is recorded as a new entry in *SCT*, an array of key-value stores with key= class ID, value= member count. Finally, the corresponding member count in *SCT* is incremented by 1 (line 15). The *MAX* operation in Algorithm 5 (line 2) returns all transcript segment indices with the maximum score. All such segment indices are used to access a look-up-table (LUT) that stores the mapping between the segment and transcript indices, to extract the corresponding transcript indices (line 6). Finally, after all reads are processed, the class information from *SCT* feeds the EM algorithm to quantify the abundance of the transcripts. All operations in Algorithm 4 and (except for LUT accesses) Algorithm 5 – AND, population count, MAX, class ID check and member count increment – are highly parallel bit-wise operations, hence can effectively be mapped to CRAM.

5.3.3 High Level Architecture

Fig.5.6 (a-c) show the functional blocks necessary for executing Algorithm 4 and Algorithm 5. CRAM-Seq is comprised of three top-level modules, namely vector transformation unit (VTU), similarity class unit (SCU) and a collection of core computational units, termed processing elements (PE). A given set of transcripts are segmented and transformed into bit-vectors in VTU (where the cost is amortized over multiple iterations of quantification with million of reads), and mapped to PEs before (global controller orchestrated) computations take place. After a RNA-Seq read arrives at CRAM-Seq from the host, the read is first transformed in VTU into the corresponding bit-vector, and mapped to all PEs to compute the similarity scores between that read and all pre-stored transcript segments. Through a sequence of *in-situ* logic and arithmetic operations, the similarity scores are computed and sifted through to derive the similarity class statistics for that particular read. SCU stores and updates the statistics, and sends the statistics back to the host once all reads are processed.

Vector Transformation Unit (VTU): This unit is responsible for receiving the data, i.e., RNA-Seq reads from the host, and for generating the corresponding bitvectors. The k-mer length used by VTU is the same as what is used in the transformation of the transcript segments in order to make all bit-vectors to have the same length for bit-wise comparisons. Fig. 5.7 illustrates the transformation operation performed by the VTU.

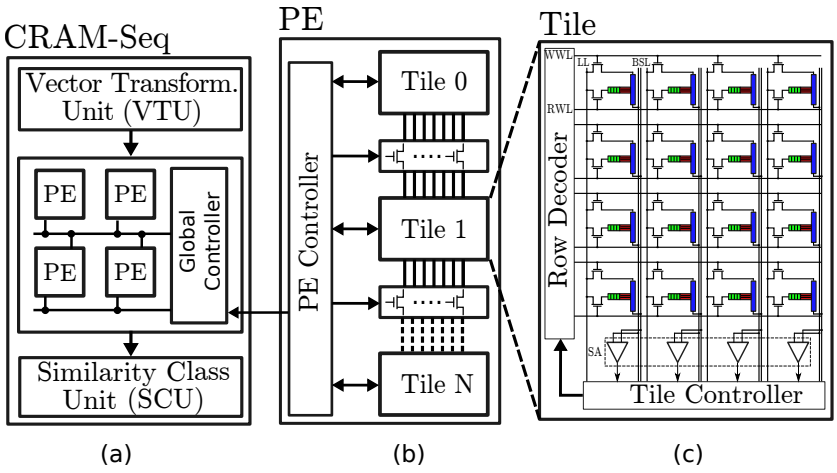


Figure 5.6: High-level architecture of CRAM-Seq.

The bit-vector in VTU is a collection of data-latches that can be $SET(RESET)$ to logic 1(0). At the beginning of the VTU operation, the bit-vector is initialized to all logic *0s*. Transformation of a RNA-Seq read entails *SET*-ting individual bit-locations on that bit-vector where each location corresponds to a k-mer along that read (k-mers are considered overlapping). k-mers in a RNA-Seq read, in 2-bit encoded format, represent overlapping groups of $2k$ bits. In Fig. 5.7, the group of 10-bits represents the first 5-mer, *ATAGC*, in a RNA-Seq read, *ATAGCTGAC*. The second k-mer, *TAGCT*, refers to the subsequent group of 10-bits– skipping the first two bits (i.e., one character). The *group of bits* refers to the location of the *bit* in the bit-vector that would be *SET*. This process is repeated for all k-mers in that read. Once all overlapping k-mers are processed, i.e., all corresponding bit locations on the bit-vector are *SET*, the transformation is complete, and VTU notifies the global controller. As soon as the PEs are done processing the previous read, the global controller writes the transposed read bit-vector(s) to PE(s) through the standard write mechanism (Section 2.2.1). Such pipelined operation hides the latency incurred by VTU.

Processing Elements (PE) are the core computational units, which are responsible for performing computations with CRAM tiles. Each PE consists of a number of connected CRAM tiles. The number of columns in a PE is the number of columns in any PE tile, whereas the number of rows in a PE is the sum of all rows of all tiles in that PE. Fig.5.6(c) shows the internal details of the *tile*, the basic building block of each PE.

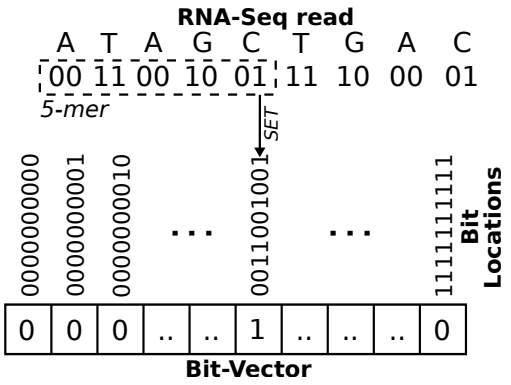


Figure 5.7: Vector transformation unit (VTU).

Each tile is a 2D array of SHE-MTJ cells, capable of performing bit-wise universal logic operations on the stored data.

Tile Connectivity: The tiles are connected through an array of switches (i.e., transistors) that are controlled by the corresponding PE controller, which column-wise connects the LL(s) of adjacent tiles (Fig.5.6(b)). When switches are *OFF*, LL(s) across adjacent tiles are effectively disconnected, hence the tiles can compute in parallel, using individual LL(s) in each tile. However, when turned *ON*, the switches effectively connect LLs across tiles and along PE columns. Therefore, logic operations such as COPY can be performed along columns across the boundary of connected tiles. The *ON* resistance of transistors is much lower than that of SHE-MTJ cells, which makes the overhead of transistor based connection between two LLs across tiles negligible. This control over computation across tiles enables us to exploit tile-level parallelism efficiently.

5.3.4 Data Layout

As shown in Fig.5.8, each PE column is organized in four compartments: transcript segment and RNA-Seq read bit-vectors, result (for storing *Match String* and similarity score) and scratch bits (to store intermediate data during computation). To exploit tile-level parallelism within a PE, a transcript segment bit-vector is divided into a number (equal to the number of tiles in a PE) of smaller sub-vectors and consecutive sub-vectors are stored in consecutive tiles in a PE column. Same applies for read bit-vectors. To summarize, a PE column stores the (transcript segment and read) bit-vectors as a whole, spread across all tiles in that PE.

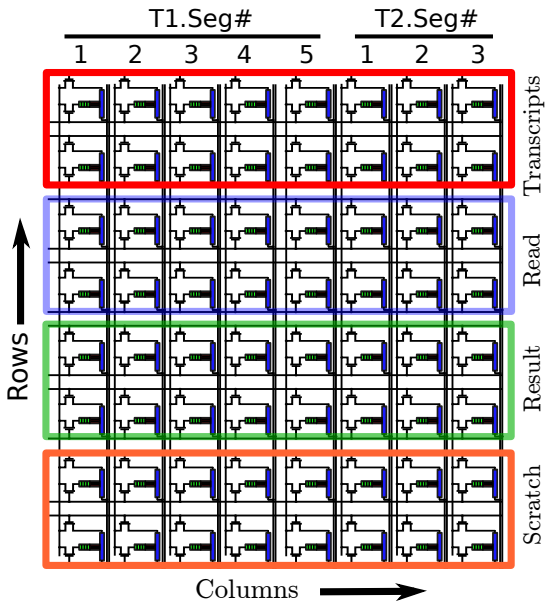


Figure 5.8: PE data layout in CRAM-Seq.

5.3.5 PE Operations

Next we describe the sequence of computational steps within a PE, as captured by Fig.5.9 considering all tiles in the PE. Each step (e.g., St_1) shows a snapshot of the PE state after a sequence of CRAM logic operations are completed. Note that, for ease of illustration and explanation, rows and columns are transposed. This figure illustrates how (transcript segment and read) bit-vectors, *T1–T4* and *R1*, are mapped to a PE, and how the corresponding similarity scores are generated. Initially at step St_0, the transcript segment bit-vectors are stored in the PE: each such bit-vector is divided into equal length sub-vectors with adjacent tiles storing consecutive sub-vectors. For instance, the transcript segment bit-vector, *T1*, is divided into 4 equal length sub-vectors *T1.0–T1.3* and stored in *row-0* of tile-0 – tile-3. Each row of each tile contains one segment of a transcript segment bit-vector. The read bit-vector (of same length) is mapped in the same way and written to rows of each tile. All rows in a tile contain the same read sub-vector (e.g., $R1.0$ in tile-0, $R1.1$ in tile-1 and so on).

In step St_1, partial *Match String*, MX.Y, is produced through bit-wise *AND* operations between the stored transcript segment sub-vectors (TX.Y) and RNA-Seq read sub-vectors (R1.Y), e.g., $M1.0$ represents the partial *Match String* between $T1.0$ and $R1.0$. The *Match String*, *MX*, which is the output of *AND* operations between $T1$

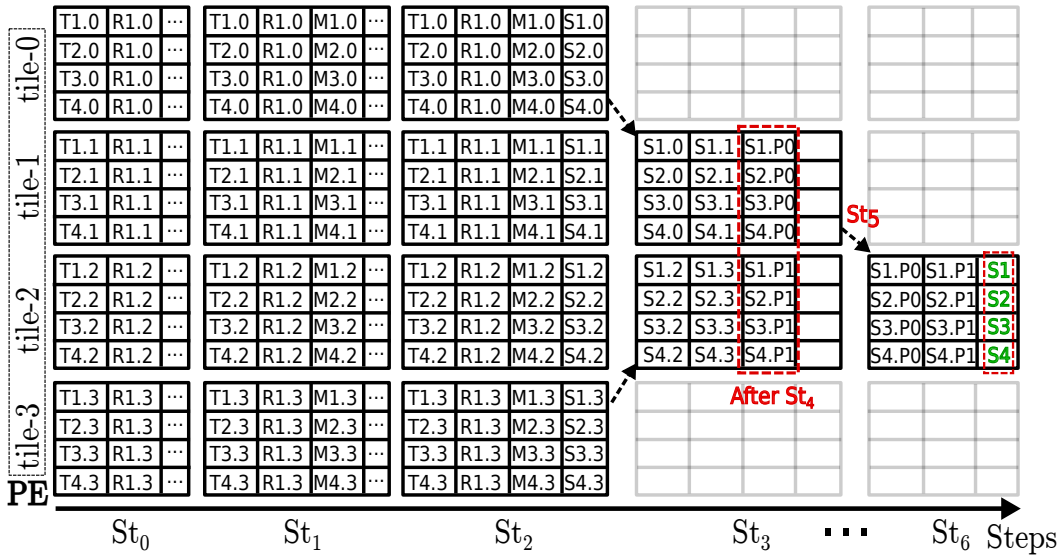


Figure 5.9: Sequence of computational steps within a PE (rows and columns are transposed).

and $R1$, is the concatenation of MX.Y, e.g., $M0$ ={*M0.0*, *M0.1*, *M0.2*, *M0.3*}. Subsequent steps compute the number of 1's in *MX*. Recall that each row of the PE tiles contains partial *Match strings*. Specifically, tile-Y computes a partial count of 1's in $MX.Y$ at step St_2. E.g., tile-0 computes number of 1's in *M1.0* through *M4.0*, in a row parallel fashion. The result is partial similarity scores (e.g., *S1.0-S4.0* in tile-0), where *S1.0-S1.3* represent the partial scores of the final score *S1*. To compute the total counts of 1's (i.e., the final similarity score) in individual *Match Strings*, these partial similarity scores in different tiles need to be reduced to single similarity score. To this end, partial scores are transferred to the adjacent tile(s) at step St_3, to perform binary addition between partial scores (i.e., a sequence of standard bit-wise additions starting from the least significant bits). E.g., *S1.0-S4.0* from tile-0 are transferred to tile-1, in corresponding rows. A similar transfer from tile-3 to tile-2 is conducted simultaneously. Step St_4 (not shown explicitly) performs another round of bit-wise addition between the partial scores, in tile-1 and tile-2 simultaneously. Step St_5 again transfers partial outputs of the binary additions from tile-1 to tile-2. A final step of binary addition, at Step St_6, produces the final similarity scores in tile-2, *S1-S4* (shown in green).

With the PE configuration shown in Fig. 5.9, a bit-vector length of 128-bits would be representative, spread across 4 tiles, with each row storing 32-bits of transcript

segment bit-vector each (Section 5.4). Not shown in the figure (to simplify illustration) are scratch bits in each row which are preset accordingly before computation starts and peripheral overhead. In this case Step St_1 encapsulates 32 AND operations. The outputs of the bit-wise (reduction) addition of *Match String* in each tile are available at Step St_2, after having performed a total of 139 more CRAM logic operations. The copy operations on the 6-bit (bit-wise reduction addition) outputs between adjacent tiles are completed at step St_3, after performing 6 more CRAM operations. The first round of binary addition on partial reduction output is completed at St_4, after performing 18 more CRAM operations. The subsequent copy at Step St_5 encapsulates 7 more operations. The final 8-bit similarity score is available after performing 21 more CRAM operations at St_6, i.e., after completing 7-bit binary additions at each row.

5.3.6 Class Extraction

Class extraction corresponds to *MAX* function in Algorithm 5. This is achieved in two steps: i) finding the transcript segment(s) with maximum similarity score, i.e., PE columns with the maximum score (across all PEs, simultaneously); and ii) finding out the transcript indices of the segment(s) identified in step i). In hardware, this is achieved through the use of sense amplifiers (SA) present in PE tiles. Fig.5.10 shows the basic idea: The tile in the figure holds the final scores along columns, in binary form, between a RNA-Seq read and all transcript segments in the corresponding PE. To find the maximum of two or more such binary values, we simply perform bit-wise comparison starting from the most significant bit (MSB) position. As an example, the tile in Fig.5.10 stores similarity scores {011 (3), 110 (6), ..., 010 (2)}.

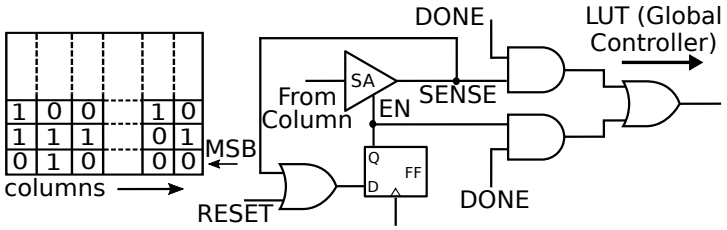


Figure 5.10: In-Memory MAX operation on similarity scores.

In each PE, each column of *one* specific tile (which keeps the multi-bit end outcome, e.g., tile 2 from Fig.5.9) needs to be scanned (at one bit position at a time, from MSB

to LSB positions, across all PEs simultaneously), using the corresponding SAs. Prior to reading out the column bits at the MSB position, PE controller sets the *RESET* feeding the input of the D-FlipFlop (D-FF), which in turn enables (via *EN*) all participating SAs in all PEs. In the first round of comparison, the bits at the MSB position in all columns are read out simultaneously. If at least one bit is 1, all read-out values are stored in the corresponding D-FFs. This in turn enables only the columns (i.e., SA) with logic 1 at the MSB position to participate in the next round of comparison at the next bit position. However, if all values are 0, the sensed data is not stored and all participating SAs remain enabled for the next round. A transistor array pulled-up by a resistor, as shown in Figure 5.11, generates a *HIGH* on *DETECT* when *all* SENSE lines are *LOW*; SENSE lines are connected to gates of individual transistors. A logic *LOW* on *DETECT* is required to store the read-out values in the D-FFs. This self-filtering process continues until the final round (reaching the LSB), when the PE controller sets *DONE*. Thereby the global controller gets either the read-out value (*SENSE*) *OR* the content of D-FF corresponding to column(s) with the maximum value only, i.e., columns that survived until the last round. Each column, i.e., transcript segment is connected to a distinct memory address in a lookup table (LUT) that stores the corresponding transcript index. Only the columns with maximum score determine which LUT address(es) to access. The area overhead of additional logic (for MAX operation and all logic 0 detection) is insignificant (only one tile per PE is involved in these operations) compared to a SOT(SHE)-MRAM substrate of similar size.

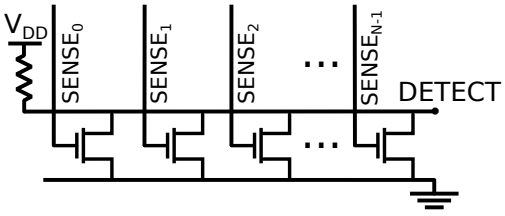


Figure 5.11: Transistor array to detect logic 0 on all SENSE lines.

5.3.7 Similarity Class Unit (SCU)

SCU keeps track of similarity class information. The {*class ID*, *member count*} values are stored in *Similarity Class IDs* and *Similarity Class Counts* modules respectively. Fig.5.12 illustrates the process. *Similarity Class IDs* and *Similarity Class Counts* are

collections of CRAM tiles (similar to PE), which have the same number of columns. For each entry (i.e., column) in *Similarity Class IDs*, there is a corresponding entry in *Similarity Class Counts* that stores and updates the member count of that class. SCU controller receives the transcript indices (which form a similarity class) from LUT (in global controller) for a RNA-Seq read, such as {25, 91} in Fig.5.12. Each similarity class is represented by an *ID bit-vector* of length equal to the total number of *transcripts* stored, where transcripts in a class are marked by setting the corresponding bits to 1. This *ID bit-vector* is used to check if a class already exists in *similarity class IDs* module through bit-wise *AND* operations with the stored *ID bit-vectors*, followed by bit-wise *OR* operations to reduce the outcome to 0 or 1: If 0, no such class exists. A new similarity class is created by storing that ID bit-vector in *Similarity Class IDs*, and incrementing the corresponding entry in *Similarity Class Counts* by 1 through *in-situ* bit-wise addition. Otherwise, the corresponding entry in *Similarity Class Counts* is incremented by 1.

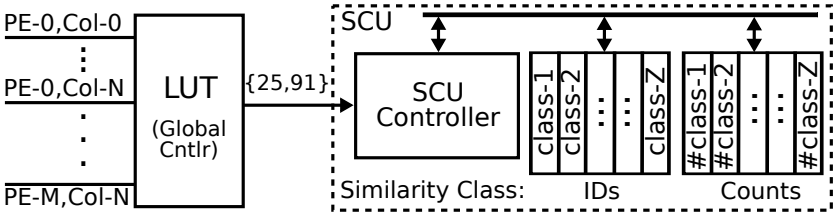


Figure 5.12: Similarity class store and update.

Optimized encoding for *class ID* using an advanced hashing algorithm can reduce SCU energy consumption by reducing #bits to store *class ID*. This would not only simplify the underlying computations but also translate into more room for transcript(-segment)s per chip. Algorithm 6 shows an example, where *class ID*s are derived from the higher-order bits of transcript indices only through shift-left and bit-wise *Exclusive OR (XOR)* operations (performed by *SCU controller*). The number of higher-order bits and shift-left operations depend on the maximum number of transcripts a class can hold.

Algorithm 6 Optimized hash

1: T_{max}: maximum #transcripts per class
2: N_{bits}: #higher-order bits in a transcript index
3: $Transcript_{index} = \{100, 157, 852, 223, ...\}$ // *list of transcript indices*
4: MBL: // *maximum length of a transcript index*
5: $hash\_val = 0$ // *to store final hash value*
6: **for** i in range(T_{max}) **do**
7: $\quad hash\_val = hash\_val \oplus Transcript_{index}[i][MBL : MBL - N_{bits}]$;
8: $\quad hash\_val << 1$
9: **end for**
10: return $hash\_val$

5.3.8 Pipeline Stages

Fig. 5.13 illustrates the life-cycle of RNA-Seq reads in the CRAM-Seq pipeline. A read flows through three functional blocks, each constituting a pipeline stage: VTU, PE+MAX and SCU. The VTU can operate on the next read while PEs process the previous one. However, PEs cannot start processing the next read until the maximum similarity score for the current read is calculated and sent to the SCU. Therefore MAX has to immediately follow PE operations, and together they form the longest latency pipeline stage, PE+MAX. Once the PE+MAX stage is done, the SCU begins immediately, and overlaps with the VTU and PE+MAX stages operating on subsequent reads. Once all reads are processed (i.e., exit the pipeline) the SCU transfers the final class information to the host where the next step of quantification (the EM algorithm) runs.

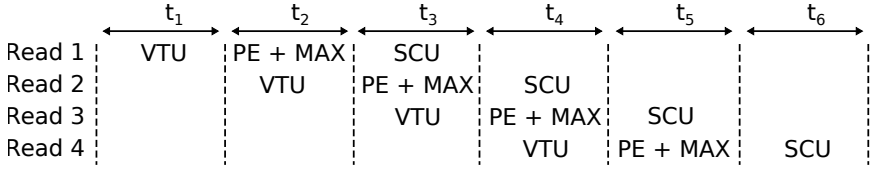


Figure 5.13: Pipeline stages in CRAM-Seq.

5.3.9 System Integration

CRAM-Seq connects to the host machine through the standard memory interface. As an accelerator substrate, CRAM-Seq does not share the virtual memory space with the host. Data (i.e., RNA-Seq reads) stream into the accelerator from the main memory for pseudo-alignment. Once CRAM-Seq computations finish, the similarity class information is sent back to the host who continues with the EM algorithm. Since no fine-grain control over CRAM-Seq operations is necessary, the programming interface

involves instructions just for sending and receiving data to/from CRAM-Seq.

5.3.10 Multi-chip Design

The number of transcript segments in a dataset may exceed the storage capacity of a single CRAM-Seq chip, necessitating multiple chip deployment. Fig. 5.14 illustrates a scale-out system with M CRAM-Seq chips. In this case, the total number of transcript segments in the dataset is divided into chunks and each chunk is assigned to a separate chip, as shown in Fig. 5.14(a). Each CRAM-Seq chip stores only the class information corresponding to the chunk of transcripts it is responsible for. The *throughput* is maintained across all chips in the system due to weak scaling (i.e., addition of more PEs across multiple CRAM-Seq chips).

As each bit-vector corresponding to a RNA-Seq read is processed by all chips in parallel, there is no need for a separate VTU in each CRAM-Seq chip. Sharing the VTU between multiple chips increases area-efficiency, leaving more room for actual computation units (i.e., PEs) in the same chip area or alternatively resulting in smaller chips for the same PE count. As the amount of memory required per chip depends on the number of transcript(segment)s stored, and all chips in the scale-out design keep the same number of transcript(segment)s, the scale-out design is as memory efficient as a single CRAM-Seq chip.

On the other hand, the energy consumption of the scale-out design is by construction higher due to the higher number of pseudo-alignments per read. While exploiting read characteristics (such as the number of unique k-mers) to assign reads only to a subset of chips for pseudo-alignment may reduce the number of pseudo-alignments, a more effective solution is combining such read scheduling with clustering of transcript(segment)s into partitions (as shown in Fig.5.15(a)) based on the pair-wise similarity between the transcripts in the dataset [87], where each transcript partition is uniquely identified by k-mer(s), called *sig-mer(s)*. This partitioning of the transcriptome is based on the following observation: For typical datasets, the bit-wise *AND*s between the transcript segment bit-vectors and a given read bit-vector tend to generate a non-sparse output only for specific subsets of transcripts. Therefore, confining quantification within such subsets of transcripts on a per read basis can cut the number of useless pseudo-alignment computations significantly. Assigning each such partition to a cluster of CRAM-Seq

chips and confining the pseudo-alignment of the reads to the chips in that cluster with the respective partitions only would be especially useful.

Transcript partitioning and subsequent mapping to chips are done only once (before storing the transcripts to CRAM-Seq chips) and therefore, the overhead of such pre-processing is amortized over many iteration of quantification with millions of reads in each iteration. In this case, successive transcripts can go into non-adjacent partitions (e.g., partition-0: {T0,T11,T526 ...}, partition-1: {T1,T2,T87,T901 ...}, ...). This does not have any impact on quantification since each chip keeps track of its transcripts using the original transcript indices within the transcript dataset.

Since a partition of transcripts can have a different #transcripts than another, a large partition might require more than one CRAM-Seq chip to store the entire partition. In the scale-out design, we pack most similar transcripts forming a partition in a single cluster of chips (where #chips in a cluster ≥ 1), and schedule reads with the same similarity signature as the partition to the corresponding cluster, thereby confining the processing of each read to a single cluster. Such clusters of chips are logical clusters where all chips within one (logical) cluster receive the same read for pseudo-alignment. If the transcript database and the corresponding partitions are changed, then simply updating the partition information in the read scheduler would suffice. Depending on the dataset, multiple partitions may reside in the same chip cluster, as well. Even then, the energy consumption per read wouldn't exceed the energy consumed in the cluster with maximum number of CRAM-Seq chips, as opposed to brute-force pseudo-alignment involving all chips in the system on a per read basis.

The *sig-mer(s)* of the transcript partitions stored by all chips are known to the read scheduler, and are used to select a particular cluster of chips for a read based on the presence of the *sig-mer(s)* within that read. The k for *sig-mer*s can be greater than the k used in PEs for fine-grain partitioning of transcripts. The read scheduler, feeding from the *RNA-Seq read pool*, thereby can schedule different reads to different chip clusters, effectively increasing the throughput by a factor of $M = \#chip\_clusters$. Fig. 5.15 (b) illustrates the idea. Partitioning of the transcript dataset (i.e., transcriptome) along with selective scheduling enable ideal performance scaling without compromising energy efficiency.

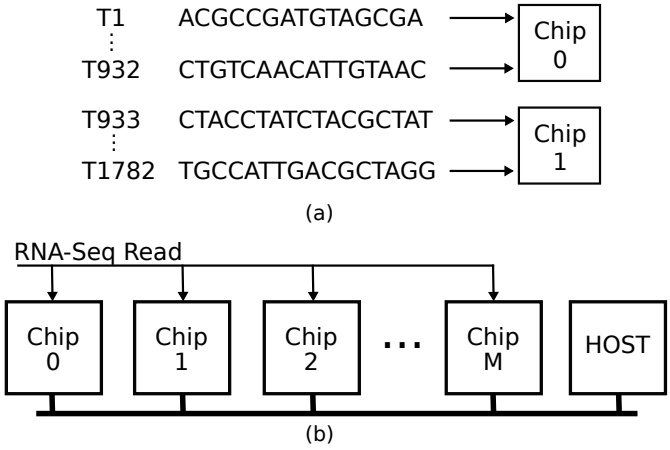


Figure 5.14: Scale-out system of CRAM-Seq chips.

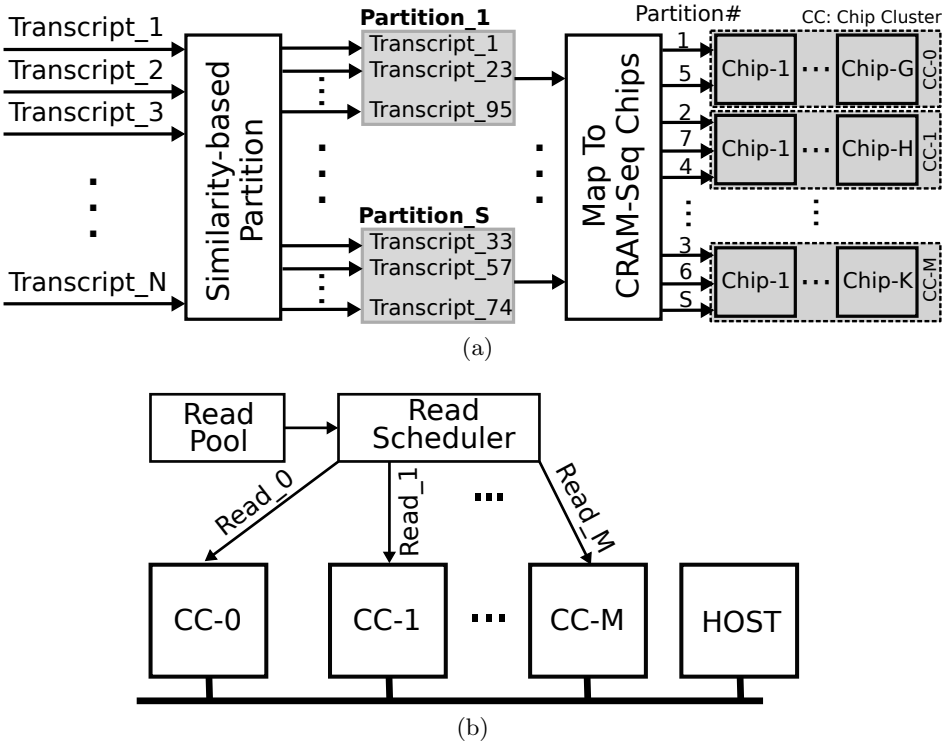


Figure 5.15: (a) Transcriptome partitioning and mapping to CRAM-Seq chip clusters; (b) read scheduling to clusters of chips.

5.4 Evaluation Setup

Without loss of generality, we consider CRAM with SHE-MTJ for CRAM-Seq. To model a PE, a step-accurate simulator is used where each step corresponds to a logic function such as *AND*. Latency and energy for sense-amplifier based MAX operation, transistors and VTU are derived from HSPICE estimates. Peripheral overheads associated with PE are obtained from NVSIM [34], including parasitic effects. All estimates use 22 nm technology node. SHE-MTJ specific parameters come from [88, 89]. Latency and energy for CRAM logic functions are extracted from equivalent circuits as shown in Fig. 2.1c. Table 5.1 provides technology parameters.

Table 5.1: Technology parameters.

Parameters	**Value**
MTJ Type	Interfacial PMTJ
MTJ Diameter (nm)	10
TMR (%)	100
RA Product ($\Omega\mu m^2$)	20
Critical Current I_{crit} (μA)	3.0 (SHE Channel)
Switching Latency (ns)	1
R_P, R_{AP} ($K\Omega$)	253.97, 507.94
R_{SHE}, $R_{trans.}$ ($K\Omega$)	64,1

PE specifics: Each tile is 128×128 that ensures signal integrity (both within the tile and across tiles in a PE), and enough tile-level parallelism while keeping the peripheral circuit overhead moderate. Each PE incorporates 32 tiles. Note that the number of PEs required is determined by the total number of transcript *segments*, therefore, the total number of PEs required may vary across different datasets even with the same number of transcripts. For simplicity we keep the transcripts in transcript datasets of particular sizes unchanged.

Chip Size: Since each 2T(ransistor)1M(agnet) SHE-MTJ cell consumes $\sim 2\times$ area of a 1T1M STT-MTJ cell (where the number of transistors used dominates the cell area), we assume a maximum chip size of 64MB, considering the maximum STT-MRAM chip size available commercially [66]. This chip size estimate is conservative based on the current state of SHE-MTJ technology which is expected to improve as the technology matures.

Problem Datasets: Table 5.2 lists the problem sizes used in the evaluation. To eliminate selection bias, for a dataset of M transcripts, the first M transcripts are taken from [90] (the nucleotide sequences of all transcripts in the reference chromosomes in a human) with lengths $\geq$ 100bp (read length). Transcript segments are selected to be maximum 200bp long (with 100bp overlap between successive segments). CRAM-Seq in this implementation uses 5-mers, i.e., 1024-bit vectors that ensure a sparse representation of transcript segments and RNA-Seq reads (unlike a 4-mer) without a big memory footprint (unlike a 6-mer). The number of RNA-Seq reads in read datasets is selected in proportion to the number of transcript segments in the transcript datasets. 5 read datasets from each transcript dataset are generated, by randomly sampling 100-bp long sub-sequences from the transcripts. Then, noise is added to the reads, assuming a single substitution rate of 0.13% and insertion/deletion error rates of 0.01% to imitate practical Illumina sequencing platforms. The maximum number of similarity classes supported by CRAM-Seq for different problem datasets is derived by profiling the datasets.

Table 5.2: Problem sizes evaluated.

#Transcripts	**#Segments**	**#PE**	**#Reads (10^6)**	**#Sim. Class**
100	931	8	0.2	1000
200	2091	17	0.45	2000
400	4724	37	1.01	6000
800	10691	84	2.3	13000
1000 (default)	14687	115	3.2	17000
3000	46065	360	10.0	34000
5000	78144	611	17.1	60000
8000	123185	963	27.0	80000

Baseline for Comparison: We use Kallisto [84], the state-of-the-art abundance quantification software which outperforms popular (exact and pseudo-alignment based) quantification approaches in terms of runtime and accuracy, and is widely adopted in standalone, as well as cloud based quantification [91, 92]. There are no other off-the-shelf baselines that are capable of performing RNA-Seq abundance quantification. Kallisto uses a pre-generated colored *de Bruijn* graph (DBG) of k-mers present in the transcripts for similarity class detection, and EM algorithm for quantification. Note that, the graph structure in Kallisto is based on the *order* of the k-mers in transcripts, and the accuracy of quantification depends on storing this *order* information correctly. Therefore,

Kallisto is significantly sensitive to k-mer length and their relative orders, unlike CRAM-Seq. There is no standard GPU baseline for abundance quantification, and alignment accelerators which map reads to (several orders of magnitude) longer references do not qualify as baselines without changing the designs to introduce problem-specific data structures. Since CRAM-Seq accelerates the pseudo-alignment part of quantification, for a fair comparison, we profile only the pseudo-alignment routine in Kallisto with 8 threads on an Intel i9-9900 system. The energy consumption of Kallisto is derived by feeding Kallisto's dynamic instruction mix to an optimistic energy model based on [93, 12].

Evaluation metrics: We characterize the performance of CRAM-Seq in terms of throughput in KSeq(uences)/s(econd) and energy efficiency in Kseq/s/mJ, and normalize to those of Kallisto. The accuracy of abundance estimates is calculated as the absolute difference between the known abundance of the synthetic read datasets and the reported estimates (by CRAM-Seq and Kallisto), and expressed as a percentage of the known abundance.

5.5 Evaluation

5.5.1 Performance Analysis

Fig. 5.16a illustrates the improvement in throughput exhibited by CRAM-Seq, over Kallisto, with 1000 transcripts. The unoptimized column represents the naive design, without any optimization, as explained in Section 5.3. The corresponding improvement in energy efficiency, over Kallisto, is shown in Fig. 5.16b. Even with an unoptimized design, CRAM-Seq outperforms Kallisto on both throughput (1.4×) and energy efficiency ($\sim 54\times$) fronts. The throughput of CRAM-Seq is determined by the PE latency and the class extraction (MAX) where PE operations make up for $\sim 98\%$ of the total latency, as shown in Fig. 5.17a. The VTU and SCU components are hidden (pipelined) and have lower latency in comparison. Fig. 5.17b captures the relative contributions to energy of PE and SCU operations (share of the VTU is negligibly small) consuming $> 99\%$ of total CRAM-Seq energy.

To improve the throughput of CRAM-Seq, we examine optimization opportunities in PE operations since PE operations dominate the latency of the CRAM-Seq pipeline

stages. A breakdown of unoptimized PE latency and energy, illustrated in Fig. 5.18s (a) and (b), respectively, reveals that PE latency is dominated by the *preset* ($\sim$ 50% of total and $\sim$ 1.5$\times$ of compute since 2-output *INV* logic requires 2$\times$ more *preset* per logic operation), an enabler step to perform the actual computations, while the actual computations, e.g., *AND* operations, consume a mere one-third of the total. The presets in the unoptimized design are performed in a sequential manner right before a logic operation, although in multiple columns at the same time. Luckily, *gang-preset*s are possible (where target output cells in multiple rows and multiple columns are preset at the same time) due to the low *preset* energy, coupled with separate read and write paths of SHE-MTJ cells that translate into a peak current draw of $\sim$ 12 mA per tile. This optimization reduces the share of the preset latency in overall PE latency by $>$ 95%, as illustrated in Fig. 5.18c. The corresponding change in throughput is reflected in the optimized column (Fig. 5.16a) that shows $>$ 3$\times$ throughput improvement over the baseline, i.e., a $\sim$ 130% increase in throughput over the unoptimized design. Since the total number of presets (and the corresponding energy) in the optimized design is unchanged, the improvement over unoptimized design in terms of the energy efficiency is similar to the throughput as observed in Fig. 5.16b.

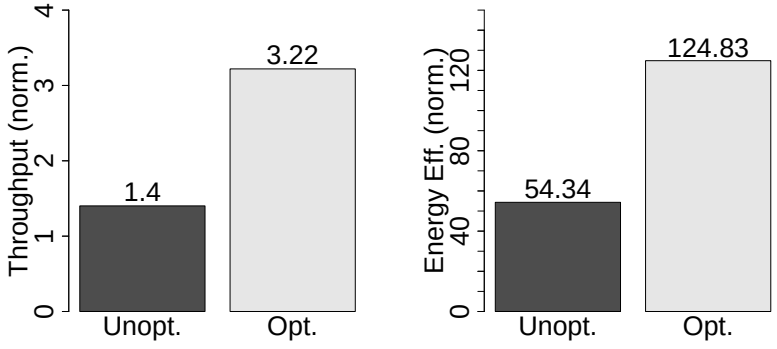


Figure 5.16: CRAM-Seq (a) throughput and (b) energy efficiency.

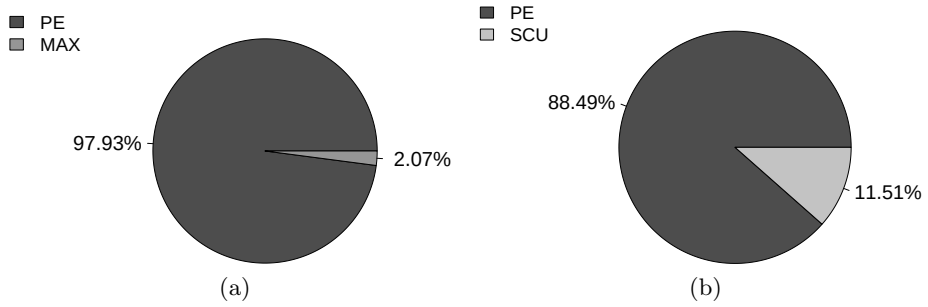


Figure 5.17: CRAM-Seq (a) latency and (b) energy breakdown.

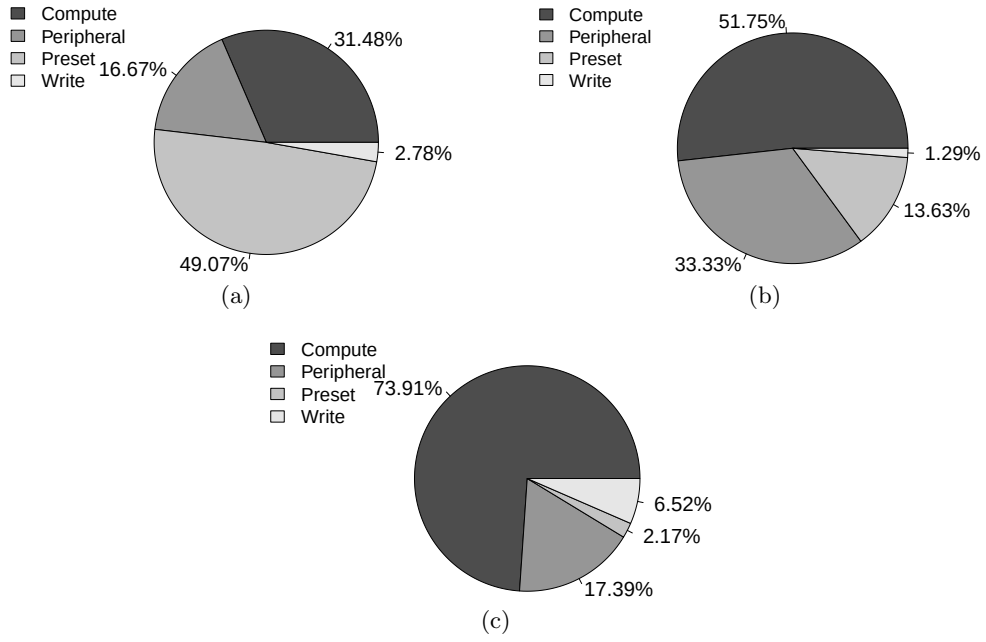


Figure 5.18: (a) Latency and (b) energy breakdown for unoptimized PE, (c) latency breakdown for optimized PE.

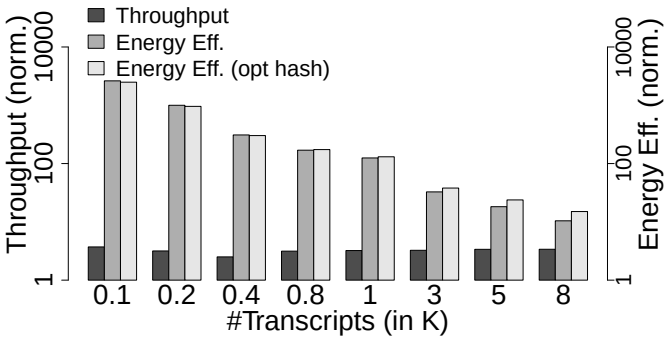


Figure 5.19: Performance scaling of CRAM-Seq.

5.5.2 Scalability

Performance scaling within a single chip: Fig. 5.19 illustrates how CRAM-Seq performance scales with the increasing number of transcripts per CRAM-Seq chip. Throughput remains stable as more transcript segments are processed by (i.e., more PEs are added to) CRAM-Seq. Such weak scaling reduces the energy efficiency improvement which decreases linearly with growing #transcripts. Energy for *class ID* check increases quickly with higher number of transcripts (e.g., $\sim 20\%$ of the total energy with 3000 transcripts) due to the naive one-hot encoding of *class IDs*. Even then, with 3000 transcripts (maximum number of segments that fits on chip) CRAM-Seq still provides $> 30X$ improvement in energy efficiency over the baseline.

The *opt hash* bars in Fig. 5.19 capture the impact of optimized hashing (Algorithm 6). Optimized hashing makes the energy-efficiency scale better (vs. the naive one-hot encoding) due to the lower #bits to encode the *class ID*, which reduces the energy consumption in the SCU. Moreover, optimized hashing itself scales better as the number of transcripts per chip increases. The maximum number of transcripts per chip also improves due to the more compact representation of *class ID*, as we will detail next. That all said, hashing in this context still has room for improvement (e.g., a hashing algorithm that converts a set of transcript indices to a unique value in a fixed 64-bit number space).

Memory scaling: The memory sizes for CRAM-Seq and the baseline with varying number of transcripts are shown in Fig. 5.20. For Kallisto only the memory required

for transcript index file is shown. PEs in CRAM-Seq, which store the transcript segments, require less memory than Kallisto (with default 31-mer) for all datasets, making CRAM-Seq more memory efficient for transcript storage even though CRAM-Seq stores multiple segments for each transcript. The contribution of LUT in system size is negligible. However, the class information storage in SCU increases quickly and with > 3000 transcripts, total CRAM-Seq storage ($\sim$ 60 MB) exceeds the Kallisto transcript index file size due to inefficient encoding of *similarity class IDs*. Optimized hashing (Algorithm 6) reduces memory overhead of SCU significantly, as shown in Fig. 5.21. With optimized hashing more (up to 5000) transcripts (i.e., more PEs) fit on chip, at the same time, SCU energy reduces accordingly. Hence, the total memory requirement still remains less than Kallisto's. This trend holds in a system of multiple CRAM-Seq chips, as well.

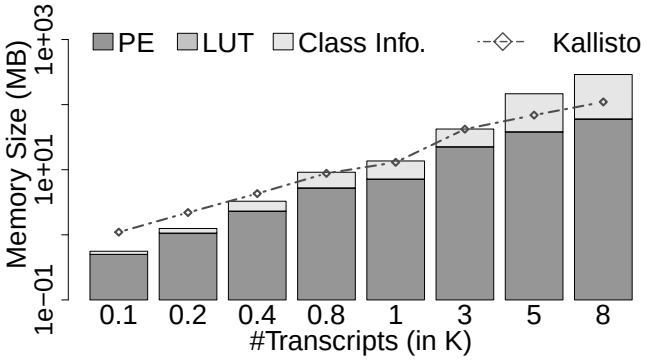


Figure 5.20: Memory required for CRAM-Seq and Kallisto.

Performance scaling across multiple chips: Fig. 5.22 captures the impact of scaling out a system of CRAM-Seq chips, following the methodology from Section 5.3.10, to handle larger transcriptomes. In this case, we assume the same number of PEs in each CRAM-Seq chip (corresponding to the 5K transcript dataset). We experiment with transcriptomes of 10K and 20K. The read scheduler considers 10 reads at a time and tries to schedule one distinct read to each cluster of chips where the most similar partition to the read resides. As Fig. 5.22 suggests, the mean throughput keeps improving with the increasing number of transcripts in the dataset. The theoretical *peak* throughput is limited by a factor equal to #clusters of chips in the system (=3 for 10K and 4 for 20K), whereas the minimum throughput is equal to that of a single CRAM-Seq chip. The

energy efficiency remains stable due to balancing between increasing energy consumption and improvement in throughput. It is possible to further improve the energy efficiency by optimizing partitioning of transcripts (i.e., changing similarity threshold [87])–to minimize the partition sizes and consequently, have smaller cluster of chips to avoid unnecessary computation within a cluster.

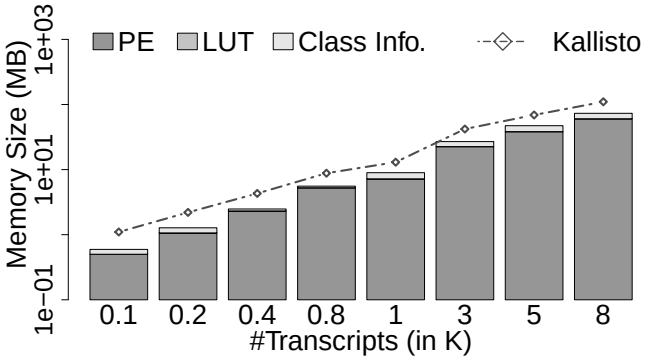


Figure 5.21: Memory required for CRAM-Seq and Kallisto with optimized hash.

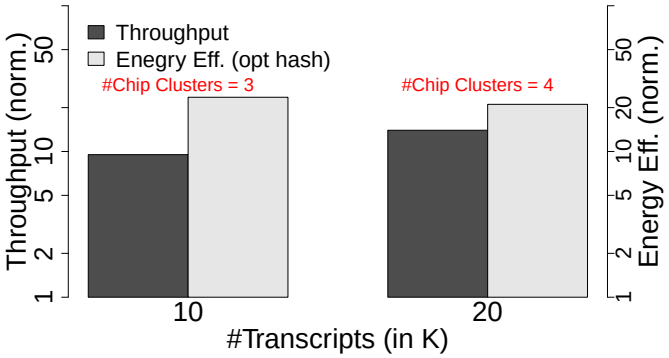


Figure 5.22: Throughput and energy efficiency of a system of CRAM-Seq chips.

5.5.3 Accuracy Analysis

Fig. 5.23 shows the trend in accuracy, i.e., mean % error in quantification for all datasets, up to 3000 transcripts (i.e., the single chip limit with naive SCU encoding scheme). The error reported by both Kallisto and CRAM-Seq remains $< 10\%$ across all datasets. Although CRAM-Seq utilizes smaller k-mers than Kallisto, the mean difference between reported errors is 0.78% (with the maximum reaching 2.61%) across all datasets. The

accuracy metric is based on the *mean absolute relative difference* (MARD) which is not robust against local outliers, leading to the ∼3% loss shown for a few transcript dataset sizes (0.4K, 0.8K, 1K) in Fig. 5.23. Pearson's correlation coefficient [94], a more robust metric against local outliers in capturing correlation between the ground truth and estimated abundances, is 0.9822633935 for CRAM-Seq, and 0.9966648441 for Kallisto (for 1K case). These values are very close to each other, indicating very high correlation in the abundance quantification estimates of CRAM-Seq and Kallisto even for the cases showing some sizable difference in accuracy. The trend in accuracy holds beyond 3000 transcripts, as well. For 5000 transcripts, CRAM-Seq experiences 8.07% of mean error as opposed to 8.11% by Kallisto.

Our experiments show that the accuracy of Kallisto with 5-mers is much worse ($> 90\%$) than CRAM-Seq, attributable to the much smaller (than transcripts) k-mer length used in building the DBG, where segmented transcripts used in CRAM-Seq help improve accuracy.

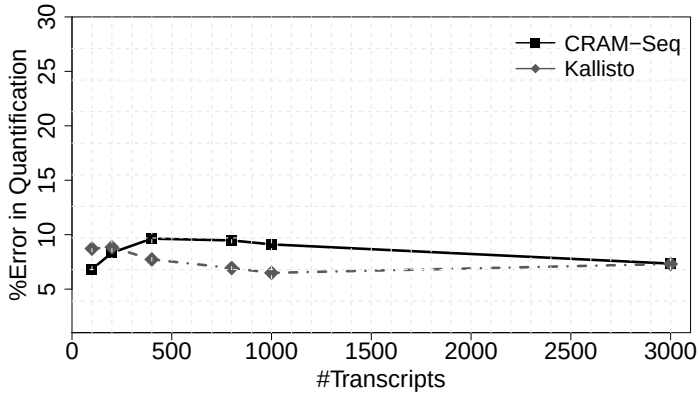


Figure 5.23: Accuracy of quantification.

Fig. 5.24 illustrates the impact of k on the accuracy. For 1000 transcripts and respective read datasets, we vary k and report accuracy. % Error in quantification reported by Kallisto is for the default k (= 31). For 4-mers, the number of distinct k-mer patterns are only 256 (not significantly higher than the RNA-Seq read length assumed), which degrades accuracy and renders a mean error of $> 10\%$, about 2X of that of Kallisto. The accuracy improves significantly as k increases to 5. As explained in Section 5.3.1, 5-mers result in 1024 distinct k-mer patterns which is sufficient to correlate reads with

individual transcript(s). However, increasing k in CRAM-Seq beyond 5 does not improve accuracy by a significant margin, but decreases throughput and increases energy consumption (Table 5.3). The improvement in accuracy becomes increasingly smaller with increasing k due to larger PEs (more CRAM tiles for each PE to accommodate longer bit-vectors resulting from longer k-mers, translating into more computation to derive similarity scores) which also increases the memory footprint of CRAM-Seq.

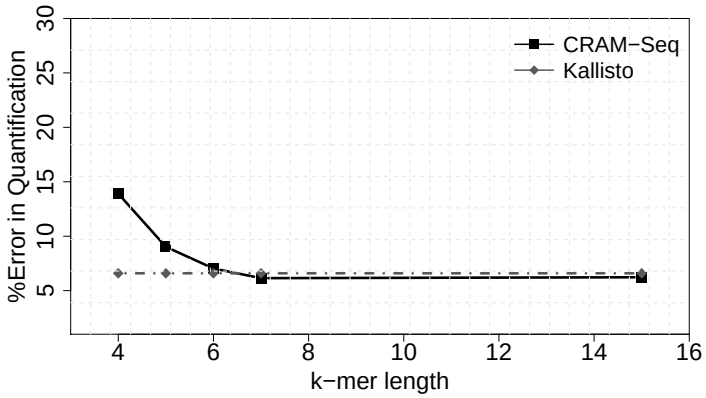


Figure 5.24: Variation in CRAM-Seq accuracy with k-mer length.

Table 5.3: Impact of k on quantification error, throughput, energy efficiency and memory size (relative to k=4). δ in %.

k	δ **Error**	δ **Throughput**	δ **Efficiency**	δ **Memory**
5	−35.42	−6.1	−6.3	+35.8
6	−50.81	−11.4	−12.0	+107.4
7	−56.70	−16.2	−16.9	+179.0
15	−56.72	−41.5	−43.2	+680.1

5.6 Related Work

CRAM-Seq is the first to accelerate RNA-Seq abundance quantification on a PIM substrate. A typical RNA-Seq abundance quantification algorithm entails alignment of each read from a set of (usually millions of) reads to a large number of long transcripts, followed by EM which quantifies abundance using the alignment outcome. CPU based exact alignment is too time consuming. Many GPU [38] and FPGA based implementations [95, 96, 97, 98] to accelerate popular exact sequence alignment algorithms such as Smith-Waterman exist. PIM based exact alignment is also gaining traction due to

potentially higher alignment throughput and lower energy consumption than both GPU and FPGA. Recent designs that use SOT-MRAM [50, 39, 99], ReRAM [49] and conventional technology [100] have reported significant throughput and energy improvement over GPU based implementations. However, exact alignment is over-kill for abundance quantification simply because the exact location of alignment is not required. Therefore, such PIM based exact alignment accelerators would result in sizable latency and energy overhead if employed in the abundance quantification pipeline. Moreover, unlike typical alignment accelerators, abundance quantification involves matching against multiple reference (i.e., transcript) sequences simultaneously as opposed to a very long one. To summarize, PIM-based based alignment accelerators, in principle, can be deployed to build a RNA-Seq quantification accelerator using the design principles presented in this chapter. However, such designs cannot be used off-the-shelf for quantification without design modifications which would warrant separate end-to-end accelerator designs. Therefore, such designs are not considered as baselines in this chapter.

The pseudo-alignment algorithms such as Kallisto [84] and Sailfish [83], on the other hand, rely on *order* or *frequency* of k-mer patterns that are common in both read and transcripts to quantify abundance. Matataki [101], another application that utilizes k-mer based approximation, is highly susceptible to errors in RNA-Seq reads and therefore limited to large-scale reanalysis, unlike CRAM-Seq that is impacted minimally by errors in reads due to the use of small k-mers. While these CPU based software solutions provide significant speedup over exact alignment based quantification, they suffer from significant data movement overhead, which is addressed by CRAM-Seq effectively.

5.7 Conclusion

RNA-Seq abundance quantification is an essential tool in gene expression analysis, useful in many applications such as differential expression analysis to infer biological function at the gene/transcript level. As a compute intensive application (due to the use of millions of RNA-Seq reads), it suffers from significant data movement overheads in classical von-Neumann systems. In this chapter, we propose computational RAM (CRAM) as a PIM substrate to accelerate abundance quantification. The resulting design, CRAM-Seq, performs the pseudo-alignment steps in CRAM. We demonstrate the key design

challenges and opportunities to achieve a significant improvement in throughput and energy efficiency, while maintaining a similar level of accuracy when compared to the state-of-the-art RNA-Seq abundance quantification algorithm Kallisto.

Chapter 6

Content Addressable Memory (CAM)

6.1 Introduction

Content addressable memory (CAM) is an extensively used functional building block in many mainstream computing systems. Rather than locating stored data using addresses (as in conventional random access memory, RAM), CAM finds information using the content itself – hence the name. Specifically, upon getting a search request for a data content, CAM performs the search on all memory locations simultaneously, and returns the location (or index) of match, if any.

On a per bit basis, Binary CAM (BCAM) can search for only two states, i.e., 0 and 1, which Ternary CAM (TCAM) expands to include also a third *don't care* state, i.e., *X*. This wildcard X makes searching for a data content, that partially matches with stored data, possible, and therefore, can generate multiple matches for a single content search request. In either case, the parallel search capability enables CAM structures to perform low latency data lookup which is desired in many contexts, including but not limited to network devices [102, 103], neuromorphic associative memory [104], big-data analytics [105, 106], pattern recognition [107, 108], data compression [109], reconfigurable computing [110] and application-specific acceleration [48].

Emergence of edge computing has further increased the range of applications where

parallel content based search is critical to overall system performance. Examples include object detection [111], neuromemristive circuits and near-sensor binary deep neural networks for edge computing devices [112, 113]. Besides fast CAM search, operation in resource constrained environments (such as wearable devices and Internet-of-Things, IoT) require very low area and energy consumption. At the same time, constrained hardware resources make reconfigurability an increasingly desired feature in these environments [114], to best match dynamically changing computational demand of the workload, specifically, to deliver the optimal performance without any waste in area and/or energy by repurposing hardware resources on demand. However, reconfiguration itself incurs an overhead which can easily become prohibitive considering the extremely tight budgets in area and energy.

Be it based on traditional CMOS or emerging technologies (STT-MTJ [115], PCM [116] or ReRAM [107, 117]), typical CAM designs suffer from either high area overhead or energy consumption (or both). Moreover, none is practically reconfigurable, hence repurposing CAM cells on demand during runtime to perform regular memory or even logic operations is out of question. On the other hand, PIM substrates –already by construction– can perform logic and regular memory operations within the same array with minimal reconfiguration overhead. By exploiting array regularity, adding CAM operations on top would be an attractive solution as long as the reconfiguration overhead can be kept at bay[1]. While various recent PIM proposals target edge-computing systems [118, 119], none explores this opportunity.

In this chapter, we propose a novel reconfigurable architecture, CAMeleon, targeting edge and embedded environments, which seamlessly adds CAM functionality to nonvolatile PIM. CAMeleon supports both BCAM and TCAM operations utilizing *in-place* logic processing capabilities of PIM. The underlying PIM substrate is the nonvolatile (spintronic) Computational RAM (CRAM) [1], which is shown to be a versatile and highly area/energy-efficient platform for resource constrained environments [118], where CAMeleon does not incur any changes to the cell architecture. CAMeleon's

[1]Designs that use CAM to perform very restricted and limited number of logic operations are not considered as PIM enabled CAM architectures in this context.

energy-efficiency comes from –unlike existing CAM proposals regardless of the underlying memory technology– not requiring specially tuned dedicated sense amplifiers to perform CAM operations. Moreover, CAMeleon can switch between PIM and (B/T)CAM operations with minimal reconfiguration overhead. In a nutshell, the contributions of this chapter are as follows:

1. To the best of our knowledge, CAMeleon is a unique reconfigurable architecture fusing generic BCAM/TCAM functionality with PIM (i.e., conventional memory and logic operations).

2. We cover the HW/SW co-design in detail, that enables integration of CAM and PIM functionality with trivial reconfiguration overheads.

3. We show that CAMeleon has comparable search latency to fastest known CAM designs, while providing a higher area/energy-efficiency.

The rest of this chapter is organized as follows: Section 6.2 presents CAM basics, Section 6.3 details HW/SW co-design of CAMeleon, Sections 6.4 and 6.5 provide the evaluation, and Section 6.6 concludes the chapter.

6.2 CAM Architecture

Fig. 6.1 illustrates basic hardware organization for CAM. CAM table comprises a 2-D array of CAM cells, which store key words (typically 32–128 bits) along each row and which are connected to *match lines*. Each CAM cell typically stores both key and inverted key bits. The query to search, stored in the query register and connected to the CAM table through search lines, connects a bit from each CAM cell to the corresponding match lines. The match line indicates whether the data stored in that row is a match for the query input. All match lines are fed to an encoder that determines the match location, i.e., index. TCAM typically requires one additional memory cell for each CAM cell that stores a wildcard bit, i.e., X (= 0/1). In case of BCAM, one match for each query is expected –unlike TCAM where more than one match is possible and therefore, a priority encoder is employed that outputs the match location with highest priority.

Logic gates to support CAM operations: The only logic gates required to perform CAM operations are *NOR* and *AND*. Refer to Table 2.2 for the corresponding truth

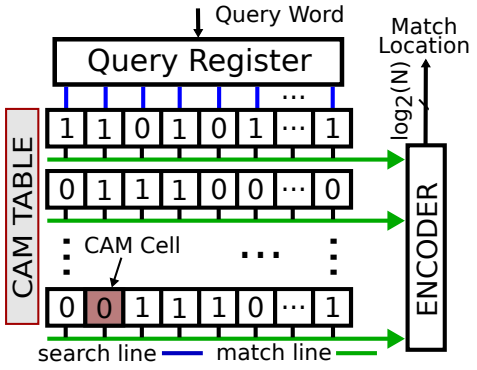


Figure 6.1: Generic CAM architecture.

table. The same biasing conditions implement an *AND* gate with a preset of 1 (and opposite current direction), where *Output* switches from 1 to 0 for all input combinations but 11.

6.3 CAMeleon Architecture

6.3.1 Overview

Fig. 6.2 shows a generic CAM algorithm. A typical CAM handles equal-length key and query words: *Key* words are stored in unique locations inside CAM structure, and *Query* words (from a query pool, one at a time) are used to search those unique memory locations simultaneously, to find the exact (partial) match(es) between the key words and the query in case of BCAM (TCAM). For every bit of a key word, both that bit and the corresponding inverted bit are stored in CAM to form a *bit-pair*. Query words are written to a structure inside CAM array, i.e., the *query register*, one at a time, before firing search. The process repeats for each query word in the pool.

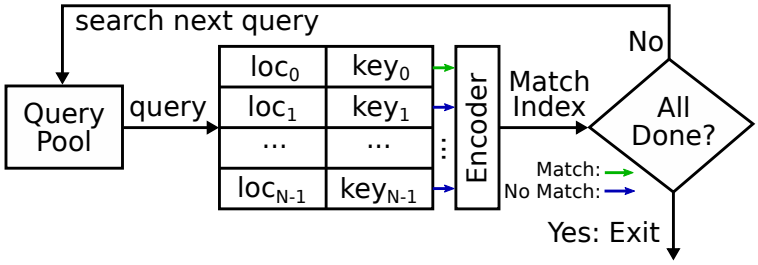


Figure 6.2: Generic CAM algorithm.

6.3.2 CAM Operations in CRAM

Basics: For each bit in a key word, there is a *bit-pair* stored in the corresponding column of the CRAM tile. Specifically, all bit-pairs of a key word go to the same column. Each bit-pair in a key word thereby forms a CAM cell along the CRAM column; multiple CAM cells residing in a CRAM column. As typically one set of key words is searched for many query words, the initial overhead of writing key words to CRAM columns is amortized over many CAM search operations. The *query register* stores a query word that generates necessary signals to perform CAM operations in CRAM. After search is complete, search outcomes are directly stored in CRAM which are subsequently read out and fed to (priority) encoders. The mechanism for CAM search in CRAM is simple: only if a bit in the query and key words match, a CRAM cell storing logic 0 at that bit position (either key or inverted key bit) is connected to the LL along the column. Therefore, by construction, a subsequent *NOR* of all such cells along the column (which keep the search outcomes for the entire key, query pair) generates a 1 to indicate a *match*, iff all bits match.

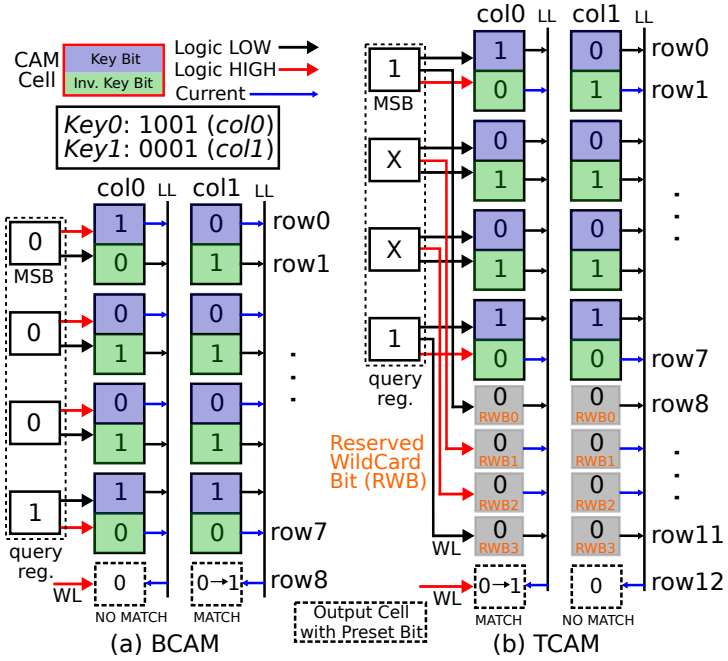


Figure 6.3: (B/T)CAM search in CAMeleon.

BCAM operation: Fig. 6.3(a) illustrates the principle of BCAM search in CRAM

with an example 9×2 tile (where red lines indicate logic HIGH). The two columns (*col0* and *col1*) store *Key0* and *Key1* respectively. The columns are 9-bit long where *row 0* - *row 7*, in each column, store the bit-pairs corresponding to 4-bit key words in adjacent CRAM cells. Each CAM cell spans two rows, hence incorporates two CRAM cells: #CAM cells/column = #bits in key or query word. For example, in Fig. 6.3(a), in each column, the two CRAM cells in *row 0* and *row1* together form a CAM cell. Each bit in the query register selects either of the rows in a CAM cell; if the query bit is 0 (1), it selects the stored (inverted) key bit. The cell in the selected row gets connected to the *LL* in the respective column, to serve as an input to the subsequent *NOR*. In Fig. 6.3(a), in each column, the *output cell* to *NOR* (in row8), connected to a different BSL group than the rest of the cells in the column, is preset to logic 0. On a per column basis, as each query bit selects one cell to connect to *LL*, applying V_{NOR} across the OBSL and EBSL (i.e., across cells storing key or inverted key bits, and the output cell) effectively creates a 4-input *NOR* gate that switches the output cell (from 0 to 1) iff all 4 input cells are logic 0. This marks a match between the query bits and the key bits stored in the respective column. In Fig. 6.3(a), the first query (0001) selects in both columns the cells in rows 0, 2, 4 and 7 (indicated in red), which renders all 0 cells in *col1*, hence, a 1 at *NOR* output to indicate a match. This is not the case for the content of the selected cells in *col0*: LL is connected to all logic 0s but one, which is not enough to generate a current to switch the output cell, hence, no match.

TCAM operation: As indicated in Fig. 6.3(b), the same search mechanism for BCAM from Fig. 6.3(a) applies to TCAM search with two changes: i) a method to search the wildcard bits in query word – stored as a bit-mask in a separate register (i.e., bit-mask register) of equal size to the query register; and ii) additional bits in each column to help in wildcard search – *Reserved Wildcard Bits* (RWB) in *row8-row11*. RWBs are equal to the number of key bits stored in a column (4 in this example). Each RWB corresponds to one unique CAM cell in that column and stores a constant value (logic 0). When a query bit is marked as a wildcard bit (by setting the corresponding bit position in the bit-mask register to logic 1), the corresponding RWB is selected instead of any row of the corresponding CAM cell, effectively bypassing the BCAM match mechanism. When a query bit is not marked as a wildcard bit, on the other hand, the search proceeds exactly in the same way as BCAM search, as depicted in Fig.6.3(a). In this case, as

well, if all cells selected to be connected to a LL are logic 0s, the *output cell* (in *row12*) in the respective column switches – marking a match. In Fig.6.3(b) this is the case for *col0* for the query (*1XX1*) since all 4 cells in *col0* selected to be connected to LL have logic 0 (in rows 1, 7, 9 and 10), which doesn't apply to *col1*.

Multi-gate logic operations: At the core of the search logic lies the *NOR* operation to generate the match outcome, which has as many inputs as the number of bits in the key (or equivalently query) words. While typical key (query) lengths tend to be fixed, CRAM logic gates cannot support arbitrary number of inputs – hence a limit applies to the #inputs of *NOR* along a column. As a workaround, we chunk query word into groups of bits and perform the search with one chunk at a time, sequentially. Each search in this case involves a n-bit *NOR* which is feasible to implement in CRAM where n is the #bits in the chunk. To generate the final match outcome, we feed the output of each such *NOR* gate to an *AND* gate, on a per column basis, which generates a 1 (to indicate a match) iff all *NOR* outputs are 1.

Row selection logic (RSL): Fig. 6.4(a) and (b) show the RSL for BCAM and TCAM, respectively (red = logic HIGH), implemented using conventional gates. Recall that each CAM cell spans two rows in a column. Each query bit in BCAM selects either of the rows in the corresponding CAM cell simply using a NOT gate. In TCAM, if a query bit is marked as a wildcard bit, neither of these rows are selected; instead the row with the corresponding *RWB* is selected. As an example, in Fig. 6.4(b), the first query bit selects *row1* and the last bit in bit-mask register selects the row *RWB3*. The TCAM RSL becomes the equivalent to the BCAM RSL if all bits in the bit-mask register are logic 0. RSL signals drive the rows of all CRAM tiles involved in search (in CAM mode).

During regular (non-CAM) CRAM operations, CRAM tile controllers (responsible for driving rows in each tile) bypass RSL signals for CAM operations– thereby making *all* cells available for CRAM operations.

6.3.3 Hardware Organization

CAMeleon incorporates a collection of CRAM tiles that store the key words – i.e., a sequence of *bit-pairs* corresponding to CAM cells – along columns. All cells along a column that store the bit-pairs (and that represent inputs to the *NOR* operation to determine match outcome) are connected to same BSL group; cells keeping the *NOR*

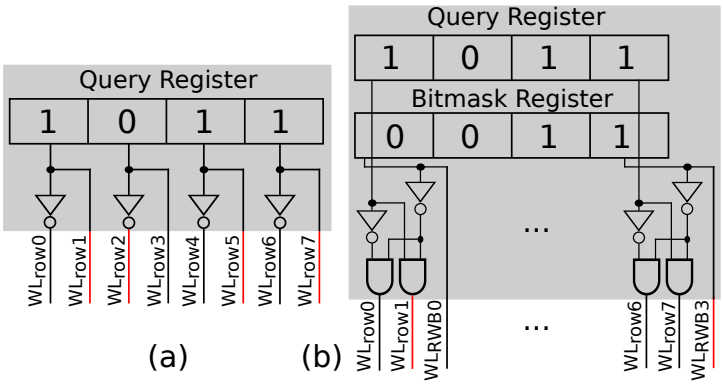


Figure 6.4: Row selection logic for (a) BCAM and (b) TCAM.

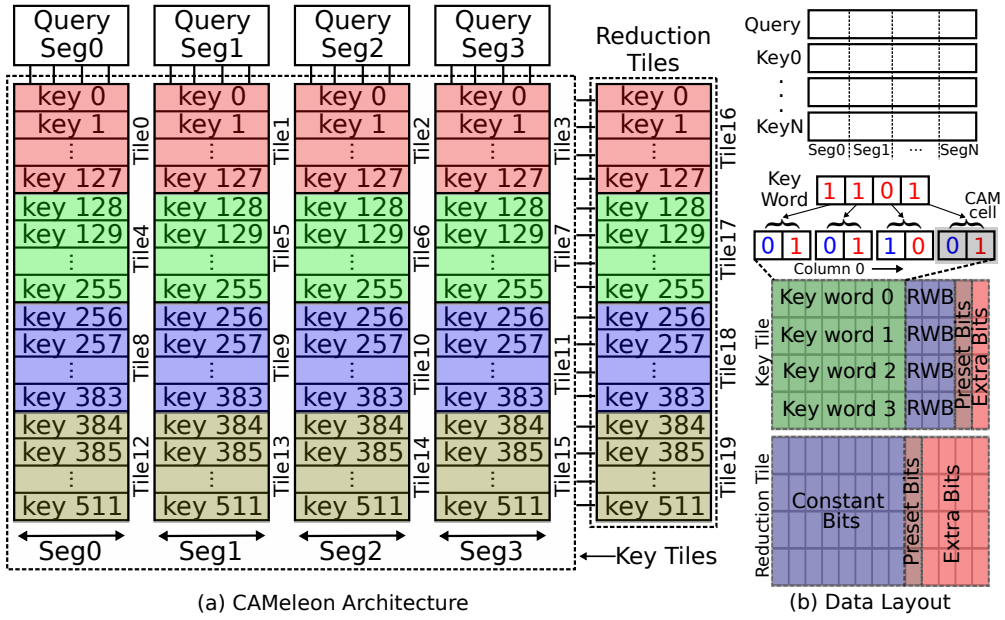


Figure 6.5: Hardware organization of CAMeleon (transposed to simplify illustration).

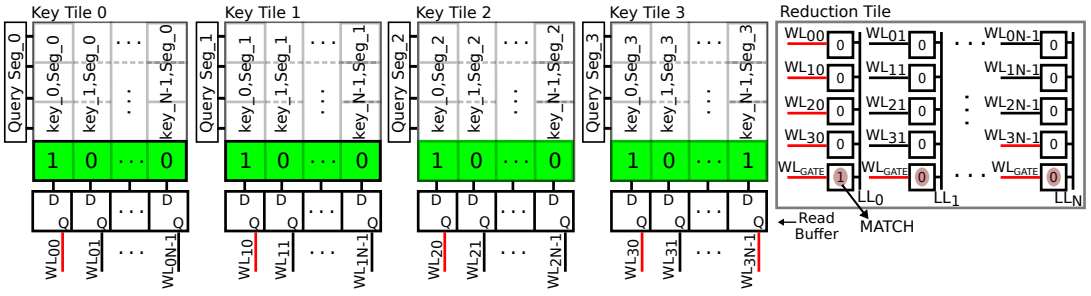


Figure 6.6: Reduction operation in CAMeleon.

output, to the opposite. The query word is stored in the query register that is shared among tiles that perform search with that query.

The high level architecture of CAMeleon is shown in Fig. 6.5(a). The first step is to divide the query and key words into smaller equal length segments (as captured by the top portion in Fig. 6.5(b)). This serves two purposes: i) Storing very long key words (i.e., bit-pairs) in one column of a tile compromises signal integrity across first and last rows, ii) Long query words would require either CRAM logic gates with large #inputs (more prone to process variation of the MTJ, the peripheral circuitry and V_{NOR}); or to perform logic operations in >2 steps – contributing to latency and energy overheads. CAMeleon stores these segments in columns in *key* tiles where CAM search is performed on each of these {Query, Key} segment pairs, in parallel. Since each query bit is matched against the corresponding bit in the key word, such parallel CAM operations are bit-independent across all key tiles.

Fig. 6.5(a) illustrates an example where 512 key words and a query are divided into 4 segments each. Each key word segment is stored in a separate key tile. For example, key tiles 0-3 store segments 0-3 of key words 0-127. Tiles 0, 4, 8 and 12 store segment 0 of keys 0-511 and share the same RSL signals corresponding to the query segment 0.

Search outcomes from individual {Query, Key} segment pairs, i.e., *partial search outcome*s, are reduced to single match/no match indication by the corresponding reduction tile (Fig. 6.5(a)). This is a special purpose CRAM tile where *WL* of individual cells can be independently selected. Similar to the key tiles, *output cells* in the last row of the reduction tile hold the outcome of the reduction operations, i.e., (0)1 for (Mis)match.

Data layout: Fig. 6.5(b) shows the data layout of key and reduction tiles, with rows and columns transposed to simplify illustration. RWBs always store logic 0 when configured for TCAM. *Preset Bits* in each (key/reduction) tile act as *output cells*. The remaining bits in each column are used as *Extra* bits which do not participate in CAM search, rather provide flexibility in scheduling regular CRAM logic operations. Both *Preset* and *Extra* bits are connected to same BSL group, whereas the rest of the bits in each column are connected to the opposite.

Reduction operation: The reduction operation is illustrated in Fig. 6.6. Each N-column key tile stores one segment each from N key words and the corresponding

segment of the query word, e.g., *Key Tile 0* stores *segment 0* from {key 0, key 1, ... key N-1} and performs search for *segment 0* of the query word. The last row in each key tile (green in figure) holds the *partial search outcome* from (B/T)CAM operation. Each key tile has a *read buffer* (RB), i.e., a D-FF array that stores the *partial search outcome* bits (which uses regular CRAM read). The outputs (Q) from RB connect to individual WLs (i.e., individual cells) along a row in the *reduction tile*. For example, the RB outputs from *Key Tile 0* connect to WL_{00}, WL_{10}, and so on, where WL_{XY} refers to the WL of a cell at *row X* and *column Y* in a reduction tile.

Accordingly, depending on the data stored in RBs (1/0), the individual WLs in the corresponding reduction tile get activated (colored *red*). In each reduction tile, all cells along a column with an active WL get connected to LL, by construction, effectively creating a logic gate configuration in that column. Except the last row, all cells in a reduction tile store a fixed 0 throughout CAM search. The last row in each reduction tile, i.e., *reduction output*, is preset to logic 0 before reduction operation begins. *Reduction output* keeps the result of the *NOR* operation where each partial match is represented by a logic 0 at its input. This *NOR* (along all columns of the reduction tile, simultaneously) outputs a logic 1 only if #cells connected to a LL = #{query,key} segment pairs. When #cells (connected to a LL) is less than #{query,key} segment pairs, the required voltage to switch the corresponding output is greater than V_{NOR}, and hence, the output retains the preset (= 0).

The #rows and #columns in a reduction tile depends on the #segment pairs and #keys stored in each key tile connected to it respectively (= $K \times S$ CRAM cells where K = #key words and S = #segment pairs). The discussion on multi-gate logic operations described earlier applies here as well, depending on S and the maximum #inputs required by CAMeleon logic operations.

When not configured to perform CAM search, all cells in a (key or reduction) tile are available for CRAM logic and memory operations. In order to use *all* cells in the key and reduction tiles in regular CRAM operations, WL signals from corresponding tile controllers are *OR*ed with the signals from RSL and RB, respectively. Fig. 6.7 illustrates the idea. In case of CAM operations, RSL (RB) signals are used to drive the rows in a key (reduction) tile; otherwise the WL signals generated by the tile controller take precedence.

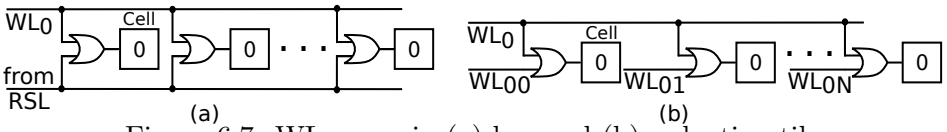


Figure 6.7: WL usage in (a) key and (b) reduction tiles.

Pipelining: Pipelining CAMeleon in order to reduce search latency is straight-forward, where RBs in key tiles can act as pipeline registers. If the latency of key (reduction) tiles is T_{ktiles} (T_{rtiles}), the total search latency of CAMeleon would be $T_{CAMeleon} = T_{ktiles} + T_{rtiles}$. Both T_{ktiles} and T_{rtiles} strongly depend on the number of logic gate operations (during search) the tiles perform. Typically, key and reduction tiles perform similar number and type of operations, so a balanced pipeline is possible, where $T_{CAMeleon} = \max(T_{ktiles}, T_{rtiles})$ applies. In this case, read buffers store the partial outcomes from key tiles, corresponding to a query, which reduction tiles use to derive the final search outcome (i.e., *reduction output*), while key tiles begin with the search of the next query from the query pool.

6.4 Evaluation Setup

We consider CRAM with STT-MTJ for CAMeleon. The performance evaluation of CAM search using CAMeleon, in terms of latency/search and energy/search/bit, is performed with an in-house step accurate simulator, where each step is a logic operation, e.g., NOR. The energy and latency of *NOR* and *AND* gates are derived from the electrical equivalent circuits (see Section 2.2.1). Table 6.1 lists all STT-MTJ parameters used in the evaluation [63, 64, 65]. The low power (LP) and high performance (HP) variants of current MTJs – CLP, CHP, CHPA(ggressive) are considered to capture the sensitivity of CAMeleon performance to device technology parameters. Similar variants are considered for future (projected) MTJ (FLP and FHP), as well. All peripheral overheads (including RSL and query register related), at 22nm technology node, are derived from NVSIM [120] and HSPICE simulations, by accounting for parasitic effects such as wire capacitance.

Dataset: As CAM key word dataset, 1024 128-bit words are randomly generated and each is used to generate 1000 128-bit query words with randomly placed (bit position has no impact on performance or correctness of search output) wildcard bits– a large enough dataset for average search performance characterization. The (default) #wildcard bits

is assumed to be 50% of the query length (= 64). 128 key tiles (64×64) are required to store the key dataset. Each key (and query) is divided into 8 (= 128/16) segments of 16-bit length. For reduction, 16 (64×64) reduction tiles are required (each reduction tile reduces 64 key words). The total memory footprint is $\sim$ 72 KB ($\sim$ 57% is used during CAM operations) – the entirety of which is available to be used as a regular CRAM array.

Logic gate configuration: The (default) #inputs to CRAM *NOR* gates is 8 (corresponding to a good trade-off between overall performance and reliability of logic operations), and each key tile performs two such logic operations in sequence (since 16 key bits are stored in each column of key tiles) before feeding a 2-input CRAM *AND* gate.

Table 6.1: Technology parameters.

Parameter	**CLP**	**CHP**	**CHPA**	**FLP**	**FHP**
MTJ Type	Interfacial PMTJ				
MTJ Diameter (nm)	45			10	
TMR (%)	133			500	
RA Product ($\Omega\mu m^2$)	5			1	
I_{crit} (μA)	40	90	180	0.79	10
Switch. Latency (ns)	3	1	0.3	1	0.3
R_P, R_P, $R_{Trans.}$ ($K\Omega$)	3.15, 7.34, 1			12.7, 76.39, 1	

Baselines for comparison: Since CAMeleon is unique in the sense that it can switch between CAM, and generic PIM modes, there is no appropriate baselines for evaluation. However, to quantify the performance improvement of CAMeleon in TCAM mode, 8 state-of-the-art baseline TCAM designs are selected (shown in Table 6.2), with different device technologies. The numbers reported for the baselines and CAMeleon exclude encoder overhead at the output.

Table 6.2: Baselines for comparison.

Baseline	SRAM	PCM	STT-MTJ[1]	STT-MTJ[2]	STT-MTJ[3]	STT-MTJ[4]	ReRAM[1]	ReRAM[2]
Reference	[3]	[121]	[122]	[123]	[124]	[125]	[126]	[107]

6.5 Evaluation

6.5.1 Performance Analysis

Fig. 6.8 provides the energy/search/bit and latency/search characterization (normalized to [3]; the lower the better). Overall, CAMeleon with CLP consumes less energy

than STT-MTJ- [123], ReRAM- [126, 107] and PCM-based [121] designs. With future (projected) MTJ-variants (FLP and FHP), the energy consumption reduces even further and CAMeleon outperforms all baselines. The baselines with STT-MTJ, ReRAM and PCM use the memory devices to only store CAM data, unlike CAMeleon which also performs computation (i.e., CAM search) with the memory devices – making CAMeleon more sensitive to device technology parameters.

On the latency front, performance of CAMeleon is dominated by the switching latency of the STT-MTJ devices. Due to longer switching latency of CLP, CAMeleon-CLP suffers from the longest search latency across the board. As MTJ variants (CHP, CHPA, FLP and FHP) exhibit increasingly lower latency, CAMeleon recovers latency significantly, e.g., by a decrease of 8.1× from CLP to CHPA. Although the baselines outperform CAMeleon in terms of search latency, it comes with a significant energy (e.g., [123] consumes 5.5× more energy than CAMeleon-CLP) and area penalty (e.g., [124] uses 5× more transistors/cell than CAMeleon). Table 6.3 compares all baselines and CAMeleon in terms of area overhead (#Transistors/cell). The SRAM-based baseline [3], while consuming less energy than most baselines, suffers from a high area overhead (16T/cell) – making it difficult to fit in a tight area budget imposed by embedded/edge hardware. CAMeleon, on the other hand, has smaller area footprint than most baselines, except for [121], [126] and [107] which have similar or slightly smaller footprint at the expense of higher energy consumption, e.g., [107] consumes 25.8× more energy than CAMeleon-CLP. Considering the finely tuned dedicated sense amplifiers for CAM search – required by all these baselines (in addition to read sense amplifiers), CAMeleon is even more area efficient.

In summary, CAMeleon outperforms a wide-range of baselines, in terms of area or energy (or both), while maintaining a comparable search latency. CAMeleon, in BCAM mode, exhibits similar energy (∼ 0.1% less than corresponding TCAM numbers) on average.

Table 6.3: CAMeleon TCAM cell comparison against baselines.

Parameter	[3]	[121]	[122]	[123]	[124]	[125]	[126]	[107]	CAMeleon
Device	SRAM	PCM	STT-MTJ	STT-MTJ	STT-MTJ	STT-MTJ	ReRAM	ReRAM	STT-MTJ
Tech. node (nm)	40	22	40	40	40	22	14	45	22
Cell	16T	3T-3R	10T-4M	9T-2M	15T-4M	6T-2M	3T-1R	2T-2R	3T-3M
Word Length (bits)	144	128	144	144	144	256	128	8	128
Logic-capable	No								Yes

Figure 6.8: Energy and latency comparison (normalized to SRAM [3]).

6.5.2 Sensitivity Analysis

TCAM energy consumption is sensitive to the #wildcard bits in the search query. Fig. 6.9 captures the relationship between energy/search/bit and % of Wildcard bits in varying lengths of query word (normalized to 25% wildcard share). The energy consumption decreases, although insignificantly ($\sim$ 1%), with increasing #Wildcard bits in query. More wildcard bits tend to yield more matches between {query, key} segment pairs– resulting in lower energy consumption due to *AND* gates with all logic 1 inputs, which doesn't incur switching.

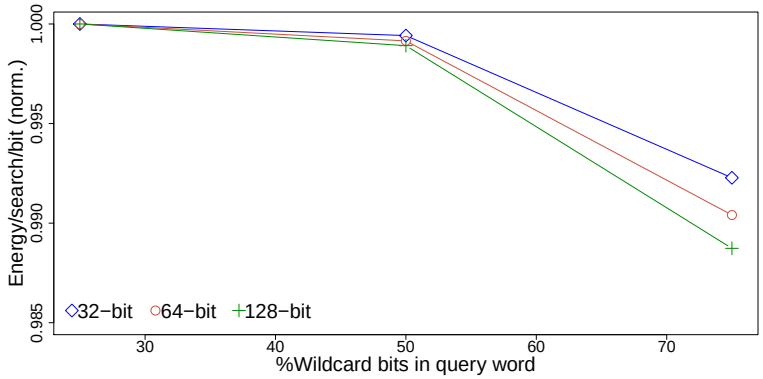


Figure 6.9: Sensitivity of CAMeleon to #wildcard bits.

Query length: Energy consumption in CAMeleon depends on the query length, as well, although insignificantly. Table 6.4 lists the energy consumption in CAMeleon when the query length is varied between 32 and 128 bits (normalized to 128-bit). The energy consumption (per search per bit) tends to decrease as query length increases – indicating good scalability. This is because the dominant search energy component (gate energy; $>$ 60% of total) does not scale in proportion to the query length, e.g.,

gate energy with 64-bit query is $\sim 1.72\times$ of that with 32-bit query – resulting in lower energy/search/bit (vs. 32-bit) for 64-bit query.

Table 6.4: Sensitivity to query length.

Query length (#bits)	32	64	128
Energy/search/bit (norm.)	1.29	1.11	1.00

Process Variation: The reliability of CAMeleon operations depend on the correct switching events in CRAM logic operations. High #inputs to logic operations could exhibit switching for incorrect input data. To understand the impact of process variation on CAMeleon functionality, we considered variation in STT-MTJ and V_{NOR}. For MTJ low resistance (R_P), we assume a σ of 10% with 0.1% variation of TMR (which captures the variability in oxide thickness and surface area), and a 5% standard deviation for V_{NOR}. Our Monte Carlo analysis for an 8-input *NOR* gate, with 10^8 iterations, shows correct switching behavior $\sim 100\%$ of the time even under our conservative assumptions. We also introduced incorrect switching behaviour in this 8-bit *NOR* gate to output a logic 1 when 7 (instead of all 8) inputs are logic 0 (with default query and key configurations). Since, in order to get an incorrect match between a {query,key} pair, all corresponding segments have to yield erroneous match through such a (faulty) gate (which is very unlikely), there was no erroneous match in CAM output for the queries.
Gate Width: Higher #inputs results in lower overall CAMeleon latency (more query bits are searched with each logic operation) and lower energy (which decreases quadratically with #inputs), however, with increasing probability of incorrect switching behavior, i.e., error in CAM search output. For example, with 16-input *NOR* gate, the latency and energy consumption of CAMeleon reduce by $1.95\times$ and $3\times$, respectively, relative to the 8-input gate based design. Such a rich trade-off space is attractive for approximate CAM search, which we leave to future work.

6.6 Conclusion

The constrained execution environment in edge and embedded computing domains, where CAM represents an ubiquitous functional block, requires low-overhead reconfigurability to re-purpose hardware resources, in order to stay within very tight area and

energy budgets. In this chapter we present CAMeleon, a unique reconfigurable hardware solution which fuses spintronic PIM and (B/T)CAM functionality in a seamless and effective fashion. We show that CAMeleon can outperform a wide-range of CAM baselines, in terms of area or energy consumption (or both), while maintaining comparable search latency – and unlike any of the baselines, while also supporting PIM functionality.

Chapter 7

Gate-Flip Errors in PIM

7.1 Introduction

Conventional von Neumann machines make a distinction between compute and memory elements. Considering performance requirements of emerging data-intensive applications, however, the data communication overhead between compute and memory elements has long become forbidding. Blurring the physical distinction between compute and memory elements, Processing-in-memory (PIM) has proven itself as an attractive solution to this drawback. Especially promising are *true* PIM substrates which directly use memory elements for computation, obviating any need for data to leave the memory array [19, 23, 25, 2, 15, 71]. These architectures typically support extremely energy efficient bit-wise logic operations which can run in parallel across all columns or rows of the memory array, and as a result, can boost the performance of numerous emerging applications including machine learning [21, 127, 128] and genomics [39, 99]. However, to unlock this potential, functional reliability is a must.

Regardless of the specific architecture or underlying memory technology, PIM functionality heavily relies on the memory devices for both data storage and computation. Therefore, any non-ideality of memory devices (including but not limited to parametric variation) directly affects the reliability of PIM functions. Moreover, most promising PIM designs are compatible with traditional (CMOS) logic to incorporate conventional circuitry for control at memory cell and array granularity, hence, are subject to traditional sources of (CMOS) variation, as well.

In this chapter we present and quantitatively characterize *gate-flips*, an acute class of PIM specific functional errors caused by parametric variation, where the logic function of a gate incorrectly mimics the logic function of another. We investigate causes of gate-flips in the form of low-level parametric variations along with propagation of gate-flip errors to the end result of computation and the impact on accuracy. While PIM systems inherit variations (and errors) from the underlying memory technology by construction, no traditional error model (typically covering memory or computation in isolation) can explain gate-flips by itself, as physics of gate-flips depends on how the PIM substrate uses memory devices for computation – usually adopting unconventional techniques.

Without loss of generality, we use the spintronic Computational RAM (CRAM) [1] as a case-study which is a reconfigurable, highly energy efficient parallel PIM substrate used to accelerate a wide-range of applications [118, 99, 61]. CRAM follows *true* PIM semantics, in that all computation happens within memory arrays, without relying on dedicated logic or sense amplifiers at the array periphery. This also makes CRAM especially prone to gate-flips, as would be the case for any true PIM architecture. On the other hand, PIM architectures performing computation at the array periphery [26, 2, 16, 15, 129] usually utilize specially designed CMOS circuitry (or dedicated CMOS logic blocks), and therefore are predominantly subject to CMOS process variations when it comes to gate-flips. In true PIM substrates like CRAM, functional reliability depends more on the underlying memory technology and the specific mechanism to perform computation using memory cells, while the CMOS circuitry used for control still has an impact.

The contributions of this chapter are as follows:

1. We present and quantitatively characterize *gate-flips*, a critical class of errors affecting functional reliability of PIM systems.

2. We investigate underlying causes of gate-flips in the form of device-level parametric variations.

3. We analyze the propagation of gate-flip errors to the end results of computation and quantify the degradation in computational accuracy.

This chapter is organized as follows: in Section 7.2, we present the types of low-level parametric variations to consider. Section 7.3 introduces gate-flips and links them to

parametric variations and functional reliability. Sections 7.4 and 7.5 cover the evaluation methodology and results. Section 7.6 discusses applying this error model to other PIM architectures, and Section 7.7 concludes the chapter.

7.2 Background

In a nutshell, logic operations in CRAM are, essentially, write operations where the input data pattern determines whether a write at the output should take place or not, according to the underlying truth table.

7.2.1 Variations and Error Modes in CRAM

Write errors: Write operation in MTJs is a stochastic process and depends on a number of factors, including the write pulse duration, thermal stability factor[1] and the write current [131]. The performance of MTJ write operation, for a given set of parameters, is typically captured by the non-switching probability, i.e., write error rate *WER*. A WER of $\leq 10^{-4}$ can be achieved with a write latency of as low as 3 ns [132, 133, 134], and experimental demonstrations for a WER of $\leq 10^{-6}$ also exist [135]. With a write current sufficiently higher than the threshold (I_{crit}) and a long enough write pulse, WER can be lowered to $< 10^{-8}$ [136]. WER as low as 10^{-11} has been reported [137] recently.

Stuck-at-faults: Stuck-at-faults usually emerge when the MTJ fails to change the stored value to the write value during a write operation and can manifest in two major ways: (i) Irreversible damage where the MTJ is stuck at one particular resistance level; and (ii) the MTJ fails to write a particular value. The first case can be due to either (a) oxide layer breakdown or (b) PVT variability (e.g., MTJ device defects or CMOS access transistor defects). Most (∼60%–70%) such faults are single isolated faults in VLSI chips [138].

Variations in critical current and write latency: Due to process variations, MTJs in a CRAM array can exhibit varying critical current (I_{crit}) and write latency. Typically, critical current follows a normal; write latency, a skewed normal distribution [139,

[1] a measure to quantify reliable retention of data in magnetic storage [130]

140]. Variations in critical current and write latency can have grave repercussions for functional reliability, hence computational accuracy.

Retention failure: Retention refers to the non-volatile nature of STT-MTJ, i.e., data retention between successive write operations. Failure to meet the retention requirement (i.e., retention failure) results in loss of data *between* memory/logic operations. Typically, a longer retention period corresponds to a higher write current through MTJ [141].

Read disturbance: Read disturbance refers to inadvertently altering the data stored in MTJ, during read operations, due to asymmetric scaling of MTJ write current and sense amplifier circuitry.

TMR variations: The variability of oxide barrier thickness in STT-MTJ leads to variation in the MTJ resistance [130], and therefore, in the tunnel magneto-resistance (TMR) ratio defined as $= (R_H - R_L)/R_L$.

Variations in V_{gate}: Process variation of the CMOS transistors in the drive circuitry may lead to variations in V_{gate}, which directly affects the write current during logic operations. Such variations can be captured by a skewed normal distribution, as well [142]. Variations in transistors dominate MTJ variations for small transistors, whereas the opposite becomes true as the transistor size increases.

7.3 Gate-Flips in Processing-In-Memory

7.3.1 Overview

Gate-flip refers to functional flipping of one logic gate to another. Gate-flips may cover the corresponding truth table entries fully or partially. Fig. 7.1 illustrates an example gate-flip between a 2-input *AND* and a 2-input *OR* gate. The table shows the 4 input patterns (00, 01, 10, and 11) along with the *Preset* and expected (correct, *Final*) output value. The preset value 1 applies to both *OR* and *AND*. After $V_{gate} = V_{AND}$ is applied, the output of *AND* should switch ($1 \rightarrow 0$) for all input patterns (i.e., current $> I_{crit}$), except for the input pattern 11. On the other hand, for *OR*, after application of $V_{gate} = V_{OR}$, the output should switch ($1 \rightarrow 0$) for input pattern 00 only. Hence, under correct operation, for input patterns 00 and 11 the switching patterns of these gates are the same. The differences in the switching patterns come from input patterns 01 and 10, and consequently, if these switching patterns, for any of these gates, mimic that of the

other gate, a gate-flip would be the case. We will refer to these input patterns as *flip patterns*. Only the output switching patterns corresponding to flip patterns can cause a gate-flip between a pair of different logic gates. Therefore, we can identify flip patterns specific to pairs of logic gates.

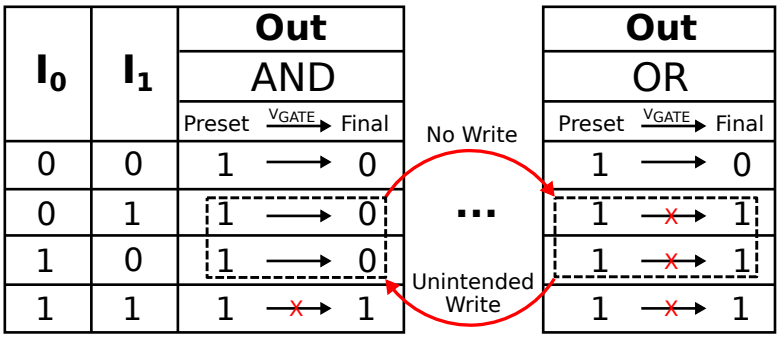


Figure 7.1: Gate-flip between logic *AND* and *OR* in CRAM.

If any of the flip patterns in *AND* result in a non-switching event instead of a switching one, i.e., a write error, the output would correspond to that of *OR*. Similarly, if the output of *OR* switches incorrectly for any of these flip patterns, the output would be same as that of *AND*. Although in this example we excluded corrupted presets, presets (essentially being write operations) are also susceptible to erroneous write events and can give rise to gate-flips by themselves. For example, if the preset (=1) operation of *AND* fails, and the output cell's last stored value was 0, there cannot be any switching, simply because the current direction through output MTJ for a $1 \rightarrow 0$ cannot achieve a $0 \rightarrow 1$ switching. On the other hand, if the last stored value was 1, the preset error would be masked.

Gate pairs with higher number of inputs are equally susceptible to gate-flips. Table 7.1 shows potential gate-flips between *MAJ(ority)3* and 3-input *AND* and *OR* gates. The input patterns that have the same expected output switching pattern between {MAJ3,AND} and {MAJ3,OR} are marked by *X*. The rest of the input patterns can cause a gate-flip: For *AND*, a gate-flip would be the case if there is no $1 \rightarrow 0$ transition, whereas for *OR*, it would be undesired $1 \rightarrow 0$ transition.

Gate-flips also apply to gate pairs with different number of inputs: For example, a 3-input *MAJ3* and 2-input *AND* gate, assuming that one of the inputs of *MAJ3* experiences a suck-at fault. We can identify the flip patterns in this case, following the same methodology in the previous examples, by (i) assuming that one input of *MAJ3*

Table 7.1: Flip patterns for MAJ3 (preset=1) to *AND* and *OR*.

I_0	I_1	I_2	**MAJ3**	**AND**	**OR**
0	0	0	0	X	X
0	0	1	0	X	$1 \rightarrow 1$
0	1	0	0	X	$1 \rightarrow 1$
0	1	1	1	$1 \rightarrow 0$	X
1	0	0	0	X	$1 \rightarrow 1$
1	0	1	1	$1 \rightarrow 0$	X
1	1	0	1	$1 \rightarrow 0$	X
1	1	1	1	X	X

is stuck at a value (0/1); and (ii) treating the remaining *MAJ3* inputs as inputs to a 2-input logic gate. We repeat (i)-(ii) for each input of *MAJ3* and finally, take the union of all of the flip patterns identified this way. Not surprisingly, the resulting flip patterns for {MAJ3,2-input AND} and {MAJ3,2-input OR} pairs are identical to the ones shown in Table 7.1.

7.3.2 Gate-Flips and Low-Level Error Modes

Gate-flip is a high-level error model that captures the impact of ambiguous behavior in PIM logic gates – in terms of non-ideal switching events at the output of the logic gates in the context of CRAM. Intuitively, there is a deeply-rooted relationship between a gate-flip error and device-level errors in MTJ. To understand this, let's look back at Fig. 7.1. The *no write* event, that causes *AND* to mimic the switching patterns of *OR*, can be caused by a number of low-level errors in STT-MTJ. Probabilistic write errors (as quantified by WER) can introduce a *no write* event randomly and result in $AND \rightarrow OR$ gate-flip. Also, if the write (i.e., drive) current is less than the critical current, due to a variation in either the critical current (due to process variation in MTJ) and/or the drive current (due to fluctuations in gate voltage V_{gate}), a *no write* event could occur. Similarly, an $OR \rightarrow AND$ gate-flip error happens if there are unintended writes ($1 \rightarrow 0$) for the flip patterns. Such unexpected writes can be the result of variation in TMR ratio (i.e., actual resistance of the output cell is lower than assumed) and/or the drive current.

Potentially a permanent stuck-at-1 fault at the output could also affect an $AND \rightarrow OR$ gate-flip, by keeping the output state constant at 1 during logic operations. Symmetrically, a stuck-at-0 fault at *OR* output could, effectively, cause an $OR \rightarrow AND$

gate-flip error. However, permanent stuck-at-faults can be characterized experimentally and avoided during mapping of an application to the PIM substrate [143]. Therefore we exclude them as a potential source of gate-flips. Temporary stuck-at-faults, on the other hand, manifest themselves as write errors and are captured by write errors in our analysis.

Computations in CRAM are concerned about stored data only as inputs to logic operations. Retention failure and read disturbance affect the states of the potential (gate) inputs between successive writes and during read operations, respectively. Errors in gate outputs (rather than the inputs), however, give rise to gate-flips.

7.3.3 Putting It All Together: Gate-Flip Matrix

We can define potential gate-flips for each gate using the method of exclusion. If Z is the set of all logic gates, then a gate in Z can flip to any gate in Z, except to itself, provided that the gate pair has the following properties:

1. Same preset: The gates that do not have the same preset, e.g., AND (preset = 1) and NAND (preset = 0), have a V_{gate} of opposite polarity, i.e., direction of current flow through output cell is different. This difference in preset makes a gate-flip harder, if not impossible.

2. Flipping gate should have a higher number of inputs: The number of inputs of a gate dictates the probable gates that are candidates for a gate-flip and whether gate-flips would be symmetric. For example, a 2-input AND gate can potentially flip to a 2-input OR gate, and the reverse is also true given both logic gates have the same number of inputs. On the other hand, for a 3-input MAJ(ority) gate and a 2-input AND, the gate-flip becomes uni-directional (i.e., $MAJ3 \rightarrow AND$).

Using these rules, the *flip-matrix* in Fig. 7.2 captures probable gate-flips between each pair in a representative (and universal) set of commonly used CRAM gates. The logic gates with both preset domains are considered, e.g., MAJ3 (preset = 1) and MAJ3B (preset = 0). The matrix reveals all pairs of gates that are vulnerable to gate-flips (along with the reason for the pairs which are not susceptible to gate-flips). The majority of the gate pairs are either protected by preset (i.e., require different presets) or difference in input counts.

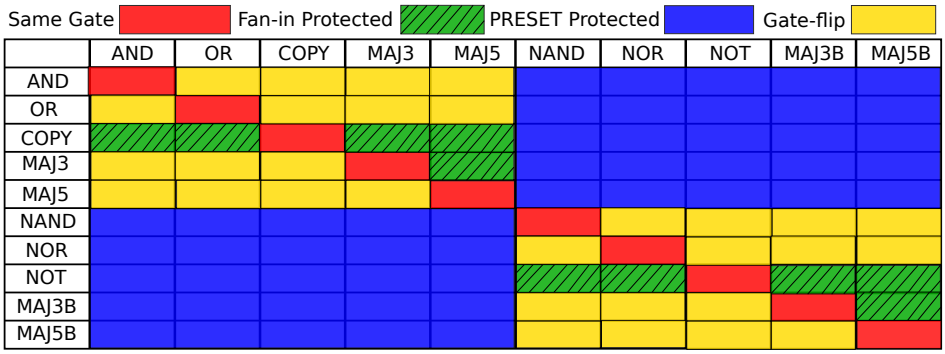


Figure 7.2: Gate-flip matrix.

7.3.4 Impact on Functional Reliability

In the worst case, the error at the output upon a gate-flip would propagate to the subsequent logic operations and reach all the way to the application output to generate an incorrect result. Along the way, masking is always possible, as, e.g., could be the case under preset errors. Masking may work in our favor to preserve the computational accuracy. Depending on the application, and the extent of algorithmic noise tolerance and masking, the impact of gate-flip errors may or may not be tolerable. Hence, functional reliability analysis represents a key step in mapping applications to PIM substrates.

7.4 Evaluation Setup

The error model is implemented in high-level languages (C++ and Python) and integrated into the selected benchmark applications– in order to capture the error exhibited by a CRAM tile(s), at the granularity of each CRAM cell. The number of tiles and tile dimension are selected based on the specific application and the corresponding inputs. We consider CRAM with STT-MTJ here. The STT-MTJ parameters assumed are same as [118]. Two main write events: no write (i.e., write fail) and erroneous write (i.e., unintended write), responsible for causing gate-flip from a high-level point of view, are captured using two respective error rates, R_{No_Write} and R_{Err_Write}. Subset of low-level errors would converge into one of these two write error events–exhibiting a gate-flip, in effect. Both error rates are swept simultaneously to observe the impact on applications' reliability. Only the gate-flips between equal fan-in gates are considered, following the gate-flip matrix.

Table 7.2: Benchmarks for evaluation.

Application	Dataset	Input
Support Vector Machine (SVM)	MNIST	Support vectors: 51

7.4.1 Benchmark Applications

As a representative benchmark, Support vector machine (SVM) is selected. Table 7.2 lists the benchmark details. This application, from a gate-flip perspective, covers a wide range of gate-flips, hence is interesting for this study.

Support Vector Machine (SVM): We use SVM for inference only, with MNIST [144] (gray scale 28×28 image/digit recognition dataset). The dataset has 10 classes for digits 0 through 9. We use a binarized version ($> threshold$ is set to 1; 0 otherwise) of the dataset, with a polynomial kernel of degree = 2. After training is done on host machine, during inference, the support vectors are mapped to separate columns of a CRAM tile. During inference, dot product between an input vector (i.e., test data) and all support vectors are performed, followed by squaring of the products and multiplying by the co-efficients, before adding them together. The dot product operations in binarized SVM are effectively bitwise *AND* operations. Therefore, the logic gates susceptible to gate-flip in SVM inference are AND, MAJ5 and MAJ3.

Data Layout: The corresponding high-level data layout of SVM is illustrated in Fig. 7.3. Each column in a CRAM array holds a support vector, the corresponding label and co-efficient– derived from the training of SVM with training data. Since inference involves computing kernel function with all support vectors (and corresponding label and co-efficient) for each test data (i.e., image), the same input test data ($= z$) is copied on all columns; as a result, the inference for a test data progresses on all columns at the same time, leveraging the column parallelism in CRAM.

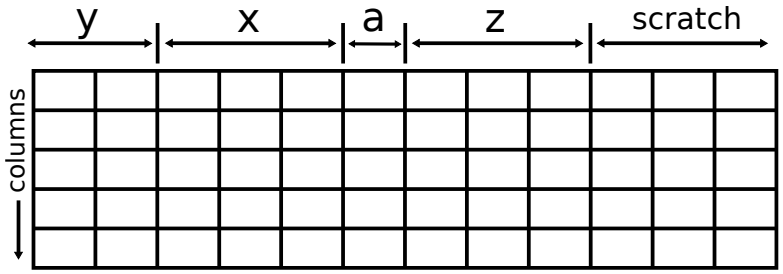


Figure 7.3: Data layout in CRAM for SVM inference (y: label, x: support vector, a: co-efficient, z: input data).

7.5 Evaluation

7.5.1 Impact on Accuracy

The accuracy of SVM inference, with the test images from MNIST dataset, is shown in Fig. 7.4. As R_{No_Write} is reduced (with R_{Err_Write} constant), the general trend shows that, for most cases, the accuracy becomes closer to that with no error ($R_{No\_Write} = R_{Err\_Write} = 0$). Also, the maximum accuracy achieved by varying R_{No_Write} is dependent on R_{Err_Write}; the lower the R_{Err_Write}, the higher the maximum accuracy. All of these observations fall in line with the intuitive understanding of how CRAM operates– more error (no write or erroneous write) contributes to the downgrading of accuracy values. Having said that, even with the pessimistic assumption of $R_{No\_Write} = R_{Err\_Write} = 10^{-5}$, the accuracy is still $\geq 60\%$ (actual error rates in STT-MTJ are much lower than this, refer to section 7.2). For $R_{No\_Write} < 10^{-5}$, the maximum accuracy (for most cases) is $\sim 85\%$, that reaches $\sim 97\%$ for $R_{No\_Write} = 10^{-6}$. Overall, SVM inference have resilience against the errors to maintain high accuracy output.

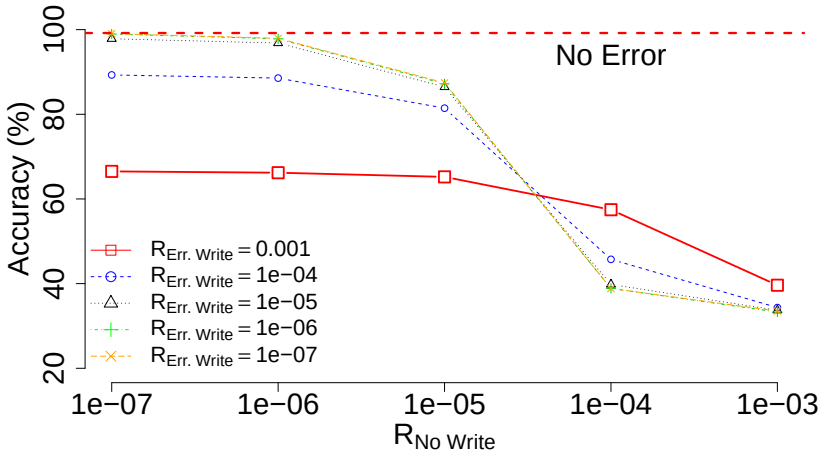


Figure 7.4: SVM inference accuracy with both *no write* and *erroneous write* events.

7.5.2 Gate-flip Statistics

The corresponding gate-flip statistics are captured in Fig. 7.5, where the #flips are presented as a % of total errors observed (i.e., not all errors cause gate-flip). The bars

are grouped by R_{No_Write}. As the trends indicate, for a given R_{No_Write}, the % of *no flips* (i.e., errors that do not correspond to flip-patterns) improves with lower R_{Err_Write}. Also, gate-flip to *AND* (particularly from *MAJ5*) decreases as R_{Err_Write} is decreased– since it is *less* likely to have a flip to *AND* with lower R_{Err_Write}, across the board for different R_{No_Write}. Since SVM inference involves a high number of 1-bit additions (used in dot-product computation, and multiplications), and *MAJ5* and *MAJ3* refer to *sum* and *carry* outputs respectively, the accuracy loss is maximum when there are significant amount of $MAJ5 \rightarrow AND/OR$ flips (refer to Fig. 7.4). Accordingly, *MAJ3* flips are less likely to result in severe accuracy loss, as also captured by relatively lower (than *MAJ5*) shares of *MAJ3* flips at higher error rates, i.e., high accuracy loss.

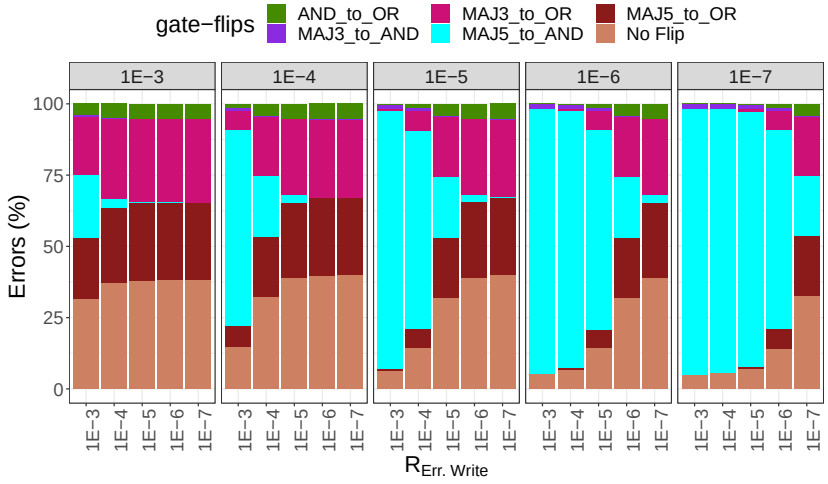


Figure 7.5: Gate-flip statistics for SVM (grouped by R_{No_Write}).

7.6 Discussion

Given that gate-flip is a high-level error model that captures the errors introduced by unexpected behavior of the logic operations, this model is easily transferable to any PIM architecture–regardless of the actual underlying mechanism at play. For example, gate-flip errors in a PIM architecture that utilize SA (sense amplifier) to perform bit-line computing, i.e., logic, can be attributed to different low-level parametric variations, e.g., SA threshold (that decides the output of a logic gate), device-to-device variability, cycle-to-cycle variations in resistance [145] etc. As long as we are able to interpret

the deviations from expected logic behaviour– in terms of the high-level events (e.g., R_{No_Write}), gate-flip model can be applied to any PIM architectures.

7.7 Conclusion

PIM architectures, although regarded as a highly attractive solution to the memory bottleneck problem in data-intensive applications, can potentially impact the reliability of an application mapped to it due to imperfect device and circuit-level variations. In order to quantify the errors introduced into the applications, by misbehaving logic operations, we present gate-flip error model that represents a novel class of functional errors in PIM architectures. This high-level model is able to capture the errors introduced at the logic gate granularity, circumventing low-level errors in the system– to quantitatively measure the error resilience of an application in the context of that PIM architecture.

Chapter 8

Conclusion

8.1 Summary

Processing-in-Memory (PIM) or in-memory computing is reinventing itself as a viable solution to the memory wall problem of von Neumann computing model, in the context of data-intensive applications. In this thesis, we explore the adoption of PIM architecture, particularly computational RAM (CRAM), for applications that use high volume of data and that benefit from the corresponding reduction in data movement overhead when mapped to a PIM architecture. We explore highly important and data-intensive *genomic pattern analysis* application– Burrows–Wheeler Aligner (BWA), through direct-mapping approach and achieved significant performance benefit over state-of-the-art baselines. Adopting HW/SW co-design approach in designing such genomics accelerators, we achieved even better performance than direct-mapping approach, with DNA sequence pre-alignment (and other similar pattern matching), and RNA-Seq abundance quantification applications. These genomics applications are essentially large-scale data-intensive pattern matching workloads that benefits from CRAM-based implementation, especially in terms of energy efficiency. Overall, we demonstrate the direct and HW/SW co-design approaches required for mapping applications in CRAM, while keeping the corresponding software features intact. We show the analytical approach required to identify the opportunities in offer, in terms of performance gain, for an application when mapped to a given PIM architecture. The challenges we experience are discussed, with details of optimal solutions to those problems. Considering the pattern matching

workloads, unconventional computing such as content addressable memory (CAM) is particularly useful in hand-held devices. To address this potential use case, leveraging the energy efficiency benefit of CRAM, CAM functionalities (based on CRAM) are also introduced in resource constraint environments, such as edge/IoT platforms. Such CAM design supports reconfigurability between CAM mode, generic logic and memory modes of operations, with very low reconfiguration and area overheads while maintaining the energy efficiency benefit of CRAM.

As wide-scale adaptation of PIM requires a deeper understanding of the functional reliability, we propose an error model that captures a novel and acute class of error– gate-flip in PIM architectures that are similar to CRAM, while the error model itself is applicable in different PIM architectures. We show that the applications suited for mapping to CRAM or similar architectures have inherent error resilience to hide or reduce the errors introduced by incorrect logic operations.

8.2 Future Research Directions

The future research directions are considered to be spread along two major axes: i) extension of the already explored ideas into other interesting application(domain)s and ii) new research directions that would benefit the ideas explored so far and probable future extensions.

8.2.1 Extension of CRAM-based Designs

The pattern matching accelerators, namely SpinPM, CRAM-Seq and BWA-CRAM, can be adopted for more interesting applications in the respective application domains and beyond. The direct-mapping design methodology presented in BWA-CRAM would be useful in mapping more state-of-the-art domain-specific applications that are suitable for PIM-based implementation from other application domains as well, such as BitFunnel (web search) [146], DBSCAN (data clustering) [147], Breadth-First-Search (graph processing) [148] etc.

On the other hand, HW/SW co-design approach explained in SpinPM and CRAM-Seq would provide guidelines for mapping other popular and memory-bound applications in CRAM, from a wide-range of domains such as genomics [149], machine learning [21]

and large-scale graph analytic [150]. It would be interesting to evaluate the potential of these designs in domain-specific accelerator context too.

Since the underlying principle of performing CAM search using in-memory logic operations, as illustrated in CAMeleon, is generic, it would be beneficial to explore the benefits of adopting CAMeleon in different HW architectures, e.g., CPU/GPU pipeline, core functional block for DNA sequence alignment etc. It would also be interesting to see how much benefit other PIM architectures can provide when modified to perform CAM search with minimal area overhead, following the row selection logic based design modifications presented in CAMeleon.

Overall, the goal of these extensions would be, mainly, to understand the impact on performance when mapped to CRAM, compared to other PIM architecture-based implementations.

8.2.2 New Research Dimensions

There are a number of new dimensions that should be explored next, in order to improve different aspects of application mapping and system integration of CRAM hardware. Majority of these ideas are also applicable to other PIM architectures, hence would constitute significant contributions to PIM literature.

Identification of PIM-compatible Kernels: Any application mapping process begins with identification of opportunities for PIM-based acceleration, in that application. Often times, it involves deeply understanding the problem and the corresponding software implementation, i.e., application– which is not straightforward for a person with no background in that application domain. An automated tool that identifies the functions/code blocks, that are PIM-compatible and dominate the performance through memory-heavy execution, would be a great asset to the PIM-based system designers. Such a tool would be useful in greatly reducing the turnaround time required for mapping an application to a given PIM architecture. In addition to the identification of PIM-compatible kernels, this tool could be extended to output an optimal data layout– for a given PIM architecture and a set of objectives, e.g., latency, throughput, energy efficiency etc. as well.

Amorphous Parallelism: Amorphous data parallelism is a form of data-parallelism that is ubiquitous in irregular algorithms, spanning a wide range of applications– from

data-mining to scientific computation to machine learning. Understanding irregular applications in term of amorphous parallelism could result in better exploiting the parallelism in irregular algorithms, as opposed to the usual way of understanding the data parallelism present in those applications. This approach would, potentially, enable more applications to be mapped in CRAM and other PIM architectures, with sizeable performance benefits.

Variation-aware Mapping of Designs: Due to presence of process variations in devices used in PIM architectures that are heavily device-dependent (especially when performing logic operations *in-situ*), it is often difficult to map different designs on the same PIM substrate of a given size. Pessimistic design choices enable reliable operation of all devices on a substrate, however at the cost of attainable performance by the faster devices on that substrate. Such problem would be significant as the packing density of emerging memory cell technologies improves and multiple interconnected designs (e.g., accelerator) are mapped to a single PIM substrate. An aggressive mapping technique would consider the distribution of parametric variations of devices on a substrate, and map the designs in a way to leverage faster devices for a design with longer latency– thereby, reducing the overall latency of the designs mapped to that substrate.

References

[1] Zamshed I Chowdhury, Jonathan D Harms, S Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P Lyle, Sachin S Sapatnekar, Ulya R Karpuzcu, and Jian-Ping Wang. Efficient in-memory processing using spintronics. *IEEE Computer Architecture Letters*, 17(1):42–46, 2017.

[2] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the IEEE International Symposium on Microarchitecture*, MICRO-50 '17, pages 273–287, New York, NY, USA, 2017. ACM.

[3] Woong Choi, Kyeongho Lee, and Jongsun Park. Low cost ternary content addressable memory using adaptive matchline discharging scheme. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2018.

[4] Maedeh Ghorbanian, Sarineh Hacopian Dolatabadi, and Pierluigi Siano. Big data issues in smart grids: A survey. *IEEE Systems Journal*, 13(4):4158–4168, 2019.

[5] Marco Aiello, Carlo Cavaliere, Antonio D'Albore, and Marco Salvatore. The challenges of diagnostic imaging in the era of big data. *Journal of Clinical Medicine*, 8(3):316, 2019.

[6] Francis X Diebold. On the origin (s) and development of the term 'big data'. *PIER Working Paper*, pages 1–6, 2012.

[7] Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access*, 2:652–687, 2014.

[8] James Bornholt, Randolph Lopez, Douglas M Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. Toward a DNA-based archival storage system. *IEEE Micro*, 37(3):98–104, 2017.

[9] Robert Colwell. The chip design game at the end of Moore's law. In *Proceedings of the IEEE Hot Chips 25 Symposium (HCS)*, pages 1–16. IEEE Computer Society, 2013.

[10] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS Biology*, 13(7):e1002195, 2015.

[11] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, 2018.

[12] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[13] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3–1, 2019.

[14] Jian-Ping Wang and Jonathan D Harms. General structure for Computational Random Access Memory (CRAM), 2015. US Patent 9224447 B2.

[15] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 88–98, 2017.

[16] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A DRAM–based reconfigurable in-situ accelerator. In *Proceedings of the IEEE International Symposium on Microarchitecture*, pages 288–301. ACM, 2017.

[17] R Nair, S F Antao, C Bertolli, P Bose, J R Brunheroto, T Chen, C Y Cher, C H A Costa, J Doi, C Evangelinos, B M Fleischer, T W Fox, D S Gallo, L Grinberg, J A Gunnels, A C Jacob, P Jacob, H M Jacobson, T Karkhanis, C Kim, J H Moreno, J K O'Brien, M Ohmacht, Y Park, D A Prener, B S Rosenburg, K D Ryu, O Sallenave, M J Serrano, P D M Siegl, K Sugavanam, and Z Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, 2015.

[18] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[19] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 600–612. IEEE, 2017.

[20] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–14. ACM, 2018.

[21] Yun Long, Taesik Na, and Saibal Mukhopadhyay. Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(12):2781–2794, 2018.

[22] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC: Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, Nov 2014.

[23] Shahar Kvatinsky, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Memristor-based material implication (IMPLY) logic: Design

principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2013.

[24] Saransh Gupta, Mohsen Imani, and Tajana Rosing. FELIX: fast and energy-efficient logic in memory. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–7. ACM, 2018.

[25] Dayane Reis, Michael Niemier, and X Sharon Hu. Computing in memory with FeFETs. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 1–6, 2018.

[26] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 1–6. IEEE, 2016.

[27] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in memory with spin-transfer torque magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483, March 2018.

[28] Shaahin Angizi, Zhezhi He, and Deliang Fan. PIMA–logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 1–6, 2018.

[29] Salonik Resch, S Karen Khatamifard, Zamshed Iqbal Chowdhury, Masoud Zabihi, Zhengyang Zhao, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. Pimball: Binary neural networks in spintronic memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[30] Charles Augustine, Georgios Panagopoulos, Behtash Behin-Aein, Srikant Srinivasan, Angik Sarkar, and Kaushik Roy. Low-power functionality enhanced computation architecture using spin-based devices. In *Proceedings of the International Symposium on Nanoscale Architectures*, pages 129–136, 2011.

[31] Ziya Arnavut and Spyros S Magliveras. Block sorting and compression. In *Proceedings DCC '97. Data Compression Conference*, pages 181–190, March 1997.

[32] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[33] Zamshed I Chowdhury, S. Karen Khatamifard, Zhengyang Zhao, Masoud Zabihi, Salonik Resch, Meisam Razaviyayn, Jian-Ping Wang, Sachin Sapatnekar, and Ulya R. Karpuzcu. Spintronic in-memory pattern matching. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 5(2):206–214, 2019.

[34] Xiangyu Dong, Cong Xu, Norm Jouppi, and Yuan Xie. NVSim: A circuit-level performance, energy, and area model for emerging non-volatile memory. In *Emerging Memory Technologies*, pages 15–50. Springer, 2014.

[35] Predictive technology model. `http://ptm.asu.edu/`.

[36] 1000 genomes project. `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/`.

[37] NA12878: `ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/NA12878/`.

[38] Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W Cheung, Hing-Fung Ting, Siu-Ming Yiu, Shaoliang Peng, Chang Yu, Yingrui Li, Ruiqiang Li, and Tak-Wah Lam. SOAP3-dp: fast, accurate and sensitive GPU–based short read aligner. *PloS One*, 8(5):e65632, 2013.

[39] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. AlignS: A processing-in-memory accelerator for DNA short read alignment leveraging SOT-MRAM. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 1–6, 2019.

[40] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):1–10, 2009.

[41] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[42] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[43] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. MEDAL: Scalable DIMM based near data processing accelerator for DNA seeding algorithm. In *Proceedings of the IEEE International Symposium on Microarchitecture*, pages 587–599, 2019.

[44] Wim Vanderbauwhede and Khaled Benkrid. *High-performance computing using FPGAs*. Springer, 2013.

[45] Guangming Tan, Chunming Zhang, Wen Tang, Peiheng Zhang, and Ninghui Sun. Accelerating irregular computation in massive short reads mapping on FPGA co-processor. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1253–1264, 2015.

[46] Lisa Wu, David Bruns-Smith, Frank A Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, et al. FPGA accelerated INDEL realignment in the cloud. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 277–290. IEEE, 2019.

[47] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. Race logic: A hardware acceleration for dynamic programming algorithms. *Proceedings of the International Symposium on Computer Architecture*, 42(3):517–528, 2014.

[48] Roman Kaplan, Leonid Yavits, Ran Ginosar, and Uri Weiser. A resistive CAM processing-in-storage architecture for DNA sequence alignment. *IEEE Micro*, 37(4):20–28, 2017.

[49] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. RADAR: A 3D-ReRAM based DNA alignment accelerator architecture. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 1–6, 2018.

[50] Farzaneh Zokaee, Hamid R Zarandi, and Lei Jiang. AligneR: A Process-In-Memory Architecture for Short Read Alignment inReRAMs. *IEEE Computer Architecture Letters*, 17(2):237–240, 2018.

[51] Andrew Lyle, Jonathan Harms, Shruti Patil, Xiaofeng Yao, David J Lilja, and Jian-Ping Wang. Direct communication between magnetic tunnel junctions for

nonvolatile logic fan-out architecture. *Applied Physics Letters*, 97(15):152504, 2010.

[52] Jianguo Wang, Hao Meng, and Jian-Ping Wang. Programmable spintronics logic device based on a magnetic tunnel junction element. *Journal of Applied Physics*, 97(10):10D509, 2005.

[53] Roman Kaplan, Leonid Yavits, and Ran Ginosar. RASSA: Resistive pre-alignment accelerator for approximate DNA long read mapping. *IEEE Micro*, 39(4):44–54, 2018.

[54] Subramanian S Ajay, Stephen CJ Parker, Hatice Ozel Abaan, Karin V Fuentes Fajardo, and Elliott H Margulies. Accurate and comprehensive sequencing of personal genomes. *Genome Research*, 21(9):1498–1505, 2011.

[55] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda-a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27, 2012.

[56] Illumina sequencing by synthesis (SBS) technology: `https://www.illumina.com/technology/next-generation-sequencing/sequencing-technology.html`, March 2017.

[57] Melanie Schirmer, Rosalinda D'Amore, Umer Z Ijaz, Neil Hall, and Christopher Quince. Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data. *BMC Bioinformatics*, 17(1):1–15, 2016.

[58] Julienne M Mullaney, Ryan E Mills, W Stephen Pittard, and Scott E Devine. Small insertions and deletions (INDELs) in human genomes. *Human Molecular Genetics*, 19(R2):R131–R136, 2010.

[59] Sung-Min Ahn, Tae-Hyung Kim, Sunghoon Lee, Deokhoon Kim, Ho Ghang, Dae-Soo Kim, Byoung-Chul Kim, Sang-Yoon Kim, Woo-Yeon Kim, Chulhong Kim, Daeui Park, Yong Seok Lee, Sangsoo Kim, Rohit Reja, Sungwoong Jho, Chang Geun Kim, Ji-Young Cha, Kyung-Hee Kim, Bonghee Lee, Jong Bhak, and Seong-Jin Kim. The first korean genome sequence and analysis: full genome sequencing for a socio-ethnic group. *Genome Research*, 19(9):1622–1629, 2009.

[60] Jeremiah Wala, Pratiti Bandopadhayay, Noah Greenwald, Ryan O'Rourke, Ted Sharpe, Chip Stewart, Steven E Schumacher, Yilong Li, Joachim Weischenfeldt, Xiaotong Yao, Chad Nusbaum, Peter Campbell, Matthew Meyerson, Cheng-Zhong Zhang, Marcin Imielinski, and Rameen Beroukhim. Genome-wide detection of structural variants and indels by local assembly. *bioRxiv*, page 105080, 2017.

[61] Masoud Zabihi, Zamshed I Chowdhury, Zhengyang Zhao, Ulya R Karpuzcu, Jian-Ping Wang, and Sachin Sapatnekar. In-memory processing on the spintronic CRAM: From hardware design to application mapping. *IEEE Transactions on Computers*, pages 1159–1173, 2018.

[62] Masoud Zabihi, Zhengyang Zhao, DC Mahendra, Zamshed I Chowdhury, Salonik Resch, Thomas Peterson, Ulya R Karpuzcu, Jian-Ping Wang, and Sachin S Sapatnekar. Using spin-hall MTJs to build an energy-efficient in-memory computation platform. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 52–57, 2019.

[63] Guenole Jan, Luc Thomas, Son Le, Yuan-Jen Lee, Huanlong Liu, Jian Zhu, Ru-Ying Tong, Keyu Pi, Yu-Jen Wang, Dongna Shen, et al. Demonstration of fully functional 8Mb perpendicular STT-MRAM chips with sub-5ns writing for non-volatile embedded memories. In *Proceedings of the Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pages 1–2, June 2014.

[64] Hiroki Maehara, Kazumasa Nishimura, Yoshinori Nagamine, Koji Tsunekawa, Takayuki Seki, Hitoshi Kubota, Akio Fukushima, Kay Yakushiji, Koji Ando, and Shinji Yuasa. Tunnel magnetoresistance above 170% and resistance—area product of $1\Omega(\mu m)^2$ attained by in situ annealing of ultra-thin MgO tunnel barrier. *Applied Physics Express*, 4(3):033002, 2011.

[65] Hiroki Noguchi, Kazutaka Ikegami, Keiichi Kushida, Keiko Abe, Shogo Itai, Satoshi Takaya, Naoharu Shimomura, Junichi Ito, Atsushi Kawasumi, Hiroyuki Hara, et al. 7.5 A 3.3ns-access-time 71.2 μW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 1–3, Feb 2015.

[66] Everspin technologies. https://www.everspin.com/.

[67] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, 19(2):23–40, 2018.

[68] SRR1153470. https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR1153470.

[69] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[70] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[71] Hybrid memory cube (HMC). http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.3-memory-FPGA/HC23.18.320-HybridCube-Pawlowski-Micron.pdf.

[72] Cortex-A5 processor. http://www.arm.com/products/processors/cortex-a/cortex-a5.php/.

[73] D. Jeon and K. Chung. CasHMC: A cycle-accurate simulator for hybrid memory cube. *IEEE Computer Architecture Letters*, 16(1):10–13, Jan 2017.

[74] Leonid Yavits, Shahar Kvatinsky, Amir Morad, and Ran Ginosar. Resistive associative processor. *IEEE Computer Architecture Letters*, 14(2):148–151, July 2015.

[75] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Christos Kozyrakis. DRAF: A low-power DRAM-based

reconfigurable acceleration fabric. *ACM SIGARCH Computer Architecture News*, 44(3):506–518, 2016.

[76] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16(1):46–50, 2016.

[77] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the International Conference on Parallel Architecture and Compilation*, pages 113–124. IEEE, 2015.

[78] Micron TN-41-01: Calculating memory system power for DDR3.

[79] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, pages 1–4, December 2013.

[80] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. Soap3: ultra-fast GPU–based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[81] Daehwan Kim, Geo Pertea, Cole Trapnell, Harold Pimentel, Ryan Kelley, and Steven L Salzberg. TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36, 2013.

[82] Cole Trapnell, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J Van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511, 2010.

[83] Rob Patro, Stephen M Mount, and Carl Kingsford. Sailfish enables alignment-free isoform quantification from RNA-Seq reads using lightweight algorithms. *Nature Biotechnology*, 32(5):462, 2014.

[84] Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-Seq quantification. *Nature Biotechnology*, 34(5):525–527, 2016.

[85] Sébastien Rodrigue, Arne C Materna, Sonia C Timberlake, Matthew C Blackburn, Rex R Malmstrom, Eric J Alm, and Sallie W Chisholm. Unlocking short read sequencing for metagenomics. *PloS One*, 5(7):1–9, 2010.

[86] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[87] Zhaojun Zhang and Wei Wang. RNA-Skim: a rapid method for RNA-Seq quantification at transcript level. *Bioinformatics*, 30(12):i283–i292, 2014.

[88] DC Mahendra, Roberto Grassi, Jun-Yang Chen, Mahdi Jamali, Danielle Reifsnyder Hickey, Delin Zhang, Zhengyang Zhao, Hongshi Li, P Quarterman, Yang Lv, et al. Room-temperature high spin–orbit torque due to quantum confinement in sputtered $Bi_xSe_{(1-x)}$ films. *Nature Materials*, 17(9):800–807, 2018.

[89] Kevin Garello, Can Onur Avci, Ioan Mihai Miron, Manuel Baumgartner, Abhijit Ghosh, Stéphane Auffret, Olivier Boulle, Gilles Gaudin, and Pietro Gambardella. Ultrafast magnetization switching by spin-orbit torques. *Applied Physics Letters*, 105(21):212402, 2014.

[90] Jennifer Harrow, Adam Frankish, Jose M Gonzalez, Electra Tapanari, Mark Diekhans, Felix Kokocinski, Bronwen L Aken, Daniel Barrell, Amonida Zadissa, Stephen Searle, et al. GENCODE: the reference human genome annotation for the ENCODE project. *Genome Research*, 22(9):1760–1774, 2012.

[91] Alexander Lachmann, Zhuorui Xie, and Avi Ma'ayan. Elysium: RNA-Seq alignment in the cloud. *bioRxiv*, page 382937, 2018.

[92] Alexander Lachmann, Daniel JB Clarke, Denis Torre, Zhuorui Xie, and Avi Ma'ayan. Interoperable RNA-Seq analysis in the cloud. *Biochimica et Biophysica Acta (BBA)-Gene Regulatory Mechanisms*, page 194521, 2020.

[93] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, pages 389–394. IEEE, 2013.

[94] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*, pages 1–4. Springer, 2009.

[95] Cuong Pham-Quoc, Binh Kieu-Do, and Tran Ngoc Thinh. A high-performance FPGA-based BWA-MEM DNA sequence alignment. *Concurrency and Computation: Practice and Experience*, page e5328, 2019.

[96] Xia Fei, Zou Dan, Lu Lina, Man Xin, and Zhang Chunlei. FPGASW: Accelerating large-scale Smith—Waterman sequence alignment application with backtracking on FPGA linear systolic array. *Interdisciplinary Sciences: Computational Life Sciences*, 10(1):176–188, 2018.

[97] Enzo Rucci, Carlos Garcia, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matias. SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. *BMC Systems Biology*, 12(5):96, 2018.

[98] James Arram, Thomas Kaplan, Wayne Luk, and Peiyong Jiang. Leveraging FPGAs for accelerating short read alignment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(3):668–677, 2016.

[99] Zamshed I Chowdhury, Masoud Zabihi, S Karen Khatamifard, Zhengyang Zhao, Salonik Resch, Meisam Razaviyayn, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. A DNA read alignment accelerator based on computational RAM. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 6(1):80–88, 2020.

[100] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan

Gomez-Luna, Amirali Boroumand, et al. GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *Proceedings of the IEEE International Symposium on Microarchitecture*, pages 951–966, 2020.

[101] Yasunobu Okamura and Kengo Kinoshita. Matataki: an ultrafast mRNA quantification method for large-scale reanalysis of RNA-Seq data. *BMC Bioinformatics*, 19(1):266, 2018.

[102] Jhih-Yu Huang and Pi-Chung Wang. TCAM-based IP address lookup using longest suffix split. *IEEE/ACM Transactions on Networking*, 26(2):976–989, 2018.

[103] Gordon Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[104] Yuriy V Pershin and Massimiliano Di Ventra. Neuromorphic, digital, and quantum computation with memory circuit elements. *Proceedings of the IEEE*, 100(6):2071–2080, 2012.

[105] Li-Yue Huang, Meng-Fan Chang, Ching-Hao Chuang, Chia-Chen Kuo, Chien-Fu Chen, Geng-Hau Yang, Hsiang-Jen Tsai, Tien-Fu Chen, Shyh-Shyuan Sheu, Keng-Li Su, et al. ReRAM-based 4T2R nonvolatile TCAM with 7x NVM-stress reduction, and 4x improvement in speed-word length-capacity for normally-off instant-on filter-based search engines used in big-data processing. In *Proceedings of the Symposium on VLSI Circuits Digest of Technical Papers*, pages 1–2. IEEE, 2014.

[106] Xuan-Thuan Nguyen, Trong-Thuc Hoang, Hong-Thu Nguyen, Katsumi Inoue, and Cong-Kha Pham. An FPGA-based hardware accelerator for energy-efficient bitmap index creation. *IEEE Access*, 6:16046–16059, 2018.

[107] Mohsen Imani, Abbas Rahimi, and Tajana S Rosing. Resistive configurable associative memory for approximate computing. In *Proceedings of the Design, Automation & Test in Europe*, pages 1327–1332. IEEE, 2016.

[108] Hsiang-Jen Tsai, Keng-Hao Yang, Yin-Chi Peng, Chien-Chen Lin, Ya-Han Tsao, Meng-Fan Chang, and Tien-Fu Chen. Energy-efficient TCAM search engine design using priority-decision in memory technology. *IEEE Transactions on VLSI Systems*, 25(3):962–973, 2017.

[109] Naoya Onizawa, Shoun Matsunaga, Vincent C Gaudet, and Takahiro Hanyu. High-throughput low-energy content-addressable memory based on self-timed overlapped search mechanism. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems*, pages 41–48. IEEE, 2012.

[110] Somnath Paul and Swarup Bhunia. Reconfigurable computing using content addressable memory for improved performance and resource usage. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 786–791, 2008.

[111] Nazgul Dastanova, Sultan Duisenbay, Olga Krestinskaya, and Alex Pappachen James. Bit-plane extracted moving-object detection using memristive crossbar-cam arrays for edge computing image devices. *IEEE Access*, 6:18954–18966, 2018.

[112] Olga Krestinskaya, Alex Pappachen James, and Leon Ong Chua. Neuromemristive circuits for edge computing: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 31(1):4–23, 2019.

[113] Olga Krestinskaya and Alex Pappachen James. Binary weighted memristive analog deep neural network for near-sensor edge processing. In *Proceedings of the IEEE International Conference on Nanotechnology (IEEE-NANO)*, pages 1–4. IEEE, 2018.

[114] Wei-Pau Kiat, Kai-Ming Mok, Wai-Kong Lee, Hock-Guan Goh, and Ramachandra Achar. An energy efficient FPGA partial reconfiguration based micro-architectural technique for IoT applications. *Microprocessors and Microsystems*, 73:102966, 2020.

[115] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G Friedman. AC-DIMM: associative computing with STT-MRAM. *ACM SIGARCH Computer Architecture News*, 41(3):189–200, 2013.

[116] Bipin Rajendran, Roger W Cheek, Luis A Lastras, Michele M Franceschini, Matthew J Breitwisch, Alejandro G Schrott, Jing Li, Robert K Montoye, Leland Chang, and Chung Lam. Demonstration of CAM and TCAM using phase change devices. In *2011 3rd IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2011.

[117] Hasan Erdem Yantır, Ahmed M. Eltawil, and Fadi J. Kurdahi. A hybrid approximate computing approach for associative in-memory processors. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(4):758–769, 2018.

[118] Salonik Resch, S Karen Khatamifard, Zamshed I Chowdhury, Masoud Zabihi, Zhengyang Zhao, Husrev Cilasun, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. MOUSE: Inference in non-volatile memory for energy harvesting applications. In *Proceedings of the IEEE International Symposium on Microarchitecture*, pages 400–414. IEEE, 2020.

[119] Wei-Hao Chen, Chunmeng Dou, Kai-Xiang Li, Wei-Yu Lin, Pin-Yi Li, Jian-Hao Huang, Jing-Hong Wang, Wei-Chen Wei, Cheng-Xin Xue, Yen-Cheng Chiu, et al. CMOS-integrated memristive non-volatile computing-in-memory for ai edge processors. *Nature Electronics*, 2(9):420–428, 2019.

[120] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, July 2012.

[121] Qing Guo, Xiaochen Guo, Yuxin Bai, and Engin Ipek. A resistive TCAM accelerator for data-intensive computing. *Proceedings of the IEEE International Symposium on Microarchitecture*, pages 339–350, 2011.

[122] Byungkyu Song, Taehui Na, Jung Pill Kim, Seung H Kang, and Seong-Ook Jung. A 10T-4MTJ nonvolatile ternary CAM cell for reliable search operation and a compact area. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 64(6):700–704, 2017.

[123] Shoun Matsunaga, Sadahiko Miura, Hiroaki Honjou, Keizo Kinoshita, Shoji Ikeda, Tetsuo Endoh, Hideo Ohno, and Takahiro Hanyu. A 3.14 μm^2 4T-2MTJ-cell fully parallel TCAM based on nonvolatile logic-in-memory architecture. In *Proceedings of the Symposium on VLSI Circuits (VLSIC)*, pages 44–45. IEEE, 2012.

[124] Chengzhi Wang, Deming Zhang, Lang Zeng, Erya Deng, Jie Chen, and Weisheng Zhao. A novel MTJ-based non-volatile ternary content-addressable memory for high-speed, low-power, and high-reliable search operation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(4):431–434, 2018.

[125] Rekha Govindaraj and Swaroop Ghosh. Design and analysis of sttram-based ternary content addressable memory cell. *ACM Journal on Emerging Technologies in Computing Systems*, 13(4):1–22, May 2017.

[126] Shuangchen Li, Liu Liu, Peng Gu, Cong Xu, and Yuan Xie. NVSim-CAM: A circuit-level simulator for emerging nonvolatile memory based content-addressable memory. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '16, pages 1–7, 2016.

[127] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 541–552. IEEE, 2017.

[128] Hüsrev Cılasun, Salonik Resch, Zamshed I Chowdhury, Erin Olson, Masoud Zabihi, Zhengyang Zhao, Thomas Peterson, Keshab K. Parhi, Jian-Ping Wang, Sachin S. Sapatnekar, and Ulya R. Karpuzcu. Spiking neural networks in spintronic computational RAM. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–21, 2021.

[129] Shaahin Angizi, Zhezhi He, Amro Awad, and Deliang Fan. MRIMA: An MRAM-based in-memory accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(5):1123–1136, 2019.

[130] WS Zhao, Yue Zhang, Thibaut Devolder, Jacques-Olivier Klein, Dafine Ravelosona, Claude Chappert, and Pascale Mazoyer. Failure and reliability analysis of STT-MRAM. *Microelectronics Reliability*, 52(9-10):1848–1852, 2012.

[131] Yunkun Xie, Behtash Behin-Aein, and Avik W Ghosh. Fokker—Planck study of parameter dependence on write error slope in spin-torque switching. *IEEE Transactions on Electron Devices*, 64(1):319–324, 2016.

[132] Daisuke Saida, Saori Kashiwada, Megumi Yakabe, Tadaomi Daibou, Naoki Hase, Miyoshi Fukumoto, Shinji Miwa, Yoshishige Suzuki, Hiroki Noguchi, Shinobu Fujita, et al. Sub-3 ns pulse with sub-100 μA switching of 1x–2x nm perpendicular MTJ for high-performance embedded STT-MRAM towards sub-20 nm CMOS. In *Proceedings of the IEEE Symposium on VLSI Technology*, pages 1–2. IEEE, 2016.

[133] S Aggarwal, H Almasi, M DeHerrera, B Hughes, S Ikegawa, J Janesky, HK Lee, H Lu, FB Mancoff, K Nagel, et al. Demonstration of a reliable 1 Gb standalone spin-transfer torque MRAM for industrial applications. In *Proceedings of the IEEE International Electronic Devices Meeting*, pages 2–1. IEEE, 2019.

[134] Takaharu Saino, Shun Kanai, Motoya Shinozaki, Butsurin Jinnai, Hideo Sato, Shunsuke Fukami, and Hideo Ohno. Write-error rate of nanoscale magnetic tunnel junctions in the precessional regime. *Applied Physics Letters*, 115(14):142406, 2019.

[135] Tatsuya Yamamoto, Takayuki Nozaki, Hiroshi Imamura, Yoichi Shiota, Shingo Tamaru, Kay Yakushiji, Hitoshi Kubota, Akio Fukushima, Yoshishige Suzuki, and Shinji Yuasa. Improvement of write error rate in voltage-driven magnetization switching. *Journal of Physics D: Applied Physics*, 52(16):164001, 2019.

[136] Jeehwan Song, Hemant Dixit, Behtash Behin-Aein, Chris H Kim, and William Taylor. Impact of process variability on write error rate and read disturbance in STT-MRAM devices. *IEEE Transactions on Magnetics*, 56(12):1–11, 2020.

[137] G Hu, JJ Nowak, MG Gottwald, SL Brown, B Doris, CP D'Emic, P Hashemi, D Houssameddine, Q He, D Kim, et al. Spin-transfer torque MRAM with reliable 2 ns writing for last level cache applications. In *Proceedings of the IEEE International Electronic Devices Meeting*, pages 2–6. IEEE, 2019.

[138] Wang Kang, Liuyang Zhang, Weisheng Zhao, Jacques-Olivier Klein, Youguang Zhang, Dafiné Ravelosona, and Claude Chappert. Yield and reliability improvement techniques for emerging nonvolatile STT-MRAM. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(1):28–39, 2014.

[139] Seyedhamidreza Motaman, Swaroop Ghosh, and Nitin Rathi. Impact of process-variations in STTRAM and adaptive boosting for robustness. In *Proceedings of the Design, Automation & Test in Europe*, pages 1431–1436. IEEE, 2015.

[140] Zhenyu Sun, Xiuyuan Bi, and Hai Li. Process variation aware data management for STT-RAM cache design. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, pages 179–184, 2012.

[141] Zhenyu Sun, Xiuyuan Bi, Hai Li, Weng-Fai Wong, and Xiaochun Zhu. STT-RAM cache hierarchy with multiretention MTJ designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6):1281–1293, 2013.

[142] Yaojun Zhang, Xiaobin Wang, Hai Li, and Yiran Chen. STT-RAM cell optimization considering MTJ and CMOS variations. *IEEE Transactions on Magnetics*, 47(10):2962–2965, 2011.

[143] Giju Jung, Mohammed Fouda, Sugil Lee, Jongeun Lee, Ahmed Eltawil, and Fadi Kurdahi. Cost- and dataset-free stuck-at fault mitigation for ReRAM-based deep learning accelerators. In *Proceedings of the Design, Automation & Test in Europe*, pages 1733–1738. IEEE, 2021.

[144] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[145] Daniele Garbin, Elisa Vianello, Quentin Rafhay, Mourad Azzaz, Philippe Candelier, Barbara DeSalvo, Gerard Ghibaudo, and Luca Perniola. Resistive memory

variability: A simplified trap-assisted tunneling model. *Solid-State Electronics*, 115:126–132, 2016.

[146] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. Bitfunnel: Revisiting signatures for search. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 605–614, 2017.

[147] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[148] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.

[149] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the Micron automata processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 415–424. IEEE, 2014.

[150] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 531–543. IEEE, 2018.