ENABLING NEAR DATA PROCESSING FOR EMERGING WORKLOADS

by

Anirban Nag

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2020

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Anirban Nag**

has been approved by the following supervisory committee members:

Rajeev Balasubramonian ,	Chair(s)	**21 Oct 2020** Date Approved
Mahdi Nazm Bojnordi ,	Member	**24 Oct 2020** Date Approved
Nuwan Jayasena ,	Member	**20 Oct 2020** Date Approved
Ryan Stutsman ,	Member	**20 Oct 2020** Date Approved
Vivek Srikumar ,	Member	**20 Oct 2020** Date Approved

by **Mary Hall** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Keida** , Dean of The Graduate School.

ABSTRACT

Emerging workloads such as data analytics, machine learning, and genomic analysis are largely data-centric; that is, they fetch large amounts of data and perform very few computations per data element. Near Data Processing (NDP) architectures that place compute-units close to the memory or perform in-situ computations within the memory array can improve the performance and energy efficiency of such data-centric workloads by reducing data movement and providing high bandwidth access to memory. However, harnessing the potential of near data processing is not straightforward and often requires maneuvering several design challenges. In this dissertation, we address some of these challenges in the context of three different NDP architectures: (1) analog in-situ accelerators for neural networks, (2) in-cache accelerators for genomics, and (3) fine-grained orchestration of compute-units within a memory module.

First, we inspect the energy-efficiency of analog in-situ accelerators. Such accelerators improve the throughput of deep neural network inference by performing fast matrix-vector multiplications using memristor crossbars. However, they have one significant shortcoming: the Analog-to-Digital Converters (ADCs) consume a significant amount of energy. We show that by leveraging techniques such as matrix transformation, heterogeneity, and smart mapping of computations, ADC usage can be significantly reduced to improve energy efficiency by 48%. Second, we explore the acceleration potential of large caches used in modern digital accelerators in the context of sequence alignment, a key computational step in the genomic analysis for healthcare. We restructure the sequence alignment algorithm to use bit-vector computations, which can be accelerated using highly parallel in-cache bit-vector operations. Thus, leveraging hardware-software codesign techniques improves the throughput of sequence alignment by $5\times$. Finally, we explore efficient ways to orchestrate requests to compute units within a memory module. In order to coschedule computation requests alongside regular load/store requests, we make a case for fine-grained offloading of computations and address the associated challenge of a

memory-centric ordering requirement for such fine-grained requests. We show that our novel lightweight ordering primitive can greatly enhance performance by delivering a 5.5× to 8.5× speedup over traditional ordering primitives. Thus, by examining multiple workload classes, multiple implementation technologies, and multiple layers of the system stack, we make a strong case for the potential of near data processing.

To my mother (Ma) and my uncle (Kakai).

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF TABLES . . . . . **viii**

ACKNOWLEDGEMENTS . . . . . **ix**

CHAPTERS

1. INTRODUCTION **1**

1.1 Introduction 1
1.2 Near Data Processing Terminology 2
1.3 Dissertation Overview 3
1.4 Layout of this Dissertation 6

2. NEWTON: GRAVITATING TOWARDS THE PHYSICAL LIMITS OF CROSSBAR ACCELERATION **7**

2.1 Introduction 7
2.2 Background 8
2.3 The Newton Architecture 12
2.4 Methodology 20
2.5 Results 22
2.6 Related Works 26

3. GENCACHE: LEVERAGING IN-CACHE OPERATORS FOR EFFICIENT SEQUENCE ALIGNMENT **39**

3.1 Introduction 39
3.2 Background 41
3.3 Proposal 47
3.4 Methodology 56
3.5 Results 58
3.6 Application to Third-Generation Alignment 59
3.7 Related Works 61

4. ORDERLIGHT: LIGHTWEIGHT MEMORY-ORDERING PRIMITIVE FOR EFFICIENT FINE-GRAINED PIM COMPUTATIONS **69**

4.1 Introduction 69
4.2 Background 72
4.3 Taxonomy of PIM Offload and Arbitration 73
4.4 Challenge: Ordering of PIM Instructions 76
4.5 OrderLight Design 78
4.6 Programmability 83

4.7 Methodology 84
4.8 Results 85

5. CONCLUSIONS AND FUTURE WORK 97

5.1 In-Situ Analog Computing for Machine Learning 97
5.2 In-Cache Computing for Genomics 98
5.3 Processing in Memory for Memory-Bound Workloads 99

REFERENCES 101

LIST OF TABLES

2.1 Key contributing elements in Newton. 37

2.2 Benchmark names are in bold. Layers are formatted as $K_x \times K_y, N_o$/stride (t), where t is the number of such layers. Stride is 1 unless explicitly mentioned. . . 38

3.1 Area and power of different components in GenAx (yellow/orange) and GenCache (white/orange) at 32 nm. 68

3.2 Comparison of latency, energy and peak throughput of different GenCache operations at 128 bp granularity for a 32 MB scratchpad (102.1 mm^2) with GenAx SWA engine (1.86 mm^2). 68

4.1 Simulator details. 96

4.2 Summary of workloads. 96

ACKNOWLEDGEMENTS

I came to Utah, knowing that I would be working with Rajeev as my Ph.D. advisor. That was the first and the best decision I made as part of my Ph.D. career. Rajeev has shown me the path to do interesting and relevant research, helped me grow as a person, and supported me through all my highs and lows. Rajeev serves me as a role model.

My mother (Ma) selflessly encouraged me to study abroad, even though she had to live on her own. Her constant care comforted me during my stay in the USA, and I can never thank her enough. My uncle (Kakai) has been my source of inspiration since childhood to pursue knowledge. My email exchanges with him during the 5 years of my Ph.D. serves to me as a reminder to never stop being curious. I also thank my sister (Amrita) and cousin brother (Sudipto) for keeping me cheerful throughout.

I thank my committee members Mahdi, Nuwan, Ryan, and Vivek, for their valuable feedback on my Ph.D. dissertation. Special thanks to all my mentors who helped me grow as a researcher: Ali during the initial years of my Ph.D., Naveen during my internship at HPE, and Nuwan and Shaizeen during my internship at AMD. I thank all the coauthors in our research papers for their valuable contributions before paper submission and during rebuttals.

Without my labmates and friends in USA, I wouldn't have had such a fun and enjoyable Ph.D. experience. I thank my labmates, Surya, Meysam, Keeton, Sumanth, Ram, Karl, Sarabjeet, Chandru, and Shirley, for the many brainstorming sessions and for playing ping-pong. I also thank my friends, Souransu, Aditi, Kushagradhi, Asmit, Abhishek, Subhankar, Sarthak, Tannistha, Yashad, Yogesh, Sunny, Nishant, Shweta, Prasid, Jahin, Shuvrajit, Sreejita, Aakanksha, Sonika, Sunipa, Debolina, David, Jarkko, Brett, and Mathias, for making my stay in USA fun and adventurous. I would like to thank the School of Computing staff, Karen, Ann, Robert, Leslie, Chethika, Lauren, and Jill, for supporting me through all the administrative hurdles.

Although I specifically thanked a few people, I am fully aware and appreciate many

others who indirectly encouraged me, inspired me, or made this journey much easier for me.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Recent years have seen a growing interest in emerging workloads such as machine learning, data analytics, and genomics. The interest is due to the influx of data that are being made available through cheap mobile and IoT devices, inexpensive and high throughput sequencing methods, and the advent of social media. Machine learning tasks such as image or speech recognition have surpassed human-level efficacy [53] and have become mainstream in applications such as language translation, healthcare, astronomy, and self-driving cars. Genomic analysis is enabling the next generation of healthcare due to affordable and portable sequencing. In the future, doctors will prescribe targeted treatment or medicine to patients by inspecting their genome, and genome sequence alignment is a major computational task associated with such analysis. Data analytics tasks such as clustering, filtering, etc. serve as primary enablers in handling large amounts of unstructured data. Most of these tasks are data-intensive or memory-bound; that is, they fetch large amounts of data and perform very few computations per data element.

On the other hand, the gap in energy consumption and performance between executing an operation in a core and accessing the operands from memory (dubbed as the "memory wall") has been a problem for many years. At 45nm technology, memory accesses consume significantly higher energy (45 pJ/bit [28]) in comparison to floating-point operations (0.3 pJ/bit). Accessing the main memory can take up to 55 ns [149], orders of magnitude higher than accessing on-chip memory (caches) or performing operations at the core. The memory wall has been the major bottleneck in modern computing systems, largely due to limited improvements in memory latency and access energy through the scaling of technology nodes. The advent of data-driven computing and the onset of the memory wall have led to renewed interest in Near Data Processing (NDP) [12].

In computer architecture, Near Data Processing (NDP) refers to moving the compute closer to the memory to reduce data movement and to provide an application with much higher data bandwidth. Data movement is reduced as the data need not traverse long wires to the processor core, which in turn reduces energy consumption. NDP also provides higher bandwidth as compute is moved closer to the memory arrays (Static Random Access Memory (SRAM) based caches or Dynamic Random Access Memory (DRAM) based main memory), which have much higher internal bandwidth (due to internal organization of the memory) in comparison to the memory bus that reaches the core. Thus, NDP has many potential benefits: (1) circumventing data traversal over long wires reduces data movement cost thereby reducing latency and energy, and (2) higher internal bandwidth of a memory organization (SRAM cache or DRAM main memory) provides high data parallelism thereby improving throughput.

There are many different flavors of NDP: (1) analog in-situ computing in memristive devices [14], [25], [129], (2) in-cache computing using SRAM caches [2], [33], (3) in-DRAM computing which performs computation within DRAM arrays [84], [85], [127], [128], and (4) Processing In Memory (PIM), which places logic near the DRAM arrays [9], [36], [41], [106], [118], [161]. However, taking advantage of these techniques is not straightforward. For example, analog in-situ computing suffers from high energy overhead of analog-to-digital converters. In-cache and in-DRAM computing require hardware changes in the peripheral circuits of the memory array to support essential bit-parallel operations. Efficient control schemes are essential to send the commands from the host (CPU/GPU) to the PIM device. In some cases, the application software needs to be modified to adapt to the available NDP operations. Given the immense future potential of NDP, this dissertation addresses some of these challenges to make NDP a viable option.

1.2 Near Data Processing Terminology

In this dissertation, we refer to Near Data Processing (NDP) as an umbrella term that encompasses both in-situ computations within a memory array using technologies such as memristors crossbars or SRAM cache, as well as near memory computations where logic is placed near a memory arrays such as SRAM cache or DRAM main memory arrays. Historically, Processing In-Memory (PIM) has been used to refer to processing in a memory

module such as DRAM main memory, which is why in Chapter 4 we use PIM.

1.3 Dissertation Overview

Near Data Processing (NDP) is a paradigm shift in the model of computing. Unless we show significant improvements and generality in our research innovations, industry is reluctant to pursue a paradigm shift. For this reason, the investment on NDP by industry has been cautious but is likely to pick up steam as killer applications emerge. A few hardware startups have shown interest: Graphcore [47] has invested in building machine learning accelerators using NDP, Mythic [101] and Syntiant [141] announced dot-product execution in flash memory for machine learning workloads, and Upmem [146] has come up with IP blocks to support Processing In Memory (PIM) for various memory-bound applications. Well established companies such as Micron have proposed implementing nondeterministic finite automata in DRAM technology using their Automata Processor [151].

Still, many challenges remain unanswered and many opportunities remain unexplored in the NDP space. In this dissertation, we explore some of these challenges and opportunities to improve the efficiency of NDP in terms of performance, energy, and generality of usage. In this section, first I define my dissertation statement and then explain in brief each of the three projects that constitute this dissertation and improves various aspects of Near Data Processing (NDP).

1.3.1 Dissertation Statement

We hypothesize that the efficiency of Near Data Processing (NDP) can be improved by (1) amortizing the energy cost of analog computing with digital structures, (2) adopting hardware-software codesign techniques that restructure the software algorithms to take advantage of NDP hardware innovations, and (3) designing efficient control schemes to orchestrate the NDP accelerator. We test this hypothesis in the context of analog in-situ accelerators for neural networks, in-cache accelerators for genomics, and Processing-In-Memory enabled memory modules.

The dissertation has two notable themes: (1) balance in computer architecture design points, and (2) application-specific knowledge in systems design. First, we try to compromise a bit on one aspect to gain a lot on another aspect to achieve balance in systems design. For example, trading area for generality, trading area for throughput, trading latency for

energy, etc. You will spot this theme in Chapters 2, 3, and 4. Second, we use insights provided by the application algorithms to design efficient accelerators. You will spot this theme in Chapters 2 and 3.

1.3.2 In-Situ Analog Computing for Machine Learning

Many recent works have designed accelerators for Convolutional and Deep Neural Networks (CNNs and DNNs). These algorithms typically involve a large number of multiply-accumulate (dot-product) operations. While digital accelerators have relied on near data processing using systolic arrays and large on-chip memory, analog accelerators have further reduced data movement by performing near data processing using in-situ computation. A recent work, In-Situ Analog Arithmetic in Crossbars (ISAAC), takes advantage of highly parallel analog in-situ computation in memristor crossbars to accelerate the many vector-matrix multiplication operations in CNNs. However, ISAAC has two significant short-comings that we address in this work. First, ISAAC is a homogeneous design where every resource is provisioned for the worst case. Second, the ADCs account for a large fraction of chip power and area. By addressing both problems, the new architecture, Newton , moves closer to achieving optimal energy-per-neuron for crossbar accelerators.

We introduce six new techniques that apply at different levels of the tile hierarchy. Two of the techniques leverage heterogeneity: one adapts ADC precision based on the requirements of every subcomputation (with zero impact on accuracy), and the other designs tiles customized for convolutions or classifiers. Two other techniques rely on divide-and-conquer numeric algorithms to reduce computations and ADC pressure. The final two techniques place constraints on how a workload is mapped to tiles, thus helping reduce resource provisioning in tiles. For a wide range of CNN dataflows and structures, Newton achieves a 77% decrease in power, 51% improvement in energy efficiency, and 2.2$\times$ higher throughput/area, relative to the state-of-the-art ISAAC accelerator.

1.3.3 In-Cache Computing for Genomics

The future of healthcare is personalized medicine, which require sequencing the genome of every individual, multiple times. Sequence alignment is the core computational challenge and constitutes up to 53% of genome analysis runtime. A high throughput accelerator will empower healthcare providers to perform genomic analysis at scale. Such an

accelerator can not only be deployed in mobile sequencing devices to provide real-time analysis but also in cloud servers for offline analysis by hospitals and doctors. Recent state-of-the-art alignment accelerators dedicate a small fraction of chip area for Arithmetic Logic Units (ALUs), control logic, and registers, and store frequently accessed data structures in lssarge SRAM caches (90% of the chip area). We observe that (1) bitwise computation can be performed in cache, (2) large caches comprising of several subarrays provide high compute parallelism, and (3) key steps in sequence alignment solely involve bit-vector computations. In this work, we introduce an accelerator, GenCache, that (1) equips the SRAM subarray peripherals with additional logic to perform bit-vector and aggregation operations essential for genomics, and (2) smartly balances the cache space between compute parallelism and storage using assistive structures such as bloom-filters. We also restructure the algorithm based on the common case output of genomic datasets to efficiently utilize the highly parallel in-cache operations. A hardware-software codesign approach yields more than additive speed-up of 5.3$\times$ over the state-of-the-art GenAx accelerator. We also show that the basic principles in GenCache can be exploited for both short reads emitted by second-generation sequencing devices and long reads emitted by third-generation sequence devices.

1.3.4 Processing in Memory for Memory-Bound Workloads

Modern workloads such as neural networks, genomic analysis, and data analytics exhibit significant data-intensive phases (low compute to byte ratio) and, as such, stand to gain considerably by using processing-in-memory (PIM) solutions along with more traditional accelerators. While PIM has been researched extensively, the granularity of computation offload to PIM and the granularity of memory access arbitration between host and PIM, as well as their implications, have received relatively little attention. In this work, we first introduce a taxonomy to study the design space whilst considering these two aspects. Based on this taxonomy, we observe that much of PIM research to date has largely relied on coarse-grained approaches that, we argue, have steep costs (incompatibility with mainstream memory interfaces, prohibition of concurrent host accesses, and more). To this end, we believe that better support for fine-grained approaches is warranted in accelerators coupled with PIM-enabled memories.

A key challenge in the adoption of fine-grained PIM approaches is enforcing memory ordering. We discuss how existing memory ordering primitives (fences) are not only insufficient but their huge overheads render them impractical to support fine-grain computation offloads and arbitration. To address this challenge, we make the key observation that the core-centric nature of memory ordering is unnecessary for PIM computations. We propose a novel lightweight memory ordering primitive for PIM use cases, OrderLight, which moves away from core-centric ordering enforcement and considerably reduces the overheads of enforcing correctness. For a suite of key computations from machine learning, data analytics, and genomics, we demonstrate that OrderLight delivers 5.5× to 8.5× speedup over traditional fences.

Thus, the dissertation investigates three unique challenges or opportunities associated with NDP: (1) energy overheads of in-situ analog computing due to analog-to-digital converters (ADCs), (2) potential to utilize large caches used in modern accelerators for highly parallel bit-vector computations, and (3) performance overheads of existing "core-centric" ordering primitives for orchestration of fine-grained PIM instructions. While addressing these challenges and opportunities, we adopt techniques such as heterogeneity, common case resource allocation, hardware-software codesign, and lightweight microarchitectural modifications. Our techniques resulted in significant improvement in either energy-efficiency or throughput, or performance for the three different NDP technologies.

1.4 Layout of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 covers improving energy-efficiency of near data processing in the context of an in-situ analog accelerator (Newton) for machine learning. Chapter 3 covers improving performance of near data processing in the context of an in-cache computing accelerator (GenCache) for genomics. Chapter 4 presents ideas on improving generality of near data processing systems in the context of designing efficient control schemes for Processing In Memory. Finally, Chapter 5 discusses our primary conclusions and presents future research directions.

CHAPTER 2

NEWTON: GRAVITATING TOWARDS THE PHYSICAL LIMITS OF CROSSBAR ACCELERATION

2.1 Introduction

Accelerators are in vogue today, primarily because it is evident that annual performance improvements can be sustained via specialization. There are also many emerging applications that demand high-throughput low-energy hardware, such as the machine learning tasks that are becoming commonplace in enterprise servers, self-driving cars, and mobile devices. The last 2 years have seen a flurry of activity in designing machine learning accelerators [22], [24], [25], [31], [71], [87], [121], [130], [158]. Similar to our work, most of these recent works have focused on inference in artificial neural networks, and specifically, deep convolutional networks, which achieve state-of-the-art accuracies on challenging image classification workloads.

While most of these recent accelerators have used digital architectures [22], [24], a few have leveraged analog acceleration on memristor crossbars [14], [25], [129]. Such accelerators take advantage of in-situ computation to dramatically reduce data movement costs. Each crossbar is assigned to execute parts of the neural network computation and programmed with the corresponding weight values. Input neuron values are fed to the crossbar, and by leveraging Kirchoff's Law, the crossbar outputs the corresponding dot product. The neuron output undergoes analog-to-digital conversion (ADC) before being sent to the next layer. Multiple small-scale prototypes of this approach have also been demonstrated [60], [94].

In this chapter, we focus on innovations in the recent ISAAC architecture [129]. While ISAAC was able to provide an order of magnitude improvement in throughput, relative to state-of-the-art digital architectures, it has a higher power density. This is primarily

because of the power and area overheads of ADC units. Therefore, we try to improve various aspects of the ISAAC design, with a special emphasis on reducing ADC and other resource requirements.

We introduce six innovations at different levels of the tile hierarchy. These innovations leverage heterogeneous requirements of different parts of the neural network computation. They avoid overprovisioning ADC, HTree, and buffer resources. And finally, they leverage numeric algorithms to further reduce the involvement of ADCs.

The new design, Newton [102], is moving the crossbar architecture closer to the bare minimum energy required to process one neuron. It does this by reducing the overheads imposed by the architecture and by large modern workloads. We define our ideal neuron as one that keeps the weight in-place adjacent to a digital ALU, retrieves the input from an adjacent single-row embedded DRAM (eDRAM) unit, and after performing one digital operation, writes the result to another adjacent single-row eDRAM unit. This energy is lower than that for a similarly ideal analog neuron because of the ADC cost. This ideal neuron operation consumes 0.33 pJ. An average DaDianNao operation consumes 3.5 pJ because it pays a high price in data movement for inputs and weights. An average ISAAC operation consumes 1.8 pJ because it pays a moderate price in data movement for inputs (weights are in-situ) and a high price for ADC. An average Eyeriss [23] operation consumes 1.67 pJ because of an improved dataflow to maximize reuse. The innovations in Newton push the analog architecture closer to the ideal neuron by consuming 0.85 pJ per operation. Relative to ISAAC, Newton achieves a 77% decrease in power, a 51% decrease in energy, and a 2.2$\times$ increase in throughput/area.

2.2 Background

2.2.1 Workloads

We consider different CNNs presented in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of image classification for the ImageNet [122] dataset. The suite of benchmarks considered in this chapter is representative of the various dataflows in such image classification networks. For example, Alexnet is the simplest of CNNs with a reasonable accuracy, where a few convolution layers at the start extract features from the image, followed by fully-connected layers that classify the image. The other networks were

designed with a similar structure but were made deeper and wider with more parameters. For example, MSRA Prelu-net [52] has 14 more layers than Alexnet [79] and has 330 million parameters, which is 5.5× higher than Alexnet. On the other hand, residual nets have forward connections with hops; that is, the output of a layer is passed on to not only the next layer but subsequent layers. Even though the number of parameters in Resnets [53] is much lower, these networks are much deeper and have a different dataflow, which changes the buffering requirements in accelerator pipelines.

2.2.2 The Landscape of CNN Accelerators

2.2.2.1 Digital Accelerators

The DianNao [22] and DaDianNao [24] accelerators were among the first to target deep convolutional networks at scale. DianNao designs the digital circuits for a basic NFU (Neural Functional Unit). DaDianNao is a tiled architecture where each tile has an NFU and eDRAM banks that feed synaptic weights to that NFU. DaDianNao uses many tiles on many chips to parallelize the processing of a single network layer. Once that layer is processed, all the tiles then move on to processing the next layer in parallel. Recent papers, e.g., Cnvlutin [5], have modified DaDianNao, so the NFU does not waste time and energy processing zero-valued inputs. EIE [130] and Minerva [121] address sparsity in the weights. Eyeriss [23] and ShiDianNao [31] improve the NFU dataflow to maximize operand reuse. A number of other digital designs [42], [71], [88] have also emerged in the past year.

2.2.2.2 Analog Accelerators

Two CNN accelerators introduced in the past year, ISAAC [129] and PRIME [25], have leveraged memristor crossbars to perform dot product operations in the analog domain. We will focus on ISAAC here because it out-performs PRIME in terms of throughput, accuracy, and ability to handle signed values. ISAAC is also able to achieve nearly 8× and 5× higher throughput than digital accelerators DaDianNao and Cnvlutin, respectively.

2.2.3 ISAAC

2.2.3.1 The Pipeline of Memristive Crossbars

In ISAAC, memristive crossbar arrays are used to perform analog dot-product operations. Neuron inputs are provided as voltages to wordlines; neuron weights are represented by preprogrammed cell conductances; and neuron outputs are represented by the currents in each bitline. The neuron outputs are processed by an ADC and shift-and-add circuits. They are then sent as inputs to the next layer of neurons. As shown in Figure 2.1, ISAAC is a tiled architecture; one or more tiles are dedicated to processing one layer of the neural network. Neuron outputs are propagated from tile to tile until all network layers have been processed to complete inference for one image.

2.2.3.2 Tiles, IMAs, and Crossbars

An ISAAC chip consists of many tiles connected in a mesh topology (Figure 2.1). Each tile includes an eDRAM buffer that supplies inputs to In-situ Multiply Accumulate (IMA) units. The IMA units consist of memristor crossbars that perform the dot-product computation, ADCs, and shift-and-add circuits that accumulate the digitized results. With a design space exploration, the tile is provisioned with an optimal number of IMAs, crossbars, ADCs, etc. Within a crossbar, a 16-bit weight is stored 2 bits per cell, across 8 columns. A 16-bit input is supplied as voltages over 16 cycles, 1 bit per cycle, using a trivial Digital-to-Analog Converter (DAC) array. The partial outputs are shifted and added across 8 columns and across 16 cycles to give the output of $16b \times 16b$ MAC operations. Thus, there are two levels of pipelining in ISAAC: (1) the intratile pipeline, where inputs are read from eDRAM, processed by crossbars in 16 cycles, and aggregated, (2) the intertile pipeline, where neuron outputs are transferred from one layer to the next. The intratile pipeline has a cycle time of 100 ns, matching the latency for a crossbar read. Inputs are sent to a crossbar in an IMA using an input h-tree network. The input h-tree has sufficient bandwidth to keep all crossbars active without bubbles. Each crossbar has a dedicated ADC operating at 1.28 $GSample/s$ shared across its 128 bitlines to convert the analog output to digital in 100 ns. An h-tree network is then used to collect digitized outputs from individual crossbars.

2.2.3.3 Crossbar Challenges

As with any new technology, a memristor crossbar has unique challenges, mainly in two respects. First, mapping a matrix onto a memristor crossbar array requires programming (or writing) cells with the highest precision possible. Second, real circuits deviate from ideal operation due to parasitics such as wire resistance, device variation, and write/read noise. All of these factors can cause the actual output to deviate from its ideal value. Recent work [59] has captured many of these details to show the viability of prototypes [60]. Section 2.2.4 summarizes some of these details.

2.2.4 Crossbar Implementations

This section discusses how crossbars can be designed to withstand noise effects in analog circuits.

2.2.4.1 Process Variation and Noise

Since an analog crossbar uses the actual conductance of individual cells to perform computation, it is critical to do writes at maximum precision. We make two design choices to improve write precision. First, we equip each cell with an access transistor (1T1R cell) to precisely control the amount of write current going through it. While this increases area overhead, it eliminates sneak currents and their negative impact on write voltage variation [160]. Second, we use a closed-loop write circuit with current compliance that does many iterations of program-and-verify operations. Prior work has shown that such an approach can provide more precise states at the cost of increased write time, even with high process variation in cells [6].

In spite of a robust write process, a cell's resistance will still deviate from its normal value within a tolerable range. This range will ultimately limit either the number of levels in a cell or the number of simultaneously active rows in a crossbar. For example, if a cell write can achieve a resistance within Δr (Δr is a function of noise and parasitic), if l is the number of levels in a cell, and *rrange* is the max range of resistance of a cell, then we set the number of active rows to $rrange/(l.\Delta r)$ to ensure there are no corrupted bits at the ADC.

2.2.4.2 Crossbar Parasitic

While a sophisticated write circuit coupled with limited array size can help alleviate process variation and noise, an IR drop along rows and columns can also reduce crossbar accuracy. When a crossbar is being written during initialization, the access transistors in unselected cells shut off sneak currents, limiting the current flow to just the selected cells. However, when a crossbar operates in a compute mode in which multiple rows are active, the net current in the crossbar increases, and the current path becomes more complicated. With access transistors in every cell in the selected rows in ON state, a network of resistors is formed with every cell conducting varying current based on its resistance. As wire links connecting these cells have nonzero resistance, the voltage drop along rows and columns will impact the computation accuracy. Thus, a cell at the far end of the driver will see relatively lower read voltage compared to a cell closer to the driver. This change in voltage is a function of both wire resistance and the current flowing through wordlines and bitlines, which in turn is a function of the data pattern in the array. This problem can be addressed by limiting the DAC voltage range and doing data encoding to compensate for the IR drop [59]. Since the matrix being programmed into a crossbar is known beforehand, during the initialization phase of a crossbar, it is possible to account for voltage drops and adjust the cell resistance appropriately. Hu et al. [59] have demonstrated successful operation of a 256×256 crossbar with 5-bit cells even in the presence of thermal noise in the memristor, short noise in circuits, and random telegraphic noise in the crossbar. For this work, a conservative model with a 128×128 crossbar with 2-bit cells and 1-bit DAC emerges as an ideal design point in most experiments.

2.3 The Newton Architecture

Similar to ISAAC, Newton employs a tiled architecture, where every tile is composed of several IMAs. Newton is targeted for mobile platforms (self-driving cars and phones) and datacenters. In both cases, the chips will be continuously fed with inputs from several cameras or users – hence our focus on throughput and energy efficiency. We first discuss three innovations that are applied within a Newton IMA, followed by three innovations that are applied within a Newton tile. The overarching theme in these ideas is the reduction in ADC and resource requirements.

2.3.1 Intra-IMA Optimizations

2.3.1.1 Mapping Constraints

ISAAC did not place any constraints on how a neural network can be mapped to its many tiles and IMAs. As a result, its resources, notably the HTree and buffers within an IMA, are provisioned to handle the worst case. This has a negative impact on power and area efficiency. Instead, we place constraints on how the workload is mapped to IMAs. While this inflexibility can waste a few resources, we observe that it also significantly reduces the HTree size and hence area per IMA. The architecture is still general-purpose; that is, arbitrary CNNs can be mapped to Newton.

2.3.1.2 Bit Interleaved Crossbars

The layout of an IMA with 16 crossbars is shown in Figure 2.2. We colocate an ADC with each crossbar. The digitized outputs are then sent to the IMA's output register via an HTree network. While ISAAC was agnostic to how a single synaptic weight was scattered across multiple bitlines, we adopt the following approach to boost efficiency. A 16-bit weight is scattered across eight 2-bit cells; each cell is placed in a different crossbar. Therefore, crossbars 0 and 8 are responsible for the least significant bits of every weight, and crossbars 7 and 15 are responsible for the most significant bits of every weight. We also embed the shift-and-add units in the HTree. So the shift-and-add unit at the leaf of the HTree adds the digitized 9-bit dot-product results emerging from two neighboring crossbars. Because the operation is a shift-and-add, it produces an 11-bit result. The next shift-and-add unit takes two 11-bit inputs to produce a 13-bit input, and so on. In addition to being an efficient pipeline and lowering the HTree widths, this approach lends itself to the heterogeneity measures we introduce shortly.

2.3.1.3 An IMA as an Indivisible Resource

We further introduce the constraint that an IMA cannot be shared by multiple network layers. If multiple layers were to share an IMA, we would need multiple HTree networks to support the simultaneous aggregation of multiple neurons. This introduces a nontrivial area overhead and offers a flexibility that is rarely useful to the workloads we examined. An IMA, therefore, represents an indivisible resource unit that is allocated to a network layer. We also restrict the number of inputs (128) to an IMA to boost input sharing and

reduce HTree bandwidth and input buffering requirements. By placing these constraints, depending on the sizes of network layers, a few crossbars in a few IMAs go unutilized, but for our workloads, this is minor and not enough to offset the power/area benefit of a smaller HTree.

2.3.1.4 Adaptive ADCs

We will first take a closer look at the dot-product being performed within an IMA. As with prior work, we are performing dot-products on 16-bit fixed-point neurons and weights, with a fixed-point scaling factor of 2^{10}. A 16-bit weight is spread across 8 cells (each in a different crossbar). The 16-bit input is iteratively fed as 16 1-bit inputs. In a single iteration, a crossbar column is performing a dot-product involving 128 rows, 1-bit inputs, and 2-bit cells; it, therefore, produces a 9-bit result requiring a 9-bit ADC.[1] We must shift and add the results of 8 such columns, yielding a 23-bit result. These results must also be shifted and added across 16 iterations, finally yielding a 39-bit output. Once the scaling factor is applied, the least significant 10 bits are dropped. The most significant 13 bits represent an overflow that cannot be captured in the 16-bit result, so they are effectively used to clamp the result to a maximum value.

What is of note here is that the output from every crossbar column in every iteration is being resolved with a high-precision 9-bit ADC, but many of these bits contribute to either the 10 least significant bits or the 13 most significant bits that are eventually going to be ignored. This is an opportunity to lower the ADC precision and ignore some bits, depending on the column and the iteration being processed. Figure 2.3 shows the number of relevant bits emerging from every column in every iteration.

The ADC accounts for a significant fraction of IMA power. When the ADC is operating at a lower resolution, it has less work to do. In every 100 ns iteration, we tune the resolution of a Successive Approximate Register (SAR) ADC to match the requirement in Figure 2.3. Thus, the use of adaptive ADCs helps reduce IMA power while having no impact on performance. We are also ignoring bits that do not show up in a 16-bit fixed-point result, so we are not impacting the functional behavior of the algorithm, thus having zero impact on algorithm accuracy.

---

[1] ISAAC introduces a data encoding that can reduce the ADC resolution by 1 bit [129].

A SAR ADC does a binary search over the input voltage to find the digital value, starting from the Most Significant Bit (MSB). A bit is set to 1, and the resulting digital value is converted to analog and compared with the input voltage. If the input voltage is higher, the bit is set to 1, the next bit is changed, and the process repeats. If the number of bits to be sampled is reduced, the circuit can ignore the latter stages. The ADC simply gates off its circuits until the next sample is provided. While we could increase sampling frequency, this is not helpful because we can only run as fast as the slowest stage in the ISAAC pipeline. It is important to note that the ADC starts the binary search from the MSB, and thus it is not possible to sample just the lower significant bits of an output neuron without knowing the MSBs. But in this case, we have a unique advantage: if any of the MSBs to be truncated is 1, then the output neuron value is clamped to the highest value in the fixed point range. Thus, in order to sample a set of Least Significant Bits (LSBs), the ADC starts the binary search with the LSB+1 bit. If that comparison yields true, it means at least one of the MSB bits is 1. This signal is sent across the HTree, and the output is clamped.

In conventional SAR ADCs [148], a third of the power is dissipated in the capacitive DAC (CDAC), a third in digital circuits, and a third in other analog circuits. The MSB decision, in general, consumes more power because it involves charging up the CDAC at the end of every sampling iteration. Recent trends show CDAC power diminishing due to the use of tiny unit capacitances (about 2fF) and innovative reference buffer designs, leading to ADCs consuming more power in analog and digital circuits [80], [100]. The Adaptive ADC technique is able to reduce energy consumption irrespective of the ADC design since it eliminates both LSB and MSB tests across the 16 iterations.

Note that by interleaving a weight across multiple crossbars, we must adapt the ADC once every 100 ns iteration, and not for every new input sample. It is also worth pointing out that the adaptive ADC technique is compatible with the encoding used by ISAAC to reduce ADC resolution by 1 bit.

2.3.1.5 Divide and Conquer Multiplication

We now introduce another technique that reduces pressure on ADC usage and hence ADC power. This can be done with linear algebra optimizations within and outside an

IMA. Here, we discuss a divide and conquer strategy at the bit level (Karatsuba's technique), applied within an IMA. Later, in Section 2.3.2.3, we apply a similar strategy at matrix granularity outside an IMA (Strassen's technique).

A classic multiplication approach for two n-bit numbers has a complexity of $O(n^2)$ where each bit of a number is multiplied with n-bits of the other number, and the partial results are shifted and added to get the final 2n-bit result. A similar approach is taken in ISAAC. However, the time complexity is $O(n)$ since the multiplication of 1-bit of input with n-bits of weight happens in parallel.

Karatsuba's divide and conquer algorithm manages to reduce the complexity from $O(n^2)$ to $O(n^{1.5})$. As shown in Figure 2.4, it divides the numbers into two halves of n/2 bits, MSB bits, and LSB bits, and instead of performing four smaller n/2-bit multiplications, it calculates the result with two n/2-bit multiplications and one (n/2 + 1)-bit multiplication.

In baseline ISAAC, the product of input X and weight W is performed on 8 crossbars in 16 cycles (since each weight is spread across 8 cells in 8 different crossbars and the input is spread across 16 iterations). In the example in Figure 2.4, W_0X_0 is performed on four crossbars in 8 iterations (since we are dealing with fewer bits for weights and inputs). The same is true for W_1X_1. A third set of crossbars stores the weights $(W_1 + W_0)$ and receives the precomputed inputs $(X_1 + X_0)$. This computation is spread across 5 crossbars and 9 iterations. We see that the total amount of work has reduced by 15%.

To implement this algorithm, we modify the IMA, as shown in Figure 2.5. The changes are localized to a single mat. Each mat now has two crossbars that share the DAC and ADC. Given the size of the ADC, the extra crossbar per mat has a minimal impact on the area. The left crossbars in four of the mats now store W_0; the left crossbars in the other four mats store W_1; and the right crossbars in five of the mats store $W_0 + W_1$; the right crossbars in three of the mats are unused. In the first 8 iterations, the 8 ADCs are used by the left crossbars. In the next 9 iterations, 5 ADCs are used by the right crossbars. The main objective here is to lower power by reducing the use of the ADC.

There are a few drawbacks as well. A computation now takes 17 iterations instead of 16. The IMA area increases because the HTree must send inputs X_0 and X_1 in parallel, each mat has an additional crossbar, the output buffer is larger to store subproducts, and 128 1-bit full adders are required to compute $(X_1 + X_0)$. Again, given that the ADC is the

primary bottleneck, these other overheads are relatively minor.

Divide and conquer can be recursively applied further. When applied again, the computation keeps 8 ADCs busy in the first 4 iterations and 6 ADCs in the next 10 iterations. This is a 28% reduction in ADC use and a 13% reduction in execution time. But, we pay an area penalty because 20 crossbars are needed per IMA.

In Figure 2.3, it is evident that most ADC undersampling is done for the subproducts X_0W_0 and X_1W_1, thus allowing the previous technique to be combined with Karatsuba's algorithm.

2.3.2 Intratile Optimizations

The previous subsection focused on techniques to improve an IMA; we now shift our focus to the design of a tile. We first reduce the size of the eDRAM buffer that feeds all the IMAs in a tile. We then create heterogeneous tiles that suit convolutional and fully-connected layers. Finally, we use a divide and conquer approach at the tile level.

2.3.2.1 Reducing Buffer Sizes

Because ISAAC did not place constraints on how layers are mapped to crossbars and tiles, the eDRAM buffer was sized to 64 KB to accommodate the worst-case requirements of workloads. Here, we design mapping techniques that reduce storage requirements per tile and move that requirement closer to the average-case.

To explain the impact of mapping on buffering requirements, consider the convolutional layer shown in Figure 2.6a. Once a certain number of inputs are buffered (shown in green and pink), the layer enters steady-state; every new input pixel allows the convolution to advance by another step. The buffer size is a constant as the convolution advances (each new input evicts an old input that is no longer required). In every step, a subset of the input buffer is fed as input to the crossbar to produce one pixel in each of the many output feature maps. If the crossbar is large, it is split across two tiles, as shown in Figure 2.6a. The split is done so that Tile 1 manages the green buffer and green inputs, and Tile 2 manages the pink buffer and pink inputs. Such a split means that inputs do not have to be replicated on both tiles, and buffering requirements are low.

Now, consider an early convolutional layer. Early convolutional layers have more work to do than later layers since they deal with larger feature maps. In ISAAC, to make the

pipeline balanced, early convolutional layers are replicated, so their throughput matches those of later layers. Figure 2.6b replicates the crossbar; one is responsible for every odd pixel in the output feature maps, while the other is responsible for every even pixel. In any step, both crossbars receive very similar inputs. So the same input buffer can feed both crossbars.

If a replicated layer is large enough that it must be spread across (say) 4 tiles, we have two options. Figure 2.6 c and d show these two options. If the odd computation is spread across two tiles (1 and 2) and the even computation is spread across two different tiles (3 and 4), the same green inputs have to be sent to Tile 1 and Tile 3; that is, the input buffers are replicated. Instead, as shown in Figure 2.6d, if we colocate the top quadrant of the odd computation and the top quadrant of the even computation in Tile 1, the green inputs are consumed entirely within Tile 1 and do not have to be replicated. This partitioning leads to the minimum buffer requirement.

The bottom line from this mapping is that when a layer is replicated, the buffering requirements per neuron and per tile are reduced. This is because multiple neurons that receive similar inputs can reuse the contents of the input buffer. Therefore, heavily replicated (early) layers have lower buffer requirements *per tile* than lightly replicated (later) layers. If we mapped these layers to tiles as shown in Figure 2.7a, the worst-case buffering requirement goes up (64 KB for the last layer), and early layers end up underutilizing their 64 KB buffer. To reduce the worst-case requirement and the underutilization, we instead map layers to tiles, as shown in Figure 2.7b. Every layer is finely partitioned and spread across 10 tiles, and every tile processes part of a layer. *By spreading each layer across many tiles, every tile can enjoy the buffering efficiency of early layers.* By moving every tile's buffer requirement closer to the average-case (21 KB in this example), we can design a Newton tile with a smaller eDRAM buffer (21 KB instead of 64 KB) that achieves higher overall computational efficiency. This does increase the intertile neuron communication, but this has a relatively small impact on computational efficiency.

2.3.2.2 Different Tiles for Convolutions and Classifiers

While ISAAC uses the same homogeneous tile for the entire chip, we observe that convolutional layers have very different resource demands than fully-connected classi-

fier layers. The classifier or Fully Connected (FC) layer has to aggregate a set of inputs required by a set of crossbars; the crossbars then perform their computation; the inputs are discarded, and a new set of inputs is aggregated. This results in the following properties for the classifier layer:

1. The classifier layer has a high communication-to-compute ratio, so the router bandwidth puts a limit on how often the crossbars can be busy.
2. The classifier also has the highest synaptic weight requirement because every neuron has private weights.
3. The classifier has low buffering requirements – an input is seen by several neurons in parallel, and the input can be discarded right after.

We, therefore, design special Newton tiles customized for classifier layers that:

1. have a higher crossbar-to-ADC ratio (4:1 instead of 1:1),
2. operate the ADC at a lower rate (10 Msamples/sec instead of 1.2 Gsamples/sec), and
3. have a smaller eDRAM buffer size (4 KB instead of 16 KB).

For small-scale workloads that are trying to fit on a single chip, we would design a chip where many of the tiles are conv-tiles, and some are classifier-tiles (a ratio of 1:1 is a good fit for most of our workloads). For large-scale workloads that use multiple chips, each chip can be homogeneous; we use roughly an equal number of conv-chips and classifier-chips. The results consider both cases.

2.3.2.3 Strassen's Algorithm

A divide and conquer approach can also be applied to matrix-matrix multiplication. Note that all of our computations are vector-matrix multiplications. But when a layer is replicated, multiple input vectors are being fed to the same matrix of weights, so replicated layer computations are matrix-matrix multiplications. By partitioning each matrix X and W into 4 submatrices, we can express matrix-matrix multiplication in terms of multiplications of submatrices. The typical algorithm, assumed in ISAAC, would require 8 submatrix multiplications, followed by an aggregation step. But as shown in Figure 2.8, linear algebra manipulations can perform the same computation with 7 submatrix multiplications, with appropriate pre- and postprocessing. Similar to Karatsuba's algorithm,

this has the advantage of reducing ADC usage and power. As shown in Figure 2.9, the computations ($P_0 - P_6$) in Strassen's algorithm are mapped to 7 IMAs in the tile. The 8th IMA can be allocated to another layer's computation. While both divide and conquer algorithms (Karatsuba's within an IMA and Strassen's within a tile) are highly effective for a crossbar-based architecture, they have very little impact on other digital accelerators. For example, these algorithms may impact the efficiency of the NFUs in DaDianNao, but the DaDianNao area is dominated by eDRAM banks and not NFUs. In fact, Strassen's algorithm can lower DaDianNao's efficiency because buffering requirements may increase. On the other hand, the computations in Newton are linked with expensive ADCs, so efficient computation does noticeably impact overall efficiency. Further, some of the preprocessing for these algorithms is performed when installing weights on the Newton chip but has to be performed on-the-fly for digital accelerators.

2.3.3 Summary

We have introduced six ideas that improve the power and area efficiency of crossbar-based architectures. Three of these ideas are applied within an IMA, and three within a tile. Two of the ideas rely on divide-and-conquer numeric algorithms – Karatsuba's within an IMA and Strassen's within a tile. Two of the ideas rely on heterogeneous hardware to meet varying computation requirements – an ADC that adapts to the precision requirement of every bitline operation in every iteration and tiles that customize resource provisioning for classifier layers. Two of the ideas introduce constraints on how network layers are mapped to resources: (1) by introducing constraints within an IMA, we reduce HTree size, and (2) by spreading a layer across many tiles, we reduce the eDRAM buffer requirements. Note again that these constraints do not limit generality, and any CNN can be mapped to a sufficiently large collection of Newton tiles.

2.4 Methodology

2.4.1 Modeling Area and Energy

For modeling the energy and area of the eDRAM buffers and on-chip interconnect like the HTree and tile bus, we use Cacti 6.5 [98] at 32 nm. The area and energy model of a memristor crossbar is based on [59]. We adapt the area and energy of shift-and-add circuits, max/average pooling block, and sigmoid operation similar to the analysis in

DaDianNao [24] and ISAAC [129]. We avail the same HyperTransport serial link model for off-chip interconnects as used by DaDianNao and ISAAC. The router area and energy are modeled using Orion 2.0 [68]. While our buffers can also be implemented with SRAM, we use eDRAM to make an apples-to-apples comparison with the ISAAC baseline. Newton is only used for inference, with a delay of 16.4 ms to preload weights in a chip.

In order to model the ADC energy and area, we use a recent survey [100] of ADC circuits published in different circuit conferences. The Newton architecture uses the same 8-bit ADC [80] at 32 nm as used in ISAAC, partly because it yields the best configuration in terms of area/power and meets the sampling frequency requirement, and partly because it can be reconfigured for different resolutions. This is at the cost of a minimal increase in the area of the ADC. We scale the ADC power with respect to different sampling frequency according to another work by Kull et al. [80]. The SAR ADC has six different components: comparators, asynchronous clock logic, sampling clock logic, data memory and state logic, reference buffer, and capacitive DAC. The ADC power for different sampling resolution is modeled by gating off the other components except for the sampling clock.

We consider a 1-bit DAC as used in ISAAC because it is relatively small and has a high Signal-to-Noise Ratio (SNR) value. Since DAC is used in every row of the crossbar, a 1-bit DAC improves the area efficiency. The key parameters in the architecture that largely contribute to our analysis are reported in Table 2.1.

This work considers recent workloads with state-of-the-art accuracy in image classification tasks (summarized in Table 2.2). We create an analytic model for a Newton pipeline within an IMA and within a tile and map the suite of benchmarks, making sure that there are no structural hazards in any of these pipelines. We consider network bandwidth limitations in our simulation model to estimate throughput. Since ISAAC is a throughput architecture, we make an iso-throughput comparison of the Newton architecture with ISAAC for the different intra-IMA or intratile optimizations. Since the dataflow in the architecture is bounded by the router bandwidth, in each case, we allocate enough resources until the network saturates to create our baseline model. For subsequent optimizations, we retain the same throughput. Similar to ISAAC, data transfers between tiles on-chip and on the HT link across chips have been statically routed to make it conflict-free. Like ISAAC, the latency and throughput of Newton for the given benchmarks can be calculated analytically

using a deterministic execution model. Since there are not any run-time dependencies on the control flow or data flow of the deep networks, analytical estimates are enough to capture the behavior of cycle-accurate simulations.

We create a similar model for ISAAC, taking into considerations all the parameters mentioned in their work.

2.4.2 Design Points

The Newton architecture can be designed by optimizing one of the following two metrics:

1. **CE:** Computational Efficiency, which is the number of fixed-point operations (in Giga Operations (GOPS)) performed per second per unit area, $GOPS/(s \times mm^2)$.
2. **PE:** Power Efficiency, which is the number of fixed-point operations performed per second per unit power, $GOPS/(s \times W)$.

For every considered innovation, we model Newton for a variety of design points that vary crossbar size, number of crossbars per IMA, and number of IMAs per tile. In most cases, the same configurations emerged as the best. We, therefore, focus most of our analysis on this optimal configuration that has 16 IMAs per tile, where each IMA uses 16 crossbars to process 128 inputs for 256 neurons. We report the area, power, and energy improvement for all the deep neural networks in our benchmark suite.

2.5 Results

As we move through the results, we will incrementally add each innovation. Each reported improvement is relative to the Newton architecture, with innovations described until that point.

2.5.1 Constrained Mapping for Compact HTree

The Newton architecture takes the baseline analog accelerator ISAAC and incrementally applies a series of techniques. We first observe that the ISAAC IMA is designed with an overprovisioned HTree that can handle a worst-case mapping of the workload. We imposed the constraint that an IMA can only handle a single layer and a maximum of 128 inputs. This restricts the width of the HTree, promotes input sharing, and enables the reduction of partial neuron values at the junctions of the HTree. While this helps

shrink the size of an IMA, it suffers from crossbar underutilization within an IMA. We consider different IMA sizes, ranging from 128×64, which supplies the same 128 neurons to 4 crossbars to get 64 output neurons, to 8192×1024. Figure 2.10 plots the average underutilization of crossbars across the different workloads in the benchmark suite. For larger IMA sizes, the underutilization is quite significant. Larger IMA sizes also result in complex HTree networks. Therefore, a moderately sized IMA that processes 128 inputs for 256 neurons has high computational efficiency and low crossbar underutilization. For this design, the underutilization is only 9%. Figure 2.11 quantifies how our constrained mapping and compact HTree improve the area, power, and energy per workload. In short, our constraints have improved area efficiency by 37% and power/energy efficiency by 18% while leaving only 9% of crossbars underutilized.

2.5.2 Heterogeneous ADC Sampling

The heterogeneous sampling of outputs using adaptive ADCs has a big impact on reducing the power profile of the analog accelerator. In one iteration of 100 ns, at max 4 ADCs work at the max resolution of 8-bits. The power supply to the rest of the ADCs can be reduced. We measure the reduction of area, power, and energy with respect to the new IMA design with the compact HTree. Since ADC contributed to 49% of the chip power in ISAAC, reducing the oversampling of ADC reduces power requirement by 15% on average. The area efficiency improves as well since the output-HTree now carries 16 bits instead of unnecessarily carrying 39 bits of the final output. The improvements are shown in Figure 2.12.

2.5.3 Karatsuba's Algorithm

We further try to reduce the power profile with divide-and-conquer within an IMA. Figure 2.13 shows the impact of recursively applying the divide-and-conquer technique multiple times. Applying it once is nearly as good as applying it twice and much less complex. Therefore, we focus on a single divide-and-conquer step. Improvements are reported in Figure 2.14. Energy efficiency improves by almost 25% over the previous design point because ADCs end up being used 75% of the time in the 1700 ns window. However, this comes at the cost of a 6.4% reduction in area efficiency because of the need for more crossbars and an increase in HTree bandwidth to send the sum of inputs.

2.5.4 eDRAM Buffer Requirements

In Figure 2.15, we report the buffer requirement per tile when the layers are spread across many tiles. We consider this for a variety of tile/IMA configurations. Even though this plot pertains to the workloads considered in this chapter, as long as the image sizes processed by the network remain 256×256 or below, it is highly representative of all workloads. The image size has a linear impact on the buffering requirement. This observation leads to the choice of a 16 KB buffer instead of the 64 KB used in ISAAC, a 75% reduction. Figure 2.16 shows a 6.5% average improvement in area efficiency because of the buffer reduction.

2.5.5 Conv-Tiles and Classifier-Tiles

Figure 2.17 plots the decrease in power requirement when FC tiles are operated at $8\times$, $32\times$, and $128\times$ slower than the conv tiles. None of these configurations lowers the throughput as the FC layer is not on the critical path. Since ADC power scales linearly with sampling resolution, the power profile is lowest when the ADCs work $128\times$ slower. This leads to a 50% lower peak power on average. In Figure 2.18, we plot the increase in area efficiency when multiple crossbars share the same ADC in FC tiles. The underutilization of FC tiles provides room for making them storage efficient, saving on average 38% of the chip area. We do not increase the ratio beyond four because the multiplexer connecting the crossbars to the ADC becomes complex. Resnet does not gain much from the heterogeneous tiles because it needs relatively fewer FC tiles.

2.5.6 Strassen's Algorithm

Strassen's optimization is especially useful when large matrix multiplication can be performed in the conv layers without much wastage of crossbars. This provides room for the decomposition of these large matrices, which is the key part of Strassen's technique. We note that Resnet has high wastage when using larger IMAs, and thus does not benefit at all from this technique. Overall, Strassen's algorithm increases energy efficiency by 4.5%, as seen in Figure 2.19.

2.5.7 Putting it All Together

Figure 2.20 plots the incremental effect of each of our techniques on peak computational and power efficiency of DaDianNao, ISAAC, and Newton. We do not include the heterogeneous FC tile in this plot because it is noncritical and forcibly operated slowly; as a result, its peak throughput is lower by definition. We see that both adaptive ADC and Divide & Conquer play a significant role in increasing PE. While the impact of Strassen's technique is not visible in this graph, it manages to free up resources (1 every 8 IMA) in a tile, thus providing room for more compact mapping of networks and reducing ADC utilization.

Figure 2.21 shows a per-benchmark improvement in area efficiency and the contribution of each of our techniques. The compact HTree and the FC tiles are the biggest contributors. Figure 2.22 similarly shows a breakdown for a decrease in power envelope, and Figure 2.23 does the same for improvement in energy efficiency. Multiple innovations (HTree, adaptive ADC, Karatsuba, FC tiles) contribute equally to the improvements. We also observed that the Adaptive ADC technique's improvement is not very sensitive to the ADC design. We evaluated ADCs where the CDAC power dissipates 10% and 27% of ADC power; the corresponding improvements with the Adaptive ADC were 13% and 12%, respectively.

2.5.8 Comparison with Google's Tensor Processing Unit (TPU)

Figure 2.24 compares the 8-bit version of Newton with Google's TPU architecture. We scale the area such that the die area is the same for both the architectures; that is, an iso-area comparison. For TPU, we perform batch processing enough to not exceed the latency target of 7ms as demanded by most application developers. Since the Newton pipeline is deterministic and as its crossbars are statically mapped to different layers, the latency of images is always the same irrespective of batch size, which is comfortably less than 7ms for all the evaluated benchmarks. We also model TPU with GDDR5 memory to allocate sufficient bandwidth.

Figure 2.24 shows the throughput and energy improvement of Newton over TPU for various benchmarks. Newton has an average improvement of 10.3× in throughput and 3.4× in energy over TPU. In terms of computational efficiency (CE) calculated using peak

throughput, Newton is 12.3× better than TPU. However, when operating on the FC layer, due to idle crossbars in Newton, this advantage reduces to 10.3× for actual workloads.

When considering power efficiency (PE) calculated using peak throughput and area, although Newton is only 1.6× better than TPU, the actual benefit goes up for real workloads, increasing it to 3.4×. This is because of TPU's low memory bandwidth coupled with reduced batch size for some workloads. As we discussed earlier, the batch size in TPU is adjusted to meet the latency target. Since large batch size alleviates memory bandwidth problem, reducing it to meet the latency target directly impacts power efficiency due to more GDDR fetches and idle processing units.

From Figure 2.24, it can also be noted that the throughput improvement of Alexnet and Resnet is not as much as the other benchmarks because of their relatively small networks. This increases the batch size, improving the data locality for FC layer weights. On the other hand, the MSRA3 benchmark has higher energy consumption than other workloads because, for MSRA3, TPU can process only one image per batch. This dramatically increases TPU's idle time while fetching a large number of weights for the FC layers. In short, Newton's in-situ computation achieves superior energy and performance values over TPU as the proposed design limits data movement while reducing analog computation overhead.

2.6 Related Works

2.6.1 Digital Accelerators

Accelerators have been designed for several machine learning algorithms to process images [79], [81] and speech [48]. DianNao [22] is one of the early works that highlight the memory wall and designs an accelerator for CNNs and DNNs that exploit data reuse with tiling. The inherent sharing of weights in CNNs is explored in ShiDianNao [31]. Origami [20] is an ASIC architecture that tries to reduce Input/Output (I/O) Bandwidth for CNNs. The PuDianNao accelerator [87] focuses on a range of popular machine learning algorithms and constructs common computational primitives. CNNs have also been mapped to Field Programmable Gate Arrays (FPGAs) [26], [34], [35]. The Convolution Engine [119] identifies data flow and locality patterns that are common among kernels of image/video processing and creates custom units that can be programmed for different applications. Minerva [121] automates a codesign approach across algorithm, architecture,

and circuit levels to optimize digital DNN hardware. RedEye [86] moves the processing of convolution layers to an image sensor's analog domain to reduce the computational burden. Neurocube [71] maps CNNs to 3D high-density high-bandwidth memory integrated with logic, forming a mesh of digital processing elements. The sparsity in CNN parameters and activations are leveraged by EIE [130] to design an accelerator for sparse vector-matrix multiplication.

2.6.2 Analog Accelerators

Memristors [138] have been primarily targeted as main memory devices [55], [150], [156]. Memristor crossbars have been recently proposed for analog dot product computations [74], [114]. Studies using SPICE models [142] show that an analog crossbar can yield higher throughput and lower power than a traditional HPC system. Mixed-signal computation capabilities of memristors have been used to speed up neural networks [75], [89], [90], [117]. In-situ memristor computation has also been used for perceptron networks to recognize patterns in small scale images [157]. Bojnordi et al. design an accelerator for the Boltzmann machine [14], using in-situ memristor computation to estimate a sum of products of two 1-bit numbers and a 16-bit weight. A few works have attempted training on memristors [7], [117], [137]. Other emerging memory technologies (e.g., Phase Change Memory (PCM)) have also been used as synaptic weight elements [18], [140]. None of the above works are targeted at CNNs or DNNs. Chi et al. [25] propose PRIME, a morphable PIM structure for ReRAM based main memory with carefully designed peripheral circuits that allow arrays to be used as memory, scratchpads, and dot product engines for CNN workloads.

2.6.3 Hardware Neural Networks.

A number of works have designed hardware neural networks and neuromorphic circuits using analog computation [17], [66], [115], [123], [124], [143] to take advantage of faster vector-matrix multiplication [44], [120]. Neural network accelerators have also been built for signal processing [13] with configurable digital-analog models or for defect tolerant cores [32], [51], [144]. Neural models have been used to speed up approximate code [49] using limited precision analog hardware [136].

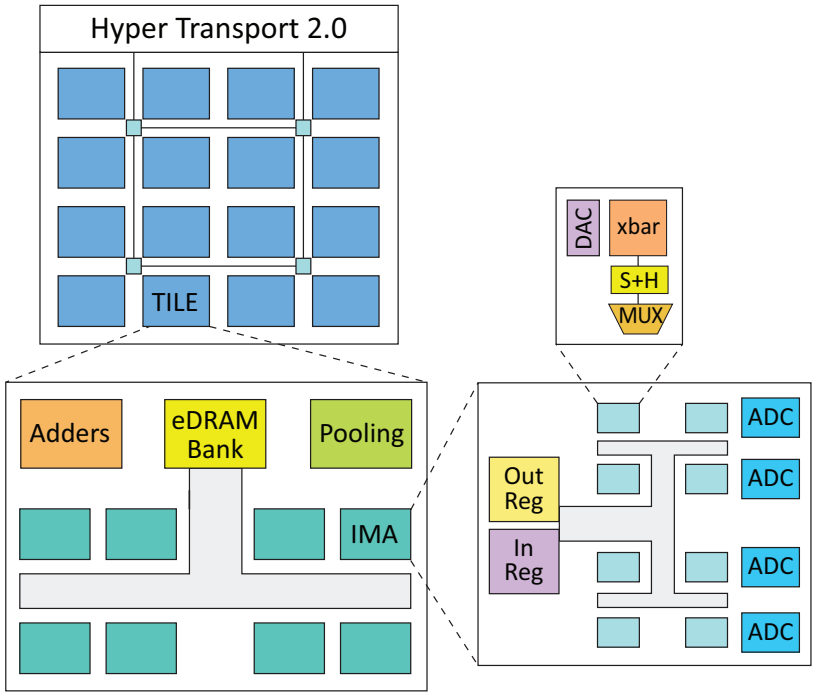

Figure 2.1. The ISAAC architecture.

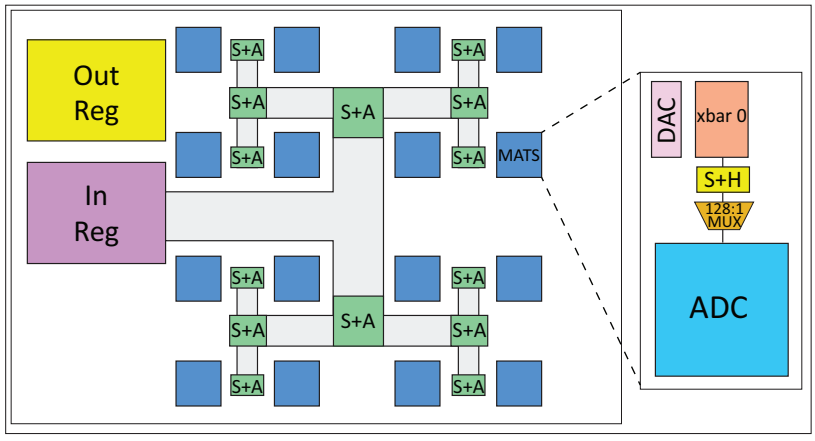

Figure 2.2. Microarchitecture of an IMA.

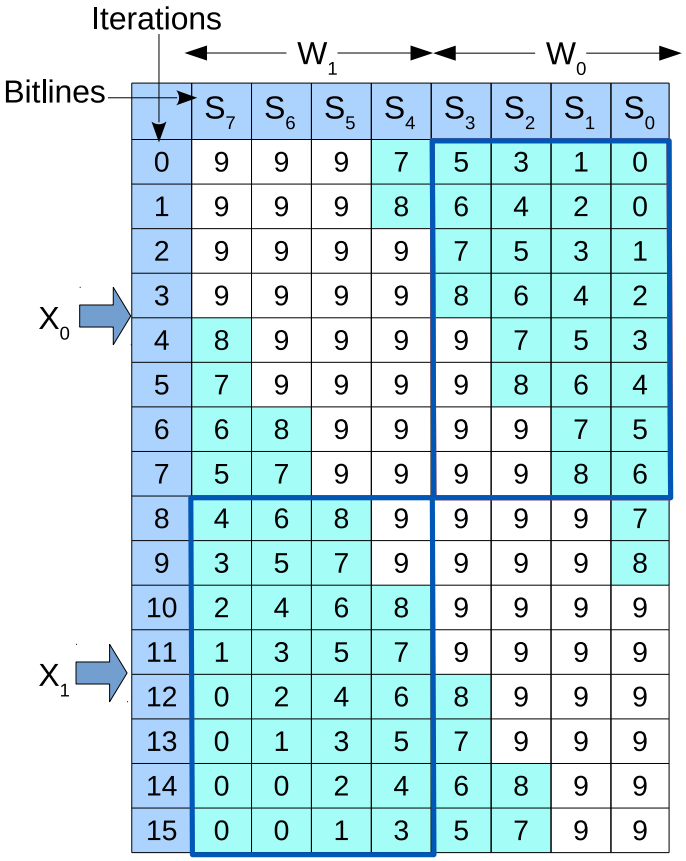

Figure 2.3. Heterogeneous ADC sampling resolution.

Divide & Conquer

$$W=2^{N/2}W_1+W_0 \quad X=2^{N/2}X_1+X_0$$

$$WX = 2^N W_1X_1 + 2^{N/2}(W_1X_0+W_0X_1) + W_0X_0$$

$$= 2^N W_1X_1 + 2^{N/2}[(W_1+W_0)(X_1+X_0)-(W_1X_1+W_0X_0)] + W_0X_0$$

$$= (2^N-1)W_1X_1 + 2^{N/2}(W_1+W_0)(X_1+X_0) + (1-2^{N/2})W_0X_0$$

Figure 2.4. Karatsuba's divide and conquer algorithm.

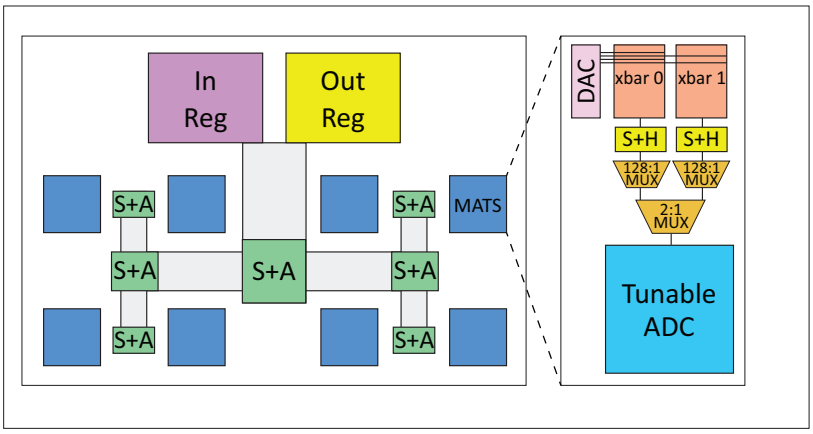

Figure 2.5. IMA supporting Karatsuba's algorithm.

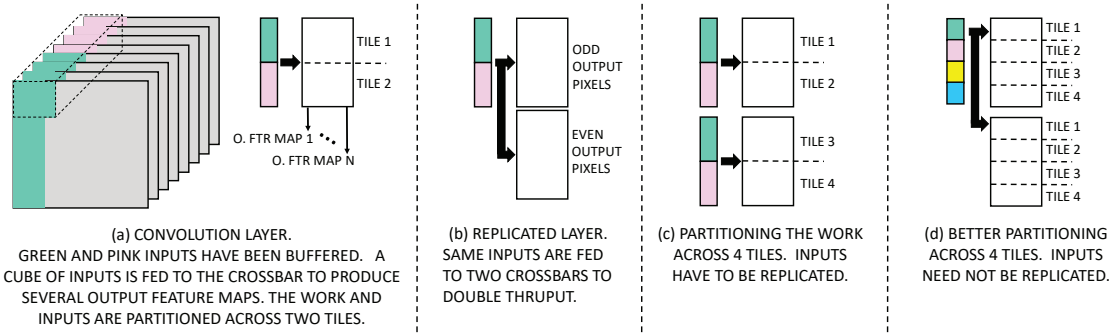

Figure 2.6. Mapping of convolutional layers to tiles.

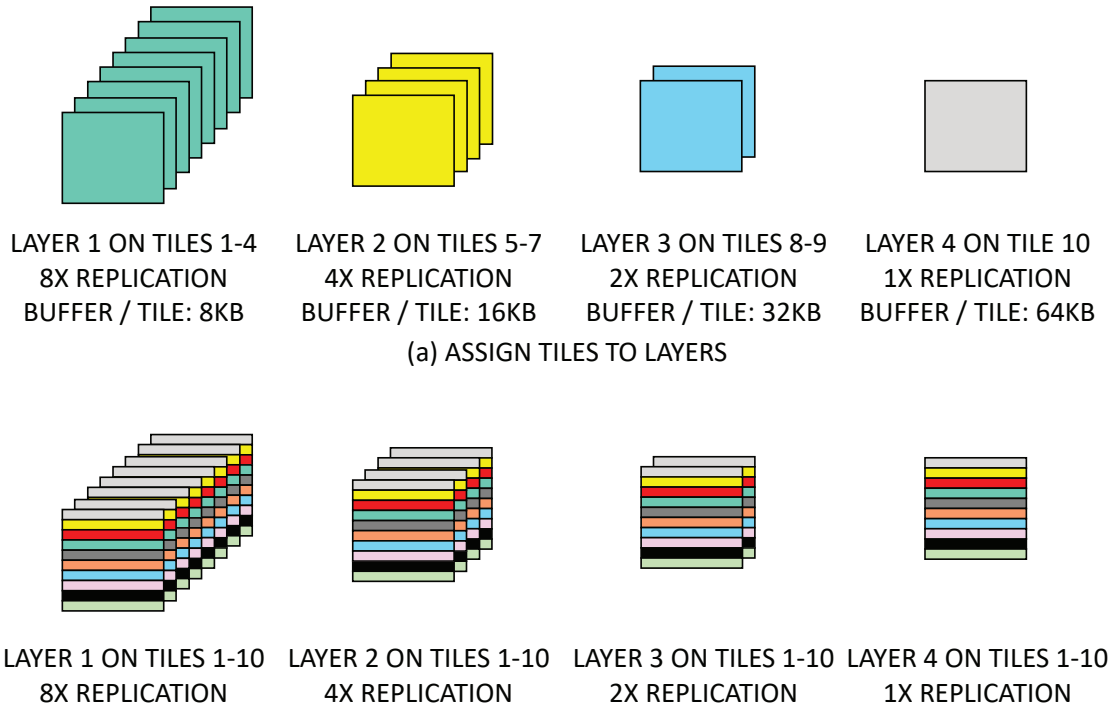

Figure 2.7. Mapping layers to tiles for small buffer sizes.

Strassen's Optimization

$$\begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix} = \begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix}$$

$P_0 \leftarrow W_{00} \times (X_{01} - X_{11})$

$P_1 \leftarrow (W_{00} + W_{01}) \times X_{11}$

$P_2 \leftarrow (W_{10} + W_{11}) \times X_{00}$

$P_3 \leftarrow W_{11} \times (X_{10} - X_{00})$

$P_4 \leftarrow (W_{00} + W_{11}) \times (X_{00} + X_{11})$

$P_5 \leftarrow (W_{01} - W_{11}) \times (X_{10} + X_{11})$

$P_6 \leftarrow (W_{00} - W_{10}) \times (X_{00} + X_{01})$

$Y_{00} = P_4 + P_3 - P_1 + P_5$

$Y_{01} = P_0 + P_1$

$Y_{10} = P_2 + P_3$

$Y_{11} = P_0 + P_4 - P_2 - P_6$

Figure 2.8. Strassen's divide and conquer algorithm for matrix multiplication.

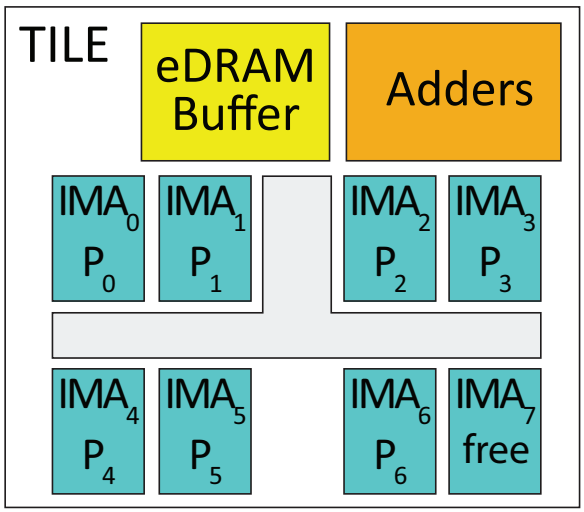

Figure 2.9. Mapping Strassen's algorithm to a tile.

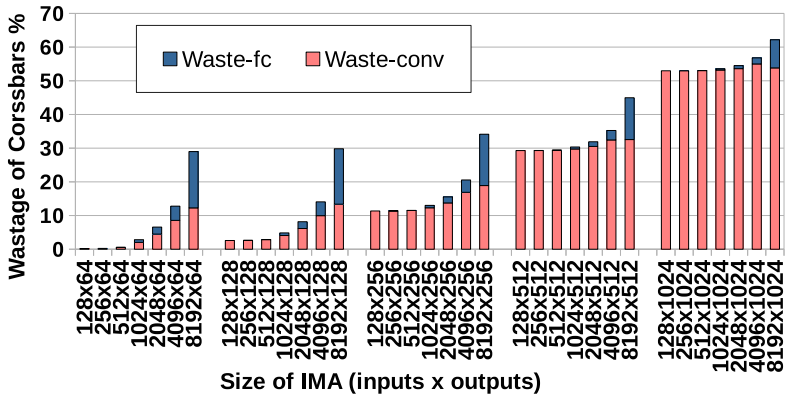

Figure 2.10. Crossbar underutilization with constrained mapping.

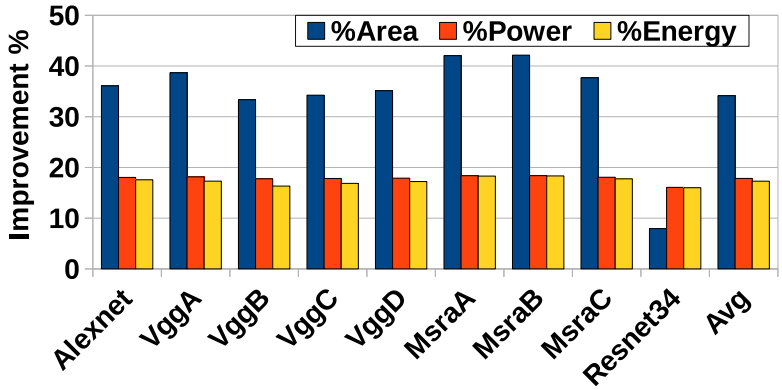

Figure 2.11. Impact of constrained mapping and compact HTree.

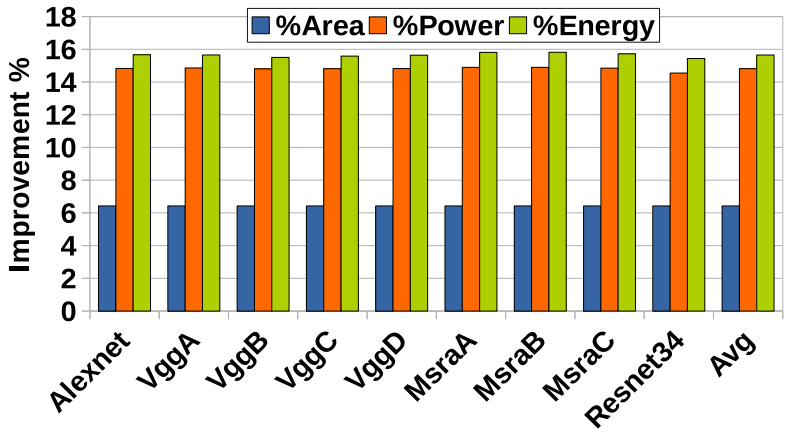

Figure 2.12. Improvement due to the adaptive ADC scheme.

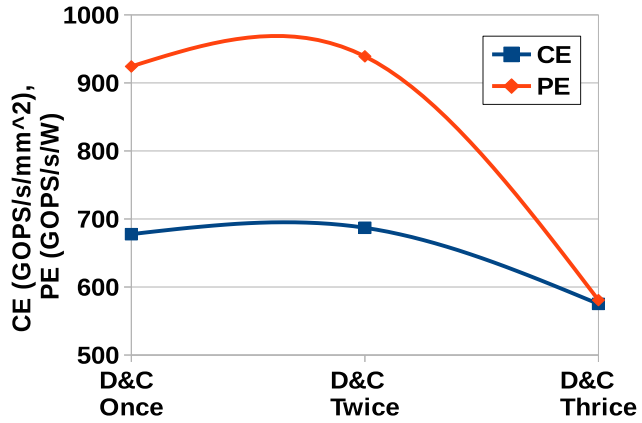

Figure 2.13. Comparison of CE and PE for divide and conquer done recursively.

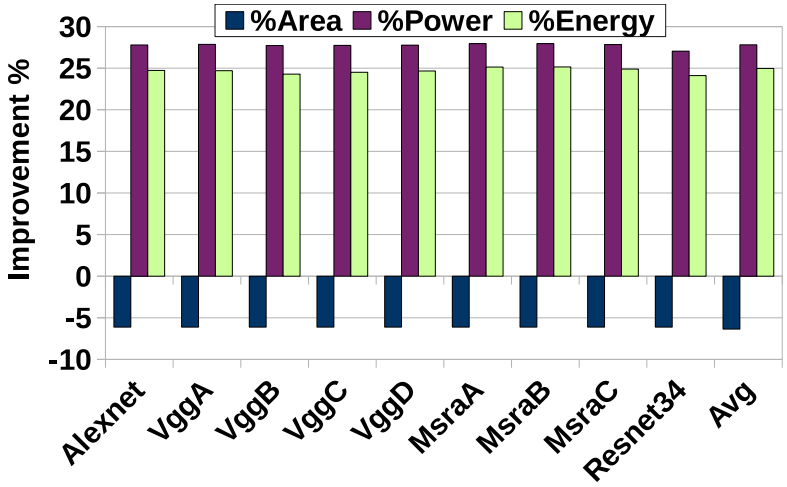

Figure 2.14. Improvement with Karatsuba's algorithm.

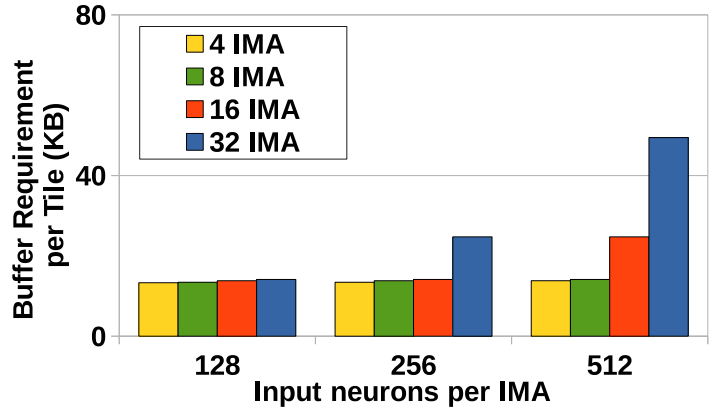

Figure 2.15. Buffer requirements for different tiles, changing the type of IMA and the number of IMAs.

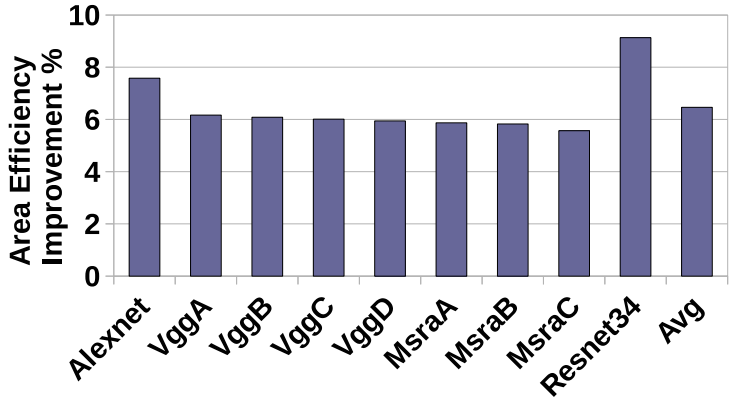

Figure 2.16. Improvement in area efficiency with decreased eDRAM buffer sizes.

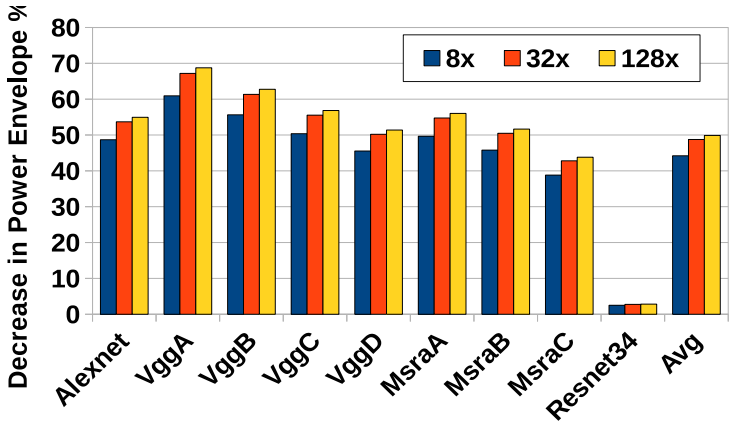

Figure 2.17. Decrease in power requirement when the frequency of FC tiles is altered.

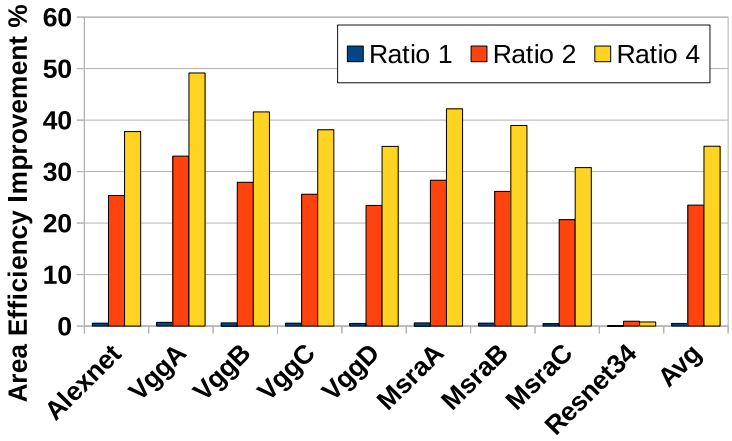

Figure 2.18. Improvement in area efficiency when sharing multiple crossbars per ADC in FC tiles.

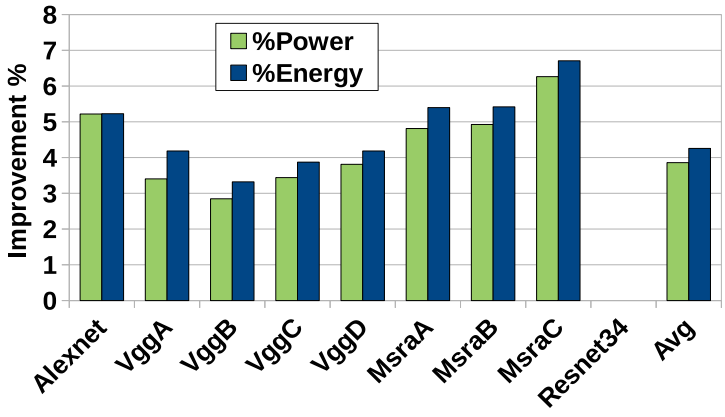

Figure 2.19. Improvement due to the Strassen technique.

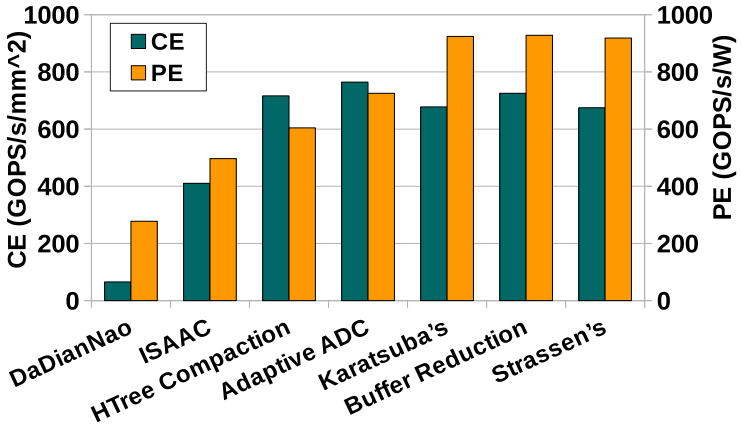

Figure 2.20. Peak CE and PE metrics of different schemes along with baseline digital and analog accelerator.

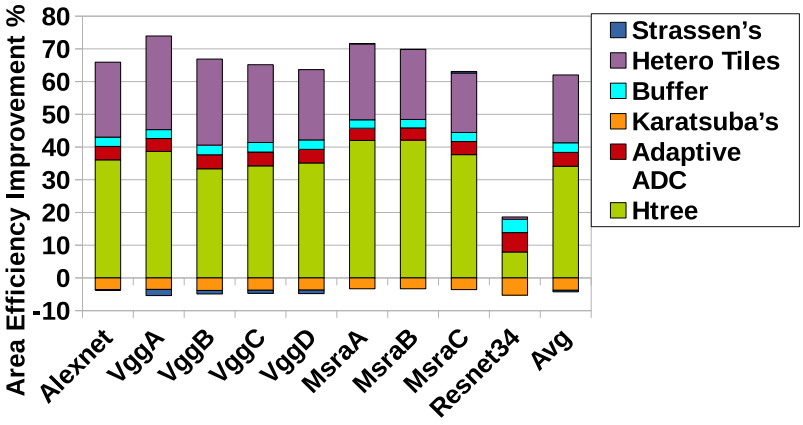

Figure 2.21. Breakdown of area efficiency.

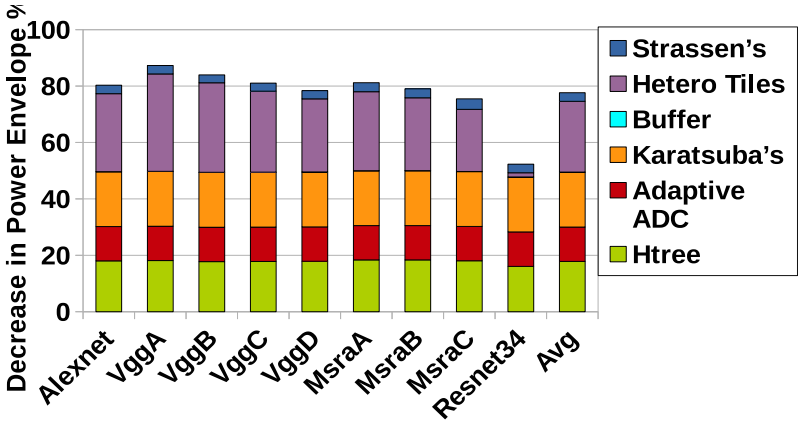

Figure 2.22. Breakdown of decrease in power envelope.

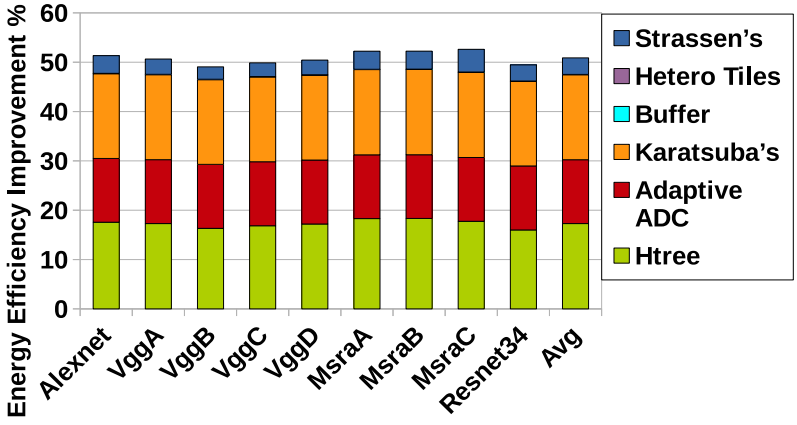

Figure 2.23. Breakdown of energy efficiency.

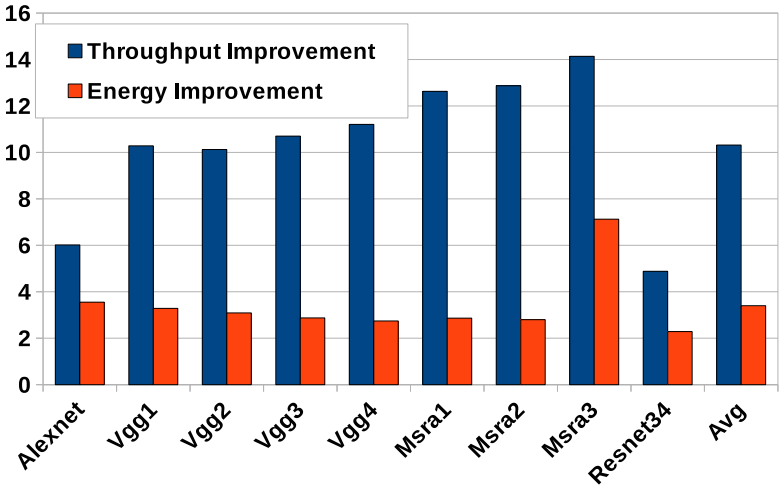

Figure 2.24. Comparison with TPU.

Table 2.1. Key contributing elements in Newton.

Component	**Spec**	**Power**	**Area** (mm^2)
Router	32 flits, 8 ports	168 mW	0.604
ADC	8-bit resolution 1.2 GSps frequency	3.1 mW	0.0015
Hyper Tr	4 links @ 1.6GHz 6.4 GB/s link bw	10.4 W	22.88
DAC array	128 1-bit resolution number	0.5 mW 8×128	0.00002
Memristor crossbar	128×128	0.3 mW	0.000025

Table 2.2. Benchmark names are in bold. Layers are formatted as $K_x \times K_y, N_o$/stride (t), where t is the number of such layers. Stride is 1 unless explicitly mentioned.

input size	**Alexnet** [79]	**VGG-A** [133]	**VGG-B** [133]	**VGG-C** [133]	**VGG-D** [133]
224	11x11, 96 (4)	3x3,64 (1)	3x3,64 (2)	3x3,64 (2)	3x3,64 (2)
	3x3 pool/2	2x2 pool/2			
112		3x3,128 (1)	3x3,128 (2)	3x3,128 (2)	3x3,128 (2)
		2x2 pool/2			
56		3x3,256 (2)	3x3,256 (2) 1x1, 256(1)	3x3,256 (3)	3x3,256 (4)
		2x2 pool/2			
28	5x5,256 (1)	3x3,512 (2)	3x3,512 (2) 1x1,256 (1)	3x3,512 (3)	3x3,512 (4)
	3x3 pool/2	2x2 pool/2			
14	3x3,384 (2) 3x3,256 (1)	3x3,512 (2)	3x3,512 (2) 1x1,512 (1)	3x3,512 (3)	3x3,512 (4)
	3x3 pool/2	2x2 pool/2			
7	FC-4096(2)				
	FC-1000(1)				

input size	**MSRA-A** [52]	**MSRA-B** [52]	**MSRA-C** [52]	**Resnet-34** [53]	
224	7x7,96/2(1)	7x7,96/2(1)	7x7,96/2(1)	7x7,64/2	
				3x3 pool/2	
56	3x3,256 (5)	3x3,256 (6) 3x3 pool/2	3x3,384 (6)	3x3,64 (6) 3x3,128/2(1)	
28	3x3,512 (5)	3x3,512 (6)	3x3,768 (6)	3x3,128 (7)	
	3x3 pool/2			3x3,256/2(1)	
14	3x3,512 (5)	3x3,512 (6)	3x3,896 (6)	3x3,256 (11)	
	spp,7,3,2,1			3x3,512/2 (1)	
7	FC-4096(2)			3x3,512 (5)	
	FC-1000(1)				

CHAPTER 3

GENCACHE: LEVERAGING IN-CACHE OPERATORS FOR EFFICIENT SEQUENCE ALIGNMENT

3.1 Introduction

Precision Medicine promises to revolutionize healthcare in the near future, relying heavily on genomic information that has the key to both diagnosis and treatment [70], [96], [108], [109]. For example, genome analysis is useful in understanding cancer-causing mutations and crafting an optimal drug cocktail that targets those cancers with minimal side effects [43]. Thanks to the reducing cost of genome sequencing [153], we will have large enough genomic databases to enable high-confidence hypothesis testing and significant discoveries. Not only will every person and every newborn be sequenced, but a single person will also be sequenced multiple times to monitor the natural progression of genetic health, growth of a tumor, and structural variations in different cells of the body. However, this introduces a significant computation and storage challenge. Infrastructures in hospitals and in the cloud will be fed with several genomic analysis queries per patient, for thousands of patients per day. Most analysis software packages consume several CPU hours to perform each of these computations [38].

Genomic accelerators have the potential to dramatically bring down the execution time and energy for these tasks. In the past year, multiple such accelerators have emerged [38], [145], [154] and led to orders of magnitude improvement. In this work, we extend this family of accelerators by primarily augmenting the memory hierarchy and tailoring the algorithm for the new hardware. The basic ideas have the potential for impact beyond the domain of genomic analysis.

A generic sequence alignment pipeline involves the following steps: a genomic segment is partitioned into small subsegments; these subsegments indexes into hash tables to

identify potential matching locations in a reference genome; filtration heuristics are used to narrow the set of candidate locations; each location is then assigned an alignment score with a dynamic programming step. The first three steps are critical because they require many parallel operations and memory fetches while also determining the pressure on the compute-intensive dynamic programming step. Prior accelerators like GenAx [38] and Darwin [145] use tiling and dedicate a large fraction of chip area for scratchpad SRAM that reduces memory bandwidth overheads imposed by the first three steps.

The proposed *GenCache* architecture [103] uses a combination of hardware and software innovations. We first employ the concept of in-cache operators to provide computational capabilities within the SRAM arrays; we introduce new operators that can be leveraged to perform many parallel filtering operations. We then design a new multiphase algorithm that reduces redundant work, uses appropriate in-cache operators in each phase, and manages scratchpad allocation and tiling to maximize reuse and parallelism. This new algorithm is based on our characterization of the common case when aligning genomic reads and seeds. We also observe that by promoting higher parallelism and reuse for some data structures, the cacheability of the largest data structure is negatively impacted; we alleviate that effect with a Bloom Filter that reduces cache pollution and bandwidth overheads.

Algorithm modifications alone or in-cache operators alone yield relatively low speedups. The combination of the two results in much more than additive speedup because the new algorithm shifts the bottleneck to compute, which is then accelerated by the new in-cache operators in GenCache.

The above hardware and software techniques help reduce data movement by more than an order of magnitude for a second-generation sequencing pipeline. Execution time is reduced by 5.3×, energy efficiency is improved by 8.6×, and the outputs of the genomic algorithm are unchanged. The new GenCache architecture imposes an area overhead of 16%, relative to GenAx. We also show that in-cache operators can be leveraged by third-generation pipelines, yielding a 1.8× improvement in third-generation filtering.

3.2 Background

3.2.1 Second- and Third-Generation Reads

Sequencing cost has dropped significantly over the past decade due to second-generation devices like Illumina HiSeq X and NovaSeq 6000. These devices typically produce small genomic segments called *reads* of size 100-200 bases at a throughput of 2300 Mbases/min. The machines introduce machine artifacts (errors) when generating reads, which range from 2-5%. Additionally, a human genome differs from other human genomes by about 0.1%. Since this difference is small, a reference-based reconstruction of the genome using these short reads is possible. In such a reconstruction, the reads are uniquely aligned against a reference genome by performing inexact matching, which accommodates for errors like mismatches and insertions/deletions (indels).

Once the reads are aligned, further statistical downstream analysis is performed to correct machine artifacts. This requires that the machine produce the same base multiple times for sufficient statistical guarantees. Thus, the machine generates multiple reads such that each base in the genome has a redundancy of $30\times - 50\times$, adding to the computational demands. Given its low error rate, the second-generation data are especially useful in discovering small variants in the genome.

Some genomic variants, called *structural variants*, span across hundreds of base-pairs. Such variants are not easily detected with second-generation reads. Third-generation sequencing devices like the PacBio and Oxford Nanopore produce long reads of size 1 to 100 kilobases at a throughput of about 240 Mbases/min. Such long reads are more effective at capturing structural variants. However, these reads have a raw error rate of 10% to 30% (machine induced), and this leads to significant computational overhead in the alignment step. Because of this trade-off, it is expected that both second- and third-generation sequencing technologies will be used in tandem to identify a broad range of variants. Initial hybrid pipelines [46], [63] have been developed that, for example, use information from second-generation alignment to improve the third-generation alignment. Accelerators for sequence alignment can therefore be an integral part of second-generation, third-generation, and hybrid pipelines.

3.2.2 Second-Generation Pipelines

3.2.2.1 Genome Analysis ToolKit (GATK) Pipeline

The Broad Institute has defined a standard second-generation pipeline – GATK. The first stage in this pipeline is sequence alignment. Even with a reference genome, this stage can consume 4.5 hours on a modern CPU [116], thus providing a throughput of 12.5 Mbases/min; that is, well below the throughput of the sequencing machine. The second stage cleans the output of alignment, and the third stage, variant calling, distinguishes real variants (mutations) from machine artifacts.

Figure 3.1 shows the breakdown of time taken in each stage of the pipeline when using 36 threads on an Intel Xeon E5 with 256 GB of memory. Roughly equal time is spent in sequence alignment (shown with the BWA-MEM algorithm), Sequence Alignment Map (SAM) file cleaning, and variant calling. SAM cleaning is typically bottlenecked by I/O and is addressed by other works using Solid-State Drive (SSD) and compression [15], [37]. In this work, we focus on the most compute-intensive task in the pipeline, sequence alignment, which impacts the first stage, and portions of the next two stages. Sequence alignment is also used in other genomic pipelines such as exome sequencing, RNA sequencing, etc. Figure 3.1 also shows that the computational overhead of alignment grows when we move from second-generation short reads to third-generation long reads.

3.2.2.2 Read Alignment Steps

As shown in Figure 3.2, a typical read alignment algorithm first partitions the read into small subsets called *seeds (step 1)*. The seed is used to index into a hash table to determine all the locations in the reference genome where this seed can be found *(step 2)*. These are the potential candidate locations where this read might align – L1 and L2 are the candidate locations in the example in Figure 3.2. An inexact match between the full read and these candidate locations is then performed. A quick inexact match is first performed to filter out a promising list of candidates *(step 3)*. In the example, only L2 advances to *step 4*, where a more compute-intensive inexact match is performed to score each candidate and identify the best alignment for the read.

3.2.2.3 First Three Steps

A number of different heuristics are used to implement the first three steps, such as Hobbes, Super-Maximal Exact Match (SMEM), Shifted Hamming Distance, and Myer's bit-vector algorithm. We will shortly discuss each of these and the trade-offs they introduce. These steps require both significant memory bandwidth and compute operations. This work primarily improves these three steps by reducing its memory bandwidth requirements, supporting many compute operations in parallel, and eliminating redundant locations that are forwarded to the fourth step.

3.2.2.4 Fourth Step

The compute-intensive inexact matching algorithm in the fourth step is usually a variant of Smith-Waterman local alignment (SWA) [134]. A read M of length m is matched against a reference fragment N of length n by filling up the cells of a matrix of size $m \times n$. The cells also store trace-back pointers such that the final alignment can be traced back from the highest-scoring cell in the matrix. Many parallel implementations of this algorithm have been proposed using FPGAs and systolic arrays [56].

3.2.3 GenAx and Darwin

GenAx [38] and Darwin [145] are hardware architectures designed to accelerate second- and third-generation read alignment, respectively. Both GenAx and Darwin use custom processing elements to accelerate the compute-intensive SWA alignment step. GenAx serves as the baseline for much of our work. It uses the SMEM algorithm for the first three steps; some of the required operations are accelerated with Content-Addressable Memory (CAM) arrays. To reduce the memory bandwidth burden of the first three steps, GenAx uses a large scratchpad and tiling to store/reuse a subset of its hash table. Darwin also has a large scratchpad for storing counters used in the filtration technique called Diagonal-band Seed Overlapping based Filtration Technique (D-SOFT) for long noisy reads. It also uses a tiling mechanism to manage the memory footprint of trace-back pointers. We later discuss more details of GenAx (Section 4.5) and Darwin (Section 3.6).

3.2.4 Bit-Line or In-Cache Computing

Bit-line computing is a hardware technique for performing in-situ computations in an SRAM array [2]. When two wordlines are activated in parallel, the result sensed at bitline-bar (BLB) is bitwise NOR, and the result sensed at bitline (BL) is the bitwise AND. The major change is an extra row decoder for simultaneous activation and a reconfigured differential sense amplifier to sense BL and BLB separately. The bits at BL and BLB can be processed further using combinational circuits to produce results like XOR, OR, SUM, $CARRY$, etc. [65]. Since multirow access can corrupt data, the wordline voltage is lowered, which slightly increases the access latency (1.5 $\times$). Such in-situ vector operations can unlock data-parallelism at the subarray level and can improve performance and energy due to reduced data movement.

3.2.5 Genomic Kernels

3.2.5.1 Seed Selection

The seed selection stage (step 1 in Figure 3.2) optimizes the number of candidate locations. The *Hobbes* algorithm fetches the frequency of all possible seeds of a read and then selects a batch of nonoverlapping seeds with minimum aggregate frequency using a lightweight dynamic programming step. The *SMEM* algorithm finds the longest exact match at each position of the read (Super Maximal Extended Matches). Hobbes is good for reads with lesser errors but produces slightly more candidate locations. SMEM is good for reads with many errors but demands a higher memory bandwidth.

3.2.5.2 Filtering

Within the first three steps, filtration (step 3 in Figure 3.2) involves a significant amount of computation. Common filtration kernels perform bit-vector computations to estimate string similarity and are amenable to in-cache operations. We, therefore, provide an overview of the major filtration kernels used in our work, each targeting a different point in the trade-off space. Hamming Distance (HD) is not suitable when indels are present; Shifted Hamming Distance (SHD) introduces a nontrivial false-positive rate; Myer's is computationally more expensive. Note that some of these kernels can be combined to reduce computation and false positives; in our work, we find that a combination of SHD and Myer's works best in some phases. Next, we explain each filtration kernel in detail:

- **Hamming Distance:** The Hamming distance between two strings indicates the number of mismatches between the strings. The Hamming distance between the read and an equal length sequence at the candidate location in the reference indicates the number of mutations, while disregarding shifts because of insertions and deletions. A hamming distance of 0 indicates an exact match.
- **Shifted Hamming Distance (SHD):** SHD is effective in identifying a small number of errors, including insertions/deletions [155]. For checking e errors, the Hamming distance is computed after shifting the read by up to e places to the left and right, indicative of e insertions or e deletions (steps 11 and 15 in Algorithm 1). After each shift, a Hamming mask is computed (steps 12 and 16) – a vector of 0s and 1s that indicate if the bases match or not. These Hamming masks are then amended using certain heuristics (steps 13 and 17) and combined using bitwise operations (steps 14 and 18) to estimate if the candidate location should be filtered or not (steps 7 and 20). SHD can successfully filter reads with up to five errors without any false negatives, but with a 7% false positive rate.
- **Myer's Bit Vector Edit Distance:** This algorithm is used to calculate the edit distance (including mismatches and indels) between two strings. This is calculated with dynamic programming, similar to SWA. Myer's algorithm is particularly amenable to in-cache operators as it uses bitwise operations. Algorithm 2 initially calculates the occurrence vector of each base (A/C/G/T) in the read-string (step 2). For example, for a read with two As in the beginning, the occurrence vector starts with two 1s. The algorithm then scans through each base in the reference string (step 6) to calculate each column of the dynamic programming matrix. Essentially, the algorithm calculates the difference between two consecutive columns with the help of auxiliary vectors (steps 3 and 4) and updates the number of edits. Apart from the addition operation in step 9, the rest are bitwise operations. In order to support Myer's algorithm in-cache or near-cache, addition and shift operations need to be supported. A minor transformation of Myer's algorithm is used to calculate banded edit distance to filter reads with more than e errors. This reduces computation by calculating only the cells of a banded diagonal.

Algorithm 1. Shifted hamming distance

1: **procedure** SHD($ref[], read[], n, E$)
2: $HM \leftarrow ref \oplus read$ ▷ Hamming Mask
3: $FM \leftarrow 1^n$ ▷ Final Mask
4: $AM \leftarrow Amend(HM)$ ▷ Amended Mask
5: $FM \leftarrow FM$ & AM
6: $e \leftarrow$ Count 1s in HM
7: **if** $e < E$ **then**
8: return *pass*
9: **else**
10: **for** all i $\leftarrow$ 1 to E **do** ▷ Loop over e errors
11: $read\_tmp \leftarrow read \gg i$
12: $HM \leftarrow read\_tmp \oplus ref$
13: $AM \leftarrow Amend(HM)$
14: $FM \leftarrow FM$ & AM
15: $read\_tmp \leftarrow read \ll i$
16: $HM \leftarrow read\_tmp \oplus ref$
17: $AM \leftarrow Amend(HM)$
18: $FM \leftarrow FM$ & AM
19: $e \leftarrow$ Count 1s in FM
20: **if** $e < E$ **then**
21: return *pass*
22: return $fail$
23: **procedure** AMEND($Mask$)
24: $M \leftarrow Mask$
25: $M_{-1} \leftarrow Mask \ll 1$
26: $M_{-2} \leftarrow Mask \ll 2$
27: $M_{+1} \leftarrow Mask \gg 1$
28: $M_{+1} \leftarrow Mask \gg 2$
29: return $(M_{-1}\overline{M}M_{+1})|(M_{-2}\overline{M_{-1}M}M_{+1})|(M_{-1}\overline{MM_{+1}}M_{+2})$

Algorithm 2. Myer's Bit-vector algorithm for levenshtein distance

procedure NUMBER_OF_EDITS($ref[], read[], n$)
  $Peq[bp](i) \leftarrow (read_i == bp)$ ▷ Occurence vector
  $Pv \leftarrow 1^n, Ph \leftarrow 0^n$ ▷ Auxilliary vectors
  $Mv \leftarrow 0^n, Mh \leftarrow 0^n$ ▷ Auxilliary vectors
  $Edits \leftarrow 0$
  for all j $\leftarrow$ 1 to n **do** ▷ Loop over n bases
    $Eq \leftarrow Peq[ref_j]$
    $Xv \leftarrow Eq$ & Mv
    $Xh \leftarrow (((Eq$ & $Pv) + Pv) \oplus Pv) \mid Eq$
    $Ph \leftarrow Mv \mid \sim(Xh \mid Pv))$
    $Mh \leftarrow Pv$ & Xh
    if Ph & 10^{m-1} **then**
      $Edits \leftarrow Edits + 1$
    else if Mh & 10^{m-1} **then**
      $Edits \leftarrow Edits - 1$
    $Ph \leftarrow Ph \ll 1$
    $Mh \leftarrow Mh \ll 1$
    $Pv \leftarrow Mh \mid \sim(Xv \mid Ph))$
    $Mv \leftarrow Ph$ & Xv
  return $Edits$

3.3 Proposal

3.3.1 Architecture Overview

Without loss of generality, in this work, we model a baseline that resembles the GenAx architecture [38]. The proposed architecture, GenCache, is shown in Figure 3.3 with new components shaded blue. It operates as a coprocessor (similar to a GPU card) and is equipped with significant memory bandwidth. The accelerator has large SRAM arrays; we add in-cache operators to these arrays. The input to the accelerator is a set of reads that must be aligned against the reference genome. These inputs are fed to a number of seeding lanes that execute the required seed selection and filtration heuristics (steps 1-3 in Figure 3.2). With help from a central controller unit, filtration commands are sent to in-cache operators in SRAM arrays, and responses are collected. The resulting candidate locations are then sent to an SWA engine where step 4 is performed to score each location and send final alignments back to the processor.

GenCache does not modify the SWA engine. It uses a new algorithm for steps 1-3 that requires fewer memory accesses and that exposes more parallelism to the in-cache

operators. The new algorithm also eliminates redundant reads that are sent to the SWA engine in GenAx. We are thus reducing the time taken in all steps and retaining a balanced pipeline. We preserve the same output quality as the baseline GenAx; that is, we do not introduce any false negatives. While we will focus on second-generation alignment here, similar approaches can be applied to third-generation alignment as well (Section 3.6).

3.3.2 GenCache Operators

In this work, we propose adding compute capabilities to the large scratchpads in genomic accelerators – this has the potential to add a large amount of distributed compute with local wiring and local data movement. In addition, we observe that genomic workloads can benefit from new in-cache operators. This subsection describes these new operators and what role they play in genomics; the next subsection describes how the algorithms can be modified to leverage in-cache operators.

3.3.2.1 Supported Operators

Compute Caches [2] were based on the principle of bit-computing, where in-situ AND and NOR operations were performed on two wordline vectors. Subsequently, more compute elements were added within the bitline peripheral circuits of the SRAM arrays to perform operations like OR, XOR, and in-place COPY [2]. These works also add nontrivial full-adder circuits to a bit-line to iteratively perform addition and multiplication to support convolution operations in neural networks [33]. All of these operators are part of our design because they are useful for various kernels. In addition, we introduce five new Reduced Instruction Set Computer (RISC)-like operators to efficiently implement SHD and Myer's algorithms. These operators are:

1. Hamming Mask (HM): computes the hamming vector between two strings, required by exact match, and in steps 2, 12, and 16 of SHD (Algorithm 1), and step 2 of Myer's (Algorithm 2).
2. Hamming Distance (HD): adds the 1s in a hamming vector; required by exact match, and in steps 6 and 19 of SHD (Algorithm 1) and steps 12 and 14 of Myer's (Algorithm 2).
3. Shift Left (SL): required in steps 15, 25, and 26 of SHD and steps 16 and 17 of Myer's.
4. Shift Right (SR): required in steps 11, 27, and 28 of SHD.

5. 32-bit addition (ADD): required in step 9 of Myer's.

For some of the above operators, we create both 1-bit and 2-bit versions. In genomic workloads, the input operand is often a base-pair (ACGT), which is represented with 2 bits.

3.3.2.2 Additional Peripheral Logic

The above operators require extra logic that is implemented adjacent to the bitline sense-amps. The overhead of this extra logic is quantified later with a synthesized design. This bitline peripheral circuit for bitline i is shown in Figure 3.4. When two wordlines A and B are activated, the results emerging from the sense-amps are $A\&B$ and $\sim A\& \sim B$. These are fed to a NOR gate (colored BLUE) to produce an XOR result. Next, we introduce connections between adjacent bitlines. This is useful to not only perform a shift but also to perform operations on 2-bit values. We introduce an OR gate (colored GRAY) for even-numbered bitlines that gathers the XOR result for two consecutive bitlines and decides if the 2-bit bases in rows A and B are the same or different (producing the signal HM or Hamming Mask). Next is a latch (colored YELLOW) that is used to perform 1-bit or 2-bit shift operations. Finally, the result of the bitline operation is sent to an adder tree used for aggregation; Hamming masks are produced at bitlines, while the hamming distance is produced at the adder tree. We implement a 128-bit adder tree (assuming inputs from odd-numbered bitlines in a 256-wide array, shown in Figure 3.3b) with 0.55 ns critical path latency at 28 nm technology.

In addition, prior work [33] has shown how word-granularity additions (say, the addition of two 32-bit words) can be performed with bitline operators. Those implementations incur overheads to map data and iteratively perform addition over several cycles. Instead, we implement a dedicated 32-bit adder adjacent to the subarray, shown in Figure 3.3b. This adder receives input operands in consecutive cycles (the first operand is buffered in a register). This allows the addition of two 128-bit rows (as required by step 9 of Myer's) in 4 cycles.

We later show that supporting these new operators results in a modest area overhead. In addition to the five new RISC-like operators introduced above, we introduce the following 4 Complex Instruction Set Computer (CISC)-like operators that perform the

filtration kernels mentioned in Section 3.2.5.2: SHD, SHD_C, $MYERS$, and $MYERS\_B$, where SHD_C performs SHD filtering for $e+1$ errors using the result of SHD_C with e errors and $MYERS\_B$ performs Myer's algorithm over a banded diagonal. Each "CISC" instruction is made up of multiple RISC operators and is parameterized by the number of errors or the read length (step 10 of SHD and step 6 of MYERS). A Read-Only Memory (ROM) stores the sequence of 4b RISC instructions for each CISC instruction.

Not shown in Figure 3.4 is a local control circuit that receives commands from the central controller and implements the RISC/CISC interface; in addition to receiving/buffering addresses, a 4-bit command is received to perform the operation and return the data response after a fixed number of cycles. The local controller "unrolls" a CISC instruction into a set of RISC instructions using the received parameter and the ROM sequence. Each RISC instruction is translated into 7b select signals for the Multiplexer (MUX) and Demultiplexer (DEMUX) logic in the bit-line peripheral circuit. We have synthesized the local controller circuit, and its area footprint is only 225 μm^2.

3.3.3 Second-Generation Read Alignment

Having introduced various in-cache operators, we describe how the second-generation read alignment algorithm can be modified to unlock the potential of GenCache.

3.3.3.1 SRAM Allocation: GenAx and GenCache

During the many steps of read alignment, the reference genome and the hash tables for seed positions are repeatedly accessed. Since these structures are too large to fit in cache, prior work GenAx uses the following tiling approach to alleviate the memory bottleneck. It partitions the reference genome into 2 MB slices, and a hash table is constructed for each slice. This hash table has a capacity of about 64 MB. GenAx thus allocates its on-chip storage for 2 MB of the reference genome and 64 MB of the hash table and processes the entire batch of input reads before bringing in the next slice of reference and hash table. The storage is organized as two scratchpads.

The values in the hash table exhibit a low computation/byte ratio, while the values in the reference genome are involved in a majority of computations. If the reference genome occupies only 2 MB of on-chip storage, it can only leverage a small fraction of in-cache operators, thus limiting the potential speedup. Therefore, we first change our resource

allocation. A much larger slice (24 to 48 MB) of the reference genome is placed in a scratchpad with compute capability. The remaining on-chip storage is then managed as a cache for the corresponding hash table. Given the large size of the hash table, the high miss rate in the cache is a potential bottleneck. We alleviate this bottleneck by restructuring the read alignment software pipeline described next.

3.3.3.2 A New Error-Aware Algorithm

Figure 3.5 shows the number of errors encountered per read when aligning against the reference genome. We show results for two popular software packages, BWA-MEM [83] and SNAP [159]. We see that about 75-80% of the reads align with the reference with 0 errors (exact matches), 15% align with one error, 5% with 2-5 errors, and the remaining with 6 or more errors. This distribution is consistent across most second-generation read datasets since two human genomes typically differ by less than 0.1%. A key observation here is that higher efficiency can be extracted by treating each of these error cases differently.

The seed-and-extend algorithm in GenAx divides a read into chunks of size 12 bp (seeds) and accesses the hash table to fetch locations where each of the seeds occurs in the reference genome. If neighboring seeds in a read return neighboring locations in the reference, the location is considered as a potential candidate. Potential candidates are then scored using the "extend" engine. It is this step of fetching locations of all the seeds from the large hash table that creates a potential memory bottleneck.

For example, consider a read of length 100 bp that is divided into 8 seeds, each of size 12. Assume that the frequency of occurrence of each seed in the reference genome is given by the array [512, 256, 128, 64, 32, 16, 8, 4]. After fetching these 8 frequency counts and pointers to the corresponding list of locations, GenAx then fetches a total of 1020 locations from the scratchpad. If the intersection of the sets of locations per seed is nonnull, GenAx declares a perfect match.

GenCache takes a different multiphase approach in selecting seeds for a read. The pigeon-hole principle states that given a read with e errors, we need $e + 1$ seeds such that at least one of the seeds will be error-free. When the read has zero errors, all seeds are error-free. We can therefore pick the least frequent seed and check for alignment against locations for that seed. In the example read above, GenCache creates seeds, then fetches

8 hash table locations (the frequency counts above). It then focuses on the four locations for the last seed and performs an in-cache perfect match for the full read against those four locations. Note that a large 32 MB slice of the reference genome has already been preloaded. We are thus fetching less than 10 memory locations (the frequency counts) in the common case and performing a handful of parallel in-cache matches, whereas GenAx reads 1000+ scratchpad locations, followed by a set intersection operation.

Similarly, in order to align a read with 1 error (mismatch or indel), we can narrow our search to the two most infrequently occurring seeds. We, therefore, employ a four-phase algorithm, shown in Figure 3.6, where each phase deals with a different error scenario. The first phase deals with exact matches, the next phase deals with 1-error matches, the next with 2-5 errors, and the fourth phase handles 6+ errors. Each phase uses a different algorithm to identify an efficient set of seeds, a different set of operations to perform the matches, and different data structures. All of this is summarized in Figure 3.6; we walk through the details of each phase in Section 3.3.4. The bottom line is that our algorithmic change avoids a lot of memory fetches in the common case (with few errors), and our hardware in-cache operators can perform highly parallel filtering operations.

3.3.3.3 Exploiting a Bloom Filter

One of the bottlenecks in our design is a large number of hash table lookups and the potential for a high cache miss rate, especially when a large fraction of on-chip space is allocated for the reference genome. As we will describe in the next subsection, futile lookups of the hash table in Phases 1 and 2 can be filtered out with a Bloom Filter.

3.3.3.4 Architectural Comparison

Architecturally, consider the contrast between GenAx and GenCache, summarized in Figure 3.7. The reference genome is a 3 billion base-pair structure that is encoded in 768 MB. GenAx partitions the reference into 384 2 MB slices and constructs hash tables specific to each slice. A slice and hash table are brought into the GenAx scratchpads, and alignment is performed for the entire batch of reads for one human (800M reads). The reads that find perfect matches are set aside and are not matched against subsequent slices. As a result, the set of reads to be processed shrinks gradually as GenAx iterate through the slices. This pipeline increases reuse of the reference and hash table but leads

to high computational overhead and high bandwidth demand when fetching the reads. *It also results in early approximate matches being sent to the SWA engine; these are futile/redundant computations if a perfect match is discovered later.*

Instead, in GenCache, the reference genome is partitioned into 16 48 MB slices. All slices are first processed in Phase 1 that looks for exact matches so that a vast majority of reads can be handled quickly with low computational overhead. *Also, unlike GenAx, these perfectly matched reads never send redundant work to the SWA engine.* The remaining 25% of reads then go through Phase 2, again iterating through 16 reference slices, performing the slightly more expensive 1-error match operations. More iterations are performed again in Phase 3 and Phase 4, with even more computations per read and an even smaller set of reads. The reference genome is therefore fetched from memory 4 times, and we also incur more memory accesses because of large hash tables that do not fit in the hash table cache. Note that in GenAx, a read is fetched 1-384 times depending on if/when it finds a perfect match, with 20-25% of reads going through all 384 iterations because they never find a perfect match. On the other hand, in GenCache, a read is fetched 1-96 times, with 75-80% of reads finding a perfect match in the 16 iterations of Phase 1.

We are thus trading higher memory bandwidth for reference and hash table for lower memory bandwidth for input reads and a lower computational burden. We show in Section 4.8 that this is a worthwhile trade-off and results in fewer overall cycles for memory fetches and compute.

3.3.4 Details of the Four-Phase Algorithm

In this subsection, we describe the design details (both algorithm and mapping to hardware) for the four phases described in Figure 3.6 for second-generation read alignment.

3.3.4.1 Seed Selection

Each phase starts with a seed solver that identifies a set of seeds that is most favorable for the next steps in the algorithm. The first two phases are most concerned with identifying 1 or 2 seeds that occur least frequently. The *Min_Search* modules walk through various seeds and track their occurrence count with hash table lookups to find seeds with minimum frequencies. Phase 3 uses the Hobbes algorithm, which performs a single (lightweight) dynamic programming operation per read. The SWA engine of GenAx is used for this. Phase 4 uses the SMEM algorithm, tailored for high-error scenarios, and used

in GenAx and BWA-MEM. Like GenAx, a Ternary CAM (TCAM) array is used to perform the SMEM operation. Figure 3.3 shows hardware units that implement the control for each of the above seed solvers.

3.3.4.2 Phases 1 and 2

Most reads are handled by Phases 1 and 2 that leverage in-cache operators. In Phase 1, we are looking for a perfect match against a few reference genome locations. We, therefore, write the read into specific cache subarrays, perfectly aligned with the candidate reference genome locations (in some cases, the read may be split across two subarrays). A Hamming Distance (HD) in-cache operation is performed for each of these locations. The central controller receives responses; when a perfect match is discovered, the read does not advance to subsequent phases and subsequent reference genome slices. Phase 2 follows a similar process but must consider the hamming masks of the left- and right-shifted versions of the read (using RISC operations SL, SR, HM).

3.3.4.3 Phase 3

In Phase 3, after using the Hobbes unit to identify seeds and candidate locations in the reference genome, the following in-cache operations are used to filter out the worst locations. An SHD CISC operation is performed at each location to filter out locations with more than 5 errors. Since SHD lets through a small number of false positives (7%), a $MYERS\_B$ CISC operation (described in Section 3.3.2) is also performed to identify the true positives. The list of true positives is then sent to the SWA engine.

3.3.4.4 Phase 4

In Phase 4, after using the hardware SMEM unit (already included in GenAx) to identify the best candidate locations in the reference, a $MYERS\_B$ operation is performed at each location. Reads with edit distance less than 40 are advanced to the SWA engine.

3.3.5 Reducing Hash Table Accesses

3.3.5.1 SRAM Allocation to Data Structures

We observe that each phase places different demands on the various data structures; there is the potential for higher efficiency if the on-chip storage is reallocated across the

data structures at the start of each phase. For example, Phase 1 handles the most reads and benefits the most from the higher parallelism afforded by many subarrays, whereas Phase 4 performs more lookups of seed locations for the SMEM seeding algorithm. In other words, Phase 1 needs a larger reference genome scratchpad (48 MB), while Phase 4 needs a larger hash table cache (48 MB). We, therefore, design the arrays so that 48 MB of SRAM support in-cache operators and leave it up to the central controller to configure/load the arrays at the start of every phase. The SRAM allocation of different data structures in the four different phases is shown in Figure 3.6.

3.3.5.2 Improving the Hash Table Cache

When a reference genome slice grows, its hash table size grows in proportion while receiving an even smaller allocation for the hash table cache. We address this problem with the following insight: the hash table for a reference slice is highly skewed in its distribution (shown in Figure 3.8), with 70-80% of seeds having zero occurrences in the reference genome, 14-17% having a single occurrence, and 6-13% having more than 1 occurrence. Seeds with 0 or 1 occurrence are especially useful during Phases 1 and 2. If we find a seed with 0 occurrences, the read cannot find a perfect match and does not need further processing in Phase 1. If we find a seed with 1 occurrence, a single HD operation on that location is enough to discover or give up on a perfect match in Phase 1. As described next, we leverage this observation to improve hash table cache efficiency.

3.3.5.3 Bloom Filter Details

We use a Bloom Filter to determine if a seed has more than 1 occurrence in the reference genome slice (a smaller set that is easier to track with a Bloom Filter). In Phase 1, we fetch Bloom Filter entries for seeds in a read until the Bloom Filter finds a seed with 0 or 1 occurrence. That seed is then looked up in the hash table (hopefully in cache). If the hash table entry is empty (0 occurrences for that seed), the read will not have a perfect match, and it is skipped over. If the hash table entry has a single occurrence, we perform an in-cache Hamming Distance operation to identify a perfect match. A similar Bloom Filter structure is also used for Phase 2.

The Bloom Filter implementation is a storage-efficient way to identify seeds with 0/1 frequency in a skewed hash table, which is crucial for Phases 1 and 2. This is a novel application of Bloom

Filters, unlike prior caching policies [113] that use Bloom Filters for a cache hit/miss prediction. Because the Bloom Filter is on-chip, it is also a more efficient structure to identify candidate locations. Without this approach, hash table entries with zero or many occurrences put pressure on bandwidth, cache capacity, and in-cache operators.

3.4 Methodology

3.4.1 Software

For short read alignment, we use the single-end ERR194147_1.fastq read dataset with 787M reads of length 101 bp, as used in GenAx. We compare the alignment accuracy of our modified pipeline against that of GenAx and BWA-MEM [83]. GenCache accuracy matches that of GenAx because (1) the four-phase algorithm only eliminates redundant SWA computations which would have had no impact on the GenAx output, (2) the techniques used in the first three phases such as Hobbes, SHD, and MYERS, have no false negatives, and (3) for complex alignments, GenCache falls back to the GenAx SMEM heuristic. Similar to GenAx, GenCache has a 0.0023% variance with the BWA-MEM output. This is because the GenAx SWA engine used in GenCache to make an apples-to-apples comparison uses a different tie-breaking mechanism than the software BWA-MEM algorithm.

3.4.2 Circuit Models

We synthesize the bitline peripheral circuit, the 128-bit adder tree, the 32-bit adder, the new seed solver circuits, the Bloom Filter hashing circuit, and the central controller using Synopsys Design Compiler. We design 8:1 and 4:1 MUXes in our peripheral circuit using smaller gates from a standard cell library. The column circuit consists of two parts: the analog component (sense amps and write drivers) and the digital component (rest of the circuit). We pitch-match the analog component with every SRAM cell to reduce variability due to noise. The digital component of the bitline peripheral is used for every even-numbered bitline and can be pitch-matched across two bitlines (10 tracks), where our 8:1 or 4:1 mux can fit easily. However, the D Flip Flop occupies 20 tracks, which is why we use two rows of peripheral circuits below the SRAM array. Each row consists of the digital component of every fourth bitline, each pitch-matched to four bitlines (20 tracks). We validate the floorplanning through a place-and-route flow. The central controller includes logic for scratchpad allocation/indexing and a finite state machine to manage the

responses of in-cache operations. The generated gate-level netlists were converted to a SPICE netlist using Fully Depleted Silicon on Insulator (FDSOI) 28 nm technology node to model the delay, power, and energy values. We model GenCache by integrating these circuit estimates within Cacti's subarray and cache models [99]. In order to model a scratchpad, we use the modified version of Cacti within the Aladdin toolset [131]. For memory fetches, we assume energy consumption of 51 pJ/bit for DDR4 [126]. Table 3.1 lists the main components in the GenCache architecture and their power and area (all scaled to 32 nm). Table 3.2 summarizes performance and energy for each operator in GenCache. The bit-line peripheral circuit adds to the area overhead of the cache subsystem by 14.7%. To accommodate the Bloom Filter, we use 4 MB more SRAM than GenAx, an additional SRAM overhead of 5.9%. A conventional 256×256 SRAM array read takes 650 ps, whereas a compute-enabled access to GenCache takes 1040 ps. The adder tree and 32-bit adder occupy a negligible area. Overall, the GenCache chip has a 16.4% higher area, a 34.7% higher peak power, and a 15% higher average power than GenAx.

3.4.3 Architecture Model

Since GenAx already shows an order of magnitude improvement over a 56-thread CPU baseline and an NVIDIA Titan Xp GPU [38], we compare our architecture against GenAx only. For the GenCache architecture, we designed a cycle-accurate simulator that models the latency and bandwidth of each component (cache HTree, cache subarray, queuing, instruction issue width of the seeding lanes, etc.). For the scratchpad, we use 16 slices of 2 MB each, where each slice has 64 banks, and each bank is equipped with 512 I/O wires. This provides high bandwidth for GenCache to send the reads to the appropriate subarrays at the cost of leakage power. Each subarray is 256×256 bits. The bitline peripheral circuit is used for every even-numbered bitline, offering parallelism of 128 bitwise operations per subarray. Each seeding lane processes both forward and reverse read at a time. The seeding lanes also buffer and overlap processing for four reads to tolerate hash table access latencies. The CAM array used by the SMEM algorithm in Phase 4 is used as a cache for each seeding lane in Phases 1-3 to temporarily store reads and locations of seeds.

3.5 Results

3.5.1 Throughput Analysis

Figure 3.9 shows the throughput improvement as each of our key innovations is incrementally introduced. By adding in-cache operators to the 2 MB reference slice in GenAx, and no algorithmic changes, we see a marginal improvement of 5% due to the limited parallelism. If the reference slice is allocated 32 MB (with in-cache operators) to create a basic GenCache, we see a 1.62× speedup over GenAx; while the operators allow high parallelism, the system continues to be bottlenecked by memory accesses because the hash tables experience frequent cache misses. The fourth bar introduces the four-phase algorithm (eliminating redundant computations) for the baseline GenAx architecture (without in-cache operators). This only yields a 1.36× speedup because of more iterations through the reads and limited parallelism for filtration. The same four-phase approach over basic GenCache yields a 2.5× speedup due to (1) fewer fetches for reads (which accounts for a large fraction of memory bandwidth) and (2) the high degree of parallelism offered by GenCache operators. *This is, therefore, a case of the whole being greater (4× speedup from in-cache operators and algorithm) than the sum of its parts (1.36× from algorithm alone and 1.62× from in-cache operators alone).* The addition of the bloom filter alleviates the remaining memory bottleneck from hash table misses in Phases 1 and 2 to yield a 5.26× speedup over baseline GenAx.

3.5.2 Energy Analysis

Figure 3.10 shows an energy comparison (in terms of reads per mJ) and memory fetches for the same six configurations. Most of the energy reduction is from a reduction in memory accesses. The in-cache operators, by themselves, do little to reduce memory accesses. In fact, they increase memory accesses because of a higher cache miss rate on hash table lookups. A larger reference slice in the third, fifth, and sixth bars leads to fewer memory fetches per read. Energy improvement is the highest (8.6×) when using the four-phase algorithm with a bloom filter due to reduced memory access and lower runtime.

Figure 3.11a further breaks down the memory accesses across phases and data structures (note log scale). In the baseline GenAx, most memory accesses are for the reads,

parts of which are fetched 384 times. In GenCache, most memory accesses are again for reads, but by eliminating redundancy, the total count is much lower. While the hash table accesses are fewer than the read fetches, the hash table accesses tend to be on the critical path, so reducing them has a large impact on performance. Figure 3.11b breaks down the execution time and energy consumed in each phase. Phases 1 and 2 take less time because of the high parallelism offered by in-cache operators HD and SHD. Phase 4 takes the most amount of time because it is bottlenecked by the CAM lookups for the SMEM based filtration. Phase 1 and Phase 4 consume high energy because of memory fetches and CAM lookups, respectively.

3.5.3 Bloom Filter Analysis

Figure 3.12a shows the Bloom Filter size required for different false positive rates and reference genome slices. The SRAM allocation for Phases 1 and 2, shown in Figure 3.6, is based on this data. Figure 3.12b shows the reduction in the miss rate for the hash table cache. Note that the hash table itself is 1.5 GB, of which only 4 MB is cached. We are using a 20 MB bloom filter to approximately track high-frequency entries in the 1.5 GB table. The bloom filter not only reduces futile memory bandwidth (captured in Figure 3.10) but also yields a higher cache hit rate by avoiding pollution. This is especially important given that most of the on-chip storage has been apportioned for storing other compute-friendly data structures.

3.5.4 Iso-Area Analysis and Effect of High Bandwidth Memory (HBM).

As a sensitivity analysis, we reduced the on-chip SRAM allocation of GenCache to 62 MB for an iso-area comparison with GenAx. In spite of the reduced parallelism and cache space for hash tables, we observe a 4.3× speedup over GenAx. Using an HBM interface with 256 GB/s bandwidth instead of a Bloom Filter does little to improve the performance (8%) of the four-phase algorithm because there are not as many hardware threads to hide the latency of memory accesses.

3.6 Application to Third-Generation Alignment

GenCache operators are useful for other genomic operations, such as Indel Realignment in the GATK pipeline. Here, we show its potential in the context of a third-generation

kernel.

3.6.1 Baseline

We assume a Darwin-like baseline that uses D-SOFT filtration for long reads. D-SOFT finds seeds from a read that map to multiple nonoverlapping bins of a reference genome. Depending on the number of bases that match in a bin, the bin is chosen as an *anchor*. The read is then iteratively aligned against the reference and extended (SWA step) on either side of the anchor using smaller tiles (512 bp).

3.6.2 In-Cache Filtration

We leverage the observation that D-SOFT produces 100-10,000 false hits per read, 97% of which can be easily pruned by using a thresholding operation on the score of the first SWA tile. *Instead of using an SWA operation, we propose to use a MYERS_B operation on the first tile as a filtration step, which is amenable to acceleration with GenCache.* This approach only advances 3% of D-SOFT hits to the Darwin SWA engine while not impacting accuracy.

3.6.3 Methodology

We simulated three sets of reads of size 10 Kbp and 30× coverage using PBSIM [111], representing the various third-generation sequencing methods and their error profiles. PacBio mimics a 15% error rate of Pacific Bioscience devices. ONT_2D and ONT_1D mimic the error rates of Oxford Nanopore devices at 30% and 40%, respectively. Darwin uses a 64 MB scratchpad for storing the counters needed for the D-SOFT algorithm for all the reference bins (for $bin\_size = 128$). In our design, we allocate 16 MB to cache the hash table and prefetch the locations for neighboring seeds in a read; we allocate 16 MB for counters; the remaining 32 MB (with in-cache operators) is used to store a slice of the 768 MB reference genome. A 32 MB slice of the reference consists of 1M bins of size 256 bp, which needs 1 MB of counters; with a 16 MB allocation for counters, we process 16 reads in parallel.

3.6.4 Results

Figure 3.13 shows the improvement from using this in-cache filtration (log scale). GenCache improves the performance of third-generation alignment by 1.8×. This is because

of two main reasons: (1) the 32 MB scratchpad provides an additional throughput of 1 BReads/s for the Myers Edit distance step along with the 21 MReads/s throughput provided by the Darwin SWA engine, and (2) the cache for the hash table and the fewer counters increases the throughput of the counter update in D-SOFT (thus creating alignment tasks at a faster rate for the SWA engine). In terms of chip area, the Darwin chip is estimated to be 264 mm^2, whereas the GenCache design is estimated to be 289 mm^2 at 32 nm.

3.7 Related Works

3.7.1 In-Cache Computing

A large body of recent work has exploited opportunities for in-memory processing [12] that embed operators into memory arrays. In-SRAM operation-based accelerators include PROMISE [135], which uses a mix of analog and digital circuits to support matrix-multiplication operations for machine learning workloads. The NAND-Net architecture [72] focuses on in-SRAM and in-DRAM implementations of binary neural networks. Architectures such as DRISA [85] and Ambit [128] based on in-DRAM operations accelerate neural networks and database search applications, respectively.

Compute Caches [2] is one of the early works that propose enabling bit-line computing using SRAM caches within traditional CPUs. Compute Caches address microarchitectural challenges, such as operand locality, memory disambiguation, cache coherence, etc., and evaluates workloads such as database query processing, cryptographic kernels, and in-memory checkpointing. Cache Automaton [139] repurposes the last-level cache in a CPU to efficiently perform state-matching and state-transition for automata processing. Neural Cache [33] and Bit-Prudent Cache [152] repurposes the last level cache for in-place fixed-point multiply-accumulate (MAC) operations to accelerate neural networks. Duality Cache [39] repurposes the last level cache for floating-point arithmetic and transcendental functions to accelerate scientific workloads.

3.7.2 Accelerating Genomics

There have been many efforts to improve the performance of genomic analysis steps (such as sequence alignment) with a variety of approaches, ranging from hardware acceleration to distributed bioinformatics runtimes. Accelerators like GenAx [38], Darwin [145],

and Medal [61] have focused on sequence alignment, while efforts like GateKeeper [8] accelerates SHD-based filtering, Wu et al. [154] use FPGAs to accelerate Insertion/Deletion (INDEL) realignment in the Cloud, and the Dragen system uses FPGAs for alignment and variant calling [97]. The FPGA approaches of accelerating filtering or alignment have the following downside in comparison to GenCache: (1) they provide lower parallelism in comparison to in-cache operators, (2) they incur additional bandwidth penalty to fetch reference segments, which remain in-place for GenCache, and (3) they incur frequent cache misses due to smaller SRAM caches. Madhavan et al. [91] explore the use of race logic to perform SWA operations by encoding information using timing delays in the circuit. Distributed systems like Persona [19] and Adam [92] take a holistic approach to combine different phases and build a cluster-scale, high-throughput bioinformatics framework.

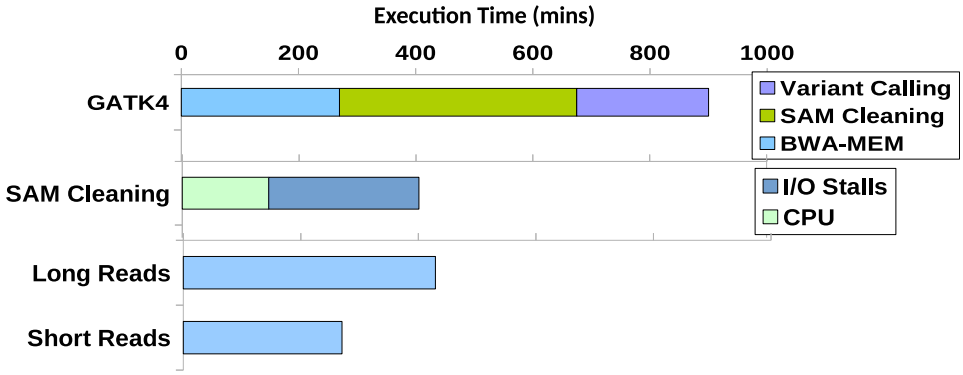

Figure 3.1. Execution time breakdown for the different stages of the GATK pipeline and comparison of time taken by long and short read alignments.

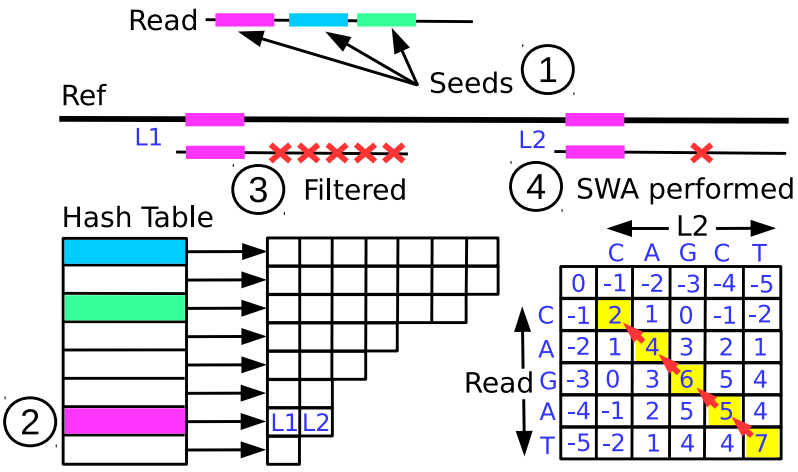

Figure 3.2. A typical four-step read alignment pipeline.

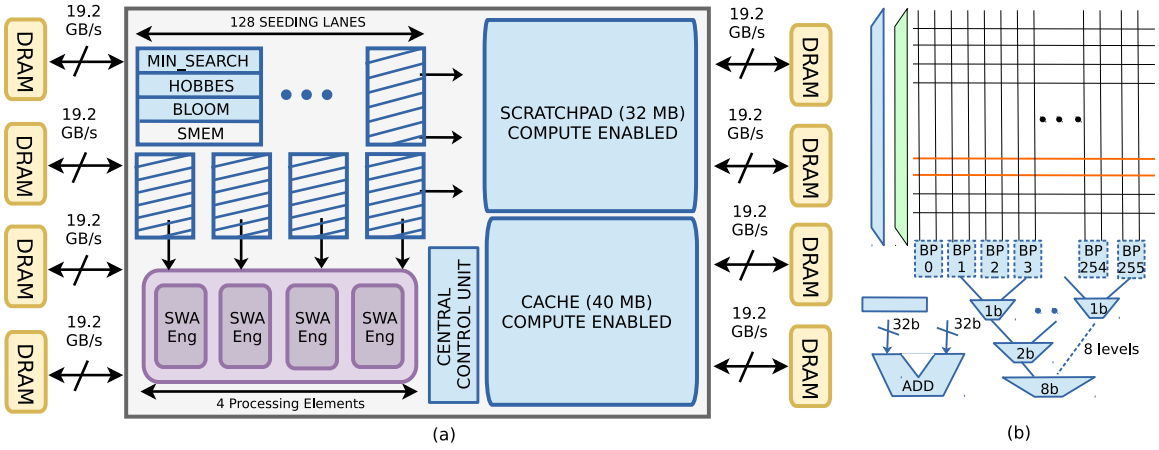

Figure 3.3. Overview of the GenCache architecture (additional components over the GenAx architecture are colored in blue): (a) block diagram of architecture and (b) bitline peripherals and added logic in a subarray.

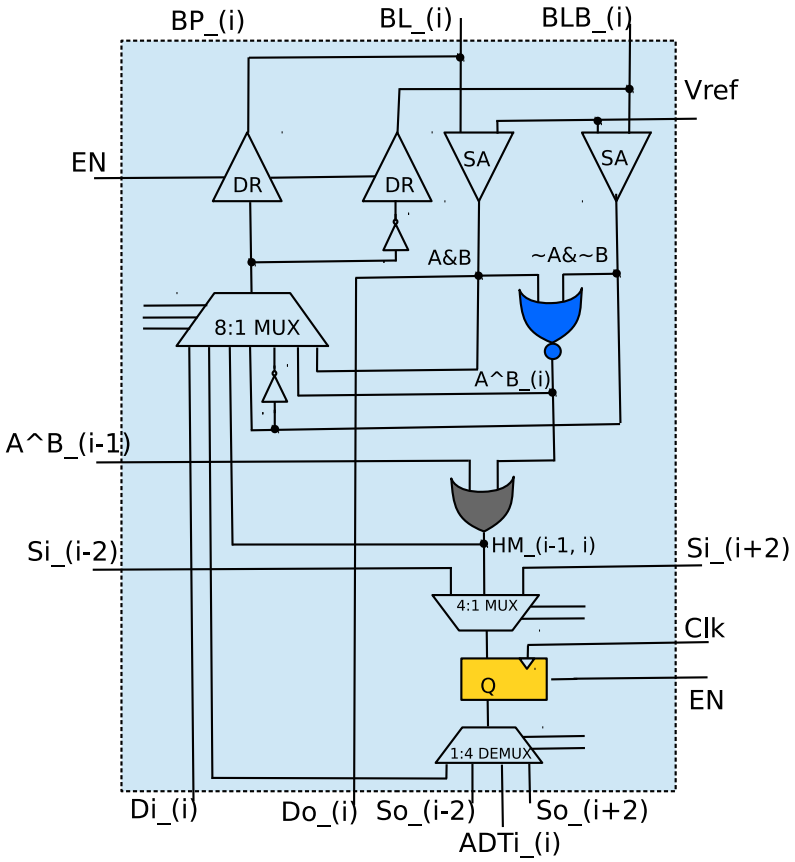

Figure 3.4. Bitline peripheral circuit for GenCache.

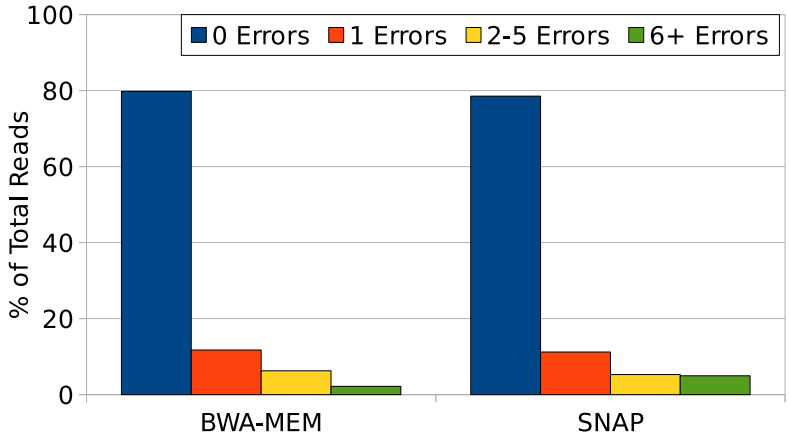

Figure 3.5. Read alignment profile for BWA-MEM and SNAP.

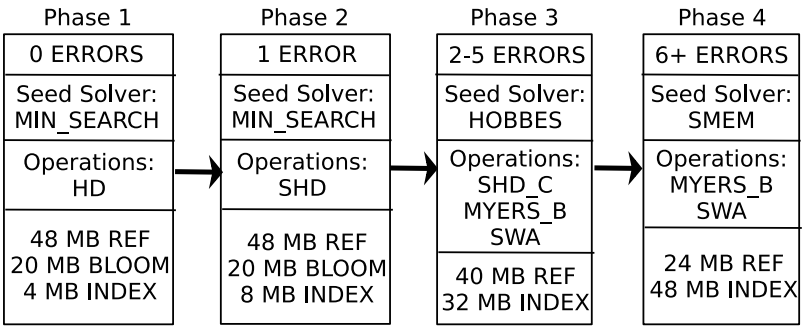

Figure 3.6. Four phases in the new alignment algorithm that exploits in-cache operators.

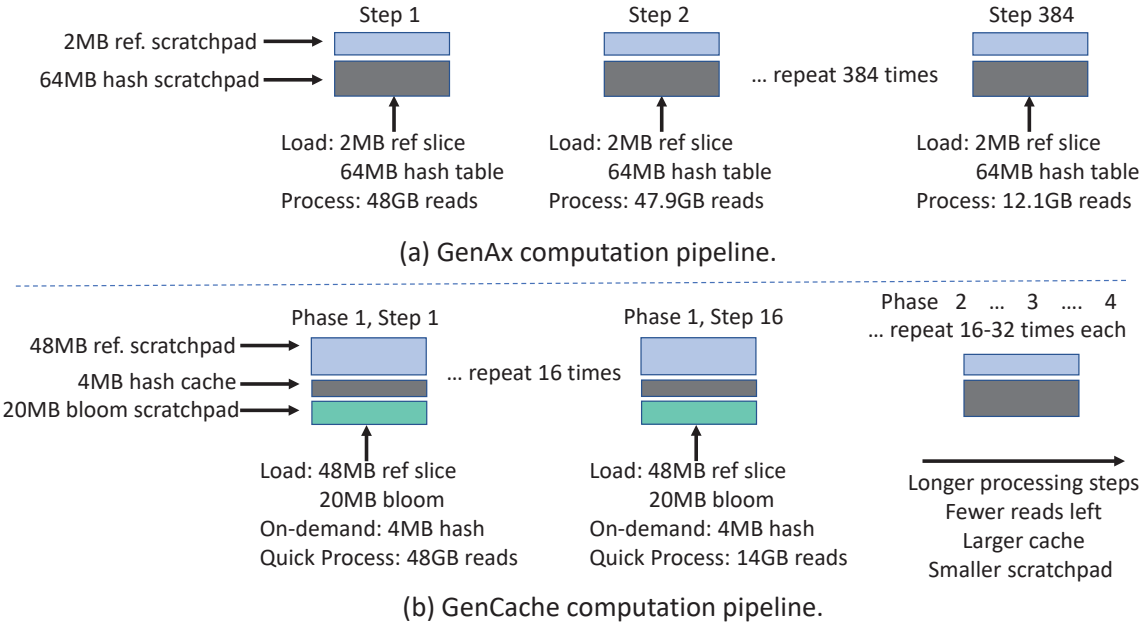

Figure 3.7. Comparison of the GenAx and GenCache computational pipelines.

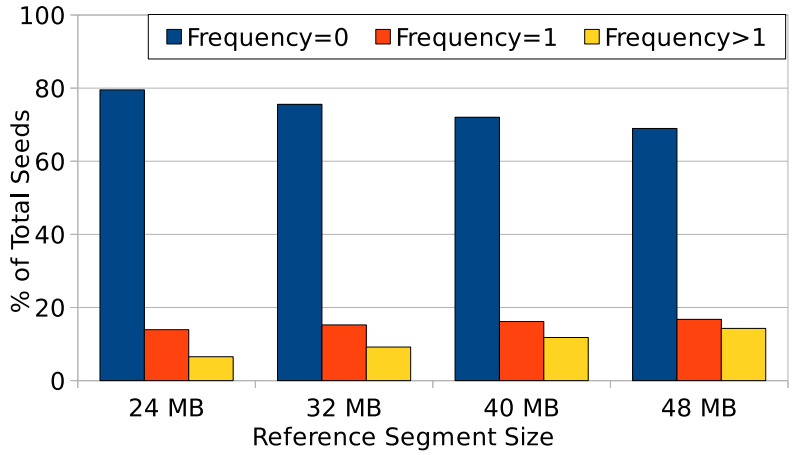

Figure 3.8. Distribution of seed occurrences for different reference genome slice sizes.

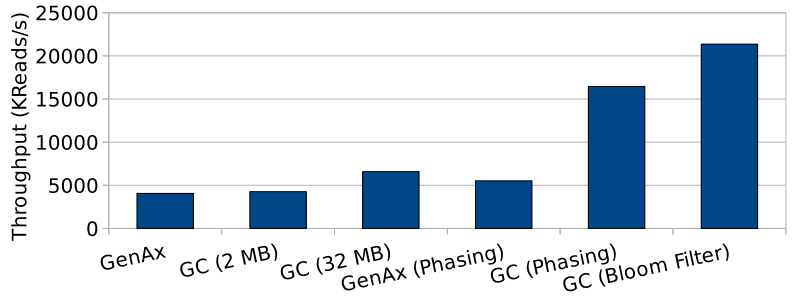

Figure 3.9. Throughput improvement of GenCache (hardware and software) for second–generation workloads.

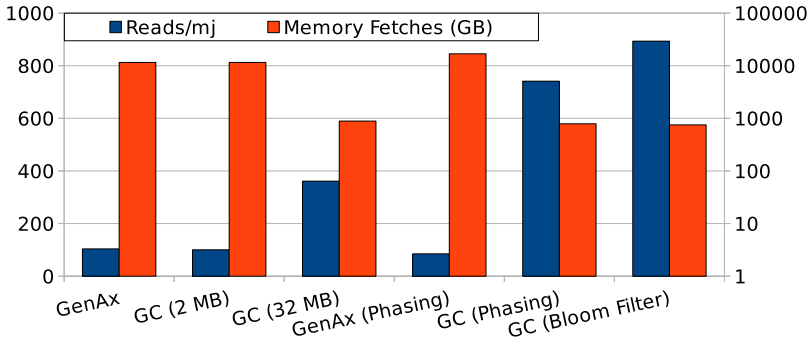

Figure 3.10. Energy improvement (in Reads/mJ) and the number of memory fetches (log scale) of GenCache (hardware and software).

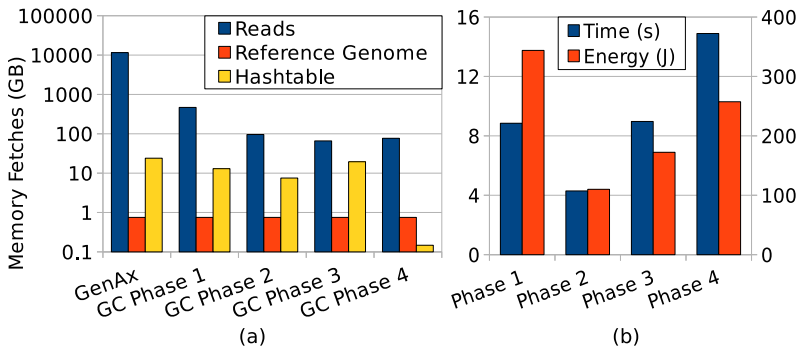

Figure 3.11. Breakdown by program phase for (a) number of memory accesses to different data structures, and (b) execution time and energy consumption.

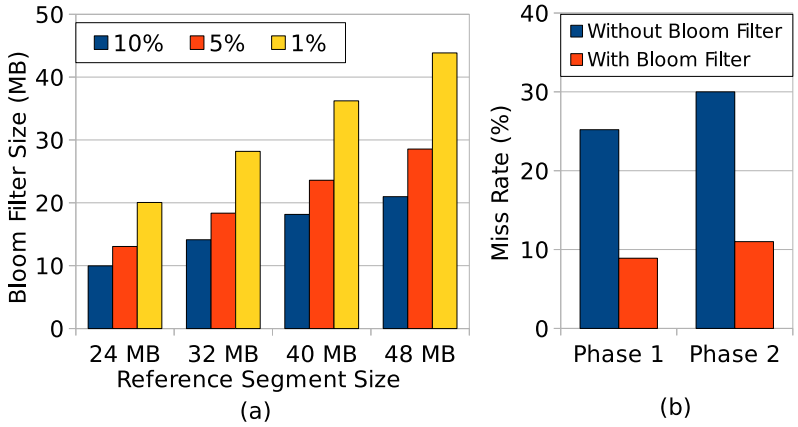

Figure 3.12. Sensitivity analysis for bloom filters: (a) bloom filter sizes for varying reference slices and false-positive rates, and (b) reduction in miss rate for Phases 1 and 2 when using a bloom filter with a 10% false-positive rate.

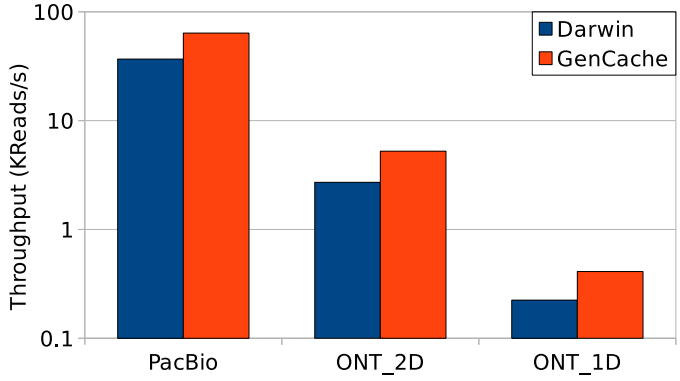

Figure 3.13. Throughput improvement of GenCache for third-generation workloads (log scale).

Table 3.1. Area and power of different components in GenAx (yellow/orange) and GenCache (white/orange) at 32 nm.

Components	Area	Peak Power
Subarray		
Bit-line Peripheral (x128)	2842 μm^2	1.98 mW
Adder Tree	672 μm^2	0.369 mW
32-bit Adder	70 μm^2	0.126 mW
SRAM memory		
GenAx Scratchpad (68 MB)	189.2 mm^2	12 W
GenCache Scratchpad (32 MB)	102.1 mm^2	8.42 W
GenCache Cache (40 MB)	117.6 mm^2	10.33 W
Seed Solvers (x128)		
Min Finder	0.015 mm^2	5 mW
Hobbes Control	0.060 mm^2	169 mW
Bloom Hashes	0.073 mm^2	247 mW
SMEM	5.78 mm^2	4.4 W
Central Control Unit	2.59 mm^2	1.53 W
SWA Engine (x4)	7.43 mm^2	8.62 W
GenAx Chip Total	202.41 mm^2	25.02 W
GenCache Chip Total	235.68 mm^2	33.72 W

Table 3.2. Comparison of latency, energy and peak throughput of different GenCache operations at 128 bp granularity for a 32 MB scratchpad (102.1 mm^2) with GenAx SWA engine (1.86 mm^2).

Operations	Cycles (@2.0GHz)	Energy/Op (pJ)	Throughput/Area ($MReads/s/mm^2$)
HD	7	46	12268
SHD (1e)	71	659	1136
SHD (5e)	328	3109	244
SHD_C	64	613	1252
MYERS_B (5e)	501	2079	160
MYERS_B (40e)	3413	14085	23.5
MYERS_B (90e)	7571	31237	10.6
GenAx SWA (40e)	456	491340	2.36

CHAPTER 4

ORDERLIGHT: LIGHTWEIGHT MEMORY ORDERING PRIMITIVE FOR EFFICIENT FINE-GRAINED PIM COMPUTATIONS

4.1 Introduction

Many critical workloads today include significant data-intensive phases (few computations per byte of data) along with compute-intensive portions. For example, convolutional neural networks are comprised of both compute-intensive convolutions and data-intensive computations such as batch normalization [1], feature map addition [1], and classifier layers [147]. Similarly, genomic analysis consists of both compute-intensive (string matching) and data-intensive (sequence filtering [145]) portions. While general-purpose or domain-specific accelerators such as GPU and TPU [67] are able to tackle the compute-intensive portions of applications, data-intensive phases remain limited by memory bandwidth.

Processing-in-memory (PIM)[1] solutions move compute close to memory arrays, providing significant memory bandwidth advantage over host[2] processors. As such, PIM solutions are uniquely positioned to speedup data-intensive portions of modern workloads. Given modern workloads comprise both data-intensive and compute-intensive phases as discussed above, it is increasingly important to couple host accelerators with memory equipped with compute capability.

While many PIM designs have been proposed, two aspects that have received limited attention are: (1) the granularity of operations offloaded to PIM, and (2) the interaction between PIM computations and host memory accesses; that is, whether host and PIM are allowed concurrent access to memory. We believe that carefully considering these

[1] In this work, we use *PIM* to refer to both in-memory-array and near-memory-array logic as the proposed work is largely agnostic to this distinction.

[2] We use the term *host* to refer to any processor or accelerator attached, but external, to PIM-enabled memory module(s).

aspects and their tradeoffs help to better serve the needs of modern workloads. To this end, we first develop a taxonomy of PIM designs based on the granularity of offloaded PIM computations and the granularity of arbitration with host memory accesses. Further, to consider both of these aspects in tandem, we focus on *temporal* (time) granularity as opposed to data granularity.

Based on our taxonomy, we observe that PIM designs spanning the past several decades [3], [16], [29], [30], [36], [42], [45], [50], [57], [58], [69], [71], [73], [76], [85], [112], [118], [132] have mostly utilized coarse-grained offload where the host ships entire computations to memory-side logic as depicted in Figure 4.1. Although such coarse-grain offload provides simplicity of design, we argue that it has steep costs. First, such an approach requires more complex memory-side logic to orchestrate offloaded computations (e.g., instruction/operation sequencing within the PIM devices). Second, assuming concurrent memory accesses from the host, it requires moving the memory controller to the memory module, along with arbitration between PIM and host accesses [95]. This is often impractical in commodity systems. Alternately, prohibiting concurrent memory accesses by host and PIM computations can lead to drastic reductions in the utilization of the host processor when PIM is heavily used.

Based on the taxonomy, we also identify an emerging class of PIM designs [4], [40], [82], [105] that perform fine-grained offload, in a temporal sense, that overcome several shortcomings of the coarse-grained approach discussed above. First, it keeps the host processor in charge of orchestrating PIM computation, thus, greatly simplifying memory-side logic to primarily cater to data-intensive computations that are most suitable for PIM. Second, it keeps the memory interface compatible with the fundamentals of existing memory interfaces, even in the presence of PIM. Finally, it allows the memory controller to schedule PIM commands interleaved with normal loads/stores, allowing concurrent operation of the host and PIM.

While the fine-grained approach for PIM is advantageous in many respects, it does have concomitant challenges. A key challenge that we focus on in this work is the *ordering* needed amongst the fine-grained PIM commands. Existing memory ordering primitives (aka fences) that are used by host computations to enforce ordering of memory operations have severe overheads [77] and are often used sparingly by programmers. As such, they

are ill-equipped to efficiently enforce ordering for fine-grained PIM commands.

To address these challenges, we make the key observation in this work that, unlike existing memory ordering primitives, which are *core-centric*, the ordering requirement for PIM commands is *memory-centric*. Consider a PIM computation where two arrays are added and stored in a third array. This computation is accomplished by a sequence of fine-grained PIM commands that read the two inputs, add them, and write the result back to memory, all of which need to be ordered with respect to one another for correctness. Unlike core-centric ordering primitives where values have to be read at the core before the write can be completed (again at the core), the ordering requirement for PIM commands has to be enforced at the PIM unit performing the computation; that is, at the memory).

We exploit this observation by proposing OrderLight [104], a lightweight memory ordering primitive where ordering is enforced at the memory controller instead of the core by sending OrderLight packets along with PIM instructions to the memory controller. In order to orchestrate PIM computations, the core simply issues a series of PIM instructions and OrderLight primitives down the memory pipe without stalling. We demonstrate the performance benefits of moving away from core-centric enforcement of memory ordering using a suite of computations from relevant applications (machine learning, genomics, and data analytics) and streaming benchmarks. Overall, we make the following contributions:

- We introduce a taxonomy of PIM designs based on the temporal granularity of both offloaded PIM computations and arbitration with host memory accesses and use it to discuss characteristics desirable in a PIM design in today's computing landscape.
- We make the observation that existing memory ordering primitives are ill-equipped to support fine-grained PIM designs (a key subclass within the above taxonomy) as existing ordering primitives are *core-centric*.
- To address this challenge, we propose a new lightweight memory ordering primitive, OrderLight, which moves away from core-centric ordering enforcement and considerably reduces the overheads of enforcing correctness for PIM.
- For a suite of applications (machine learning, data analytics, genomics) and streaming benchmarks, we demonstrate that OrderLight delivers 5.5× to 8.5× improvement over existing memory ordering primitives.

4.2 Background

We first discuss relevant workloads and GPUs as potential host accelerators that can be coupled with PIM.

4.2.1 Data-Intensive Phases in Modern Workloads

Increasing data-centric processing has led to many modern workloads having both data-intensive (few computations per byte of data) and compute-intensive phases. While general-purpose cores and specialized accelerators can tackle compute-intensive portions, PIM solutions stand best to speedup data-intensive portions. We discuss some of these workloads below.

4.2.2 Machine Learning

Workloads such as neural network (Deep Neural Network (DNN), Recurrent Neural Network (RNN), etc.) training and inference consist of multiple layers of computation. Layers such convolutions are often formulated as matrix-matrix multiplication operations and are compute-intensive. Other layers, such as feature-map addition (e.g., in residual networks [54]), batch normalization [62], and fully-connected (during inference) have low compute-to-byte ratios and are data-intensive [1]. Feature-map addition sums neuron activations of two layers (two vectors) to feed as input to a third layer (output vector). Batch normalization scales and add biases to an input vector of neuron activations. Fully-connected layers in inference perform a series of dot product operations of a large input activation vector with a large number of weight vectors. Profiling shows that data-intensive computations constitute approximately 32% of the training runtime of ResNet50 on current GPU hardware [1].

4.2.3 Genomic Analysis

Sequence alignment is a key step in genomic analysis and is performed by aligning small sequences (called reads) against a reference genome. A read is aligned against the reference at a set of candidate locations using a dynamic programming step (compute-intensive). A filtering algorithm is used to reduce the set of candidate locations. The filtering algorithm computes simpler operations such as Hamming distance or dot-product at a large number of candidate locations and is data-intensive. Filtering has been shown

to constitute 65% of sequence alignment runtime [73].

4.2.4 Data Analytics

Data analytics for unstructured data, such as unlabeled text or images, typically requires two main steps: feature extraction and clustering. Feature extraction is often performed using neural networks that provide a feature vector for a word, a sentence, or an image (compute-intensive convolutions). The feature vectors obtained from this first step are then used to cluster the data points within a large dataset using algorithms such as Kmeans and Histogram. Kmeans and Histogram are data-intensive as they require sifting through a large amount of data with simple computations (Kmeans: distance from centers, Histogram: bin update).

4.2.5 Host Accelerator - GPU

Several of the modern workloads discussed in Section 4.2.1 benefit from having a GPU as the host accelerator (e.g., GPU is preferred for ML training [10]). As such, the system we evaluate assumes a GPU coupled with PIM-enabled (Section 4.4.1) High Bandwidth Memory (HBM) [64] as our baseline (we discuss further reasons for this baseline in Section 4.4.3). However, the ideas and architectural innovations discussed in this chapter are applicable broadly across other forms of hosts and PIM organizations as well.

A GPU consists of multiple cores, known as Streaming Multiprocessors (SMs) or Compute Units (CUs) in NVIDIA and AMD terminology, respectively. Each SM has one or more Single Instruction Multiple Data (SIMD) units to issue instructions from a vector of threads called warps or wavefronts. The GPU SMs issue memory requests through their load/store (LDST) units. Each SM has a private L1 cache, a texture cache, a constant cache, a software managed scratchpad, and a shared instruction cache. The SMs are connected to a shared L2 cache via an interconnection network. Each memory channel is associated with its own L2 slice. There are multiple memory controllers, one per channel. Physical memory is interleaved at chunk granularity (e.g., 256B chunks) across memory channels.

4.3 Taxonomy of PIM Offload and Arbitration

In this section, we present our taxonomy for PIM designs with a focus on the temporal granularity of two aspects: (1) offloaded PIM computations and (2) arbitration of PIM

computation and host memory accesses. We also use this taxonomy to discuss characteristics desirable in a PIM design in today's computing landscape. Note that "temporal granularity" refers to the amount of time consumed by an offloaded PIM computation and not the amount of data it operates on.[3] Further note that, as discussed in Section 4.2, given the heterogeneous nature of modern workloads, we consider PIM designs coupled with host accelerators that tackle both compute and data-intensive phases. We do not consider designs where memory is the main compute unit [85], [129]; that is, no host processor.

4.3.1 Coarse-Grain Offload and Fine-Grain Arbitration

We term designs which ship entire PIM computations to memory-side logic but allow host and PIM computation to arbitrate for memory accesses at fine granularity (typically, individual load/stores) as coarse-grain offload and fine-grain arbitration (CGO/FGA) designs [3], [16], [42], [57], [58], [71], [118]. Such designs require complex memory-side logic to orchestrate PIM computation. Further, they enable fine-grain arbitration between PIM and host accesses by moving memory scheduling from the host to the memory module using transactional host memory interfaces, often based on the Hybrid Memory Cube [95]. We note that none of the currently available mainstream memory interfaces provide the transactional semantics required by these designs [125].

4.3.2 Coarse-Grain Offload and Coarse-Grain Arbitration

Coarse-grain offload and coarse-grain arbitration (CGO/CGA) designs follow offload mechanisms similar to C GO/FGA designs above, but they disallow concurrent memory accesses from host and PIM computations [29], [30], [36], [69], [76], [112]. As a result, these designs (depicted in Figure 4.2a) render system memory inaccessible to the host during PIM computations. This is undesirable in datacenters (and other multiuse environments) as it can impact QoS guarantees of all tasks scheduled on the host and adversely affect the achievable utilization of datacenter resources. Further, this approach also places a lower bound on the minimum computation granularity for PIM offloads as they must be

[3]While there is some correlation between the temporal granularity of a task and the amount of data accessed, we note that the relationship is not uniform across PIM architectures. For example, a temporally fine-grain bitwise operation on an entire DRAM row may touch several KB of data but still complete within a single row operation's worth of time.

large enough to justify draining the host's memory pipeline prior to launching the PIM computation and refilling it after completion of PIM execution.

4.3.3 Fine-Grain Offload and Coarse-Grain Arbitration

PIM designs with fine-grain offload and coarse-grain arbitration (FGO/CGA) offload PIM computations at fine granularity (typically temporally equivalent to individual loads/stores), which simplify the memory-side logic to only support data-intensive computations and not associated orchestration logic [45], [50], [73], [85], [132]. However, such designs suffer from the drawbacks of coarse-grain arbitration, as outlined in Section 4.3.2.

4.3.4 Fine-Grain Offload and Fine-Grain Arbitration

PIM designs with fine-grain offload and fine-grain arbitration (FGO/FGA) keep memory-side logic devoid of PIM orchestration overheads and, at the same time, allow fine-grain arbitration of host and PIM memory accesses (depicted in Figure 4.2b). While historically scarce, a few recent research efforts fall into this class [4], [40], [82], [105]. Furthermore, this approach shows good versatility, as the designs utilizing this approach span transactional memory interfaces [4] as well as minor variations of mainstream Joint Electron Device Engineering Council (JEDEC) memory standards [82].

4.3.5 Desirable PIM Characteristics

With the above taxonomy in mind, in this work, we observe that there is significant value in FGO/FGA PIM designs as they truly enable host and PIM computations to execute concurrently, which is often highly beneficial for modern workloads. Further, they reduce memory-side logic complexity (no orchestration needed within memory modules) and also broaden PIM usage by allowing even small segments of computation to be effectively offloaded to PIM. In addition, FGO/FGA designs are compatible with mainstream memory interfaces such as Double Data Rate (DDR), High Bandwidth Memory (HBM), Graphics DDR (GDDR), and Low Power DDR (LPDDR). As such, we believe that improving the efficiency of such designs is warranted.

4.4 Challenge: Ordering of PIM Instructions

While in Section 4.3, we discussed the desirability of FGO/FGA PIM designs, we discuss in this section a key challenge associated with these designs, which is the focus of this work. To appropriately highlight this challenge, we first discuss a generic and parameterized PIM unit, followed by an example set of fine-grained PIM operations the PIM unit performs. Finally, we use both of the above to discuss how existing memory ordering primitives fall short of efficiently supporting ordering requirements of fine-grained PIM instructions needed in FGO/FGA designs, thus motivating the need for an efficient memory ordering primitive geared towards this use case.

4.4.1 Generic and Parameterized PIM Compute Unit

Our proposed work is agnostic to the specific memory-side logic and is applicable whenever FGO/FGA PIM designs are employed (more details in Section 4.4.3). Consequently, in this work, we consider a generic PIM compute unit which processes fine-grained PIM commands as depicted in Figure 4.3. Further, we parameterize this unit to study a variety of PIM solutions. The PIM unit we consider consists of a SIMD ALU coupled with temporary storage (labeled TS in Figure 4.3). The SIMD nature of the ALU allows effective utilization of the high bandwidth typically available to PIM designs, and the temporary storage buffers operands read from memory or results to be stored in memory. Note that, while Figure 4.3 shows a PIM unit that may reside on a 3D-stacked die separate from the memory arrays, alternatively, the PIM unit may be placed close to the arrays (e.g., near a memory bank or a memory subarray) representing a broad swath of different PIM solutions. Further, specializations and different cardinalities of the PIM unit can also be considered. Depending on the placement and number of units, different bandwidth multiplication factors over host-available memory bandwidth is realized collectively by the PIM units. In our evaluations, we sweep this bandwidth multiplier to study disparate PIM solutions. Further, we also study the efficacy of our ideas by varying the size of temporary storage associated with these PIM units.

4.4.2 Fine-Grained PIM commands

We discuss in this section the nature of fine-grained PIM commands that we consider in this work. As with any memory access to DRAM, fine-grained PIM commands incur *precharge* and *row activate* operations on operand accesses from memory. Further, we assume data movement operations similar to loads and stores are used to move data from/to an activated DRAM row to temporary storage associated with PIM, and RISC-like instructions are used to orchestrate PIM computations.

Consider a simple PIM computation where two vectors (a and b) are added and stored in a third vector (c): $c[i] = a[i] + b[i]$. We envision the orchestration of such a computation using our generic PIM compute unit from Section 4.4.1 by using a sequence of fine-grained PIM commands, as shown in Figure 4.4. Specifically, it will first load the input values into temporary storage. However, unlike normal host loads, these commands achieve higher bandwidth to the memory associated with the PIM unit (line 2). This is followed by PIM computation commands (line 5) to add the input operands together and store the result back to temporary storage. Finally, the result is stored to memory (*vector c*) (line 7).

Note that based on available temporary storage size, multiple chunks of input operands can be saved (N commands, line 2) before adding them (N back-to-back compute commands, lines 4-5). In the same vein, several of the calculated results could be stored back to memory as well (N commands, line 7). Further, based on placement and cardinality of PIM compute units, several such computations (across different channels/banks/subarrays) can be orchestrated in parallel.

4.4.3 Ordering Requirement for PIM Commands

With the fine-grained PIM offloading that we envision, the host accelerator (GPU in this work) issues PIM memory instructions (akin to loads, stores) to accomplish the PIM computation. We refer to this host executed kernel (GPU parlance) as a *PIM kernel* (Figure 4.4). GPUs rely on fine-grain hardware scheduling of many threads, which is particularly useful in issuing PIM memory operations to several generic PIM compute units that can be placed within the memory hierarchy. PIM memory instructions issued by the host get translated into fine-grained PIM commands discussed in Section 4.4.2 at the memory controller.

As with existing memory instructions, PIM instructions issued by the host also have ordering requirements. For example, in Figure 4.4, loading/fetching *vectors a* and *b* should happen before the computation operation(s), which in turn should happen before the store operations for *vector c*. Further, as in host computations, PIM instructions issued by the host can be reordered to target several performance optimizations. For example, instructions can get reordered within the host pipeline, in the network from the host to the memory controller, within the memory controller transaction queue, and more. As such, any required order between PIM instructions has to be explicitly enforced.

Host computations rely on memory ordering primitives, aka fences, to ensure the ordering of memory instructions. Fences are inserted in the host PIM kernel in Figure 4.4 to ensure ordering for PIM execution. However, fences are impractical from a performance point of view and insufficient functionally for fine-grained PIM operations. First, fences have considerable performance overheads and are generally used sparingly and judiciously. Fine-grained PIM offloading, however, considerably increases the number of needed fence instructions leading to severe overheads. With more storage, a larger number of PIM instructions (N) can be issued (lines 2, 4, and 7) before issuing a fence. Note that the size of the temporary storage (N) determines the number of loop iterations required to complete the task in Figure 4.4. Each iteration requires three fences; if N is high, fewer iterations and fences are required. For a range of values of N (as a fraction of row-buffer size), fences can slow down execution by 4.5× to 25× as depicted in Figure 4.5 (see Section 4.7 for methodology details). Second, for memory instructions which do not return data to the host (stores), existing fences only ensure ordering up until the global serialization point (coherence directory or memory controller), which is insufficient for PIM instructions as they have to be issued to the memory module by the memory controller in the desired order for correctness. As such, existing memory ordering primitives remain a strong impediment to support of FGO/FGA PIM designs. Addressing this challenge is the key focus of this work.

4.5 OrderLight Design

In this section, we discuss our proposed approach to tackle the challenge of providing efficient memory ordering while enabling FGO/FGA PIM designs.

4.5.1 Key Insight: Memory-Centric Ordering Enforcement

In order to design an efficient memory ordering primitive for fine-grained PIM approaches, we make the key observation that the *core-centric* nature of existing memory ordering primitives is **neither necessary nor sufficient** for PIM instructions. Consider a load instruction followed by a fence instruction. Fence semantics have to ensure that any subsequent memory instructions happen after the load is complete. The completion point for a load is at the core (core receives requested data block), and the core incurs wait-cycles to ensure this ordering. In contrast, PIM instructions (e.g., *Fetch-and-Add*) are completed at the PIM compute unit. This provides a unique opportunity to push ordering enforcement for PIM instructions to the memory controller, thus, freeing the core to issue PIM instructions without incurring wait cycles. Thus, PIM instructions need *memory-centric* ordering enforcement.

Furthermore, existing memory ordering primitives enforce ordering until the global serialization point (coherence directory or memory controller). However, as PIM instructions are completed at the PIM compute unit, ensuring proper ordering necessitates that the corresponding PIM commands are issued to memory subject to the ordering constraints (to avoid reordering at the memory controller). This further makes a case for memory-centric ordering enforcement.

4.5.2 OrderLight Overview

To overcome the shortcomings of core-centric memory ordering primitives and exploit the opportunities of memory-centric ordering, we propose the OrderLight memory ordering primitive. In our proposed design, in order to express ordering between PIM instructions, the programmer employs the novel OrderLight instruction instead of a regular fence. On encountering this instruction, the core generates an OrderLight packet, which percolates all the way to the memory controller. The relative ordering of OrderLight packet and PIM instructions is maintained at every step of the memory pipe until the PIM requests reach the memory controller. This packet is also preserved by the memory controller in its transaction queue to enforce ordering amongst PIM commands. Finally, as the order is conveyed to and enforced at the memory controller, the core does not incur any wait cycles and can issue PIM instructions unabated.

We highlight the benefits of an OrderLight primitive in comparison to a fence in Figure 4.6. We show the ordering of *Load* and *Fetch-and-Add* instructions of the *vector_add* PIM kernel discussed in Section 4.4.2. With existing fence primitives, we observe that the core incurs wait cycles (165 to 245 cycles as depicted in Figure 4.5) to ensure the ordering requirement. In contrast, using OrderLight instructions, the OrderLight packets are delivered in-order to the memory controller, which enforces memory ordering. As such, the host does not incur wait cycles and can issue PIM instructions with high throughput.

In the following sections, we describe the architectural changes needed to realize memory-centric ordering using our proposed OrderLight primitive in the context of GPUs as host accelerators. Figure 4.7 depicts a typical GPU architecture along with core and memory pipe (path from core/SM to memory controller) that we consider in this work. We first discuss the design changes needed for our new OrderLight instruction in the core and then highlight the changes needed to support the OrderLight packet in the memory pipe.

4.5.3 Architectural Changes in the Core

4.5.3.1 OrderLight Instruction and OrderLight Packet

Our proposed work introduces a new OrderLight instruction that is employed by the programmer to express memory ordering between PIM instructions. An OrderLight instruction inserts an OrderLight packet into the memory pipe. Figure 4.8 shows the different fields in an OrderLight packet, with example bit widths. The packet is distinguished from normal load/store requests using a 2-bit packet ID. A channel ID (shown as a 4b field in Figure 4.8) identifies the memory channel for which order is to be enforced. The next field is an optional 4b memory-group ID. A memory-group can be a subset of banks, an HBM stack[4], a subset of subarrays, etc. Memory-group ID helps enforce ordering for a particular memory-group only. For example, if PIM data structures are mapped to one memory-group, and non-PIM data structures are mapped to a different memory-group, the memory-group ID in the OrderLight packet informs the architecture to not constrain non-PIM requests whenever possible as they need not be ordered. Finally, the fourth field is the packet number within a channel and memory-group. This helps the memory-

[4]HBM groups vertically-stacked memory dies into groups of four referred to as *stacks*, somewhat analogous to ranks in DDR memory systems

controller to perform sanity checks and collect statistics.

4.5.3.2 Operand Collector

The operand collector logic in the core schedules operand access to a multibanked register file. It consists of multiple collector units, each of which buffers operands accessed from the register file for one instruction. After all the operands are buffered, an instruction is ready to be issued. Each memory instruction is allocated a collector unit, and its register access requests are queued in an arbitration logic block. The arbitration logic can issue register accesses out of order to schedule accesses to multiple banks in parallel.

Fences halt execution before instructions are sent to the operand collector. However, to avoid instruction reordering with OrderLight primitive, the OrderLight packet is issued only after all preceding PIM requests have left the operand collector. This is achieved by keeping a count of the number of PIM requests residing in the operand collector. The count is incremented every time a PIM request is allocated to a collector unit and is decremented when the request is issued. An OrderLight packet does not need to go through the operand collector phase and is issued to the LDST queue when the count is zero. Thus, the instruction issue is halted by the OrderLight packet for a much shorter period of time in comparison to normal fences. A separate counter is used for each memory-channel and memory-group. To reduce the number of counters, an implementation may limit the number of channels/memory-groups that can be controlled per SM.

4.5.4 Architectural Changes in the Memory Pipe

4.5.4.1 Caches

Caches contribute to reordering due to cache hits for later requests. A memory request that experiences a cache hit is serviced faster. However, PIM computation requests are meant to reach the memory and do not affect the cache. We consider PIM requests to behave the same way nontemporal loads and stores do. Thus, these requests bypass the caches and are directed to the main memory. For the L1 cache, the requests move from the LDST queue to the interconnect network. For the L2 cache, the requests move from the interconnect-to-L2 queue to the L2-to-DRAM queue.

4.5.4.2 Diverging Paths in the Memory Pipe

The memory pipe may consist of one or more diverging paths. For example, many architectures have multiple subpartitions per L2 slice and separate input/output queues for each subpartition. L2 subpartitions are often used to cater to different memory-groups within a memory channel. PIM requests navigated to different subpartitions may merge later in the memory pipe out of order.

To maintain order through divergent paths, we use a *copy-and-merge* technique for the OrderLight packet, as shown in Figure 4.9. When an OrderLight packet reaches a divergence point in the memory pipe, the packet is copied into multiple packets that traverse each of the relevant memory subpaths. For example, if a PIM kernel is issuing PIM requests to a particular memory-group within a channel, and the requests of the memory group traverse through two of four L2 subpartitions, then the OrderLight packet is copied to generate two OrderLight packets, each of which traverses the queues of the relevant subpartition. The copied packets are merged at the convergence point in the memory pipe. Any requests that follow the OrderLight packets in the different memory subpaths are not allowed to proceed until all the OrderLight copies are merged, and the merged packet moves forward.

The copy-and-merge technique is achieved using finite state machines (FSM) at divergence and convergence points. The FSM at the divergence point uses information in the OrderLight packet, such as channel ID and memory-group ID, to replicate the packet on each relevant subpath. The FSM at the convergence point issues a merged packet down the pipe once all copies of the packet are received (number of copies to merge is determined similarly to the divergence point). Note that path divergence among L2 slices does not require reconvergence as each L2 slice is associated with one memory channel with no subsequent merging of paths.

4.5.4.3 Memory Controller

The memory controller may implement a unified queue or separate queues for read and write requests. It also contains a scheduler which tries to balance increased DRAM efficiency (via techniques such as prioritizing row hits and reducing read/write turnarounds) and fairness. Commands for scheduled requests are enqueued in command queues (which

may be separate for each bank), which are then issued to the memory module subject to timing constraints. For separate read and write queues, the copy-and-merge technique has to be adopted, where two OrderLight packets are generated and pushed to each of the queues and get merged at the scheduler stage of the controller.

The memory controller is allowed to reorder requests in the following cases: (1) requests are not separated by an OrderLight packet, or (2) requests belong to different memory-groups. The scheduler is augmented with a request counter and an OrderLight flag for each PIM memory-group. The counter associated with a memory-group is incremented when a request to that memory-group is dequeued by the scheduler and decremented when it is scheduled. When the scheduler receives an OrderLight packet, the OrderLight flag for the appropriate memory-group is set. Any subsequent request to that memory-group is not scheduled until the flag is unset. The flag is unset when the counter for the memory-group is decremented to zero; that is, all requests preceding the OrderLight packet have been scheduled. Once the OrderLight flag is reset, the scheduler is free to process subsequent requests.

4.6 Programmability

As discussed in Section 4.4.3, unlike coarse-grain PIM approaches, programming for fine-grained PIM approaches necessitates the host to execute a PIM kernel comprising of a stream of PIM instructions. In the long-term, we envision compile-time software tools that can automate the generation of PIM instructions based on specially-annotated regions of code expressing computation to be executed on PIM units. In the near-term, intrinsics-like low-level primitives can be embedded in high-level code that generates the appropriate fine-grain PIM instructions when compiled.

In a GPU-like host processor with many cores, hardware contexts, and concurrent threads, a subset of the cores or hardware contexts can be set-aside for executing the PIM kernel. Further, to reduce the need for synchronization among host software warps, the control of each PIM unit (or a set of PIM units) can be limited to a single host warp. In other words, each PIM unit receives PIM instructions from a single host warp, avoiding the need for synchronization among multiple host warps in orchestrating PIM computation. We utilize such a model in our evaluations.

As with any PIM computation, FGO/FGA requires keeping data in different levels of the memory hierarchy coherent. For example, dirty data (of PIM operands) should be flushed to the main memory before PIM computation on that data is invoked, and data updated by PIM computation should be invalidated in the host's caches. Both of these aspects are orthogonal to the granularity of PIM offloading and arbitration and is similar to previous PIM research. The application could issue (selective) cache flushes before launching a PIM kernel to ensure a consistent view of memoryi, or the PIM architecture may support such functionalities as part of the PIM instructions.

4.7 Methodology

We use GPGPU-Sim [11] to evaluate the performance impact of our proposed OrderLight primitive when used with a GPU host. The GPU microarchitecture and the memory parameters assumed in this work are summarized in Table 4.1.

4.7.1 Workloads

PIM is useful for data-intensive applications that have a low compute-to-memory ratio. As such, we first analyze the behavior of OrderLight and fence using the stream benchmark [93], which is representative of many kernels in General Purpose GPU (GPGPU) applications such as feature_map addition, scalar_product, activation_functions, etc. Next, we evaluate the performance improvement due to OrderLight on a set of GPGPU kernels in machine learning, data analytics, and genomic applications, as discussed in Section 4.2. Table 4.2 shows a summary of our workload suite, which represents a range of different compute-to-memory ratios.

4.7.2 Modelling PIM kernels

For our evaluation, we write the PIM kernel (CUDA code) for each workload so as to mimic the sequence of PIM instructions, similar to the pseudocode shown in Figure 4.4. As with any PIM architecture, the code requires knowledge of the memory organization, such as the interleaving granularity of physical memory across channels, size of PIM temporary storage (Section 4.4.1), etc. The fields for an OrderLight packet such as channel ID and memory-group ID are also populated using such information.

The PIM kernels use one warp per memory channel and utilize the Single Instruction

Multiple Thread (SIMT) parallelism to generate N PIM instructions in parallel (lines 2, 5, and 8 in Figure 4.4). Our evaluation shows that using one SM per two warps is sufficient to execute the PIM kernel (8 SMs for 16 memory channels). We use nontemporal store instructions to mimic PIM instructions. Similar to most fine-grained PIM instructions, nontemporal stores bypass the host caches and avoids allocating the lines in the caches. We assume that the GPU driver allocates large pages for the PIM data structures and ensure that all of the operands needed for a PIM computation align within the memory regions associated with each PIM unit.

4.7.3 Evaluation Metrics

In the results section, we use two new metrics: (1) PIM Command Bandwidth (GigaCommands/s or GC/s), which represents the number of PIM commands sent to the memory per second, and (2) PIM Data Bandwidth (GB/s), which represents the bandwidth at which PIM processes data within the memory module. Note that the PIM Data Bandwidth reflects the product of PIM command bandwidth and the bandwidth multiplication factor (BMF) over host bandwidth. PIM Command Bandwidth highlights the gains using OrderLight in the throughput of sending PIM commands, which in turn provides gains in PIM Data Bandwidth.

4.8 Results

In this section, we first discuss the benefits of using OrderLight for stream kernels, followed by its benefits for a suite of key computations from modern applications.

4.8.1 OrderLight Speedups for Stream Benchmark

Figure 4.10a shows the improvement in PIM command bandwidth and data bandwidth for four different temporary storage (TS) sizes for each of the stream kernels when using fence versus the OrderLight primitive (with a BMF of 16). Figure 4.10b shows a similar comparison but for execution time and the number of stall cycles at the core.

4.8.1.1 Improved Command Bandwidth

Ordering primitives slow down the rate at which PIM commands are sent to the memory modules due to ordering constraints, which in turn limits PIM benefits. As such, we

study the improvement in command bandwidth that OrderLight achieves over existing fence primitives. To do so, we use the Add kernel as an exemplar of other kernels we study.

The Add kernel is representative of the *vector_add* kernel shown in Figure 4.4. We observe that the command bandwidth when using the OrderLight primitive is, on average, 2.6× higher than when using the fence primitive. This is largely due to the elimination of the stall cycles induced by the fence at the core, because of which the memory controller is also starved from issuing PIM commands.

The size of TS associated with a PIM compute unit dictates two important factors that influence the command bandwidth, and as such, we vary this parameter to study its effect. First, it dictates the number of ordering primitives that are necessary. The larger the TS size, more PIM instructions can be sent before issuing an ordering primitive: when copying data from memory to TS prior to performing computation on that data, no ordering is necessary among multiple independent copy commands. Ordering is only required at the end of the copy commands before computations are performed on the data now in TS. The decreasing number of fences with increasing TS size improves the command bandwidth for fences, as evident in Figure 4.10a. This factor has a negligible impact on OrderLight as it does not pay a significant penalty of wait cycles at the core per OrderLight instruction.

Second, TS also dictates the peak achievable command bandwidth. This is so, as it dictates the DRAM locality. Consider the Add kernel again. It accesses three different vectors, a, b, and c, each of which gets mapped to a different DRAM row due to the memory mapping policy. The kernel issues a series of instructions to one vector before switching to the next vector (Figure 4.4). Figure 4.11 shows the timing constraints for a similar example that writes two vectors (p and q). For a TS sized to hold 256B of data from p, the memory controller can issue 8 row-hit accesses (32B each) for vector p before switching to a different row for vector q, within a period of 44 cycles. The latency of opening and closing a row $(t_{RCDW} + t_{WTP} + t_{RP})$ limits the command bandwidth. The achievable peak command bandwidth, in this case, is 2.3 GC/s (8/44 * peak). We observe in Figure 4.10a that, while the command bandwidth achieved with fences is far below that of the peak, OrderLight comes close to this peak (2.1 GC/s).

An equally important factor that also affects peak command bandwidth is the number

of operands read or written from/to memory by a computation as different operands typically map to different DRAM rows. As an example, Scale benchmark only works on a single DRAM row in comparison to Add benchmark, which works on three DRAM rows. Consequently, the overheads of row open/close are much lower for the Scale benchmark, increasing its achievable peak command bandwidth. Further, as Scale only works on a single DRAM row, TS (and its size) is immaterial for this benchmark. We see that the command bandwidth of Scale is around 5.6 GC/s, which is roughly half of the peak (peak is 12.6 GC/s). This can be attributed to the t_{CCDL} limitation, which limits sending one command to a memory array per two cycles. Other benchmarks follow trends similar to the Add benchmark because of accessing multiple vectors, which leads to row misses.

Overall, across kernels, OrderLight considerably improves the command bandwidth for PIM commands and often reaches the peak command bandwidth possible.

4.8.1.2 Improved Data Bandwidth

Figure 4.10a also shows the data bandwidth offered by PIM. Recall that memory bandwidth available for PIM compute units is often much higher than that available to the host and is the key benefit of PIM. We see that the data bandwidth with OrderLight is higher than the peak external data bandwidth of the memory module (405 GB/s) by 4.3× on average and outperforms the data bandwidth with fences by 3.8×.

4.8.1.3 Improved PIM Speedup Over Host

Figure 4.10b shows that the use of fences slows down PIM execution drastically to show little to no benefits over GPU execution (green bars in the graph). PIM execution with fences shows improvement over GPU only when using larger TS (1/4 or 1/2 of row-buffer) by 2× to 3.4×. On the other hand, OrderLight consistently outperforms GPU execution time for every TS size by 3.5× to 7.4× on average.

The key reason for improved PIM speedups is attributed to reduced stall cycles at the core in sending PIM instructions, which in turn leads to improved command bandwidth. This is depicted in Figure 4.10b, which shows that the number of stall cycles for the two different primitives closely resemble the plot for execution time. When using OrderLight, the number of stall cycles decreases with bigger TS because the memory controller can issue commands at a faster rate, thereby decreasing backward pressure on queues in the

memory pipe.

4.8.1.4 Speedups for Varied PIM Solutions

As discussed in Section 4.4.1, by varying placement and cardinality of PIM compute units, we can study the benefits of OrderLight for varied PIM solutions. We study three different BMF (bandwidth multiplication factor) for PIM over the host in Figure 4.12 using the Add kernel. We observe in this figure that OrderLight consistently performs better than fences by 1.9× to 3.1×. In fact, the performance with fences is worse or comparable to GPU execution in 8 of the 12 cases, whereas OrderLight provides improvement over GPU in 10 of the 12 cases. This is because, with decreasing BMF, a greater number of PIM commands have to be sent to the memory to perform the same job, which increases the burden with fences.

4.8.2 Speedups for Data-Intensive Applications

Figure 4.13 shows the performance improvement with OrderLight primitive over fences for a set of important data-intensive computations from GPGPU applications. We observe that OrderLight provides 5.5× to 8.5× improvement in execution time over the fence primitive for the suite of computations evaluated and considerably improves PIM speedups over host as compared to fence primitive.

From Table 4.2, we see that the FC and Kmeans access only one data structure per computation, which is why they experience more row locality than the other kernels. For the same reason, they show little variability in performance with different *TS* size when using the OrderLight primitive. Gen_Fil issues irregular PIM requests on 128-B data granularity (1/16th of RB). This is why Gen_Fil shows no variability with bigger *TS* size for both fence and OrderLight primitive.

We observe that FC, Kmeans, and Gen_Fill kernels show significant improvement in execution time with OrderLight even for bigger *TS* size. As shown in Figure 4.13, the number of ordering primitives issued per PIM instruction decreases with an increase in *TS* size at a much slower rate for these kernels in comparison to the other kernels (rate of decrease: FC: 33%, Kmeans: 22%, Gen_Fil: 0%, Others: 50%). This is because of the computation structure of these kernels. Thus, a lot more ordering primitives are issued even for bigger *TS* sizes, which hurts the performance when using fences.

4.8.3 Related Works

In this section, we briefly discuss some FGO/FGA PIM designs from the literature and also contrast OrderLight with fences for persistent memory ordering, which, at a high-level, appear related.

4.8.4 Fine-Grain Offload and Fine-Grain Arbitration PIM

Although alternate non-FGO/FGA PIM designs have merits as discussed in Section 4.3, FGO/FGA PIM designs are particularly suitable for modern workloads with compute- and data-intensive phases. These designs are also compatible with mainstream memory interfaces. These and other characteristics discussed in Section 4.3.5 make them particularly desirable in the current computing landscape. Consequently, we focus on research works that fall under this subclass here.

Lee et al. [82] focus on designing a PIM architecture for matrix-multiplication that can be finely interleaved with commodity DRAM command behaviors. Other works have used PIM operations using slight variants of existing DRAM operations [40], [128] or instruction-granularity offload of application-specific operations to PIM [4], [105]. However, these efforts focus on the PIM architectures and do not address the requirements for efficient host-side control of PIM, which is our focus in this work. As a result, OrderLight is applicable for these (and more) FGO/FGA PIM designs and stands to complement them by providing efficient PIM operation ordering capabilities.

4.8.5 Memory Ordering for Persistence

Systems with a persistent nonvolatile memory (NVM) requires that writes "at memory" appear to have occurred in the specified order to ensure recoverability of persistent data structures. However, such an ordering is "core-centric," requiring the memory to observe writes in the same order as the core issues them. Research works such as Delegated Persist Ordering [77] track core-centric constructs such as intrathread and interthread dependencies, coherence traffic, etc. using persist buffers and bloom filters in traditional CPUs to offload epochs of persistent writes to the NVM controller. BPFS [27] uses epoch barriers to divide the program into epochs in which stores can persist concurrently. They do so by tagging all cache blocks using an epochID and using a novel cache replacement policy to write epochs to persistent memory in order. We discuss in Section 4.5.1 of Chapter

4 that the ordering requirement for fine-grained PIM instructions is ”memory-centric” and requires addressing a different set of challenges from these NVM-related approaches.

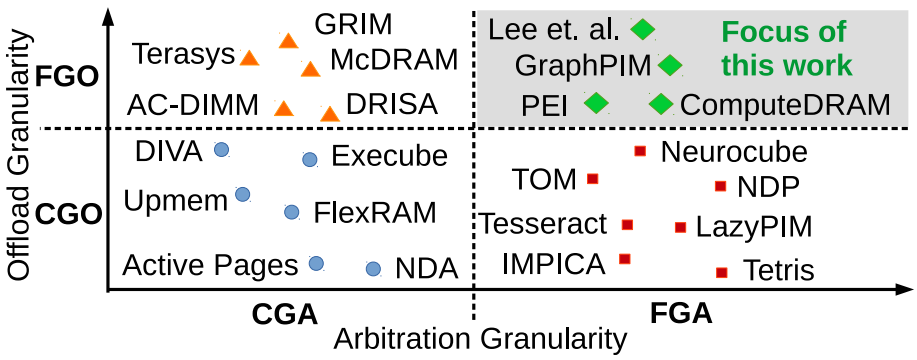

Figure 4.1. Taxonomy of PIM based on *temporal* coarse-grained and fine-grained offload (CGO & FGO) and coarse-grained and fine-grained arbitration (CGA & FGA) for host memory accesses with examples from the literature.

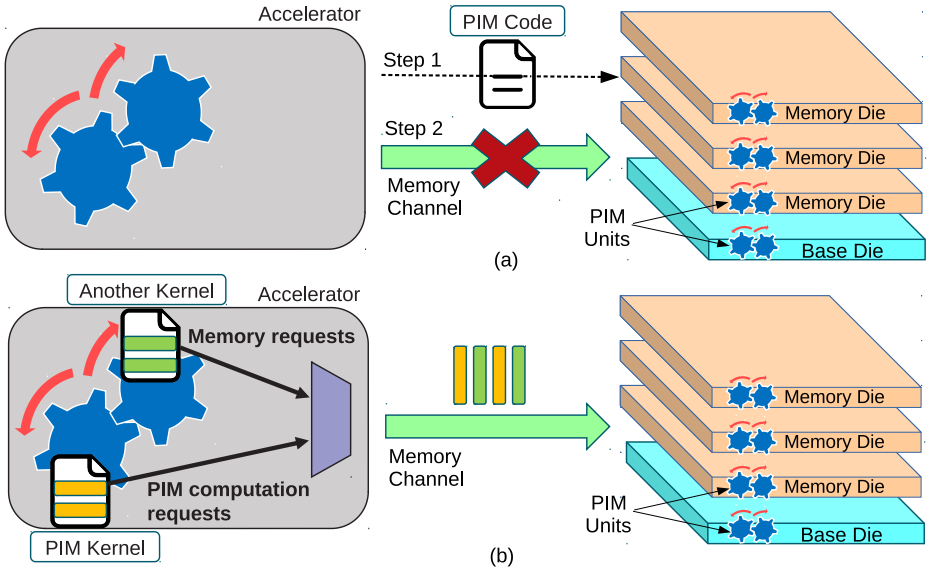

Figure 4.2. Difference between coarse-grained and fine-grained arbitration: (a) coarse–grained arbitration disallows concurrent host and PIM memory accesses, and (b) fine–grained arbitration interleaving of PIM and host memory accesses.

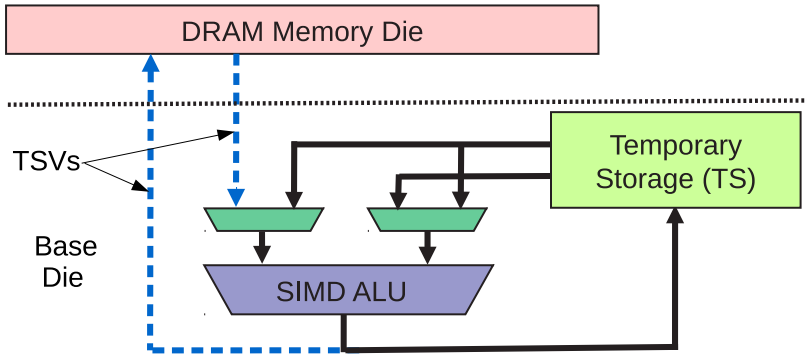

Figure 4.3. A generic and parameterized PIM compute unit with temporary storage (TS) and SIMD ALU. One or more such compute units may be placed in multiple locations per channel to obtain bandwidth multiplication over the host.

Vector_Add: c[i] = a[i] + b[i]

Host PIM Kernel	Memory Controller
while(computations not done within a memory channel)	
Load *a* ← N times	1. Precharge, Activate //Open row *a* 2. PIM_Load *a* (from DRAM to TS) ← N times
Insert fence here	
Fetch-and-Add *b* ← N times	3. Precharge, Activate //Open row *b* 4. Fetch *b* (to ALU) 5. PIM_Add *b* to *a* (store result *c* in TS) ← N times
Insert fence here	
Store *c* ← N times	6. Precharge, Activate //Open row *c* 7. PIM_Store *c* (from TS to DRAM) ← N times
Insert fence here	

Figure 4.4. Host PIM kernel and fine-grained commands sent by the memory controller for the vector_add kernel.

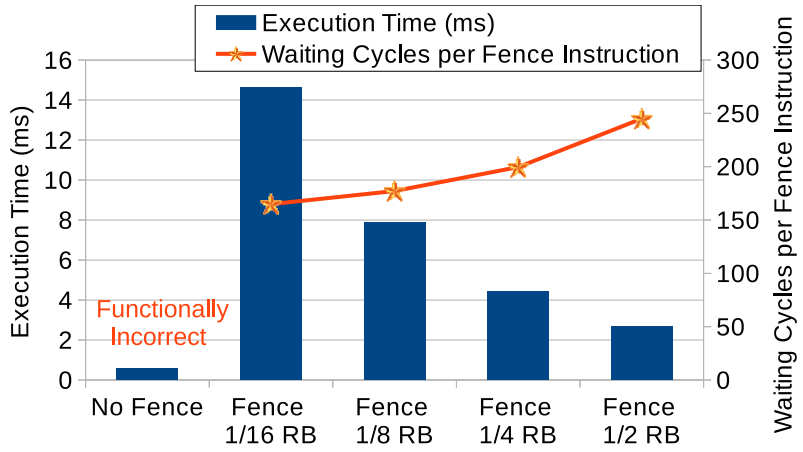

Figure 4.5. Fence overhead for vector_add kernel.

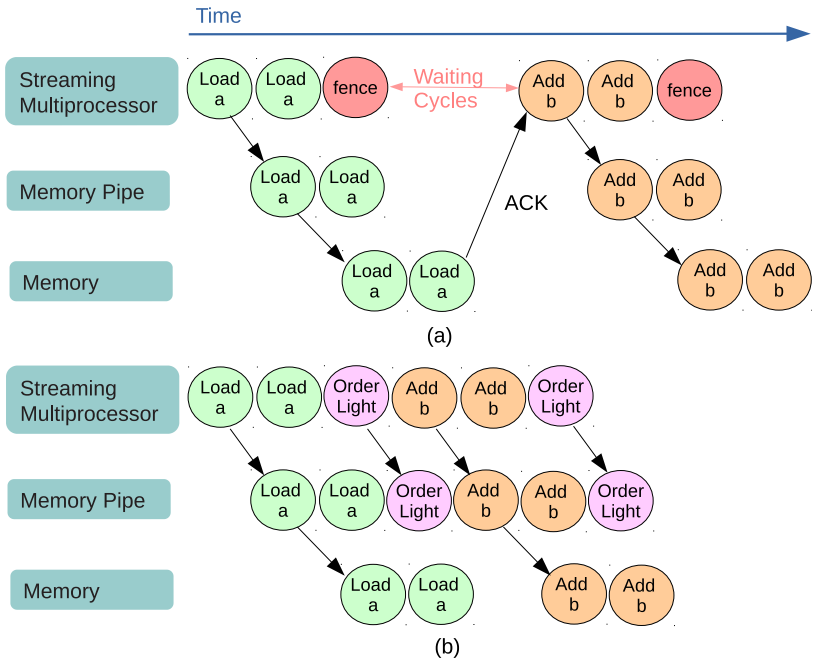

Figure 4.6. Behavior of fence versus OrderLight: (a) fence keeps the host (SM in GPU) stalled to enforce ordering, and (b) OrderLight packet ensures order at the memory controller by percolating through the memory pipe obviating stalls at the host.

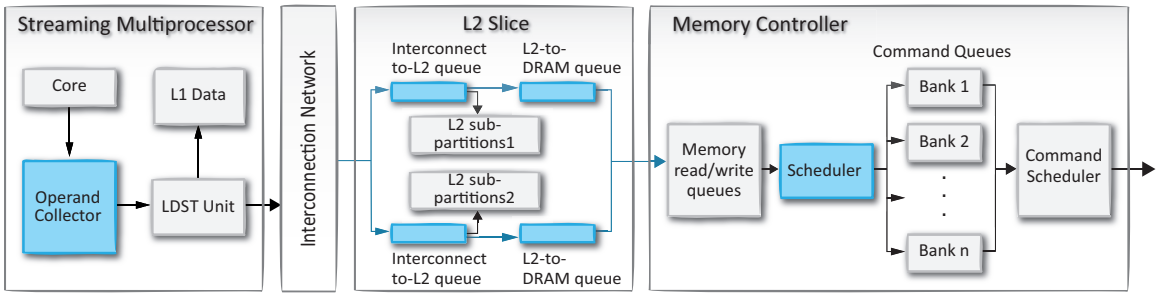

Figure 4.7. Core and memory pipe in GPU. Modules that reorder are highlighted in blue.

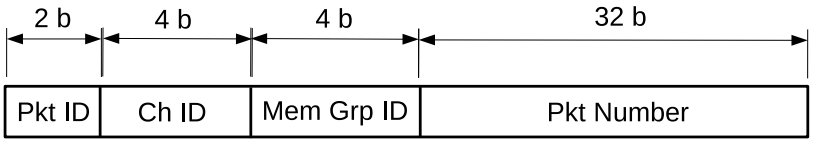

Figure 4.8. Different fields of an OrderLight packet.

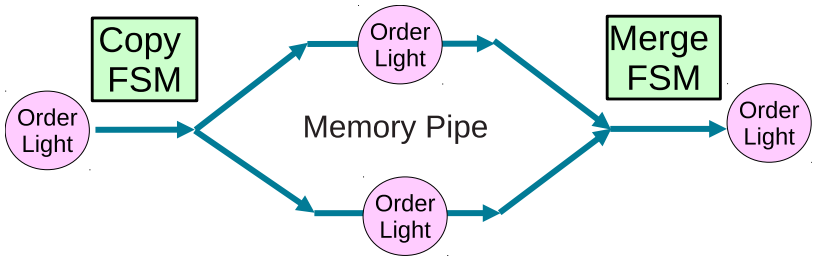

Figure 4.9. The copy-and-merge technique used for OrderLight packet when divergence is encountered in the memory pipe.

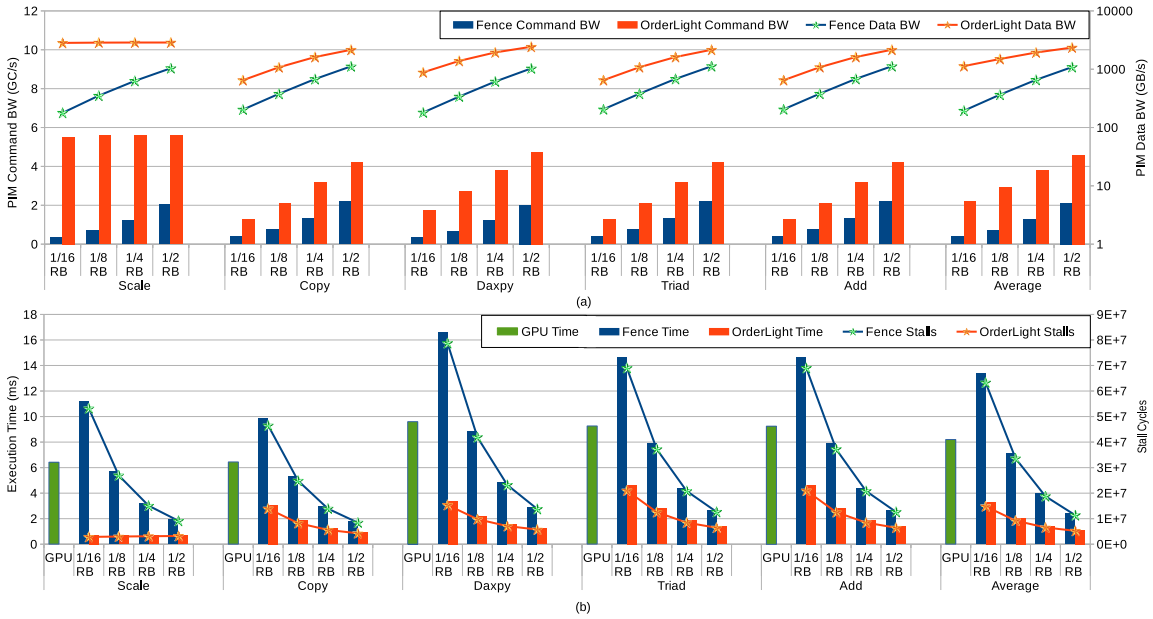

Figure 4.10. Comparison of fence versus OrderLight for stream benchmarks using the following metrics: (a) PIM command bandwidth (linear scale) and data bandwidth (log scale), and (b) execution time and the number of stall cycles at the core.

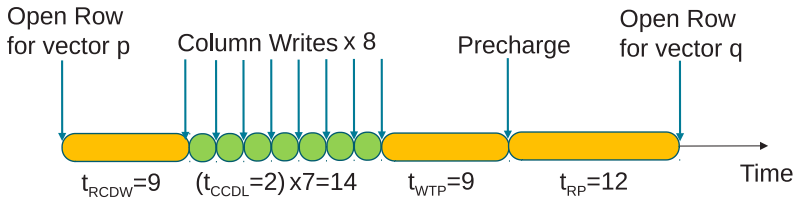

Figure 4.11. DRAM timing for opening the DRAM row for vector p, sending 8 write requests (equivalent to PIM commands), followed by opening the row for vector q.

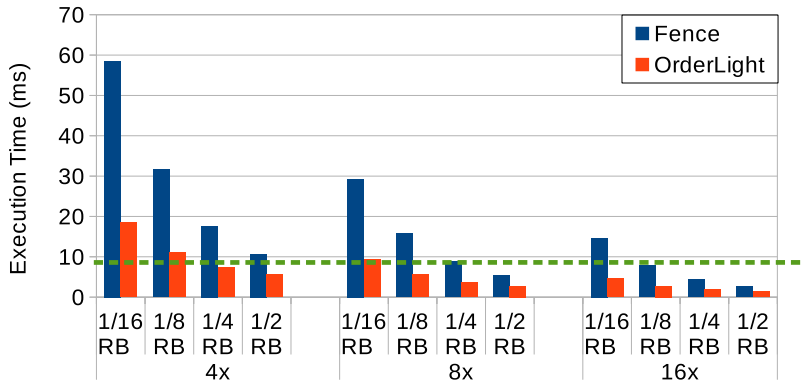

Figure 4.12. Comparison of fence and OrderLight using different Bandwidth Multiplication Factor for the Add kernel.

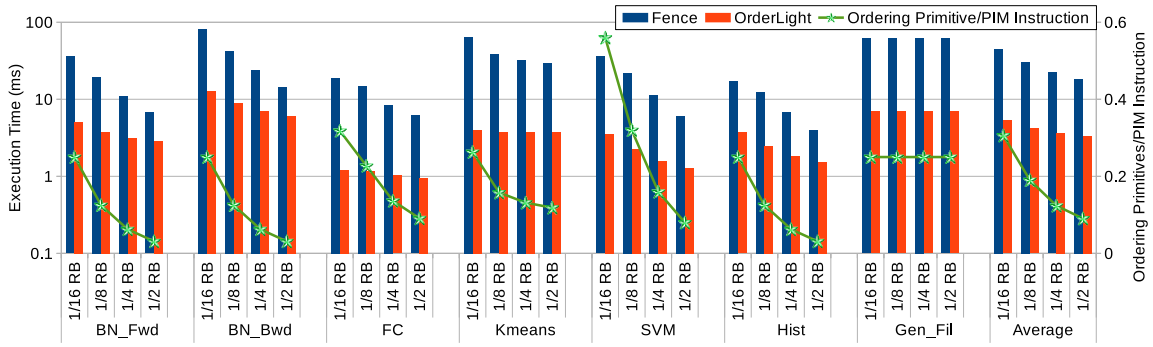

Figure 4.13. Improvement in execution time using the OrderLight primitive over fences for a set of data-intensive computations in GPGPU applications. The graph also plots the number of ordering primitives issued per PIM instruction for each kernel.

Table 4.1. Simulator details.

GPU Parameters			
GPU Model:		Volta Titan V	
Number of SMs:	80	Core Frequency:	1200 MHz
L1 Data Size:	32 KB	Shared Memory Size:	96 KB
L2 Size:	3 MB	L2 Queue Size:	64
Memory Scheduler:	FRFCFS	R/W Queue Size:	64
Memory Parameters			
Memory Model:		HBM	
Memory Channels:	16	DRAM Bus Width:	16
Banks per Channel:	16	Memory Frequency:	850 MHz
Memory Timing: (in cycles)	CCD=1:RRD=3:RCDW=9:RAS=28:RP=12: CL=12:WL=2:CDLR=3:WR=10:CCDL=2:WTP=9		

Table 4.2. Summary of workloads.

Kernels	Description	Compute: Memory Ratio	More than one data structure accessed?
Stream Benchmark			
Scale	a[i] = scalar*a[i]	1:1	No
Copy	b[i] = a[i]	0:2	Yes
Daxpy	b[i] = b[i] + scalar*a[i]	2:2	Yes
Triad	c[i] = a[i] + scalar*b[i]	2:3	Yes
Add	c[i] = a[i] + b[i]	1:3	Yes
Other Workloads			
BN_Fwd [62]	Batch Normalization Forward Phase	7:3	Yes
BN_Bwd [62]	Batch Normalization Backward Phase	14:6	Yes
FC [79]	Fully Connected	2:1	No
KMeans [21]	KMeans Clustering	10:1	No
SVM [107]	Support Vector Machine	2.5:2	Yes
Hist [110]	Histogram	3:2	Yes
Gen_Fil [73]	Genomic Sequence Filtering (GRIM Algo)	3:1	No

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The growing demands for data-centric computations will usher a new era of research in Near Data Processing. Moving forward, research in significantly improving the efficiency and performance of distant-term NDP technologies (memristive computing and in-cache computing) will lead to more investment from the industry. On the other hand, Processing In Memory (PIM) is already a viable option for big companies and startups (such as Upmem), but the barriers to entry are the low-level microarchitectural details such as control mechanisms to orchestrate PIM. This dissertation addresses challenges in both ends of the NDP spectrum and makes a case for NDP in the near and distant future.

In the next three sections, I conclude this dissertation with a high-level overview and closing remarks on each of the three projects. These sections also highlight the key insights that can be concluded from the research ideas proposed. This is followed by discussions on future research directions related to each of the projects.

5.1 In-Situ Analog Computing for Machine Learning

In the Newton accelerator, we show that a toolkit that exploits heterogeneity and common-case resource requirements can unearth a 2× improvement. We target resource provisioning and efficiency in a crossbar-based deep network accelerator. Starting with the ISAAC architecture, we show that three approaches – heterogeneity, mapping constraints, and divide and conquer – can be applied within a tile and within an IMA. This results in smaller eDRAM buffers, smaller HTree, energy-efficient ADCs with a varying resolution, energy- and area-efficiency in classifier layers, and fewer computations. The Newton architecture cuts the current gap between ISAAC and an ideal neuron in half.

Many of the ideas proposed in Newton would also apply to a general accelerator for matrix-matrix multiplication, as well as to other neural networks such as Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), etc. Divide and conquer

based techniques can also be used to reduce the complexity of crossbar implementation to accelerate postquantum cryptography algorithms. Memristor-based accelerators such as ISAAC and Newton can be used in the Dual In-line Memory Module (DIMM) form factor and installed in modern rack architectures to design large scale systems. Such systems can be useful to simulate the 100 billion neurons of a human brain in order to achieve human-level intelligence.

5.2 In-Cache Computing for Genomics

In the GenCache work, we show that in-cache operators can be leveraged to provide significant speedups for sequence alignment. In particular, the filtration operations require access to large datasets, and prior work has overcome this bottleneck with tiling. With new in-cache operators and a reconfigured memory hierarchy, we show the potential for high parallelism. We analyze the sequence alignment workload to identify redundant work and create a new error-aware four-phase pipeline that is a better fit for the GenCache architecture. The algorithm alone, or the in-cache operators alone yield small speedups of 1.36× and 1.62×; their combination yields a more than additive speedup of 5.26×. The improvements are caused by higher parallelism, fewer memory fetches, and elimination of redundant work. The 15× reduction in memory accesses is especially important when future security/privacy measures will further penalize off-chip accesses. Our circuit analysis shows that the additional in-cache logic has a small area/power overhead. We also show significant benefits by exploiting in-cache operators in third-generation pipelines. We have thus helped alleviate the significant memory bottleneck noted by both recent genomic accelerators, GenAx and Darwin.

The introduced operators in GenCache can be exploited by other stages of the genomic pipeline for additional improvements and are left for future work: reference-based compression, INDEL Realignment, variant calling, and protein sequencing. The combination of in-cache operators, algorithm restructuring, and the proposed SRAM allocation into cache/Bloom/scratchpad can also apply to other workloads that rely on bitwise operators, and that must manage irregular memory accesses: Viterbi search for speech recognition, similarity search in databases, hash-based joins for database queries, and document filtering in web search. This work, therefore, provides further evidence that in-cache operators

are useful for a broad class of applications.

5.3 Processing in Memory for Memory-Bound Workloads

With crucial applications exhibiting both compute and data-intensive phases, it is imperative that accelerators be coupled with PIM-enabled solutions. To that end, the OrderLight work first introduces a taxonomy to better understand the design space of an accelerator (e.g., a GPU) interacting with PIM-enabled memory when considering the temporal granularity of both computation offloads to near-memory logic and arbitration of PIM and host memory accesses. Based on this taxonomy, we observe that prior PIM proposals largely focus on coarse-grain approaches, which can have steep costs. On the other hand, while fine-grain PIM approaches avoid these costs, a key impediment to realizing them is support for efficient memory ordering primitives for fine-grained PIM instructions. As such, we propose a novel memory-centric ordering primitive, OrderLight, which overcomes the shortcomings of traditional core-centric memory ordering primitives such as fences. Evaluations based on key computations from several application domains show that OrderLight delivers 5.5× to 8.5× speedup over traditional fences.

The OrderLight ordering primitive addresses challenges associated with ordering PIM instructions. However, due to the inherent nature of memory-bound processes, a lot of PIM computation requests may starve a few memory requests from compute-bound processes and lead to high latency accesses, thereby reducing overall throughput. A possible solution is to allow the compute-bound processes to provide hints to the hardware system when they are about to start a memory access phase or a computation phase. Such hints will help the hardware to prioritize the few memory requests over PIM requests. Techniques such as Decoupled Access Execute (DAE) [78] can hoist memory accesses and group them to create distinct phases of memory accesses or computations. Such grouping can help to appropriately send hints at the beginning of a phase.

Future systems will consist of a redefined memory hierarchy with near data processing capabilities; that is, core, caches, and memory will be equipped to perform computations. Computations will be performed where data resides, which leads to low latency execution for many irregular workloads such as machine learning, graph processing, etc. To achieve this, we have to figure out what computation instructions to support, the efficient pro-

gramming model, the ideal location for performing each computation, and how the cache and memory resources should be dynamically partitioned for computing versus regular memory accesses.

REFERENCES

[1] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ML: A case for collaborative ML acceleration using near-data processing," in *Proc. Int. Symp. Memory Syst.*, 2019, pp. 506–517.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proc. 23rd Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 481–492.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 105–117.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 336–348.

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. Jerger, and A. Moshovos, "Cnvlutin: Zero-neuron-free deep convolutional neural network computing," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 1–13.

[6] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnol.*, vol. 23, no. 7, p. 075201, 2012.

[7] F. Alibart, E. Zamanidoost, and D. B. Strukov, "Pattern classification by memristive crossbar circuits using ex-situ and in-situ training," *Nature*, vol. 4, no. 1, pp. 1–7, 2013.

[8] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: A new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinf.*, vol. 33, no. 21, pp. 3355–3363, 2017.

[9] H. Asghari-Moghaddam, Y. Son, J. Ahn, and N. Kim, "Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems," in *Proc. 49th Int. Symp. Microarchit.*, 2016, pp. 1–13.

[10] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of Caffe, Neon, Theano, and Torch for deep learning," *arXiv preprint arXiv:1511.06435*, 2015.

[11] A. Bakhoda, G. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.

[12] R. Balasubramonian *et al.*, "Near-data processing: Insight from a workshop at MICRO-46," in *IEEE Micro's Special Issue on Big Data*, vol. 34, no. 4, 2014, pp. 36–42.

[13] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *Proc. 40th Int. Symp. Comput. Archit.*, 2013, pp. 1–12.

[14] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Proc. 22nd Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 1–13.

[15] J. K. Bonfield and M. V. Mahoney, "Compression of FASTQ and SAM format sequencing data," *PLoS One*, vol. 8, no. 3, p. e59190, 2013.

[16] A. Boroumand *et al.*, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 46–50, 2016.

[17] B. E. Boser, E. Sackinger, J. Bromley, Y. Le Cun, and L. D. Jackel, "An analog neural network processor with programmable topology," *IEEE J. Solid-State Circuits*, vol. 26, no. 12, pp. 2017–2025, 1991.

[18] G. W. Burr *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element," in *Proc. 60th Int. Electron Devices Meeting*, 2014, pp. 3498–3507.

[19] S. Byma *et al.*, "Persona: A high-performance bioinformatics framework," in *Proc. 26th USENIX Annu. Tech. Conf.*, 2017, pp. 153–165.

[20] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proc. 25th Great Lakes Symp. VLSI*, 2015, pp. 199–204.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[22] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Arch. Support Program. Lang. and Oper. Syst.*, 2014, pp. 269–284.

[23] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 367–379.

[24] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Int. Symp. Microarchit.*, 2014, pp. 609–622.

[25] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 27–39.

[26] J. Cloutier, S. Pigeon, F. R. Boyer, E. Cosatto, and P. Y. Simard, "VIP: An FPGA-based processor for image processing and neural networks," in *Proc. Int. Conf. Microelectron. Neural Netw.*, 1996, pp. 330–336.

[27] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 133–146.

[28] W. Dally, "Challenges for future computing systems," 2015. [Online]. Available: https://www.cs.colostate.edu/ cs575dl/Sp2015/Lectures/Dally2015.pdf

[29] F. Devaux, "The true processing-in-memory accelerator," in *Proc. 31st Symp. Hot Chips*, 2019, pp. 1–24.

[30] J. Draper *et al.*, "The architecture of the DIVA processing-in-memory chip," in *Proc. 16th Int. Conf. Supercomputing*, 2002, pp. 14–25.

[31] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 92–104.

[32] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proc. 19th Asia and South Pacific Des. Automat. Conf.*, 2014, pp. 201–206.

[33] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. 45th Int. Symp. Comput. Archit.*, 2018, pp. 383–396.

[34] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. Conf. Comput. Vision and Pattern Recognit. Workshop*, 2011, pp. 109–116.

[35] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Programmable Log. and Appl.*, 2009, pp. 32–37.

[36] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 283–295.

[37] P. Foley *et al.*, "Accelerate genomics research with the Broad-Intel genomics stack," 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerate-genomics-research-with-the-broad-intel-genomics-stack-paper.pdf

[38] D. Fujiki *et al.*, "GenAx: A genome sequencing accelerator," in *Proc. 45th Int. Symp. Comput. Archit.*, 2018, pp. 69–82.

[39] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 397–410.

[40] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-memory compute using off-the-shelf DRAMs," in *Proc. 52nd Int. Symp. Microarchit.*, 2019, pp. 100–113.

[41] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. 24th Int. Conf. Parallel Archit. and Compilation Techn.*, 2015, pp. 113–124.

[42] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proc. 22nd Int Conf. Arch. Support for Program. Lang. and Oper. Syst.*, 2017, pp. 751–764.

[43] Genetics science learning center, "Your doctor's new genetic tools," 2017. [Online]. Available: http://learn.genetics.utah.edu/content/precision/example/

[44] R. Genov and G. Cauwenberghs, "Charge-mode parallel architecture for vector-matrix multiplication," *IEEE Trans. Circuits and Syst. II: Analog and Digit. Signal Process.*, vol. 48, no. 10, pp. 930–936, 2001.

[45] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Comput.*, vol. 28, no. 4, pp. 23–31, 1995.

[46] S. Goodwin, J. Gurtowski, S. Ethe-Sayers, P. Deshpande, M. C. Schatz, and W. R. McCombie, "Oxford nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome," *Genome Res.*, vol. 25, no. 11, pp. 1750–1756, 2015.

[47] Graphcore, "Graphcore intelligence processing unit," 2017.

[48] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. 38th Int. Conf. Acoust., Speech, and Signal Process.*, 2013, pp. 6645–6649.

[49] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *Proc. 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 615–626.

[50] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. Friedman, "AC-DIMM: Associative computing with STT-MRAM," in *Proc. 40th Int. Symp. Comput. Archit.*, 2013, pp. 189–200.

[51] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Automatic abstraction and fault tolerance in cortical microachitectures," in *Proc. 38th Int. Symp. Comput. Archit.*, vol. 39, no. 3, 2011, pp. 1–10.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. Int. Conf. Comput. Vision*, 2015, pp. 1026–1034.

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vision and Pattern Recognit.*, 2016, pp. 770–778.

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. Eur. Conf. Comput. Vision*. Springer, 2016, pp. 630–645.

[55] Y. Ho, G. M. Huang, and P. Li, "Nonvolatile memristor memory: Device characteristics and design implications," in *Proc. 28th Int. Conf. Comput.-Aided Des.*, 2009, pp. 485–490.

[56] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm," in *Proc. 15th Int. Conf. Embedded Comput. Syst.: Archit., Model., and Simul.*, 2015, pp. 221–227.

[57] K. Hsieh *et al.*, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 204–216.

[58] K. Hsieh *et al.*, "Accelerating pointer chasing in 3D-Stacked memory: Challenges, mechanisms, evaluation," in *Proc. 34th Int. Conf. Comput. Des.*, 2016, pp. 25–32.

[59] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *Proc. 53rd Des. Automat. Conf.*, 2016, pp. 1–6.

[60] M. Hu *et al.*, "Memristor-based analog computation and neural network classification with a dot product engine," *Adv. Mater.*, vol. 30, p. 1705914, 2018.

[61] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, "MEDAL: Scalable DIMM based near data processing accelerator for DNA seeding algorithm," in *Proc. 52nd Int. Symp. Microarchit.*, 2019, pp. 587–599.

[62] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 448–456.

[63] M. Jain *et al.*, "Nanopore sequencing and assembly of a human genome with ultra-long reads," *Nature Biotechnol.*, vol. 36, no. 4, pp. 338–345, 2018.

[64] JEDEC, "High bandwidth memory DRAM (HBM1, HBM2)," 2019. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[65] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory," *IEEE J. Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[66] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" in *Proc. Int. Joint Conf. Neural Netw.*, 2012, pp. 1–5.

[67] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[68] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration," in *Proc. 12th Des., Automat., and Test Eur. Conf.*, 2009, pp. 423–428.

[69] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *Proc. 17th Int. Conf. Comput. Des.*, 1999, pp. 192–201.

[70] M. J. Khoury, "Cancer precision medicine: More population sciences ahead!" 2016. [Online]. Available: https://blogs.cdc.gov/genomics/2016/01/20/cancer-precision-ahead/

[71] D. Kim, J. H. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 380–392.

[72] H. Kim, J. Sim, Y. Choi, and L.-S. Kim, "NAND-Net: Minimizing computational complexity of in-memory processing for binary neural networks," in *Proc. 25th Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 661–673.

[73] J. S. Kim *et al.*, "GRIM-filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, pp. 23–40, 2018.

[74] K.-H. Kim *et al.*, "A functional hybrid memristor crossbar-array/CMOS system for data storage and neuromorphic applications," *Nano Lett.*, vol. 12, no. 1, pp. 389–395, 2011.

[75] Y. Kim, Y. Zhang, and P. Li, "A digital neuromorphic VLSI architecture with memristor crossbar synaptic array for machine learning," in *Proc. 3rd Symp. Cloud Comput.*, 2012, pp. 328–333.

[76] P. Kogge, "The EXECUBE approach to massively parallel processing," in *Proc. 23rd Int. Conf. Parallel Process.*, 1994, pp. 77–84.

[77] A. Kolli *et al.*, "Delegated persist ordering," in *Proc. 49th Int. Symp. Microarchit.*, 2016, pp. 1–13.

[78] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs," in *Proc. 25th Int. Conf. Compiler Constr.*, 2016.

[79] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 26th Annu. Conf. Neural Inform. Process. Syst.*, 2012, pp. 84–90.

[80] L. Kull *et al.*, "A 3.1 mW 8b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS," *IEEE J. Solid-State Circuits*, vol. 48, no. 12, pp. 3049–3058, 2013.

[81] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.

[82] W. J. Lee, C. H. Kim, Y. Paik, J. Park, I. Park, and S. W. Kim, "Design of processing-"inside"-memory optimized for DRAM behaviors," *IEEE Access*, vol. 7, pp. 82 633–82 648, 2019.

[83] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[84] S. Li *et al.*, "SCOPE: A stochastic computing engine for DRAM-based in-situ accelerator," in *Proc. 51st Int. Symp. Microarchit.*, 2018, pp. 696–709.

[85] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *Proc. 50th Int. Symp. Microarchit.*, 2017, pp. 288–301.

[86] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "RedEye: Analog convnet image sensor architecture for continuous mobile vision," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 255–266.

[87] D. Liu *et al.*, "PuDianNao: A polyvalent machine learning accelerator," in *Proc. 20th Int. Conf. Arch. Support Program. Lang. and Oper. Syst.*, 2015, pp. 369–381.

[88] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 393–405.

[89] X. Liu *et al.*, "A heterogeneous computing system with memristor-based neuromorphic accelerators," in *Proc. 18th High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.

[90] X. Liu *et al.*, "RENO: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Proc. 52nd Des. Automat. Conf.*, 2015, pp. 1–6.

[91] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," in *Proc. 41st Int. Symp. Comput. Archit.*, vol. 42, no. 3, 2014, pp. 517–528.

[92] M. Massie *et al.*, "Adam: Genomics formats and processing patterns for cloud scale computing," Univ. California, Berkeley, Tech. Rep. UCB/EECS-2013-207, 2013.

[93] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comput. Soc. TCCA Newslett.*, vol. 2, pp. 19–25, 1995.

[94] E. J. Merced-Grafals, N. Dávila, N. Ge, R. S. Williams, and J. P. Strachan, "Repeatable, accurate, and high speed multi-level programming of memristor 1T1R arrays for power efficient analog computing applications," *Nanotechnol.*, vol. 27, no. 36, p. 365202, 2016.

[95] Micron, "Hybrid memory cube specification 2.0," 2018. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf

[96] M. Might and M. Wilsey, "The shifting model in clinical diagnostics: How next-generation sequencing and families are altering the way rare diseases are discovered, studied, and treated," *Genet. Med.*, vol. 16, no. 10, pp. 736–737, 2014.

[97] N. A. Miller *et al.*, "A 26-hour system of highly sensitive whole genome sequencing for emergency management of genetic diseases," *Genome Med.*, vol. 7, no. 1, pp. 1–16, 2015.

[98] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Int. Symp. Microarchit.*, 2007, pp. 3–14.

[99] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to understand large caches," Univ. Utah, Tech. Rep., 2007.

[100] B. Murmann, "ADC performance survey 1997-2015 (ISSCC & VLSI symposium)," 2015. [Online]. Available: http://web.stanford.edu/ murmann/adcsurvey.html

[101] Mythic, "Mythic AI technology," 2020. [Online]. Available: https://www.mythic-ai.com/technology/

[102] A. Nag *et al.*, "Newton: Gravitating towards the physical limits of crossbar acceleration," *IEEE Micro Special Issue Memristor-Based Comput.*, vol. 38, no. 5, pp. 41–49, 2018.

[103] A. Nag *et al.*, "GenCache: Leveraging in-cache operators for efficient sequence alignment," in *Proc. 52nd Int. Symp. Microarchit.*, 2019, pp. 334–346.

[104] A. Nag, S. Aga, N. Jayasena, and R. Balasubramonian, "OrderLight: Lightweight memory-ordering primitive for efficient fine-grained PIM computations," 2020, manunscript under internal review at AMD.

[105] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. 23rd Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.

[106] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM J. R&D*, vol. 59, no. 2/3, pp. 17–1, 2015.

[107] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *Proc. IEEE Int. Symp. Workload Characterization*, 2006, pp. 182–188.

[108] National Cancer Institute, "The genetics of cancer," 2017. [Online]. Available: https://www.cancer.gov/about-cancer/causes-prevention/genetics

[109] A. C. Need *et al.*, "Clinical application of exome sequencing in undiagnosed genetic conditions," *J. Med. Genet.*, vol. 49, no. 6, pp. 353–361, 2012.

[110] Nvidia, "Nvidia CUDA SDK 4.2," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-archive

[111] Y. Ono, K. Asai, and M. Hamada, "PBSIM: PacBio reads simulator toward accurate genome assembly," *Bioinf.*, vol. 29, no. 1, pp. 119–121, 2013.

[112] M. Oskin, F. Chong, and T. Sherwood, "Active pages: A model of computation for intelligent memory," in *Proc. 25th Int. Symp. Comput. Archit.*, 1998, pp. 192–203.

[113] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proc. 16th Int. Conf. Supercomputing*, 2002, pp. 347–356.

[114] Y. V. Pershin and M. Di Ventra, "Experimental demonstration of associative memory with memristive neural networks," *Neural Netw.*, vol. 23, no. 7, pp. 881–886, 2010.

[115] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "NeuFlow: Dataflow vision processing system-on-a-chip," in *Proc. 55th Midwest Symp. Circuits and Syst.*, 2012, pp. 1044–1047.

[116] A. Prabhakaran *et al.*, "Infrastructure for deploying GATK best practices pipeline," 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/deploying-gatk-best-practices-paper.pdf

[117] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[118] S. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2014, pp. 190–200.

[119] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. 40th Int. Symp. Comput. Archit.*, 2013, pp. 24–35.

[120] S. Ramakrishnan and J. Hasler, "Vector-matrix multiply and winner-take-all as an analog classifier," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 2, pp. 353–361, 2014.

[121] B. Reagen *et al.*, "Minerva: Enabling low-power, high-accuracy deep neural network accelerators," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 267–278.

[122] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vision*, vol. 115, no. 3, pp. 211–252, 2014.

[123] E. Sackinger, B. E. Boser, J. Bromley, Y. LeCun, and L. D. Jackel, "Application of the ANNA neural network chip to high-speed character recognition," *IEEE Trans. Neural Netw.*, vol. 3, no. 3, pp. 498–505, 1991.

[124] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, 2008, pp. 431–438.

[125] A. Schlapka, "Micron announces shift in high performance memory roadmap strategy," 2018. [Online]. Available: https://www.micron.com/about/blog/2018/august/micron-announces-shift-in-high-performance-memory-roadmap-strategy

[126] T. Schmitz, "The rise of serial memory and the future of DDR," 2015. [Online]. Available: https://www.xilinx.com/support/documentation/white-papers/wp456-DDR-serial-mem.pdf

[127] V. Seshadri *et al.*, "Fast bulk bitwise AND and OR in DRAM," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 127–131, 2015.

[128] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Int. Symp. Microarchit.*, 2017, pp. 273–287.

[129] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 14–26.

[130] S.Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 243–254.

[131] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. 41st Int. Symp. Comput. Archit.*, 2014, pp. 97–108.

[132] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "McDRAM: Low latency and energy-efficient matrix computations in DRAM," *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.*, vol. 37, no. 11, pp. 2613–2622, 2018.

[133] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[134] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[135] P. Srivastava *et al.*, "PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *Proc. 45th Int. Symp. Comput. Archit.*, 2018, pp. 43–56.

[136] R. St Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *Proc. 41st Int. Symp. Comput. Archit.*, vol. 42, no. 3, 2014, pp. 505–516.

[137] J. Starzyk and Basawaraj, "Memristor crossbar architecture for synchronous neural networks," *IEEE Trans. Circuits and Syst. I*, vol. 61, no. 8, pp. 2390–2401, 2014.

[138] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[139] A. Subramaniyan, J. Wang, E. R. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proc. 50th Int. Symp. Microarchit.*, 2017, pp. 259–272.

[140] M. Suri, V. Sousa, L. Perniola, D. Vuillaume, and B. DeSalvo, "Phase change memory for synaptic plasticity application in neuromorphic systems," in *Proc. Int. Joint Conf. Neural Netw.*, 2011, pp. 619–624.

[141] Syntiant, "Syntiant NDP101 neural decision processor," 2020. [Online]. Available: https://www.syntiant.com/ndp101

[142] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean, "Exploring the design space of specialized multicore neural processors," in *Proc. Int. Joint Conf. Neural Netw.*, 2013, pp. 1–8.

[143] S. M. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *Proc. Int. Conf. Syst., Man and Cybern.*, 1990, pp. 701–703.

[144] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. 39th Int. Symp. Comput. Archit.*, 2012, pp. 356–367.

[145] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *Proc. 23rd Int. Conf. Arch. Support Program. Lang. and Oper. Syst.*, 2018, pp. 199–213.

[146] Upmem, "PIM unlocks big data apps efficiency," 2019. [Online]. Available: https://www.upmem.com/technology/

[147] S. Venkataramani *et al.*, "SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks," in *Proc. 44th Int. Symp. Comput. Archit.*, 2017, pp. 13–26.

[148] N. Verma and A. P. Chandrakasan, "An ultra low energy 12-bit rate-resolution scalable SAR ADC for wireless sensor nodes," *IEEE J. Solid-State Circuits*, vol. 42, no. 6, pp. 1196–1205, 2007.

[149] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat caches for scale-out servers," *IEEE Micro*, vol. 37, no. 2, pp. 90–103, 2017.

[150] P. O. Vontobel, W. Robinett, P. J. Kuekes, D. R. Stewart, J. Straznicky, and R. S. Williams, "Writing to and reading from a nano-scale crossbar memory based on memristors," *Nanotechnol.*, vol. 20, no. 42, p. 425204, 2009.

[151] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the Micron Automata processor," in *Proc. 13th Int. Conf. Comput. Frontiers*, 2016, pp. 135–144.

[152] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *Proc. 25th Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 81–93.

[153] K. A. Wetterstrand, "DNA sequencing costs: Data from the NHGRI genome sequencing program (GSP)," 2017. [Online]. Available: http://www.genome.gov/sequencingcostsdata

[154] L. Wu *et al.*, "FPGA accelerated INDEL realignment in the cloud," in *Proc. 25th Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 277–290.

[155] H. Xin *et al.*, "Shifted hamming distance: A fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," *Bioinf.*, vol. 31, no. 10, pp. 1553–1560, 2015.

[156] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 476–488.

[157] C. Yakopcic and T. M. Taha, "Energy efficient perceptron pattern recognition using segmented memristor crossbar arrays," in *Proc. Int. Joint Conf. Neural Netw.*, 2013, pp. 1–8.

[158] Y.Li *et al.*, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *Proc. 49th Int. Symp. Microarchit.*, 2016, pp. 1–13.

[159] M. Zaharia *et al.*, "Faster and more accurate sequence alignment with SNAP," *arXiv preprint arXiv:1111.5572*, 2011.

[160] M. Zangeneh and A. Joshi, "Design and optimization of nonvolatile multibit 1T1R resistive RAM," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 8, pp. 1815–1828, 2014.

[161] D. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. 23rd Int. Symp. High-Perform. Parallel and Distrib. Comput.*, 2014, pp. 85–98.