In-Memory Acceleration for General Data Parallel Applications

by

Daichi Fujiki

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Daichi Fujiki
dfujiki@umich.edu
ORCID iD: 0000-0001-7949-0417

ACKNOWLEDGMENTS

The completion of this study could not have been possible without the expertise and guidance of my advisor, Dr. Reetuparna Das. Reetu was always willing and enthusiastic to assist and encourage me throughout the research project, and her deep insights in this field have made the inspiring experience for me. I would also like to give my warmest thanks to my co-advisor, Dr. Scott A. Mahlke, for his engagement and input throughout this in-memory computing project.

Aside from my dissertation project on in-memory computing architecture, I enjoyed working on genome sequencing acceleration projects. I owe a deep sense of gratitude to Dr. David Blaauw and Dr. Satish Narayanasamy for their support on the genomics projects.

I greatly acknowledge the funding received towards my PhD from the Nakajima Foundation. Thanks to Dr. Kohei Itoh and Dr. Masahiro Ono, who enlightened me on the career path with PhD in my freshman year.

I am extremely thankful to my labmate and my friend, Arun Subramaniyan, for his pioneering spirit, encouragement, and kindness. Discussions with the lab members in Mbits research group were always exciting and led to new ideas and directions. Thanks to Harini, Vidushi, Xiaowei, Charlie, Kush, Nate, Shunhao, Jack, Tim, Hari, Yufeng, and Ali!

Finally, I must express my very profound gratitude to my family. I would like to give special thanks to my wife Haruka Fujiki and my parents as a whole for their continuous support and understanding. This dissertation stands as a testament to your patience, unconditional love, and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

General purpose processors and accelerators including system-on-a-chip and graphics processing units are composed of three principal components: processor, memory, and interconnection of these two. This simple but powerful architecture model has been the basis of computer architecture for decades. However, the recent data-intensive trend in computation workloads has observed bottlenecks in this fundamental paradigm of computers. Studies show that data communication takes 1,000$\times$ time and 40$\times$ power compared to arithmetic performed in the processors.

Processing-in-Memory (PIM) has long been an attractive idea that has the potential to break the well known memory wall problem. PIM moves compute logic near the memory, and thereby reduces data movement. In contrast, certain memories have been shown that they can morph themselves into compute units by exploiting the physical properties of the memory cells, making them intrinsically more efficient than PIM. Modern computing systems devote a large portion (more than 90%) of aggregate die area for passive memories; thus, re-purposing them for active computing units brings substantial benefits. However, prior work has only provided low-level interfaces for computation or relied on a manual mapping of machine learning kernels to the compute-capable memories. The main goal of this dissertation is to extend the compute capability of memory arrays and make them applicable to a wide range of data-parallel applications.

First, a processor architecture is proposed that re-purposes resistive memory to support data-parallel in-memory computation. The proposed execution model seeks to expose the available parallelism in a memory array by supporting a programming model that merges the concepts of data-flow and vector processing. This is empowered by a compiler that transforms Data Flow

Graphs of tensor programs to a set of data-parallel code modules with memory ISA. Second, this dissertation presents Duality Cache architecture that flexibly transforms caches on demand into an in-memory accelerator that can execute arbitrary data-parallel programs. The proposed architecture adopts the SIMT execution model and uses CUDA/OpenACC framework as the programming frontend. We develop a backend compiler that compiles PTX, the intermediate representation for CUDA, for the proposed architecture. Finally, this dissertation presents a multi-layer in-memory computing framework. In-memory computing can be implemented across multiple layers of the memory hierarchy, and in such a system figuring out the right place to compute is an important question to be answered. We propose a framework that determines the appropriate level of memory hierarchy for in-memory computing and maximizes resource utilization.

We compare the performance and energy efficiency of our in-memory accelerators with server class CPU and GPU using variety of data-parallel applications. Our experimental results show that in-ReRAM computing achieves 7.5$\times$ average speedup for PARSEC applications and in-SRAM computing achieves 3.6$\times$ average speedup for Rodinia applications. Multi-layer in-memory computing can provide an overall speedup of 4.8$\times$ for Graph Neural Networks applications with a significant workload dynamism. Our multi-faceted approaches, mainly composed of enhanced arithmetic operations, parallel programming models with compilers, and parallel execution models, unlock massive compute capabilities and energy efficiency of in-memory computing for general data-parallel applications.

CHAPTER 1

Introduction

Evolution in computer hardware has been boosted by technology scaling and architectural innovations. While the phenomenal growth in computer hardware has been sustained by microfabrication advancement in the circuit followed by Moore's law combined with Dennard scaling, the golden age of technology scaling has been ceased, leading to poor voltage scaling and rising chip power density. Microarchitecture in general purpose processors has been matured, and architectural innovations have started to taper off. To attain further efficiency, specialization is the key.

There has been a rich body of work that exercises the idea of hardware specialization. Graphics Processing Unit (GPU) and System-on-Chip (SoC) are representative examples of remarkable success. GPUs first appeared as a key enabler of richer graphics interfaces and applications, providing APIs that drive fixed function units and programmable shader processors. Later, their abundant parallel computation resources have been exploited for general purpose parallel computing and enabled today's compute intensive workloads such as machine learning.

As digital devices become ubiquitous and more data is gathered and processed, data-centric applications dominate today's computing, and specialization for this important domain has been demanded. These workloads exhibit high data level parallelism and deal with a large amount of data. Thus, the performance of data-centric applications depends critically on efficient access and processing of data. The target of this works is to seek and design a specialized data-centric computing system.

General purpose processors and accelerators including GPUs follow Von-Neumann architecture. It is composed of three main components: processor, memory, and interconnection of these two. While this simple but powerful model has been the basis of computer architecture since the very first computer was invented, the recent data-centric trend in computation has witnessed bottlenecks in this fundamental paradigm of computers. Some study shows that processors expend 1,000$\times$ time and 40$\times$ power on data communication compared to arithmetic. The well known "memory wall" problem originally referred to the problem of growing disparity in speed between fast processors and slow memories. While architects have tried a variety of strategies to overcome the memory wall by exploiting locality and building deeper memory hierarchies, it has not solved the fundamental problem behind the data communication. To achieve efficiency in data-centric applications, a paradigm shift from the one that decouples memory from computation is essential.

In a modern general purpose computing system, a large portion (more than 90%) of the aggregate die area is devoted to memory systems in the memory hierarchy (on-chip caches, main memory, and storage class memory). For instance, SRAMs are utilized as low-latency temporary storage and occupy a large fraction (over 70%) of the CPU's die area. The latest Intel's server class Xeon processors devote more than 30MB SRAM just for the last level cache (LLC). This work targets to transform these large passive memory modules into active computation units for general data-centric applications by repurposing the elements used in the memory cells and the peripheral circuits.

Moving Computation Closer to Memory For decades, Processing-in-Memory (PIM) has been an attractive idea that has the potential to break the memory wall. PIM moves compute logic near the memory by leveraging technologies such as 3D stacking, and thereby reduces data movement. In contrast, certain memories have been shown that they can morph themselves into compute units by exploiting the physical properties of the memory cells, making them intrinsically more efficient than PIM.

Enabling in-place computing in memory without transferring data outside unlocks massive parallelism and computation horsepower thanks to dense memory arrays, in addition to significantly reduced the data communication cost. Bitline computing is one of the representative methods to enable in-memory computing. By simultaneously activating multiple wordlines, the voltage or current on a bitline changes based on the contents of the activated memory cells sharing the same bitline. By reading them out with a special peripheral circuit, several important operations are accomplished. Recent works have leveraged compute capability of SRAM, DRAM, and NVMs to perform bitline computing and to accelerate compute and data intensive applications such as machine learning.

While compute-capable memories offer significant benefits, previous work has only provided a low-level interface for in-memory operation or relied on a manual mapping of machine learning kernels to the memory arrays. This work extends the compute capability of the memory arrays, and makes them applicable to a wide range of data-parallel applications. As GPGPU generalizes GPU's shader architecture, this work seeks to design general in-memory computation system stacks, primarily for SRAM and emerging NVMs, to make in-memory computing accessible to a wide range of data-parallel programs.

Towards General Data-Parallel Computing in Memory We envision that large SRAM persists in modern processors as a crucial enabler of high performance computing, and NVMs will achieve popularity as the next generation main or storage-class memory. Given a wide spectrum of memories that feature small latency or large memory density, re-purposing them into a general vector processing unit will provide ample opportunities for a variety of applications to reap the benefits of in-situ computing. To that end, the holistic approach of in-memory computation stack is essential.We design simple but powerful memory ISA, execution model that can expose ILP, DLP, and TLP, and compilers that take advantage of the in-memory parallelism and transform high level arithmetic operations in TensorFlow or CUDA/OpenACC programs into low-level memory ISA. This software stack approach is mostly agnostic to the underlying memory technologies and thus can broadly help develop a software stack for in-memory computing on a spectrum of emerging memory technologies.

This work demonstrates that the holistic approach to general in-memory computing provides a huge efficiency gain for general data-parallel programs. It is unlocked by exposing the large thread/SIMD resources of in-memory computing to the kernels. For example, our Resistive RAM (ReRAM) based in-memory SIMD accelerator can expose 2 million SIMD slots, and cache based in-memory bit-serial processor can support 1,146,880 parallel floating-point operations at 3.5% processor die area overhead for a Xeon E5-2697 with 35MB cache. This is orders of magnitude higher than server class CPUs (e.g., Xeon E5-2597 has 448 slots) and GPUs (e.g., Titan-Xp has 3840 slots).

Barebone compute memories can provide limited functionalities. For example, compute SRAM supports vector logical operations (and, or, nor, xor, comparison, and not), search, data migration (copy and swap), and compute ReRAM supports analog dot-product computing of the input voltages and cell conductance following Ohm's and Kirchhoff's laws. To enable general purpose computing, this work extends the memory arrays to support in-situ operations beyond the basic computation primitives and design a simple but powerful ISA.

The execution model and programming model need to be carefully designed, considering hardware complexity, network resource requirements, and programming flexibility. To exploit the density of ReRAM, our ReRAM based accelerator employs a VLIW SIMD execution model using TensorFlow as the programming frontend. The execution model exposes DLP by unrolling and vectorizing tensors, and ILP by a compiler-assisted instruction scheduler. We parameterize the number of processing elements assigned to each vector element to balance resource usage for ILP and DLP. Moreover, TensorFlow serves as a powerful programming model for general purpose programs, not limiting itself to machine learning. With its powerful programmability and wide compatibility of the user frontend and hardware backend, Tensor Flow has grown in popularity as a programmer-oriented Domain Specific Language (DSL). It achieves programmer friendliness (e.g., programmers do not have to learn DSL for each backend architecture), while exposing parallelism explicitly and providing opportunities for fine-grained optimizations for each backend and compiler, including ours.

On the other hand, for SRAM based acceleration, the SIMT model was chosen as the programming and execution model. We observe and demonstrate that SIMT serves as a powerful and flexible programming model for data parallel programs with irregular memory access patterns or divergent execution flow. In addition, since CUDA/OpenACC are popular and widely used frameworks across different fields from scientific computing to machine learning, leveraging it for in-cache computing architecture with direct translation or trivial source code changes will archive great portability and opportunity to use the existing software. Moreover, having independent Cooperative Thread Arrays (CTAs) entails minimum network resources for inter-thread communications that happen locally within a CTA, which avoids over provisioning of hardware resources to handle all communication patterns and execution order enforcement. This work shows that a variety of parallel programs written in CUDA/OpenACC can be ported for in-memory acceleration.

In-memory computing can be implemented across multiple layers of the memory hierarchy. In such a multi-layer in-memory system, it is necessary to determine the appropriate memory layer to perform computation, as different memory substrates have different characteristics and trade-offs. We observe that efficient job processing in multi-layer in-memory computing cannot be accomplished without careful job scheduling and performance prediction to maximize resource utilization. To this end, we design a framework to take full advantage of multi-layer in-memory computing, using architecture models, static and dynamic workload analysis, and a job scheduler.

In summary, this work offers the following contributions:

In-Memory Computing with ReRAM

- IMP, a processor architecture that re-purposes resistive memory to support data-parallel in-memory computation, is designed. In the proposed architecture, memory arrays store data and act as vector processing units. ReRAM memory array is extended to support in-situ operations beyond the dot product and design a simple ISA with limited compute capability.

- A compiler that transforms DFGs in Google's TensorFlow to a set of data-parallel modules and generates module code in the native memory ISA is developed for IMP. The compiler implements several optimizations to exploit underlying hardware parallelism and unique features/constraints of ReRAM-based computation.

- Although the in-memory compute ISA is simple and limited in functionality, it is demonstrated that with a good programming model and compiler, it is possible to off-load a large fraction of general-purpose computation to memory. For instance, it can execute in memory an average of 87% of the PARSEC applications studied.

In-Memory Computing with SRAM

- *Duality Cache* architecture that re-purposes caches on demand to data-parallel accelerators capable of executing arbitrary programs is designed. The proposed architecture adopts the SIMT execution model. Cache data arrays act both as vector processing units and register files. Each thread supports in-order VLIW instructions.

- *Duality Cache* features a Turing complete ISA similar to NVIDIA's PTX [5]. SRAM arrays are extended to support floating point operations and leverage the dynamic range of operands to reduce bit-serial operation latency. In-SRAM transcendental functions (sin, cos, etc.) are supported using CORDIC algorithms [6, 7].

- A compiler backend that translates CUDA/OpenACC programs to native *Duality Cache* ISA is developed. The compiler implements several optimizations to enhance parallelism and efficiency within the constraints of in-cache computation, exploiting the unique architectural features.

Multi-Layer In-Memory Computing

- We design multi-layer in-memory computing that re-purpose multiple memories in the memory hierarchy on demand for applications with workload dynamism and diverse compute intensity. The proposed architecture offers a common programming interface and the ability to co-exist in-memory computing with a general cache or memory system.

- We conduct a detailed workload analysis for Graph Neural Networks (GNNs) in in-memory processors and show an interesting case study where GNNs can significantly benefit from multi-layer in-memory computing. We design a kernel mapping of GNN's critical kernels such as GEMM and SpMM for in-memory computing. We also show general data-parallel applications can benefit from multi-layer in-memory computing.

- Efficient job processing in multi-layer in-memory computing cannot be accomplished without careful job scheduling and performance prediction to maximize the resource utilization. We develop heuristics for the job scheduler using an analytical scaling model and a neural network based performance predictor.

The remainder of this dissertation is as follows. We discuss the background and related research work in Chapter 2. Following that, we describe our proposed research in ReRAM based in-memory computing in Chapter 3 and SRAM cache based in-memory computing in Chapter 4. Then, we propose our multi-layer in-memory computing scheme in Chapter 5. Finally, we conclude the dissertation in Chapter 6.

CHAPTER 2

Background

Modern computing systems have a deep and complex memory hierarchy that comprises different memory technologies. Such composition is targeted to exploit spatial and temporal locality of memory access that commonly exists in many workloads. Each memory has different features and tradeoffs, which are adopted to enable area hungry but fast cache system (SRAM), cheap and dense main memory (DRAM), and slow but dense, non-volatile storage class memory (NVMs). Table 2.1 summarises the features of different memory technologies. In-memory computing has been explored in multiple memory substrates, and in this chapter, we will present important baseline in-memory computing techniques for SRAM, DRAM, and ReRAM, showing representative innovations that enabled in-memory computing. The in-memory computing paradigm is also compared with near-memory computing.

Table 2.1: Comparison of various memory technologies.

	SRAM	DRAM	NAND Flash	NOR Flash	PCM	STT-MRAM	ReRAM
Cell area (F^2)	>100	6-10	<4 (3D)	10	4-20	6-50	≤2
Voltage (V)	<1	<1	<10	<10	<3	<2	<3
Read time	∼1 ns	∼10 ns	∼10 μs	∼50 ns	<10 ns	<10 ns	<10 ns
Write time	∼1 ns	∼10 ns	100 μs - 1 ms	10 μs - 1 ms	∼50 ns	<10 ns	<10 ns
Write energy (J/bit)	∼1 fJ	∼10 fJ	∼10 fJ	100 pJ	∼10 pJ	∼0.1 pJ	∼0.1 fJ
Retention	N/A	∼64 ms	>10 y	>10 y	>10 y	>10 y	>10 y
Endurance	$> 10^{16}$	$> 10^{16}$	$> 10^{4}$	$> 10^{5}$	$10^{8} \sim 10^{15}$	$> 10^{15}$	$10^{8} \sim 10^{12}$
Multibit	1	1	>4	>4	>2	1	2-7
Non-volatility	No	No	Yes	Yes	Yes	Yes	Yes
F: Feature size of lithography							

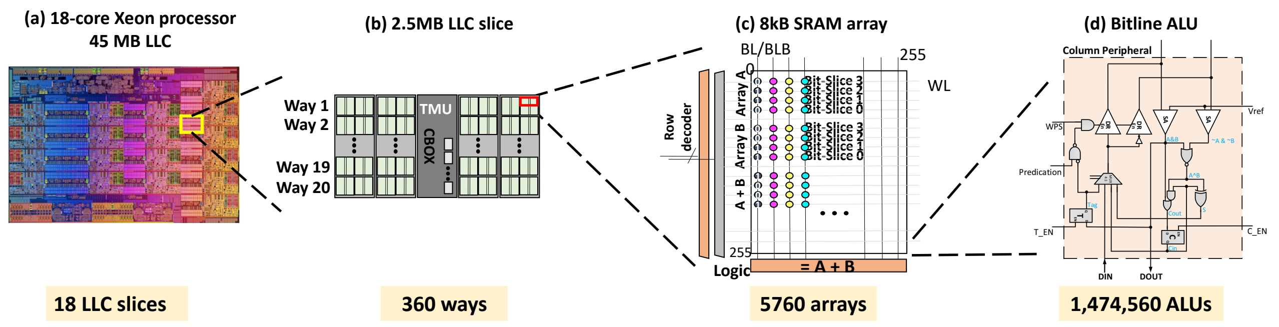


Figure 2.1: Neural Cache architecture [1].

2.1 In-SRAM Computing

Static RAM (SRAM) is one of the important types of memory used widely in today's computer systems. Data stored is static, and periodic refreshing is not needed. Data in SRAM is also volatile. SRAMs are used for the on-chip cache and register files of most of the modern processors. SRAM has larger and more complex bit cells compared to other types of memories. Therefore, SRAM has lower data density and higher manufacturing cost, and is usually used for memory structures with smaller capacity. However, the access speed of SRAM is faster than other memories. Thus, SRAM is suitable for the high-speed cache and registers that are in the top part of the memory hierarchy. In-SRAM computing has its unique advantages over other memory-centric acceleration approaches, such as cost, technology friendliness, and flexibility.

In-SRAM computing activates multiple wordlines of SRAM arrays and performs logic or arithmetic operations on vertically aligned bit cells within a column. Compute Caches [8] introduces an in-cache computation framework that supports `copy`, `zeroing`, `xor`, `compare`, and `search`. Jeloka *et al.* [9] shows data corruption due to multi-row access is prevented by lowering the word-line voltage to bias against write of SRAM array. Their measurement across 20 test chips fabricated using 28 *nm* technology demonstrates no data corruption even with activating 64 word-lines simultaneously for in-place computation. They also demonstrate the stability of six sigma robustness, equivalent to industry standard robustness against process variation, by Monte Carlo simulation.

Logic operations can be sequenced to perform arithmetic operations. Neural Cache [1] expands on compute cache's logical operation capabilities to support arithmetic operations inside the

SRAM arrays for machine learning workloads. Neural Cache vertically aligns operands in each bitline and perform computation in a bit-serial manner. As opposed to bit-parallel computing, which processes multiple bits in a single data word, bit-serial computing processes bit-by-bit, taking multiple cycles to produce results. However, bit-serial computing can store carries in a latch along the bitline, saving the complexity of communication for carry propagation across bitlines and allowing to support configurable precision. Data is transposed by a transpose memory unit (TMU) placed in the cache control box [1]. TMU design is based on an 8T SRAM array.

Data is vertically mapped to each bitline. Each n-bit element is stored across n wordlines, and thus each wordline holds one *bit-slice* of 256 vector elements as shown in Figure 2.1 (c). The bits in each bit-slice are of the same bit position of the data type.

By activating two word-lines in the SRAM, we are able to sense `logical and` at bit-line (BL) and `logical nor` at bit-line complement (BLB). Note, a re-configurable differential senseamp [8] is used to sense BL and BLB independently. A 1-bit *full adder* can be implemented using a few extra gates at the peripheral as shown in Figure 2.1 (d). When activating two wordlines, the values in each wordline are added together with the carry in the latch, and a new sum and carry are generated. The sum can be written to a different wordline in the same cycle. By adding each bit iteratively, we can perform the addition of two n bit numbers in n cycles. Multiplication takes $n^2 + 3n - 2$ cycles and is implemented as a series of additions of partial products.

Bit-serial computing in cache provides massive throughput. In the above SRAM architecture, 256 bit-lines in one 8 KB SRAM array are turned into 256 *bit-line ALUs* in a vector unit, and 5760 such 8 KB arrays in a 45MB LLC transform to 1,474,560 bit-serial ALUs (Figure 2.1) operating at a frequency of 2.5 GHz when computing. Note, while a 45 MB LLC cache access from the cores takes 20-30 ns, the smaller 8 KB SRAM arrays can themselves operate at a frequency up to 4 GHz [10].

2.2 In-ReRAM Computing

Emerging non-volatile memories (NVMs) have been an attractive memory substrate due to their high density and the potential to replace DRAM main memory. Some advanced technologies of non-volatile memories use programmable resistive elements referred to as *memristors*.

Memristors are characterized by linear current-voltage (IV) characteristics called memristance. Memristance is defined in terms of the relationship between magnetic flux linkage Φ_m and the amount of charge that has flowed q, characterized by the following memristance function, which describes the charge-dependent rate of change of flux with charge:

$$M(q) = \frac{d\Phi_m}{dq}. \tag{2.1}$$

Using the time integral of voltage $V(t) = d\Phi_m/dt$ and the time integral of current $I(t) = dq/dt$, we obtain,

$$\begin{aligned} M(q(t)) &= \frac{d\Phi_m/dt}{dq/dt} = \frac{V(t)}{I(t)}, && (2.2) \\ V(t) &= M(q(t))I(t). && (2.3) \end{aligned}$$

By regarding memristance as charge-dependent resistance, we obtain a similar relationship as Ohm's law

$$V(t) = R(t)I(t), \tag{2.4}$$

and by solving for current as a function of time,

$$I(t) = V(t)/R(t) = V(t)G(t). \tag{2.5}$$

Various memristors use material systems that exhibit thermal or ionic resistive switching effects, such as phase-change chalcogenides and solid-state electrolytes. By applying a sufficiently high voltage, the memristor cell forms conductive filaments, which enables it to transition between high resistance (reset) state and low resistance (set) state. This internal state change is retained without power, providing non-volatility. The data is read by injecting the reference voltage and sensing the current from the memristor cell through the bitline, following the relationship of Equation 2.5. Due to the linear IV characteristics of memristor (Figure 2.2 (b)), one cell can be programmed to 2^n different states (typically n is $1 \sim 5$), and decode n bit data by sensing the current magnitude. In other words, a memristor cell can function as a multi-level cell (MLC) device.

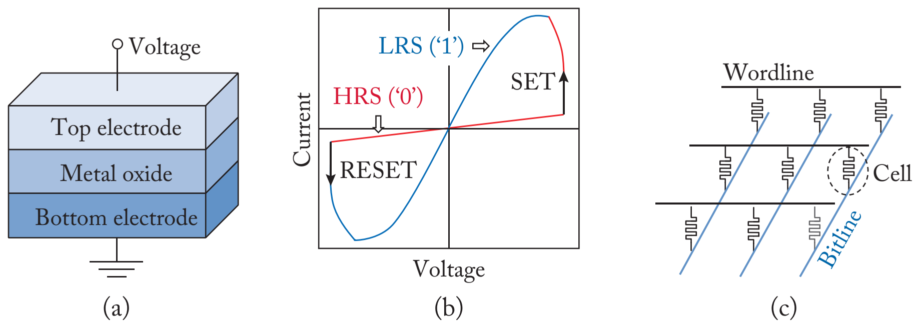


Figure 2.2: ReRAM cell and array architecture (adopted from [2]). (a) Conceptual view of a ReRAM cell; (b) I-V curve of bipolar switching; (c) schematic view of a crossbar architecture.

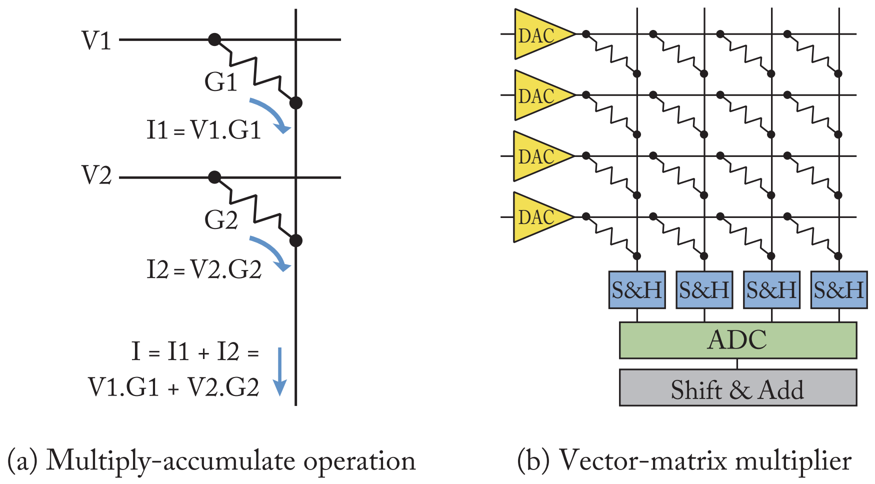


Figure 2.3: In-memory computing in ReRAM (adopted from [3]).

The linear IV characteristics of memristors are further exploited for in-memory computation in the analog domain (Figure 2.3). Changing the reference voltage within the region below the threshold voltages for set and reset still holds the memristance in Equation 2.3. As Equation 2.5

explains, the bitline current can be interpreted as the result of the multiplication of cell conductance and the input voltage. Furthermore, by activating multiple rows, currents that flow from different memristor cells sharing a bitline accumulate in the bitline, following Kirchhoff's law. This analog computing capability of memristors is first proposed for accelerating neural network workloads of which computation is dominated by multiply-accumulate (MAC) operations that compose dense matrix multiplications. For example, in such a system, weights are stored as the conductance of memristor cells, and a voltage proportional to the input activation is applied across the cells. The accumulation is performed in each bitline, as described above.

ReRAM is one of the representative memory technologies using memristors. Since ReRAMs have been introduced [11], several works have leveraged its dot-product computation functionality for neuromorphic computing [12, 13]. ISAAC [3] and PRIME [2] utilize ReRAMs to accelerate several Convolutional Neural Networks (CNNs). ISAAC proposes a full-fledged CNN accelerator with a carefully designed pipelining and precision handling scheme. PRIME studies a morphable ReRAM based main memory architecture for CNN acceleration. PipeLayer [14] further supports training and testing of CNN by introducing an efficient pipelining scheme. Aside from CNN acceleration, ReRAM arrays have been used for accelerating Boltzmann machine [15] and perception network [16]. While it has been shown analog computation in ReRAM can substantially accelerate the machine learning workloads, none have targeted general purpose computing exploiting the analog computation functionality of ReRAM. Pinatubo [17] modifies the peripheral sense-amplifier circuitry to accomplish logical operations like AND and OR in NVMs. While this approach appears promising to build complex arithmetic operations using binary primitives, doing arithmetic on multi-bit ReRAM cells using bitwise operations comes with several challenges.

ReRAM has also been explored to map the functionality of binary logic gates on memristor cells and their connectivity. This approach is referred to as gate mapping techniques in this dissertation. A single memristor can implement stateful material implication logic [18, 19], using its polarity. Assuming a memristor with binary states, its state transition will be determined based on the voltage (positive or negative) applied to the top electrode p and the bottom electrode q, and the ReRAM's internal state z. PLiM computer [20] regards this behavior as a 3-input majority gate function with an inverted input. Transformation of a memristor cell state z into a disjunctive normal form leads to the following equation using majority function M_3:

$$\begin{aligned} z_n &= (p \cdot \overline{q}) \cdot \overline{z} + (p + \overline{q}) \cdot z && (2.6) \\ &= p \cdot z + \overline{q} \cdot z + p \cdot \overline{q} && (2.7) \\ &= M_3(p, \overline{q}, z). && (2.8) \end{aligned}$$

Using Resistive Majority function ($RM_3(p, q, z) := M_3(p, \overline{q}, z)$), PLiM implements functionally complete operators and performs computation sequentially accessing a single bit in an array at a time. A compiler for PLiM computer [21] and a PLiM-based parallelized architecture using VLIW-like instruction set [22] are also proposed.

One of the limitations of the approaches above is that input data to the memristor-mapped gate has to be read out if it is stored in the memory. IMPLY [23] and MAGIC [24] propose gate mapping techniques without the need to read out the operands.

Gate-mapping techniques enable bulk logic operations with very low overhead. For some designs, the peripheral circuit need not add computation logic. However, given the fixed connectivity in the crossbar array, it is difficult to support a variety of logic operations in a small number of cycles. Thus, a single arithmetic operation composed of several logic operations can take hundreds to thousands of cycles to complete. For example, a multiply is implemented using ≈56000 majority-gate operations (majority-gate operation requires one memory cycle) and 419 ReRAM cells [21], while our analog approach based on [3] performs a multiplication in 18 memory cycles. While NVMs do not need to copy data thanks to non-destructive reads, they have limited endurance, slower operation frequency, and high write latency and energy. Thus, the gate-mapping techniques would be favorable to workloads that operate on massive read-only data with simple logic operations.

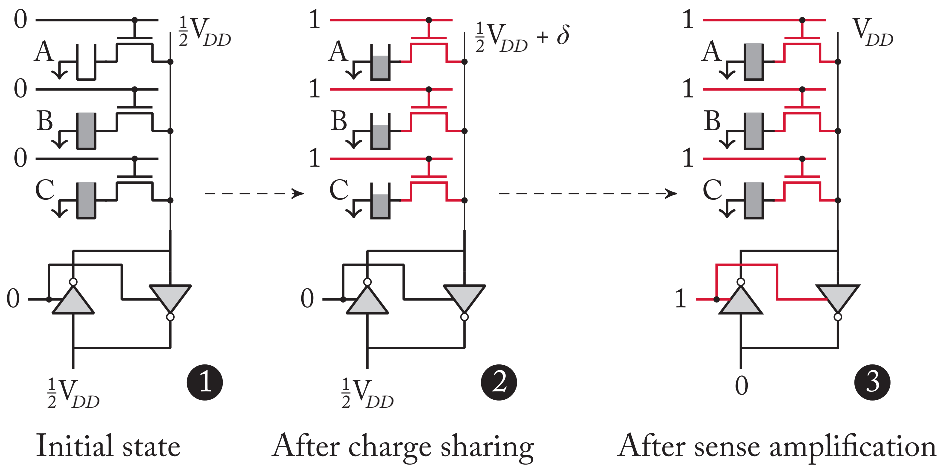


Figure 2.4: Triple Row Activation (TRA) [4]

2.3 In-DRAM Computing

While there is a large body of work that explores near-memory computing using DRAM, including 3D stacked memory and bit-serial ALUs attached to each bitline or sense amplifier ([25, 26]), there are several known obstacles for DRAM-based *in-memory* computing, such as logic cost and memory density issue. Charge sharing techniques are proposed as a key enabler of DRAM-based in-memory computing. Charge sharing techniques activate more than one wordline and perform bitwise operations by exploiting altered charges in capacitors connected to the same bitline. Hence, it can provide some important logic operations with a small area cost.

Ambit [4] proposes charge sharing based bitwise AND and OR operation. Ambit simultaneously activates three wordlines (referred to as triple-row activation or TRA), as illustrated in Figure 2.4. Based on the charge sharing principles [27], the bitline deviation δ when multiple-rows are activated is calculated as

$$\delta = \frac{kC_cV_{DD} + C_b 1/2V_{DD}}{3C_c + C_b} - \frac{1}{2}V_{DD} \quad (2.9)$$

$$= \frac{(2k-3)C_c}{6C_c + 2C_b}V_{DD}, \quad (2.10)$$

where C_c is the cell capacitance, C_b is the bitline capacitance, and k is the number of cells in the fully charged state. We assume an ideal capacitor (no capacitance variation, fully refreshed), transistor, and bitline behaviour (no resistance). According to Equation 2.10, the bitline deviation is positive (sensed as 1) if $k = 2, 3$ and negative (sensed as 0) if $k = 0, 1$. Therefore, if there are at least two fully charged cells before the charge sharing, V_{DD} is sensed, and since the sense amplifier drives the bitline to V_{DD}, all three cells will be fully charged. Otherwise, they will be discharged to 0.

The behavior of TRA is the same as a 3-input majority gate. Given A, B, and C represent the logical value of the three cells, it calculates $AB + BC + CA$, which can be transformed into $C(A + B) + \overline{C}(AB)$. Hence, by controlling C, TRA can perform AND ($C = 0$) and OR ($C = 1$). Ambit also supports NOT operation using a dual-contact cell that has an additional transistor. A combination of AND and NOT forms NAND, a functionally complete operator. Therefore, Ambit can support any logical operations.

ComputeDRAM [28] demonstrates that off-the-shelf unmodified commercial DRAMs can perform charge-sharing-based computation. They manage to activate more than one wordline of a DRAM sub-array by manipulating command sequences violating the nominal timing specification and by activating multiple rows in rapid succession. They find multiple ACTIVATE commands interposed by PRECHARGE command can activate multiple rows within a timeframe that the charge sharing is possible. Not all DRAM chips support charge-sharing by their manipulated command sequences, and not all columns always result in the desired computation result. However, their findings cast a new light that charge-sharing-based in-DRAM computation can be supported stably with minimal or no hardware changes to the DRAM DIMM.

ROC [29] takes a different approach of charge sharing to further reduce latency for logic operations, leveraging the characteristics of a diode connected with capacitors. ROC has a smaller latency than Ambit since it takes only two copy operations to compute the result. It also requires a smaller area because ROC needs only two cells during computation. To make a functionally complete operator, ROC attaches one access transistor at the bottom of a column, similar to Ambit. They also propose an enhanced ROC design with propagation and shift support by adding additional transistors and horizontal connectivity to the compute capacitors. By performing copy and computation simultaneously, ROC can reduce the computation cycles (e.g., four cycles for XOR, two cycles for AND), avoiding data corruption and result instability due to initial charges remaining in cells.

2.4 Near-Memory Computing

The early explorations of near-DRAM processing (the 1970s-1990s) were inspired by the idea of making data movement between DRAM and a processor faster by integrating the processor and DRAM into the same chip. It enables replacing the inter-chip communication with on-chip data movement. A near-DRAM processing system mainly consists of three components [30]: the scalar processing system, the memory system, and vector processing units. The scalar processing system has a similar structure as a CPU with the processor core and the cache system. The memory interface unit has a wider I/O width and interacts with the DRAM modules and caches, serving as a bridge enabling near-DRAM processing. The vector processing units are placed to perform data-parallel operations to fully take advantage of the high bandwidth of the integrated DRAM. Prior work has explored near-DRAM processing elements near DRAM arrays [30, 31, 32, 33, 34] and processing elements coupled with DRAM cells [35, 36, 37, 38]

There are multiple advantages that arise from near-memory computing. First, the memory bandwidth has significantly increased, because the bottleneck in the external memory data bus no longer exists. Second, the latency for memory access becomes shorter, since it can eliminate the long wire delay in the external data bus. Finally, the system energy efficiency is improved, as the on-chip data movement consumes less energy than the off-chip data bus.

However, such an integrated processor has a weaker performance than the normal processor in a standalone chip. This is because the processor is implemented with the DRAM process to integrate it in the same chip. The logic circuit and the SRAM arrays of the processor are slower when implemented in the DRAM process, because it is optimized for memory cost and energy, but not logic speed. Furthermore, the in-DRAM logic only has access to the two to three metal layers used in the DRAM process; thus, even a simple logic circuit results in a larger footprint than usual.

Near-Memory Computing with 3D Stacked Memory 3D stacked memory has evolved as a novel high bandwidth memory technology with the growth of data-intensive applications such as big data analytics and bandwidth greedy architectures such as GPUs. A 3D stacked memory is a 3D integrated circuit with multiple heterogeneous 2D dies stacked on each other. While the majority of the layers are memory layers that contain DRAM modules, the bottom layer of the stack consists of controlling logic, known as the logic layer. The dies communicate vertically with each other using through-silicon vias (TSV) and micro-bumps. Such vertical communication reduces the travel distance of data and improves latency and energy efficiency. Hybrid Memory Cube (HMC) [39] and High-Bandwidth Memory (HBM) [40] are two major examples of 3D stacked memories. They were developed and standardized by different entities, but they share a similar design as described above.

Since DRAM and logic are built on separate dies, they can use different transistor technology. Thus, logic layers can continue to achieve high performance, while memory layers optimize for cost and power efficiency. This is the key advantage of 3D stacked DRAM for near-memory computing: there no longer exists the disadvantage of prior work where the computing logic and memory need to be integrated on the same die, which reduces the performance of the logic built with DRAM technology nodes.

Many near-DRAM processing designs have been proposed [41, 42, 43, 44, 45, 46, 47, 48], targeting data-intensive application domains such as data analytics, graph processing, and deep learning. The logic layer is implemented with a high-speed logic process, so any near-memory computing logic can be instantiated here. At the same time, the logic layer may communicate with the DRAM modules with a high data transmission rate in TSV, thanks to the 3D-stacked structure [49].

Comparison of Near-Memory and In-Memory Computing Compared to near-memory computing, in-memory computing leverages an emerging style of in-memory computing referred to as bit-line computing or directly utilizes the physical characteristics of the dense memory arrays for computation. Since bit-line computing re-purposes memory structures to perform computation in-situ, it is intrinsically more efficient than near-memory computing which augments logic near memory. More importantly, it unlocks massive parallelism at minimal silicon cost. Compared to near-memory computing, it is not entirely true that near-memory computing can significantly minimize the data movement cost.

First, a large amount of energy is consumed by interconnects, which transfer data internally from a memory subarray to I/O. For example, DRAM's row activation takes 0.11 pJ/bit (909 pJ/row) for HBM2, while data movement and I/O take 3.48 pJ/bit at 50% toggle rate [50]. The data movement cost consists of pre-GSA (Global Sense Amplifier) data movement cost (43%), post-GSA data movement cost (34%), and I/O cost (23%). Neither in- nor near-memory computing can avoid the row activation cost and pre-GSA data movement cost. While in-memory computing can save the majority of post-GSA data movement cost and I/O cost, near-memory computing can only reduce the I/O cost unless it is placed very close to the memory array.

Second, For SRAM caches, the energy consumption of data movement is 1985 pJ while cache access consumes 467 pJ (L3 slice) [8]. Thus, H-Tree, which is the interconnect used for data transfer within a cache slice, consumes nearly 80% of cache energy spent in reading from a 2MB L3 cache slice. In-memory computing can reduce the majority of the data transfer cost, while near-memory computing near an SRAM slice cannot.

Third, the savings of near-memory computing vary based on the proximity of the near-memory computing logic to the memory arrays. While a tight coupling of the logic and memory array can provide maximum cost savings for data movement to near-memory computing as well, it may not be optimal and cost-effective from other aspects such as memory density and process technology.

Lastly, near-memory computing can reduce the latency to fetch data from memory. However, reduced communication overhead is often traded off by reduced computation throughput due to less performant cores or limited area for custom logic with enough parallelism and throughput.

CHAPTER 3

In-Memory Computing with Resistive RAM

Recent developments in Non-Volatile Memories (NVMs) have opened up a new horizon for in-memory computing. While prior work reports in-ReRAM computing can make considerable performance improvement of some specific workloads such as machine learning, its benefits originating from massive parallelism and reduction in data movement have not yet been exposed to applications in other important domains. In this chapter, we present IMP, an in-memory data parallel processor for general data parallel acceleration in ReRAM.

3.1 General Purpose In-ReRAM Computing

Non-Volatile Memories (NVMs) create opportunities for advanced in-memory computing. By repurposing memory structures, certain NVMs have been shown to have in-situ analog computation capabilities. For example, resistive memories (ReRAMs) store the data in the form of resistance of titanium oxides, and by injecting voltage into the word line and sensing the current on the bit-line, the *dot-product* of the input voltages and cell conductances is obtained using Ohm's and Kirchhoff's laws.

Recent works have explored the design space of ReRAM-based accelerators for machine learning algorithms by leveraging this *dot-product* functionality [3, 2]. These ReRAM-based accelerators exploit the massive parallelism and relaxed precision requirements, to provide orders of magnitude improvement when compared to current CPU/GPU architectures and custom ASICs, in-spite of their high read/write latency. *In this chapter, we seek to answer the question, to what extent is resistive memory useful for more general-purpose computation?*

Despite the significant performance gain offered by computational NVMs, previous works have relied on manual mapping of convolution kernels to the memory arrays, making it difficult to configure it for diverse applications. We combat this problem by proposing a programmable in-memory processor architecture and programming framework. A general purpose in-memory processor has the potential to improve performance of data-parallel application kernels by an order of magnitude or more.

3.1.1 Design Goals

The efficiency of an in-memory processor comes from two sources. The first is massive data parallelism. NVMs are composed of several thousands of arrays. Each of these arrays are transformed into a single instruction multiple data (SIMD) processing unit that can compute concurrently. The second source is a reduction in data movement, by avoiding shuffling of data between memory and processor cores. Our goal is to design an architecture, establish the programming semantics and execution models, and develop a compiler, to expose the above benefits of ReRAM computing to general purpose data parallel programs.

Architecture The in-memory processor architecture consists of memory arrays and several digital components grouped in tiles, and a custom interconnect to facilitate communication between the arrays and instruction supply. Each array acts as a unit of storage as well as a vector processing unit. The proposed architecture extends the ReRAM array to support in-situ operations beyond dot product (i.e., addition, element-wise multiplication, and subtraction). We adopt a SIMD execution model, where every cycle an instruction is multi-casted to a set of arrays in a tile and executed in lock-step. The Instruction Set Architecture (ISA) for in-memory computation consists of 13 instructions. The key challenge is developing a simple yet powerful ISA and programming framework that can allow diverse data-parallel programs to leverage the underlying massive computational efficiency.

Programming Model The proposed programming model seeks to utilize the underling parallelism in the hardware by merging the concepts of data-flow and vector processing (or SIMD). Data-flow explicitly exposes the Instruction Level Parallelism (ILP) in the program, while vector processing exposes the Data Level Parallelism (DLP). Google's TensorFlow [51] is a popular programming model for machine learning. We observe that TensorFlow's programming semantics is a perfect marriage of data-flow and vector-processing that can be applied to more general applications. Thus, our proposed programming framework uses TensorFlow as the input.

Compiler We develop a *TensorFlow compiler* that generates binary code for our in-memory data-parallel processor. The TensorFlow (TF) programs are essentially Data-Flow Graphs (DFG) where each operator node can have multi-dimensional vectors, or tensors, as operands. A DFG that operates on one element of a vector is referred to as a module by the compiler. The compiler transforms the input DFG into a collection of data-parallel modules with identical machine code. Our execution model is coarse-grain SIMD. At runtime, a code module is instantiated many times and processes independent data elements. The programming model and compiler support restricted communication between modules: reduce, scatter and gather. Our compiler explores several interesting optimizations such as unrolling of high-dimensional tensors, merging of DFG nodes to utilize n-ary ReRAM operations, pipelining compute and write-backs, maximizing ILP within a module using VLIW style scheduling, and minimizing communication between arrays.

For general purpose computation, we need to support a variety of compute operations (e.g., division, exponent, square root). These operations can be directly expressed as nodes in TensorFlow's DFG. Unfortunately, ReRAM arrays cannot support them natively due to their limited analog computation capability. Our compiler performs an instruction lowering step in the code-generation phase to translate higher-level TensorFlow operations to the in-memory compute ISA. We discuss how the compiler can efficiently support complex operations (e.g., division) using techniques such as the Newton-Raphson method which iteratively applies a set of simple instructions

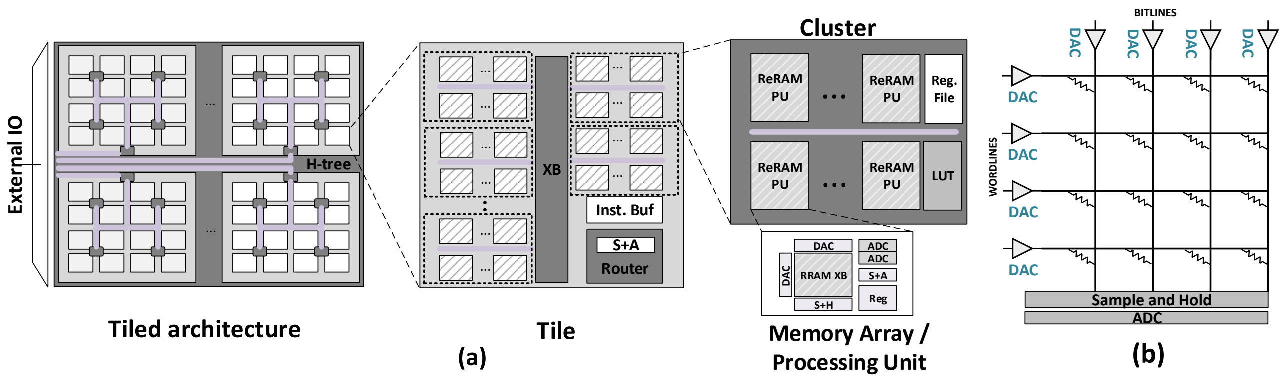


Figure 3.1: In-Memory Processor Architecture. (a) Hierarchical Tiled Structure (b) ReRAM array Structure

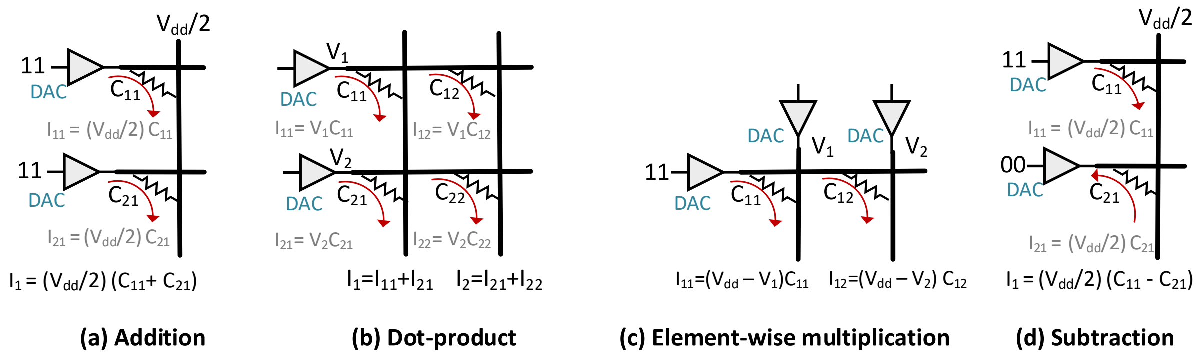


Figure 3.2: In-situ ReRAM array operations.

(add/multiply) to an initial seed from the look-up table and refines the result. The compiler also transforms other non-arithmetic primitives (e.g., square and convolution) to the native memory SIMD ISA.

3.2 Processor Architecture

We propose an in-memory data-parallel processor on ReRAM substrate. This section discusses the proposed microarchitecture, ISA, and implementation of the ISA.

3.2.1 Micro-architecture

The proposed in-memory processor adopts a tiled architecture as shown in Figure 3.1. A tile is composed of clusters of memory nodes, few instruction buffers and a router. Each cluster consists of a few memory arrays, a small register file, and look-up table (LUT). Each memory array is shown in Figure 3.1 (b). Internally, a memory array in the proposed architecture consists of multiple rows of resistive bit-cells, a set of digital-analog converters (DACs) feeding *both* the word-lines and bit-lines, sample and hold circuit (S+H), shift and adder (S+A) and analog-digital converters (ADCs). The process of reading and writing to ReRAM memory arrays remains unchanged. We refer the reader to ReRAM literature for details [11, 3]. The memory arrays are capable of both data storage and computation. We explain the compute capabilities of the memory arrays and the role of digital components (e.g. register file, S+A, LUT) in Section 3.2.2.

The tiles are connected by an H-Tree router network. The H-Tree network is chosen to suit communication patterns typical in our programming model (Section 3.3) and it also provides high-bandwidth communication for external I/O. The clusters inside a tile are connected by a router or a crossbar topology. A shared bus facilitates communication inside a cluster. A hierarchical topology inside the tile limits the network power consumption, while providing sufficient bandwidth for infrequent communication typical in data-parallel applications.

Each memory array can be thought of as a vector processing unit with few SIMD lanes. The processor adopts a SIMD execution model. Each array is mapped to a specific instruction buffer. All arrays mapped to the same instruction buffer execute the same instruction. Every cycle, one instruction is read out of the each instruction buffer and multi-casted to the memory arrays in the tile. The execution model is discussed in detail in Section 3.4.

The processor evaluated in this work consists of 4,096 tiles, 8 clusters per tile, and 8 memory arrays per cluster. Each array can store 4KB of data and has 8 SIMD lanes of 32 bits each. Consequently, the processor has aggregate SIMD width of two million lanes, aggregate memory capacity of 1GB and 494 mm^2 area. The resolution of ADC and DAC is set to 5 and 2 bits.

Opcode	Format	Cycles
add	<mask><dst>	3
dot	<mask><reg_mask><dst>	18
mul	<src><src><dst>	18
sub	<mask><mask><dst>	3
shift{l|r}	<src><dst><imm>	3
mask	<src><dst><imm>	3
mov	<src><dst>	3
movs	<src><dst><mask>	3
movi	<dst><imm>	1
movg	<gaddr><gaddr>	Variable
lut	<src><dst>	4
reduce_sum	<src><gaddr>	Variable

Table 3.1: In-Memory Compute ISA. The instructions use operand addresses specified by either <src>, <dst> or <gaddr>. The <src> and <dst> is a 8-bit local address (1-bit indicates memory/register + 7-bit row number/register number). The <gaddr> is a 4 byte global address (12-bit tile # + 6-bit array # + 7-bit row # + reserved bits). The <imm> field is a 16 byte immediate value.

3.2.2 Instruction Set Architecture

The proposed Instruction Set Architecture (ISA) is simple and compact. Compared to a standard SIMD ISA, In-memory ISA does not support complex (e.g. division) and specialized (e.g. shuffle) instructions because these are hard to do in-situ in-memory. Instead, compiler transforms complex instructions to a set of `lut`, `add` and `mul` instructions as discussed later. The ISA consists of 13 instructions as shown in Table 3.1. Each ReRAM arrays executes the instruction locally, hence the operand addressing modes reference rows inside the array or local registers. The instructions can have a size of up to 34 bytes. Now we discuss the functionality and implementation of individual instructions.

1) add The `add` instruction is an n-ary operation that adds the data in rows specified by <mask>. The <mask> is a 128-bit mask which is set for each row in the array that participates in addition. Figure 3.2 (a) shows an add operation. The mask is fed to word-line DACs, which is used to apply a Vdd ('11') or Vdd/2 ('10') to the word-lines. A '1' in the mask activates a row. Each bit-cell in a ReRAM array can be abstractly thought of as variable resistor. Addition is performed inside the array by summing up currents generated by conductance (=resistance^{-1}) of each bit-cell. A sample and hold (S + H) circuit receives the bit-line current and feeds it the ADC unit which outputs the digital value for the current. The result from each bit-line represents the partial sum for bits stored in that bit-line. A word or data element is stored across multiple bit-lines. An external digital shifter and adder (S + A) combines the partial sums from bit-lines. The final result is written back to <dst> memory row or register. Each of ReRAM crossbar (XB), ADC and S+A takes 1 cycle, resulting in 3 cycles in total.

2) dot The `dot` instruction is also an n-ary operation which emulates a dot product over the data in rows specified by <mask>. A dot product is a *sum of products*. The *sum* is done using current summation over the bit-line as explained earlier. Each row computes a *product* by streaming in the multiplicand via the word-line DAC in a serial manner as shown in Figure 3.2 (b). The multiplicands are stored in register file and the individual registers are specified using <reg_mask> field.

Robust current summation over ReRAM bit-lines has been demonstrated in prior works [52, 53]. We adapt the dot product architecture from ISAAC [3] for our `add` and `dot` instructions. We refer the reader to these works for further implementation details.

3) mul The `mul` instruction is 2-ary operation that performs element-wise multiplication over elements stored in the *two* <src> memory rows and stores the result in <dst>. To implement this instruction we utilize the row of DACs at the top of the array feeding the bit-lines (Figure 3.1 (c)). The multiplicand is streamed in through the DACs serially 2-bits at time and the product is accumulated over bit-lines as shown in Figure 3.2 (c). The word-line DACs are set to Vdd ('11').

Note that element-wise multiplication was not supported in prior works on memristor-based accelerators, and is a new feature we designed for supporting general purpose data-parallel computation. Since dot product uses the same multiplicand for all elements stored in a row, it can not be utilized for element-by-element multiplication. We solve this problem by using an additional set of DACs for feeding bit-lines. As in ISAAC, the operation is pipelined into 3 stages: XB, ADC and S+A, processing 2 bits per cycle, resulting in 18 cycles in total for 32 bit data.

4) sub The `sub` instruction performs element-wise subtraction over elements stored in the two set of memory rows (minuends and subtrahends) specified by <mask>s and stores the result in <dst>. Subtraction in ReRAM arrays has not been explored before. We support this operation by draining the current via word-line as shown in Figure 3.2 (d). The output voltage for word-line DAC of the subtrahend row is set to ground allowing for current drain. Hence the remaining current over the bit-line represents the difference between minuend and subtrahend. For this operation we reverse the voltage across memristor bit-cell. Fortunately, several reports on fabricated ReRAM demonstrate the symmetric V/I properties of memristor with reverse voltage across terminals [54, 55].

5) lut The `lut` instruction sends the value stored in <src> as an address to the lookup table (LUT), and write back the data read from the LUT to <dst>. The multi-purpose LUT is implemented for supporting high-level instructions. LUT is utilized for nonlinear functions such as sigmoid, and initial seeding of division and transcendental functions (Section 3.5.1). The LUT has 512 entries of 8-bit numbers to suffice the precision requirement of the arithmetic algorithms implemented [56]. LUT is a small SRAM structure which operates at much higher frequency than ReRAM arrays and hence shared by multiple arrays. Its contents are initialized by the host at runtime. `lut` takes 4 cycles, adding 1 cycle on top of the basic XB, ADC, S+A pipeline.

6) mov, movi, movg, movs The `mov` family of instructions facilitates movement of data between memory rows of an array, registers, and even across arrays via global addressing (<gaddr>). The global addresses are handled by the network, hence the latency of gobal moves (`movg`) is variable. Immediate values can be stored to <dst> as well via `movi` instruction. These instructions are implemented using traditional memristor read/write operations. The selective mov (`movs`) instruction selectively moves data to elements in <dst> based on an 8-bit mask. Recall that any <dst> row can store 8 32-bit elements in the prototype architecture.

7) reduce_sum The `reduce_sum` instruction sums up the values in the <src> row of different arrays. The reduction is executed outside the arrays. This instruction utilizes the H-tree network and the adders in the routers to reduce values across the tiles.

8) shift / mask The `shift` instruction shifts each of the vector element in <src> by <imm> bits. The mask instruction logically ANDs each of the vector element in <src> with <imm>. These instructions utilize the digital shift and adder (S+A) outside the arrays.

Discussion Our goal is two-fold. First, keep the instruction set as simple as possible to reduce design complexity and retain area efficiency (hence memory density). Second, expose all compute primitives which can be done in-situ inside the memory array without reading the data out. The proposed ISA does not include any instructions for looping, branch or jump instructions. We rely on the compiler to unroll loops wherever necessary. Our SIMD programming model ensures small code size, in spite of unrolling. Control flow is facilitated via condition computation and selective moves (Section 3.3). The compute instructions in the ISA are restricted to add, sub, dot, mul. Our programming model based on TensorFlow, supports a rich set of compute operations. Our compiler transforms them to a combination of ISA instructions (Section 3.5.1) and hence enables general purpose computation.

3.2.3 Precision and Signed Arithmetic

Floating point operations need normalization based on exponent, hence in-memory computation for the floating point operands encumbers huge complexity. We adopt a fixed point representation. We give the flexibility for deciding the position of the decimal point to trade-off between precision and range. But the responsibility to prevent bit overflow and underflow is left to the programmers. We developed a testing tool that can calculate the dynamic range of the input that assures the required precision. Note that under the condition that overflow/underflow does not happen, fixed point representation gives better accuracy compared to floating point. Section 3.6 discusses the impact on application output.

For general purpose computation, it is important to support negative values. Prior work [3] uses a biased representation for numbers, and then normalizes the bias via subtraction outside the memory arrays. This approach is perhaps reasonable for CNN dot products, because the overhead of subtraction outside the array for normalizing the bias, is compensated by multi-row addition within the array. In general, data-parallel programs' additions need not span multiple rows (often 2 rows are sufficient). In such a scenario, subtraction outside the array needs additional array read which offsets the benefit of biased addition inside the array.

We observe that for b-bit bit-cells (i.e. 2^b resistance levels), current summation followed by shift+adder across bit-lines outputs the correct results as long as negative numbers are stored in 2^b's complement notation. In our prototype design, arrays have 2-bit bit-cell, hence addition over negative numbers stored as 4'complement will yield correct results. Furthermore it can be mathematically proved that 4's complement is exactly equal to 2's complement in base-4 representation. Thus there is no need for conversion between number formats. The same principle holds true for multiplication as long as the DAC used for streaming in the multiplicand has same resolution as resistance level of ReRAM bit-cells. In our design, 2-bit DACs are required.

Input nodes	Const	Placeholder	Variable						
Arithmetic Operations	Abs Sigmoid Tensordot⋆	Add Sqrt	ArgMin Square	Div Sub	Exp Sum	FloorDiv Conv2D⋆	Less ExpandDims⋆	Mul MatMul⋆	RealDiv Reshape⋆
Control Flow etc.	Assign	AssignAdd	Gather	Identity	Pack	Select	Stack	NoOp	

Table 3.2: Supported TensorFlow Nodes. (⋆ has restrictions on function/data dimension.)

3.3 Programming Model

We choose Google's TensorFlow [51] as the programming front-end for proposed in-memory processor. By using TensorFlow, programmers write the kernels which will be offloaded to the memory. TensorFlow expresses the kernel as a *Data Flow Graph* (DFG). Since TensorFlow is available for variety of programming languages (e.g. Python, C++, Java, Go), programmers can easily plug in the TensorFlow kernels in their code. Also, since TensorFlow supports variety of target hardware systems (e.g. CPU, GPU, distributed system), programmers can easily validate the functionality of the kernel and scale the system depending on the input size.

TensorFlow (TF) offers a suitable programming paradigm for data parallel in-memory computing. First, nodes in TF's DFGs can operate on multi-dimensional matrices. This feature embeds the SIMD programming model and facilitates easy exposure of Data Level Parallelism (DLP) to the compiler. Second, irregular memory accesses are restricted by not allowing subscript notation. This feature benefits both programmers and compilers. Programmers do not have to convert high-level data processing operations (e.g., vector addition) into low-level procedural representations (e.g., for-loop with memory access). The compiler can fully understand the memory access pattern. Third, the DFG naturally exposes Instruction Level Parallelism (ILP). This can be directly used by a compiler for Very Long Instruction Word (VLIW) style scheduling to further utilize underlying parallelism in the hardware without implementing complex out-of-order execution support. Finally, TensorFlow supports a persistent memory context in nodes of the DFG. This is useful in our merged memory and compute architecture for storing persistent data across kernel invocations.

Our programming model and compilation framework support the following TensorFlow primitives (See Table 3.2 for the list of supported TF nodes.):

Input nodes The proposed system supports three kinds of input: Placeholder, Const, and Variable. Placeholder is a non-persistent input and will not be used for future module invocations. Const is used to pass constants whose values are known at compile time. Scalar constants are included in ISA, and vector constants are stored in either the register file or an array based on the type of their consumer node in the DFG. Variable is the input with persistent memory context, of which data can be used and updated in the future kernel invocations. Variables are initialized at kernel launch time.

Operations The framework supports a variety of complex operation nodes including transcendental functions. We discuss the process of lowering these operation nodes into native memory ISA in Section 3.5.1.

Control Flow Control flow is supported by a select instruction. A select instruction takes three operands and generates output as follows:

$$O[i] = Cond[i] \ ?\ A[i] : B[i]. \tag{3.1}$$

A select instruction is converted into multiple selective move (`movs`) instructions. The Condition variable is precomputed and used to generate the mask for the selective moves.

Reduction, Scatter, Gather A reduction node is supported by the compiler and natively in the micro-architecture. Scatter and gather operations are used to implement an indirect reference to the memory address given in the operand. These operations generate irregular memory accesses and require synchronizations to guarantee consistency. Because of the non-negligible overhead, these operations should be used rarely. We observe in many cases that these operations can be eliminated before offloading the kernel by sending gathered data from CPU.

3.4 Execution Model

The proposed architecture processes data in a SIMD execution model at the granularity of *module*. At runtime, different instances of a module execute the same instructions on different elements of input vectors in a lock-step manner. Our compiler generates a module by unrolling a single dimension of multi-dimensional input vectors as shown in Figure 3.3. Intuitively, a DFG generated by TensorFlow can represent one module. At kernel launch time, the number of module instances are dynamically created in accordance with the input vector length.

The proposed execution model allows restricted communication between instances of modules.

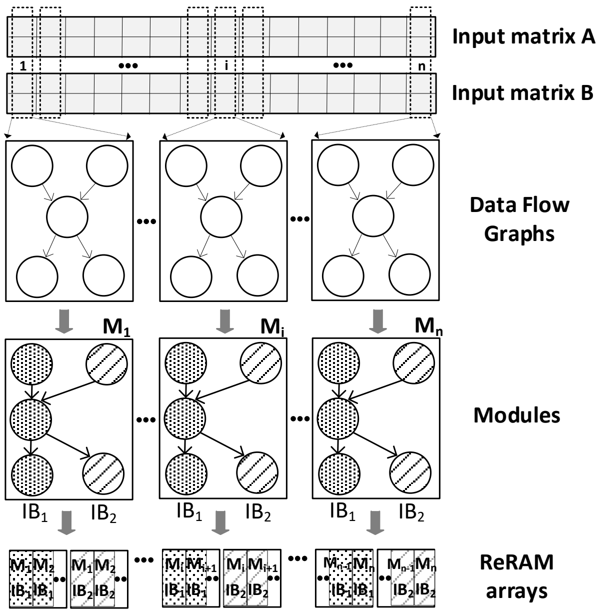


Figure 3.3: Execution Model.

Such communication is only allowed using scatter/gather nodes or reduction nodes in the DFG. We find these communication primitives are sufficient to express most TensorFlow kernels.

Each module is composed of one or more Instruction Blocks (IB) as shown in Figure 3.3. An IB consists of a list of instructions which will be executed sequentially. Conceptually, an IB is responsible for executing a group of nodes in the DFG. Multiple IBs in a module may execute in parallel to expose ILP. The compiler explores several optimizations to increase the number of concurrent IBs in a module and thereby exposes the ILP inside a module.

We view rows in the ReRAM array as a SIMD vector unit with multiple lanes or *SIMD slots*. Each IB is mapped to a single lane or one slot. To ensure full utilization of all SIMD lanes in the array, the runtime maps identical IBs from different instances of the same module to an individual array as shown in the last row of Figure 3.3. This mapping results in correct execution because all instances of a module have the same set of IBs. Furthermore, IBs of a module are greedily assigned to nearby arrays so that the communication latency between IBs is minimized.

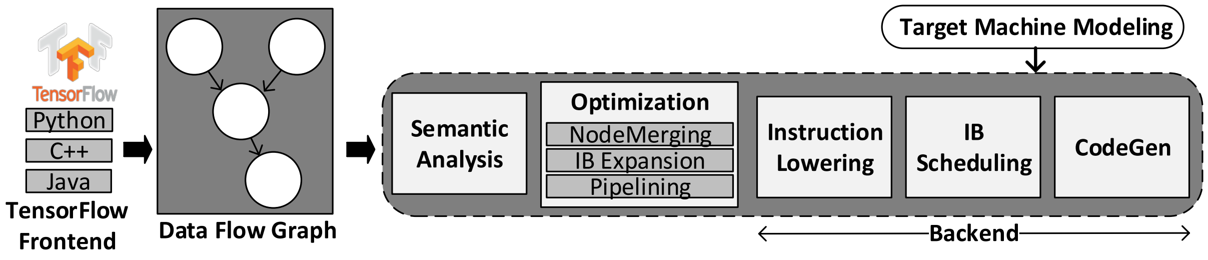


Figure 3.4: Compilation Flow.

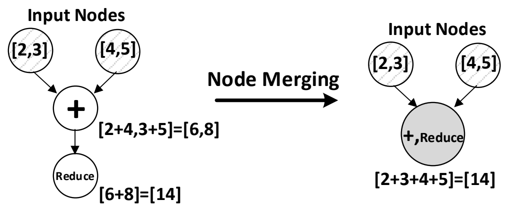


Figure 3.5: Node Merging.

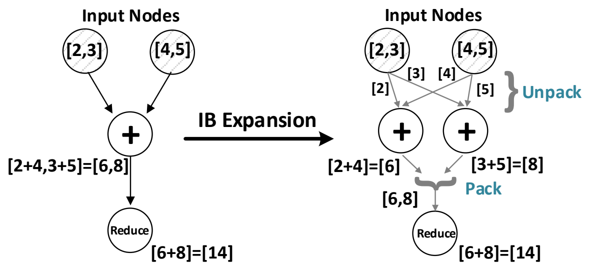


Figure 3.6: IB Expansion.

3.5 Compiler

The overall compilation flow is shown in Figure 3.4. Our compiler takes Google's TensorFlow DFG in the protocol buffer format as an input, optimizes it to leverage parallelism that the in-memory architecture offers, and generates executable code for the in-memory processor ISA. The compiler first analyzes the semantics of input DFG which has vector/matrix operands and creates a module with a single IB with required control flow. Several optimizations detailed later expand a module to expose intra-module parallelism by decomposing and replicating the instructions in the single IB into multiple IBs and merging redundant nodes. This is followed by instruction lowering, scheduling of IBs in a module, and code generation. Instruction lowering transforms complex DFG nodes into simpler instructions supported by in-memory processor ISA. Instruction lowering is also done by promoting the specific instructions (e.g. ABS) to general ones (e.g., MASK) and expanding the instruction into a set of native memory ISA instructions.

The compiler tool-chain is developed using Python 3.6 and C++. The compiler front-end uses

TensorFlow's core framework to parse the TensorFlow Graph. TensorFlow nodes supported at this time are listed in Table 3.2.

3.5.1 Supporting Complex Operations

The target memory ISA is quite simple and supports limited number of compute instructions as described in Section 3.2.2. Natively, the arrays can execute dot product, addition, multiplication and subtraction. However, general purpose computation requires supporting a diverse set of operations ranging from division, exponents, transcendental functions, etc. We support these complex operations by converting them into a set of LUT, addition and multiplication instructions based on algorithms used in Intel IA-64 processors [57, 58].

The compiler uses either Newton-Raphson or Maclaurin-Goldschmidt methods that iteratively apply a set of instructions to an initial seed from the look-up table and refine the result. Our implementation chooses the best algorithms based on the precision requirement. We could have used simpler algorithms (e.g., SRT division), but we employ iterative algorithms because (1) bit shift cannot be supported in the array, so for each bit shift operation the values need to be read out and written back, (2) supporting bit-wise logical operations (and, or) are challenging because of multi-level resistive bit-cells, and (3) simple algorithms often require more space, which is challenging for the data carefully aligned in the array.

Finally, the compiler also lowers convolution nodes in the DFG to the native memory ISA. Prior works [3] have mapped convolution filter weights to the array and performed dot product computation by streaming in the input features. Because filters used for general-purpose programs are typically small (e.g. 3x3 for HotSpot and Sobel filter), we map the input data to the array and stream in the filter. This approach reduces buffering for the input data and improves array utilization. Furthermore, the compiler decomposes the convolution into a series of matrix-vector dot-products done simultaneously on different input matrix slices, thereby reducing the convolution time significantly.

3.5.2 Compiler Optimizations

Node Merging A node merging pass is introduced to fill the gap between the capabilities of the target in-memory architecture and the expressibility of the programming language. The proposed in-memory ISA can support compute operations over *n*-operands. A node merging pass promotes a series of 2-operand compute nodes in the DFG of a module, to a single compute node with many operands as shown in Figure 3.5. The maximum number of operands *n* is limited by the number of array rows and the resolution of ADCs. ADCs consume a significant fraction of chip power, and their power consumption is proportional to their resolution. Our compiler can generate code for an arbitrary resolution *n* and the chip architects can choose a suitable *n* based on the power budget.

The node merging pass also combines certain combinations of nodes to reduce intermediate writes to memory arrays. For example, a node which feeds its results to a multiplication node need not write back the results to memory. This is because multiplicand is directly streamed into the array from registers.

Instruction Block Scheduler Independent Instruction Blo-cks (IBs) inside a module can be co-scheduled to maximize ILP as shown in the third row of Figure 3.3. Our compiler adapts the Bottom-Up-Greedy (BUG) algorithm [59] for scheduling IBs. BUG was first used in the Bulldog VLIW compiler [59] and has been adapted in various schedulers for VLIW/data-flow architecture, e.g. Multiflow compiler [60] and compiler for the tiled data-flow architecture, WaveScalar [61]. Our implementation of the BUG algorithm first traverses the DFG through a bottom-up path, collecting candidate assignments of the instructions. Once the traversal path reaches the input (define) node, it traverses a top-down path to make a final assignment, minimizing the data transfer latency by taking both the operand location and successor location into consideration. We modify the original BUG algorithm to introduce the notion of in-memory computing, where a functional unit is identical to the data location. We also modified the algorithm to take into account read/write latency, network resource collision latency, and operation latency.

Instruction Block (IB) Expansion Instruction Blocks that use multi-dimensional vectors as operands can be expanded into several instruction blocks with lower-dimension vectors to further exploit ILP and DLP. For example, consider a program that processes 2D matrices of dimension sizes [2, 1024]. The compiler will first convert the program to a module which will be instantiated 1,024 times and executed in parallel. Each module will have an IB that processes 2D vectors. The expansion pass will further decompose the module into 2 IBs that process 1D scalar value.

The expansion pass traverses the nodes in a module's DFG in a bottom-up/breath-first order and detects the subtrees that process multi-dimensional vectors of the same size. The subtree regions detected are expanded. To ensure the dimensions are consistent between the sub-tree regions, pack and unpack pseudo operations are inserted between these regions. Pack and unpack operations are later converted to `mov` instructions. A simplified example is shown in Figure 3.6.

Pipelining A significant fraction of the compute instructions goes through two phases: compute and write-back. Unfortunately, these two phases are serialized, since an array cannot compute and write simultaneously. Our compiler breaks this bottleneck by pipelining these phases and ensuring the destination address for the write-backs are in a separate array. By using two arrays, one array computes while writing back the previous result to the other array. In the worst case, this optimization lowers the utilization of arrays by half. Thus, this optimization is beneficial when the number of modules needed for the input data is lower than the aggregate SIMD capacity of the memory chip.

Balancing Inter-Module and Intra-Module Parallelism Some of the optimizations discussed above attempt to improve performance by exposing parallelism inside a module. Because of Amdahl's law, increasing the number of IBs in a module will not result in linear speedup. Depending on the data characteristics, the SIMD slots assigned to a module may not be fully utilized in every cycle. In fact, expanding a module could slow down the total execution time when the number of IBs across all module instances exceeds the aggregate SIMD slots in the memory chip. In such a scenario, multiple iterations may be needed to process all module instances, resulting in a

	Benchmark	Input data shape	# IB insts.
PARSEC	Blackscholes	[4, 10000000]	163
	Canneal	[2, 600, 4096]	6
	Fluidanimate	[3, 17, 229900]	294
	Streamcluster	[2, 128, 1000000]	6
Rodinia	Backprop	[16, 65536]	117
	Hotspot	[1024, 1024]	26
	Kmeans	[34, 494020]	91
	StreamclusterGPU	[2, 256, 65536]	6

Table 3.3: Evaluated workloads. Numbers in bracket indicates size of respective x,y,z dimensions

performance loss.

Our compiler can generate code for arbitrary upper bounds on the number of IBs per module, and can flexibly tune the intra-module parallelism with respect to inter-module parallelism. We develop a simple analytical model to compute the approximate execution time given the number of IBs per module and number of module instances. The number of module instances is dependent on input data size, and is only known at runtime. Thus, the optimal code is chosen at runtime based on the analytical model and streamed in to the memory chip from host.

3.6 Methodology

Benchmarks We use a subset of benchmarks from PARSEC multi-threaded CPU benchmark suite [62] and Rodinia GPU benchmark suite [63] as listed in Table 3.3. We re-write the kernels of the benchmarks in TensorFlow code and then generate in-memory ISA code using our compiler. We choose to port the applications which could be easily transformed to Structure of Array (SoA) code for the ease of porting to TensorFlow and a data-parallel architecture. We leave the remaining benchmarks to future work. For the benchmarks which use floating point numbers in the kernel, we assess the effect of converting it into fixed point numbers. By tuning the decimal point placement, we ensure that the input data is in the dynamic range of fixed point numbers. We ensure that the quality of result requirement defined by the benchmark is met. We use the native dataset for each benchmark and compare it with the native execution on the CPU and GPU baseline systems. The

Component	**Params**	**Spec**	**Power**	**Area**(mm^2)
ADC	resolution frequency number	5 bits 1.2 GSps 64×2	64 mW	0.0753
DAC	resolution number	2 bits 64×256	0.82 mW	0.0026
S+H	number	64×128	0.16 mW	0.00025
ReRAM Array	number	64	19.2 mW	0.0016
S+A	number	64	1.4 mW	0.0015
IR OR Register	size size size	2KB 2KB 3KB	1.09 mW 1.09 mW 1.63 mW	0.0016 0.0016 0.0024
XB	bus width size	16B 10×10	1.51 mW	0.0105
LUT	number	8	6.8 mW	0.0056
Inst. Buf	size	$8 \times$ 2KB	5.83 mW	0.0129
Router	flit size num_port	16 9	0.82 mW	0.00434
S+A	number	1	0.05 mW	0.000004
1 Tile Total			101 mW	0.12
Inter-Tile Routers	number	584	0.81 W	2.50
Chip total			416 W	494 mm^2

Table 3.4: In-Memory Processor Parameters

size of the input for each kernel invocation ranges from 8MB to 2GB.

Area and Power Model All power/area parameters are summarized in Table 3.4. We use CACTI to model energy and area for registers and LUTs. The energy and area model for ReRAM processing unit, including ReRAM crossbar array, sample-and-hold circuits, shift-and-add circuits are adapted from the ISAAC [3]. We employ energy and power model in [64] for the on-chip interconnects and assume an activity factor of 5% for TDP (given that the network operates at 2 GHz and memory at 20 MHz). The benchmarks show an order of magnitude lower utilization of network. ADC/DAC energy and power are scaled for 5-bit and 2-bit precision [65]. While the state-of-the art ReRAM device supports 4 to 6 resistance levels [66], strong non-uniform analog resistance due to process variation makes it challenging to program ReRAM for analog convolution, resulting in

Parameter	**CPU** (2-sockets)	**GPU** (1-card)	**IMP**
SIMD slots	448	3840	2097152
Frequency	3.6 GHz	1.58 GHz	20 MHz
Area	912.24 mm^2	471 mm^2	494 mm^2
TDP	290 W	250 W	416 W
Memory	7MB L2; 70MB L3 64GB DRAM	3MB L2 12GB DRAM	1GB RRAM

Table 3.5: Comparison of CPU, GPU, and IMP Parameters

convolution errors [67]. We conservatively limit the number of cell levels to two and use multiple cells in a row to represent one data.

Performance Model For determining the IMP performance, we develop a cycle accurate simulator which uses an integrated network simulator [68]. Note ReRAM array executes instructions in order, instruction latency is deterministic, network communication is rare, and compiler schedules instruction statically after accounting for network delay. Thus estimated performance for IMP is highly accurate.

3.7 Results

3.7.1 Configurations Studied

In this section we evaluate the proposed In-Memory Processor (**IMP**), and compare it to state-of-art **CPU** and **GPU** baselines. We use an Intel Xeon E5-2697 v3 multi-socket server as CPU baseline and Nvidia Titan XP as the GPU baseline. The IMP configuration (shown in Table 3.4) evaluated has 4,096 tiles and 64 128×128 ReRAM arrays in each tile.

Table 3.5 compares important system parameters of the three configurations analyzed. IMP has significantly higher degree of parallelism. IMP enjoys 546× (4681×) more SIMD slots than GPU (CPU). The massive parallelism comes at lower frequency, IMP is 80× (180×) slower than GPU (CPU) in terms of clock cycle period. IMP is approximately area neutral compared to GPU, and about 2× lower area than the 2-socket CPU system. The TDP of IMP is significantly higher,

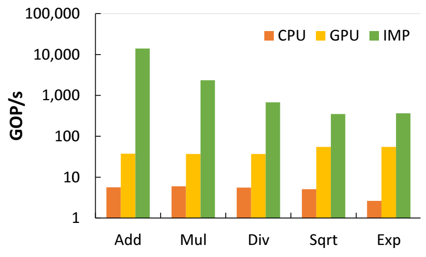


Figure 3.7: Operation throughput (log scale).

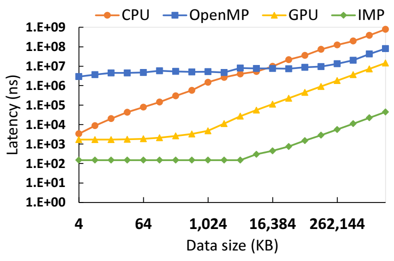


Figure 3.8: Addition Latency.

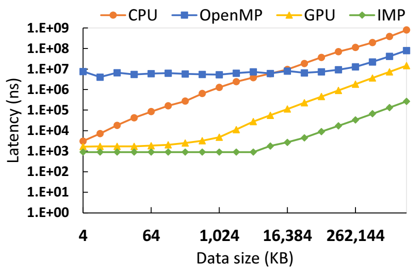


Figure 3.9: Multiplication Latency.

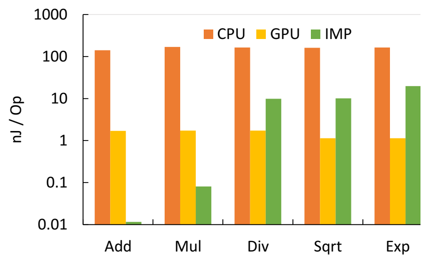


Figure 3.10: Operation energy.

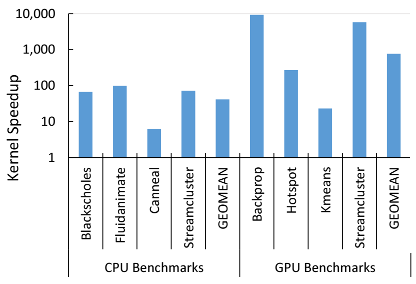


Figure 3.11: Kernel speedup.

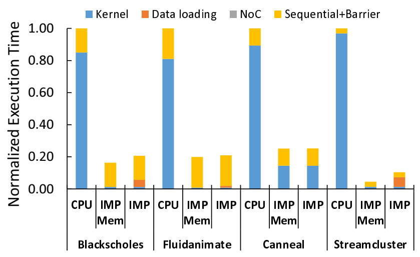


Figure 3.12: CPU Application performance.

however we will show that IMP has lower average power consumption and energy consumption (Section 3.7.3).

3.7.2 Operation Study

Figure 3.7 presents the operation throughput of CPU, GPU, and IMP, measured by profiling microbenchmarks of add, multiply, divide, sqrt and exponential operations. We compile the microbenchmarks with -O3 option and parallelize it using OpenMP for the CPU. We find IMP achieves orders of magnitude improvement over the conventional architectures. The reason is two fold: massive parallelism and reduction in data movement. The proposed architecture IMP has 546$\times$ (4681$\times$) more SIMD slots compared to GPU (CPU) as shown in Table 3.5. Although IMP has lower frequency, it more than compensates this disadvantage by avoiding data movement. CPU and GPU have to pay a significant penalty for reading the data out of off-chip memory and passing it along the on-chip memory hierarchy to compute units.

IMP speedup is especially higher for the simple operations. The largest operation throughput is achieved by addition (2,460× over CPU and 374× over GPU), which has smallest latency in IMP. On the other hand, division and transcendental functions take many cycles to produce the results. For example, it takes 62 cycles for division and 115 cycles for exponential, while addition takes only 3 cycles. Therefore, the throughput gain becomes smaller for complex operations. While CPU and IMP per-operation throughput reduces for higher latency operations, GPU throughput increases. This is because the GPU performance is bounded by the memory access time, and unary operators (exponential and square root) have less amount of data transfer from the GPU memory.

Figure 3.8 and 3.9 show the operation latency of addition and multiplication for different input size. We compare the execution time of single-threaded CPU, multi-threaded CPU (OpenMP), and GPU. IMP offers the highest operations performance among the three architectures, even for the smallest input size (4KB).

Figure 3.10 shows the energy consumption for each operation. Because of the high operation latency and the large energy consumption of ADC, we observe higher energy consumption for the complex operations relative to GPU. Ultimately, the instruction mix of the application will determine the energy efficiency of the IMP architecture.

3.7.3 Application Study

In this section we study the application performance. First, we analyze kernel performance shown in Figure 3.11. For CPU benchmarks, the figure shows performance for hot kernels in PARSEC benchmarks. We assume that non-kernel code of PARSEC benchmarks are executed in the CPU. Note that this data transfer overhead is taken into account in the results of IMP. The GPU benchmarks from Rodinia are relatively small, hence we regard them as application kernels. We observe a 41× speedup for CPU benchmarks and 763× speedups for GPU benchmarks.

GPU benchmarks obtain higher performance improvement in IMP because of the opportunity

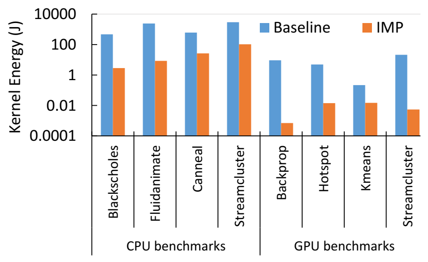


Figure 3.13: Energy consumption.

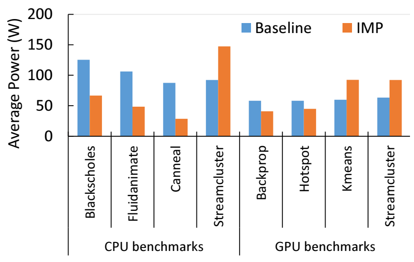


Figure 3.14: Average power.

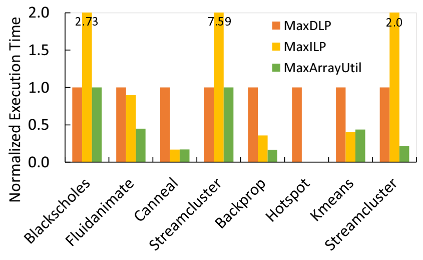


Figure 3.15: Compiler optimizations.

to use dot product operations and higher data level parallelism. On the other hand, the speedup for kmeans is limited to 23×. kmeans deals with Euclidean distance calculation of 34 dimensional vectors, and this incurs many element-wise multiplications. Although kmeans shows significant DLP available in the distance calculations, we could not fully utilize the DLP of the application because of the capacity limitation of the IMP's SIMD slots. This series of multiplications of distance calculation increases its critical latency and limits the speedup. As suggested in the operation throughput evaluation on Figure 3.7, IMP achieves higher performance especially when the kernel has significant DLP and many simple operations. We observe in general mul, add, and movl instructions are most common, while movg, reduce_sum and lut are less frequent. For example, a blackscholes kernel has 14% add, 21% mul, and 58% local move instructions. The rest are mask and lut.

The performance results for the overall PARSEC application are presented in Figure 3.12. For this result, we assume two scenarios: (1) IMP (memory) assumes IMP is integrated into the memory hierarchy and the memory region for the kernel is allocated in IMP. (2) IMP (accelerator) is a configuration when IMP is used as an accelerator and requires data copy as GPUs do. While we believe IMP (memory) is the correct configuration, IMP (accelerator) is a near-term easier configuration which can be a first step towards integrating IMP in host servers.

On average, IMP (accelerator) yields a 5.55× speedup and IMP (memory) provides 7.54× for the Region of Interest (ROI). We observe that 41× kernel speedup does not translate to similar application speedup due to Amdahl's law. Figure 3.12 also shows the breakdown of the execution

Config	Blackscholes	Fluidanimate	Canneal	Streamcluster	Backprop	Hotspot	Kmeans	Streamcluster
MaxDLP	665 / 1	1015 / 1	7220 / 1	2698 / 1	1028 / 1	1081893 / 1	3623 / 1	5386 / 1
MaxILP	377 / 5	437 / 9	1216 / 1212	159 / 129	184 / 32	3125 / 1024	134 / 38	287 / 257
MaxArrayUtil	665 / 1	437 / 4	1228 / 444	2698 / 1	171 / 27	1024 / 3125	1584 / 3	1169 / 6
Lifetime (years)	8.89	20.1	32.2	22.1	15.7	250	5.88	12.8

Table 3.6: (1) IB latency (cycles) and # of IBs for different optimization targets. (2) Lifetime

time, which is divided into kernel, data loading, communication on NoC, and the non-kernel part of the ROI. The non-kernel part is mainly composed of time for barrier and unparalleled parts of the program. It can be seen that 88% of execution time can be off-loaded to IMP. We also observe that large fraction of the execution time on ReRAM is used for data loading ($4\times$ of the kernel at maximum). Thus, as suggested before, in-memory accelerator is better coupled with the existing memory hierarchy to avoid data loading overhead. We also find the NoC time is not the bottleneck, because of the efficient reduction scheme supported by the reduction tree network integrated in the NoC.

Figure 3.13 shows the total energy consumption of the entire application (thus includes both kernel and non-kernel energy for PARSEC). We find $7.5\times$ and $440\times$ energy efficiency for CPU benchmarks and GPU benchmarks, respectively. This energy reduction is partly due to energy efficiency of IMP for kernel's instruction mix and partly due to reduced execution time.

Figure 3.14 shows the average power consumption of the benchmarks. The TDP of IMP is high when compared to GPU and CPU (Table 3.5). ADCs are the largest contributer to peak power. The required resolution for ADCs is a function of maximum number of operands supported for *n*-ary instructions in our ISA. To contain the TDP, we limit the ADC resolution to 5-bits and thereby limiting the number of operands for *n*-ary instructions (`add`, `dot`). While this may affect the performance of a customized dot-product based machine learning accelerator significantly, it is not a serious limitation for general purpose computation. Although IMP's TDP is high due to the ADC power consumption, the average power consumption is dependent on the instruction's requirement for ADC resolution. For example, the ADCs consume less power for instructions with fewer operands. We find that the average resolution for ADC is 2.07 bit (maximum resolution is 5-bit). Overall, the average power consumption for IMP is estimated to be 70.1 W. The average power consumption measured for the benchmarks in the baseline is 81.3 W.

3.7.4 Effect of Compiler Optimizations

We introduce three optimization targets to the compiler and evaluate how each optimization affects the results. The first optimization target is **MaxDLP**, which creates one IB per module to maximize DLP. This policy is useful when the data size is larger than the SIMD slots IMP offers. However, the module does not have an opportunity to exploit ILP in the program. Also, IB expansion is not applied for this policy.

The second optimization target is **MaxILP**, which fully utilizes the ILP and lets IB expansion expand all multi-dimensio-nal data in the module. This will create largest number of IBs per module and shortest execution time for single module. However, because of the sequential part of the IB, array utilization becomes lower. This policy can increase the overall execution time when the kernel is invoked multiple times due to insufficient SIMD slots in IMP.

The third optimization target, **MaxArrayUtil**, maximizes the array utilization considering the number of SIMD slots needed by input data. For example, if the incoming data consumes 30% of the total SIMD slots in IMP, each module can use 3 IBs to fully utilize all the arrays while avoiding multiple kernel invocations. The compiler optimizes under the constraint of maximum IBs available per module

Table 3.6 shows the maximum IB latency and the number of IBs per module. Figure 3.15 presents the execution time of different optimization policies normalized to MaxDLP (baseline). MaxArrayUtil represents the best possible performance provided by the compiler optimizations under resource constraints imposed by IMP. Overall it provides an average speedup of 2.3$\times$.

Two other optimizations not captured by above graph are node merging and pipelining. On average, the module latency is reduced by 13.8% with node merging and 20.8% with pipelining.

3.7.5 Memory Lifetime

We evaluate the memory lifetime by calculating the write intensity of the benchmarks (last row in Table 3.6). Based on the assumption in [69], we consider the ReRAM cells to wear out beyond 10^{11} writes. The compiler balances the writes to the arrays by assigning and using ReRAM rows in a round-robin manner. Assuming the arrays are continuously used for kernel computation (but not while the host is processing), the median of expected lifetime is *17.9 years*.

3.8 Summary

This chapter proposed novel general-purpose ReRAM-based In-Memory Processor architecture (IMP), and its programming framework. IMP substantially improves the performance and energy efficiency for general-purpose data parallel programs. IMP implements simple but powerful ISA that can leverage the underlying computational efficiency. We propose the programming model and the compilation framework, in which users use TensorFlow to develop a program and maximize the parallelism using the compiler's toolchain. Our experimental results show IMP can achieve 7.5$\times$ over PARSEC CPU benchmarks and 763$\times$ speedup over Rodinia GPU benchmarks.

CHAPTER 4

In-Memory Computing with SRAM

This chapter presents *Duality Cache*, an in-cache computation architecture that enables general purpose data parallel applications to run on caches. We propose a holistic approach of building *Duality Cache* system stack with techniques of performing in-cache floating point arithmetic and transcendental functions, enabling a data-parallel execution model, designing a compiler that accepts existing CUDA programs, and providing flexibility in adopting for various workload characteristics.

4.1 In-SRAM Computing

Modern general purpose processors and accelerators are integrated with large on-chip caches to fully exploit locality. They are utilized as a low-latency temporary storage and occupy a large fraction (over 70%) of the die area. For example, the latest Intel's server class Xeon processors devote more than 30MB SRAM just for the last level cache (LLC). Furthermore, data-movement over the cache hierarchy is costly, both in terms of time and energy.

To tackle these inefficiencies, recent works re-purpose the elements in cache structures and transform them into large data-parallel compute units. Compute Caches [8] introduces an in-SRAM computing technique referred to as bit-line computing, which activates multiple word lines and performs *logical operations*. Neural Cache [1] further augments compute capability to efficiently support *fixed point arithmetic* operations. Neural Cache transforms a 35 MB Xeon Cache into 1,146,880 *bit-line ALUs* with a die area overhead of 2%. The proposed *bit-line ALU* operates on transposed or vertically aligned data in a bit-serial manner. These additional compute resources improve the efficiency of Convolutional Neural Networks (CNNs) by 679× (speedup 18.3×, energy savings 37.1×) over a CPU (Xeon E5) and 128× over a GPU (Titan Xp). The source of the efficiency is the combined effect of reduced data movement and massive parallelism.

4.1.1 Challenges and Opportunities

While compute-capable caches offer significant benefits, previous works have just provided low-level interface for in-cache operation [8] or relied on a manual mapping of convolution kernels to the cache arrays [1]. This chapter proposes the *Duality Cache* system stack that makes in-cache computing accessible to general purpose data-parallel programs.

Our proposed system solves several challenges to make caches capable of general purpose data processing. *First*, to address a wide set of data-intensive applications, having a rich set of computation primitives is essential. Prior work is limited to logical and fixed-point arithmetic operations. Most data-parallel workloads require floating point operations. Manipulation of mantissa based on exponents in an in-cache vector architecture is a non-trivial challenge. We devise techniques that support bit-serial floating point operations for applications with high precision or large dynamic range demands. We present techniques that reduce the latency of bit-serial operations based on the dynamic range of operands. The proposed techniques can support 1,146,880 parallel floating-point operations at 3.5% processor die area overhead for a Xeon E5-2697 with 35MB cache. CORDIC algorithms [6, 7] are leveraged to support in-cache transcendental functions.

Second, a critical challenge for in-cache computing is the design of the interface between the CPU cores and compute caches, execution model, and cache addressing structure. Operands of in-cache operations need to be aligned on a bit-line ALU (constraining them to specific locations in cache). We address these problems by developing a single instruction multiple thread (SIMT) architecture, where each thread is mapped to bit-line ALUs. The data bit-cells on a bit-line ALU become thread-local bit-serial registers which are directly addressable in the instruction set architecture (ISA). Compute operations are allowed only on the thread-local registers. *Duality*

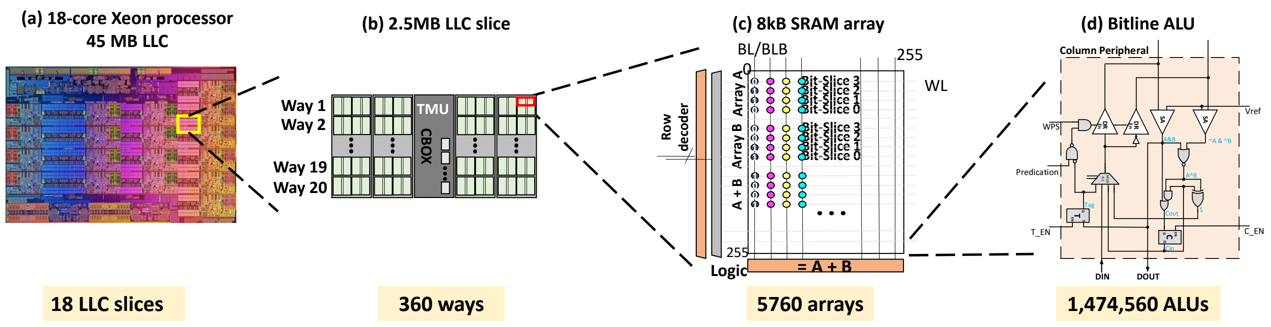


Figure 4.1: Neural Cache architecture [1].

Cache threads are organized into control blocks and mapped to cache ways. We design a micro-architecture that orchestrates control block SIMT instructions. The processor can switch between cache mode and accelerator mode. The SIMT *Duality Cache* architecture is activated only in accelerator mode. *Duality Cache* extensions incur a modest area overhead (3.5%) but do not affect the functionality or performance of conventional cache mode operation.

Finally, compute capable caches require a programming model and compiler that are capable of *exposing parallelism* in applications to the underlying hardware and harnessing its full potential. We adopt CUDA/OpenACC as a programming model and develop a compiler which can translate arbitrary CUDA/OpenACC programs to the *Duality Cache* ISA. The compiler allocates resources, schedules VLIW instructions, and conducts several optimizations exploiting unique opportunities in our in-cache architecture. We also develop compiler assisted techniques to flexibly allocate a fraction of cache to be used as SIMT compute units and regular cache storage.

4.1.2 Benefits

Duality Cache can morph general purpose processors into data-parallel accelerators. In this context, its compute resources are comparable to GPGPU, a representative throughput-oriented parallel accelerator. Although performance bottlenecks of GPGPU are workload dependent, commonly claimed causes include CPU-GPU communication through PCIe bus, load imbalance, on-chip storage size, bandwidth utilization, and compute flops [70, 71, 72]. *Duality Cache* can alleviate these bottlenecks.

Data movement between CPU host and accelerator device. Since disjoint address space of GPU and CPU necessitates explicit data transfer through PCIe, workloads with fine-grained interleaving of serial and parallel phases are difficult to achieve speedups due to communication overheads. Likewise, initial data transfer between the host and device memory is costly especially when data reuse is not high. *Duality Cache* has an advantage of tight integration with the host memory hierarchy and can minimize these overheads.

Cost. While tighter integration of GPU and CPU can alleviate the above problems, the area of modern GPUs (e.g. Titan Xp die area is 471 mm^2 in 16 nm) makes on-die integration with CPU impractical. In contrast, *Duality Cache* extensions require an area of 15.8 mm^2 in 22 nm, while providing nearly 9.3$\times$ more compute resources, making it a cost effective solution. Besides area savings, cost manifests itself in terms of power usage and maintenance. The TDP of a server with Xeon E5 dual socket processor and Titan XP GPU is 640 W, whereas TDP of a server with Xeon E5 processor extended with *Duality Cache* is 296 W (Table 4.2).

Increased on-chip memory capacity. On-chip SRAM can alleviate external memory bandwidth pressure and help reduce memory access latency. GPU's cache size is limited compared to CPU as its die area is dominated by compute units. *Duality Cache* can provide flexible partitioning of compute and cache allocation, which enables memory bounded applications to benefit from a large cache allocation. GPU's memory bandwidth resources can potentially be underutilized by not having enough kernels that request memory accesses. In such case, *Duality Cache* can increase bandwidth utilization by having enough active kernels exploiting its higher compute resources.

4.2 System Stack

In this section, we present a system stack for *Duality Cache* for accelerating data-parallel applications. This section discusses the proposed bit serial arithmetic primitives, execution model, microarchitecture, compiler, and programming model.

4.2.1 ISA

Prior works support a limited set of logical and integer operations. Compute caches [8] introduces basic bit-parallel operations which perform logical operations to horizontally aligned data in caches. Neural cache [1] proposes bit-serial computation that enables several integer operations of vertically aligned data. Our work proposes general ISA for in-cache architecture, leveraging the bit-serial computation scheme. Proposed ISA adopts an early version of PTX (SM2.x), an ISA for low-level parallel thread execution virtual machine employed in NVIDIA's compiler for Fermi GPU family [5]. Our ISA is thus Turing complete. This design choice is made to maximize portability of existing source code while minimizing hardware complexity; any other operations that are not natively supported by the ISA are dealt with by compiler lowering and/or a software library.

Table 4.1 lists major arithmetic operations supported by our ISA, their algorithm, and baseline latency. Machine learning workloads, which Neural Cache targets, can provide reasonable results using reduced precision datatypes (e.g. 8-bit fixed point). However, a class of scientific applications requires more precision in computation, which necessitates full 32-bit integer or floating point support. In this work, we develop in-cache floating point arithmetic. Since some operations listed take latency that scales quadratic with the size of data, native implementation of the algorithms presented in the past work may critically impact performance. Below we discuss our techniques to minimize the bit-serial latency for these operations based on their dynamic range.

4.2.1.1 Floating Point Arithmetic

Prior in-cache architectures do not support floating point (FP) arithmetic. Unlike integer and fixed-point arithmetic, floating point needs normalization of exponents, which requires shift operations of mantissa by an arbitrary value for addition and subtraction.

The proposed algorithm for floating point addition is shown in Algorithm 1. The algorithm leverages bit-serial fixed point addition and subtraction operations discussed in Neural Cache [1]. A floating point addition first requires normalization or shifting of mantissa by the difference of exponents. Note a compute SRAM array is a SIMD unit which does exactly same operation on

Operation	Type	Algorithm	Latency
add	uint, int	[1]	n
sub	uint, int	Bit-serial*	$2n$
mul	uint	[1]	$n^2 + 3n - 2$
mul	int	Bit-serial*	$n^2 + 5n$
div, rem	uint	[1]	$1.5n^2 + 5.5n$
div, rem	int	Bit-serial*	$1.5n^2 + 9.5n$
and, or, xor	uint	[8]	n
shl, shr	uint, int	Bit-serial*	n^2
add, sub	float	Bit-serial*	$\mathcal{O}(n^2)$ - variable
mul	float	Bit-serial*	$\mathcal{O}(n^2)$ - variable
div	float	Bit-serial*	$\mathcal{O}(n^2)$ - variable
sin, cos	fixed point	CORDIC*	$(7k + 1)n + 7k + 1$
exp	fixed point	CORDIC*	$4kn + 4k + 2$
log	fixed point	CORDIC*	$4kn + 4k$
sqrt	fixed point	CORDIC*	$4kn + 4k$
rsqrt	float	Fast inverse square root*	$\mathcal{O}(n^2)$ - variable

Table 4.1: Supported in-cache arithmetic operations. (* = This work. n = #of bits of datatype. k = iteration count.)

256 operands (vectors `A` and `B`) at the same time (Figure 4.2). The proposed design first computes the difference in exponents for all vector elements (vector `ediff`).

The next step needs to shift second operands' mantissa (`B[i].mnt`) by the difference of exponents (`ediff[i]`) for all i such that `ediff[i]` > 0 and then add it to the first operand's mantissa (`A[i].mnt`). We introduce `arshadd` (arithmetic right shift and add) primitive to accomplish this in few cycles. `arshadd` is equivalent to $a + (b \gg d)$. *For given d, shift operation is free for bit-serial architecture.* For example, $a + (b \gg 1)$ can be performed by activating correct bits (a_i and b_{i+1}) and adding them.

Since the *vector* architecture of compute SRAM arrays forces all threads in an array to perform exactly the same operation, arithmetic shift by the values in `ediff` vector may take in the worst case $\mathcal{O}(n^2)$ cycles for n-bit data, since there are 256 values in `ediff` vector and each element of vector is shifted serially. We observe that the dynamic range of exponents is small in real-world workloads and leverage this to reduce the operation latency. The algorithm takes $\mathcal{O}(dn)$ cycles by

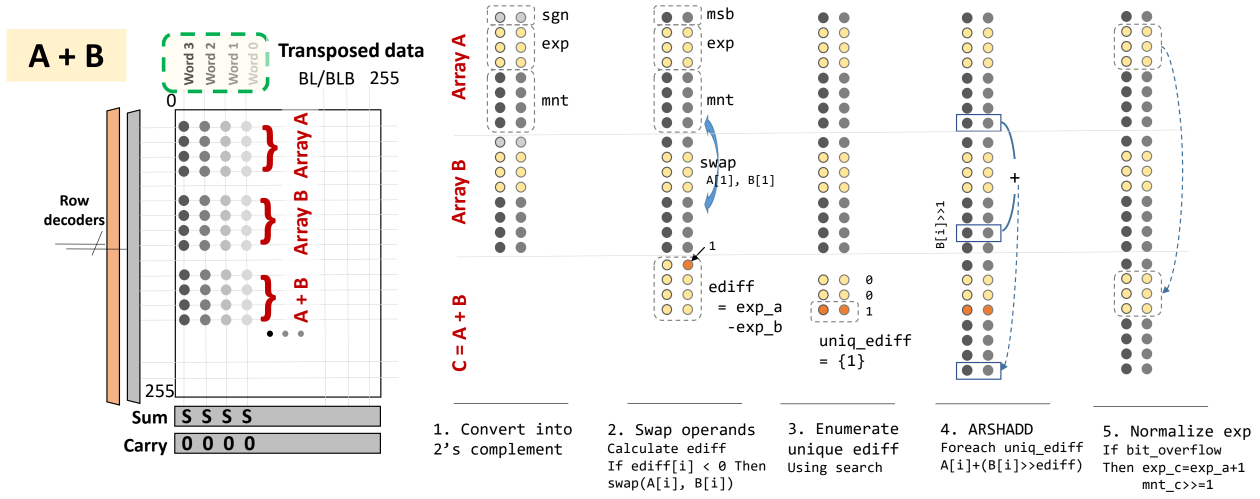

A + B
Word 3
Word 2
Word 1
Word 0
Transposed data
BL/BLB 255
0
Row decoders
Array A
Array B
A + B
255
Sum S S S S
Carry 0 0 0 0
Array A
Array B
C = A + B
sgn
exp
mnt
msb
exp
mnt
swap A[1], B[1]
1
ediff = exp_a -exp_b
0
0
1
uniq_ediff = {1}
B[i]>>1
+
1. Convert into 2's complement
2. Swap operands Calculate ediff If ediff[i] < 0 Then swap(A[i], B[i])
3. Enumerate unique ediff Using search
4. ARSHADD Foreach uniq_ediff A[i]+(B[i]>>ediff)
5. Normalize exp If bit_overflow Then exp_c=exp_a+1 mnt_c>>=1

Figure 4.2: In-SRAM floating point addition overview. The mantissa (mnt) is normalized with the difference in exponents (exp) using a search operation.

searching for all unique d ediff values instead of the worst-case. Note that the worst case variation of d is equal to the number of mantissa bits (23 for IEEE 754 FP32). We do a leading zero search to find the upper-bound value of `ediff` to be searched.

The `search` operation for each unique value is executed in two cycles as follows. In the first cycle, all word-lines that correspond to bit 1s in the search value are activated and logical `AND` of the bit-positions is sensed on each bit-line. In the second cycle, all word-lines that correspond to bit 0s in the search value are activated and `NOR` result is sensed on each bit-line-bar. A logical `AND` of the results from these two cycles produces the final search hit vector.

Additionally, we swap operands with negative `ediff` as shown in Figure 4.2, to avoid divergent execution. Without this swap, we have to repeat the for-block (Line 10 in Algorithm 1) twice, which dominates the processing time of the naive algorithm. Therefore, it is worth doing a swap.

Floating point addition and subtraction require conversion of sign bit format to 2's complement format, also unhiding the implicit leading digit. The mantissa of input values to Algorithm 1 is in 2's complement format, and this conversion is handled by an instruction that precedes. We also introduce an instruction that does re-conversion to sign bit format and mantissa normalization. We minimize the number of conversions by skipping re-conversions between operations. This is helped by a compiler analysis, which scans through the input code and inserts these conversion

Algorithm 1 Floating Point Add (C=A+B)

```
1:  procedure VECTOR_FPADD
2:      ARSHADD(X, Y, k) = X + (Y $\gg$ k)
3:      type float $\{.exp, .mnt\}$
4:      vector <float> $A, B, C$
5:      vector ediff $\leftarrow A.exp - B.exp$
6:      if ediff[i] $< 0$ then
7:          SWAP(A[i], B[i])
8:          ediff[i] $\leftarrow A[i].exp - B[i].exp$
9:      end if
10:     for each unique $k$ in ediff do
11:         if ediff[i] == $k$ then
12:             $C[i].mnt =$
13:             ARSHADD($A[i].mnt$, $B[i].mnt$, $k$)
14:         end if
15:     end for
16:     if overflow$_i$ then
17:         $C[i].exp = A[i].exp + 1$; $C[i].mnt \gg 1$
18:     else
19:         $C[i].exp = A[i].exp$
20:     end if
21: end procedure
```

operators.

We also support efficient floating point multiplication and division. Floating point multiplication (division) is a combination of addition (subtraction) of exponent bits and multiplication (division) of mantissa bits, where we can apply the same technique as integer multiplication (division) which we will discuss in the following section.

We do not support denormal floating point numbers (non-zero numbers with magnitude smaller than the smallest normal number). Note that since denormal number handling significantly reduces process speed in general, some systems omit this hardware support. Intel's SIMD instruction set handles it by calling a software exception, also providing a knob to disable the exception call [73].

4.2.1.2 Integer Arithmetic Optimization's

We apply several optimizations to the baseline integer algorithm to skip redundant cycles depending on the data. For example, when performing multiplication, we can avoid calculating partial sums if i-th bit are all zero across all the data entry. Below lists other optimizations we introduce for multiplication and division.

- We perform a leading zero search on multiplicands and dividends to identify the effective data size. Leading k zeros will reduce more than $n \times k$ cycles.
- We perform a leading zero search on divisors. Since we know the number of digits of quotient Q of A/B is at most $x = \lfloor \log A - \log B \rfloor + 1$, we can skip the first $n - x$ iterations, which saves more than $\sum_{i=0}^{x-1}(n+i)$ cycles. Leading zeros of divisors can also contribute to reducing cycles by interpreting it as data with a smaller datatype. Note that the leading zero searches can be done in parallel for both operands using `search`.
- We perform `search` on the partial residues to judge whether they are zero. For example, 1001/10 will see zero partial residue after calculating Q=01xx. We set 0 to the third MSB of Q without performing subtraction in the iteration.

4.2.1.3 Transcendental Functions

In addition to floating point operations, we support transcendental functions. Previous work on in-memory memristive computing [74] utilizes look-up tables (LUTs) for those functions to get initial guess and refines it by an iterative process such as the Newton-Raphson method. However, this approach not only requires a large area for LUTs for each cache bank but also makes LUTs a serialization point which ends up in limiting computation throughput.

For our in-cache architecture, we utilize a different algorithm called COordinate Rotation DIgital Computer or CORDIC [6, 7, 75, 76]. CORDIC does not require accessing LUTs for each operand value but calculates and refines the result digit-by-digit using pre-calculated constant numbers that can be shared by any operand value. CORDIC does not make any serialization point, which makes it highly efficient for the *Duality Cache* 's massively parallel vector architecture. Furthermore, with using pre-calculated constants, CORDIC only involves addition, subtraction, and fixed-amount-bitshift, but not multiplication, thus being suitable for bit-serial computing, as their latency is $\mathcal{O}(n)$ (mul is $\mathcal{O}(n^2)$). Further, our compiler exposes the ILP in CORDIC algorithms with its VLIW instruction scheduling.

CORDIC approach can be applied to various operations including exp, log, trigonometric / hyperbolic functions, and square root. We set the iteration count to 17 to retain the accuracy of FP32 format. While our CORDIC implementation accept fixed point numbers with fixed region (e.g. $[0°, 90°]$), it is trivial to normalize data to fit within the region (e.g. $\sin(120°) = \sin(120° - 90°)$, $\log 1234 = 3 \times \log 1.234$). This normalization and type conversion to the fixed point are handled by a software library.

4.2.2 Programming Model

Programming model creates a direct and significant impact on programmability and architecture design. While simple models (e.g. wide SIMD [1]) simplify the hardware, it limits flexibility. On the other hand, guaranteeing too much freedom may result in over provisioning of hardware resources in order to handle all communication patterns. To expose the massive parallelism of *Duality Cache* to applications with irregular (or data-dependent) memory access, we adopt a SIMT programming model of CUDA (NVIDIA's GPGPU programming framework) and OpenACC.

CUDA describes kernels as multi-threaded programs and groups threads into warps. In a warp, threads are executed in a synchronized manner. Inter-thread synchronization and sharing are allowed within a group of threads called thread block or Cooperative Thread Array (CTA). In other words, different CTAs are independent and can be scheduled and executed in any order.

Proposed architecture benefits from this programming model from two aspects. First, CUDA is a popular and widely used framework across different fields spanning from scientific computing to machine learning. Leveraging it for *Duality Cache* architecture with direct translation or trivial source code changes will archive portability and opportunity to use the existing software. Second, having independent CTAs entails minimum network resources for inter-thread communications that happen locally within a CTA.

On top of CUDA, we support OpenACC. OpenACC provides OpenMP-like pragma to programmers, making it easier to convert existing serial programs to parallel programs. Currently,

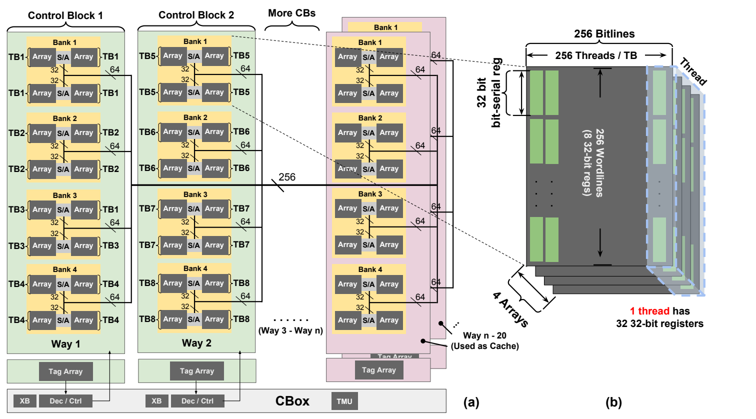


Figure 4.3: In-Cache SIMT execution model and architecture overview.

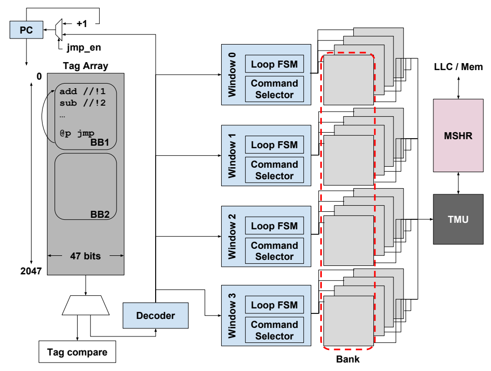


Figure 4.4: Frontend architecture.

commodity OpenACC compilers support multi-thread, multi-core CPUs and NVIDIA GPUs. OpenACC is characterized by its ability to describe fine-grained interleaving of serial computation on the host and parallel kernels to be executed on an accelerator (e.g. GPU) using pragma. While GPUs tends to face communication bottleneck for those OpenACC programs with frequent host-device communication, *Duality Cache* enables seamless execution between host code and kernel code, as caches share the same memory hierarchy as the host.

4.2.3 Execution Model and Architecture

Our execution model reflects the programming model, but when compared to GPUs it is simpler and coarser-grained. The cache acts in two modes: accelerator mode and cache mode. In *accelerator mode* a bit-line in a cache array becomes one thread lane of a SIMT processor. In our architecture, registers and compute units are identical. We assign registers in a thread to the bit-line and perform computation in-place. Operands are vertically aligned within the registers mapped on the bit-line. In *cache mode*, the cache arrays are part of the processor's traditional multi-level memory hierarchy. Note that the accelerator mode does not change the functionality or performance of the cache mode.

SIMT Architecture. Instruction issue is performed at the **Control Block (CB)** granularity. Figure 4.3 shows the *Duality Cache* architecture. A CB consists of a group of 1,024 threads and is allocated to a single way of a cache slice. Each way consists of 4 banks, each bank is capable of executing 256 threads referred to as **Thread Block (TB)**. Hereafter, we use TB to refer to this 256 thread group, and CTA to *software* thread block of CUDA or a *gang* of OpenACC.

We choose to dedicate an entire bank with four SRAM arrays to a TB to provide a sufficient number of registers per thread and prevent frequent register spilling. One SRAM array has 256 bit-cells along a bit-line, thus can afford only 8 32-bit bit-serial registers as shown in Figure 4.3 (b). By allocating 256 threads to a bank of four SRAM arrays we can afford 32 32-bit bit-serial registers per thread. Thus each thread in a TB is virtually mapped to multiple arrays in a bank, and each member array has a slice of registers. The proposed architecture restricts the maximum number of threads in a CTA to 1,024.

We only allow inter-thread communication within CB. This design choice is made to balance programmability and hardware complexity. We utilize a 256$\times$256 local crossbar in the C-Box to shuffle / broadcast CTA local data as shown in Figure 4.3 (a). Although the throughput of the crossbar is limited by the interconnect bandwidth, it can service arbitrary inter-thread communication within a CB in a fixed time-frame. Kernels that do not require inter-thread communication can span across multiple CBs.

A CB and GPU's Streaming Multiprocessor (SM) are similar in the thread and register capacity. While latest GPUs have large register files (64K 32-bit reg / SM) so that cores can time multiplex different warps in blocks assigned to the SM, we directly execute instructions in-situ in numerous *Duality Cache* threads mapped within the registers.

In-Order VLIW Architecture. Mapping a TB to a bank of four arrays opens up an interesting opportunity: each array can execute a different operation in the same cycle. Thus we can perform *VLIW-like instruction scheduling*, allowing *Duality Cache* to exploit ILP in the program. All the banks in a way (i.e. CB) process the same (VLIW) instructions. Instructions are buffered in the tag array in each way which is continuously fed entries by a host processor core. The buffered instructions are then decoded and issued through four issue windows where each instruction is broadcasted to corresponding member arrays of all threads in the CB. *Duality Cache* performs computation in a bit-serial manner. Each bit-line acts as a computation unit, and all bit-lines in an array perform the same operation as in a SIMD processor.

Instruction Sequencing. All threads in a Control Block (CB) perform blocking execution, including memory accesses, with implicit synchronization between threads. On the other hand, different CBs can execute different instructions at a time. This implements compute and memory access overlap at a coarse level: while GPUs schedule for warps to overlap compute and memory access within the block, we fire a lot more CBs at one time using rich compute/register resources ($9.3\times$ more than Titan Xp) and overlap memory accesses with other CB's computations.

This means each CB maintains its own programming counter (PC). PC is incremented before fetching next instruction and points to an entry in the tag array, which works as a ring buffer. The frontier PCs are monitored by the host processor to prevent overwriting instructions that have not been executed. While fixed length loops are unrolled by host run-time or by the compiler, data-dependent loops in applications which iterate under a condition (e.g. convergence) are handled inside CB. For these loops, we maintain entire loop body block in the tag array, and a conditional jump (predicated jump) instruction resets the PC to the loop entry. The loop exits upon negative jump_en, which receives wired-OR of predicate bits stored in the cache peripheral. A CPU core can continue to fill successor instructions of the loop, but it cannot overwrite the loop block until after PC exits the loop.One CPU core is sufficient to launch and feed all CBs. Control flow is handled by predication, and indirect jumps (branches) are not supported, following the PTX language.

The control box in a cache slice is implemented with finite state machines (FSMs) in hardware that can dispatch low-level control signals to the cache banks for performing cycle-by-cycle bit-serial operations based on issued instructions.

Load and Store Instructions. *Duality Cache* interfaces with memory hierarchy through Transpose Memory Unit (TMU) [1]. TMUs have 8T transpose bit-cells which can read and write data in both horizontal and vertical directions to enable the conversion between regular bit-parallel layout and transposed bit-serial format. TMUs are placed in cache control box (CBox in Figure 4.3). When performing load instruction, target addresses are first read out from an array that belongs to one of the issue windows. Unlike other compute operations, only one memory instruction can be included in one VLIW instruction because of the interconnect bandwidth. The bit-serial addresses are transposed in TMU, registered in Miss Status Handling Register (MSHR), and sent to the memory. MSHR enables simple memory coalescing; duplicated accesses to a cache line are treated as an MSHR hit and suppressed. MSHR keeps track of source thread numbers. When the target cache line arrives from the memory, it is first sent to TMU. The destinations are set by configuring the local crossbars so that it can rearrange or multicast data into the data banks. The data is then read out from TMU in bit-serial format and sent to awaiting threads through the crossbar.

Data that can be accessed by *Duality Cache* has to be stored in specially allocated pages (DC-pages) in the main memory address space. A simple MMU placed at the memory controller performs address translation. The address translation is mainly aimed to balance the DRAM load by shuffling the physical address allocation.

4.2.4 Compiler

We develop a backend compiler that transforms PTX, NVIDIA's low-level parallel thread execution virtual machine ISA, into VLIW-style code for *Duality Cache* which we refer to as DC-PTX. Opcodes of DC-PTX are a subset of PTX opcodes; some instructions designed specifically to GPUs are eliminated. On the other hand, DC-PTX adds several fields to PTX to include operand locations.

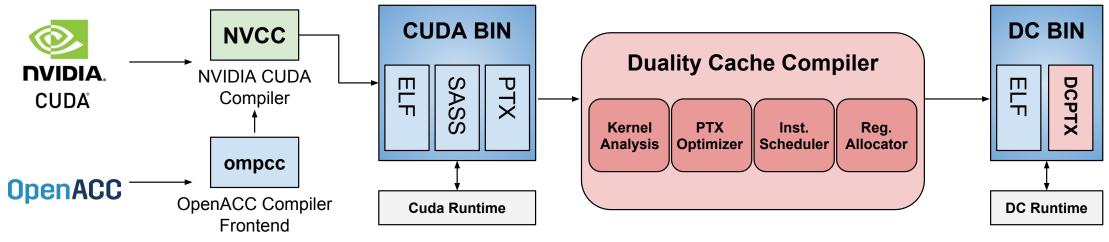


Figure 4.5: Compilation tool flow. CUDA source code is first compiled by NVIDIA CUDA compiler (nvcc). *Duality Cache* compiler extracts PTX assembly from CUDA executable and generates DC-PTX code. OpenACC program is compiled by an OpenACC compiler which generates GPU code that is then compiled by nvcc.

Figure 4.5 shows the overall compilation flow. CUDA source code is first compiled by NVIDIA's CUDA compiler (nvcc). The output CUDA executable includes three kinds of object files (i.e. elf, PTX, and SASS). Our backend compiler extracts and parses the PTX files, applies several optimization passes to PTX IR, schedules instructions, allocates resources, and generates DC-PTX code. DC-PTX kernel is loaded and executed by API calls to DC-Runtime library in a similar way as CUDA runtime.

The compiler is currently built on top of GPU Ocelot dynamic compilation framework [77]. We choose PTX as IR since most of the CUDA compilation tool flow is closed-source (including ptxas which performs resource allocation and scheduling). Currently, GPU Ocelot is the only compilation framework academically available to work on GPU object files. We also utilize Rose compilation framework [78] to perform source-to-source compilation to apply optimization passes to the source code before nvcc compilation.

OpenACC programs can also be compiled using the same infrastructure, except that the source code is first compiled by an OpenACC compiler which extracts the accelerator code and generates GPU code that is then internally compiled by nvcc.

Duality Cache compiler framework translates a CUDA code to VLIW SIMD ISA. Although VLIW is not as efficient as out-of-order execution for exploiting ILP, it enables ILP to be exploited with lower hardware complexity since complicated ILP aware scheduling is handled by the compiler. Unlike traditional VLIW architecture, the proposed *Duality Cache* architecture has to take operand locality into account; all operands need to reside in the member array where the operation is executed, otherwise, we have to explicitly copy the operands to the member array.
Following are the implemented features of our compiler:

Register Pressure Aware Instruction Scheduling

Register pressure and efficient VLIW instruction scheduling are an inseparable problem. In our design, instruction scheduling is tightly coupled with resource allocation. While many compilers for VLIW architecture schedule instructions first before register allocation to maximize parallelism utilizing abundant register resources shared by many execution units, our execution model has limited number of private registers, which may result in frequent register spilling. On the other hand, resource-allocation-first approaches often introduce many false dependencies in return for minimized register usage, which can reduce available parallelism. We tackle this problem by performing resource allocation and instruction scheduling at the same time. We use Bottom-Up Greedy (BUG) [59] as the baseline scheduling algorithm, and linear scan register allocation as the baseline resource allocation algorithm. By taking register pressure into account while performing instruction scheduling, the compiler can pick better strategy to balance parallelism and register spilling. In our design, we allocate computation units considering register pressure as well as operand movement overhead. This approach balances the register pressure of each member array and maximizes parallelism as long as there are available registers. When register pressure is too high for all the member arrays, we start spilling a register according to the spill policy of the linear scan algorithm. Parallelism can be sacrificed due to high data movement cost.

PTX Optimizations

AST balancing: To maximize ILP, it is better to distribute operands of a chain of associative binary operations evenly to available VLIW slots. Generally compiler frontend left-folds an expression of binary operation chain if it does not have parentheses when constructing Abstract Syntax Tree (AST) (e.g. a + b + c + d ⇒ (+ (+ (+ a b) c) d)), making a true dependency between the temporary value (partial sum) and the next operand. One of the optimizations we apply reconstructs the AST to form a balanced tree (e.g. a + b + c + d ⇒ (+ (+ a b) (+ c d))) so that unnecessary dependencies will not hinder exploring ILP when scheduling instructions for our in-order VLIW architecture.

Thread independent variable isolation: We further include an optimization to reduce register pressure by not storing thread independent variables. For example, a fixed length loop is unrolled by *Duality Cache* runtime and the induction variable is provided as a constant if necessary. DC compiler identifies thread independent variables by conducting dependency analysis and affixes metadata as a marker for the instruction that only processes thread independent variables.

4.2.5 Cache Partitioning

Duality Cache architecture can utilize memory arrays in LLC for both computing and caching. Generally, CUDA programs are optimized for GPUs, which typically have 88-144KB SRAM storage in SM for L1+texture cache and shared memory (Pascal GPUs). Therefore, reserving one way (128KB) per CB provides a similar configuration as GPUs. However, cache utilization is highly dependent on applications, and our architecture is able to flexibly adjust the cache resource allocation based on reuse patterns. Prior works [79, 80] have shown that some classes of GPU applications are known to receive small benefits from caches because of less locality. Also, compute intensive kernels can underutilize memory bandwidth. In those cases, we can increase the compute allocation in the cache. On the other hand, we observe many applications with irregular memory access patterns benefit from caches if the working set fits in the caches. Here we can increase the cache allocation to reduce memory bandwidth pressure.

Our compiler can analyze kernel dimension and shared memory usage to determine the cache allocation so that we can leverage the locality of the applications which is explicitly specified by programmer in the form of shared, constant, or texture memory (Note that these local memories are remapped to global memory by the compiler.) We also analyze memory access patterns, and estimate memory traffic and data reuse through static kernel code instrumentation.

4.3 Methodology

Benchmarks: We use applications from Rodinia GPU benchmark suite [71] and PathScale OpenACC benchmark [81] as listed in Table 4.3. We compile the CUDA applications using nvcc 4.2 using default compile options of the benchmark suite (except for the target architecture which we set to sm_20 to make CUDA binary compatible with GPU Ocelot [77]). The OpenACC applications are compiled using Omni Compiler [82], an open-source academic OpenACC compiler, which is internally linked with CUDA Toolkit. We modify the source code of Omni Compiler to disable the automatic insertion of cache configuration API calls which are not supported by CUDA

Model	Die					Benchmark Servers			
	mm^2	nm	GHz	TDP	LLC	Dies	DRAM Size	mm^2	TDP
Xeon E5-2697 v3	456	22	2.6	145 W	35 MB	2	64 GB DDR4	912	290 W
NVIDIA Titan Xp	471	16	1.6	250 W	3 MB	1 + 2 (host)	12 GB GDDR5 + 64 GB DDR4	1,383	640 W
Duality Cache	471	22	2.6	148 W	35 MB	2	64 GB DDR4	942	296 W

Table 4.2: Benchmark server configuration.

	Applications	Dataset	Custom Optimization
Rodinia	backprop, bfs, b+tree, dwt2d, hotspot, hotspot3D, hybridsort, nw, streamcluster	default	None
	gaussian	omp_default	Increased CTA size
	heartwall, leukocyte	default	Loop unrolling
	lud, nn	large (1k, 2k)	CPU hybrid (lud)
acc	divergence, gradient, lapgsrb, laplacian, tricubic, tricubic2, uxx1, vecadd, wave13pt	256 128 1024	Increased CTA size
	gameoflife, gaussblur, matvec, whispering	256 1024	Increased CTA size

Table 4.3: Evaluated workloads. (acc = OpenACC Benchmark)

Toolkit 4.2. While we use the old CUDA version to work with GPU Ocelot dynamic compilation tool [77], we use latest CUDA Toolkit 9.2 [83] and community standard PGI OpenACC compiler 18.10 [84] for the native run on GPU.

We mostly use the dataset preset by the benchmark suite. For some benchmarks, such as gaussian, lud, nn from Rodinia, we use the OpenMP dataset or a data-generator generated large dataset to augment the utilization of computation unit. Moreover, we modify the source code of some benchmarks to further expose the parallelism of *Duality Cache* . These custom optimizations are discussed in detail in Section 4.4.4.

Area and Power Model: Area and power parameters are summarized in Table 4.4. The energy and power model of *Duality Cache* peripherals and Transpose Memory Unit is from Neural Cache [1]. We synthesize the controller and state machine using Synopsis Design Compiler in a 45nm process. We assume average ILP 1.25 and 10% activity factor for TDP. We employ the energy and power model in [85] for the local crossbar and assume an activity factor of 5% for TDP. We use CACTI [86] to model energy and area for scratch SRAM used in MSHR.

Power and energy for CPU and DRAM activity are measured by profiling microbenchmarks

	Area (mm^2)	**Power** (W)	**Area Overhead**
CPU	456	145	-
Duality Cache Peripheral	3.15	2.96	0.69 %
TMU	5.32	0.06	1.17 %
Controller / FSM	6.16	0.33	1.35 %
MSHR	0.86	0.05	0.19 %
Local Crossbar	0.28	0.01	0.06 %
Total	471.77	148.40	3.5 %

Table 4.4: *Duality Cache* parameters.

using Intel Rapl interface. We use NVIDIA nvprof to measure GPU power.

Performance Model: We develop a *Duality Cache* timing model and functional model on GPU Ocelot's tracer framework and PTX emulator. Since some of the in-cache operations are data dependent, the timing model interacts with the functional model in the emulator. Target applications are executed on the DC-PTX emulator in GPU Ocelot and we obtain front-end and back-end traces using our tracer for each CTA. We then rerun the traces using our simulator and feed the trace files to Ramulator [87]. We perform CPU-trace driven DRAM simulation on Ramulator with a modified processor model.

4.4 Results

4.4.1 Configurations Studied

In this section, we evaluate the proposed *Duality Cache* and compare it to two baselines. The first baseline (CPU) uses Intel Xeon E5-2697 v3 multi-socket server. The second baseline (GPU) is a server with host Xeon E5 and NVIDIA Titan Xp GPU. The details of both configurations are shown in Table 4.2. We assume *Duality Cache* to be implemented in the 2-socket Xeon server system. When entire LLC geometry is allocated for computation, *Duality Cache* has 150$\times$ more threads than GPU. The massive parallelism comes at the cost of larger operation latency of the bit-serial algorithms (Section 4.4.5).

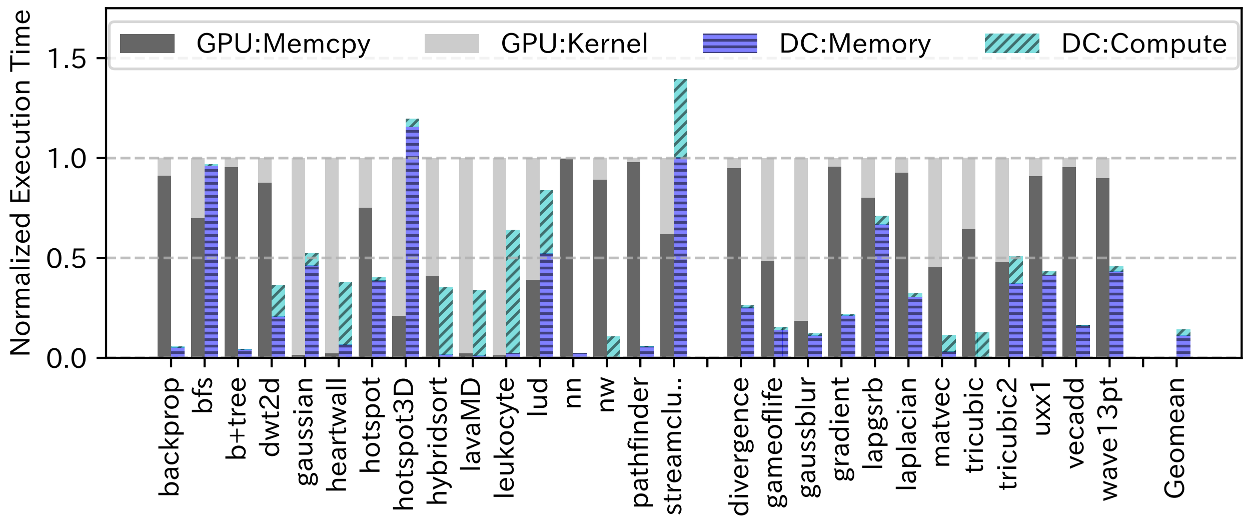


Figure 4.6: System performance. GPU:GDDR5+memcpy, DC:DDR4.

4.4.2 Performance

In this section, we study the application performance. The execution time for the GPU server and *Duality Cache* server is shown in Figure 4.6 (normalized to GPU, lower is better). It shows the breakdown of memcpy time and kernel execution time for GPU. We consider the memory transfer (cudaMemCopy) time, but time spent on GPU initialization, CUDA API calls (including CUDA malloc), and OpenACC API calls is not included. GPU's kernel time includes memory access time. For *Duality Cache* , we show compute and memory access time. The compute time of *Duality Cache* is the aggregate latency of issued instructions on the critical path, and the memory access time is total time − compute time.

Duality Cache provides a 3.6× average speedup for the Rodinia benchmarks and 4.0× speedup for the OpenACC benchmarks, compared to GPU. Figure 4.10 shows the average system speedup of *Duality Cache* compared with CPU. *Duality Cache* provides a 72.6× speedup compared to CPU for the Rodinia benchmarks, and 9.6× for the OpenACC benchmarks. The OpenACC benchmarks can have fine-grain serial and parallel interleaving making their GPU acceleration less effective compared to Rodinia benchmarks.

We discuss the key factors that contribute to the *Duality Cache* performance below:

A. Reduced Memcpy Time: Memcpy time takes a substantial portion of the GPU execution time

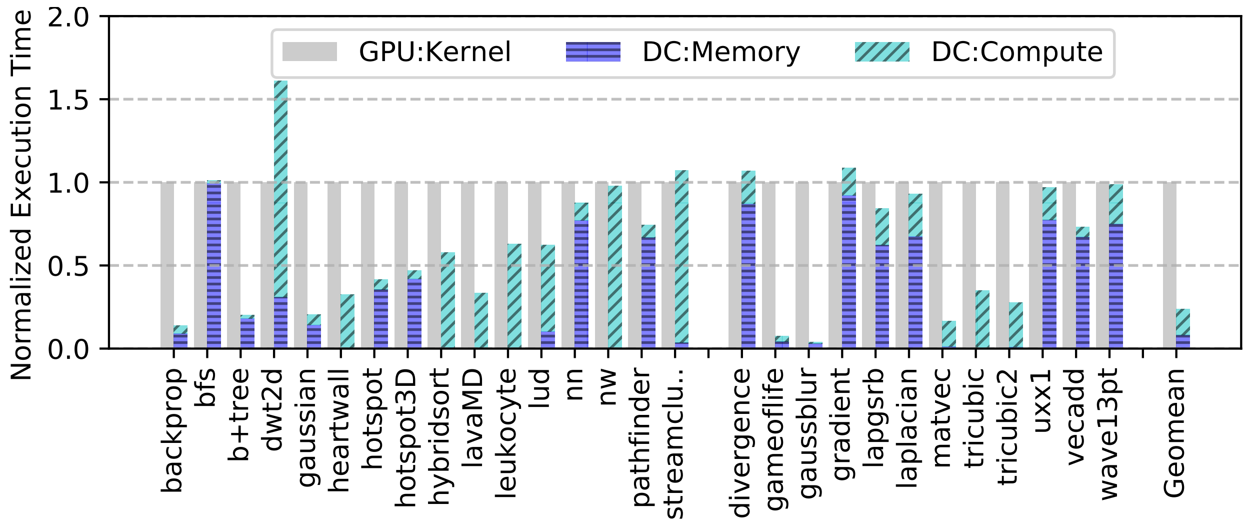


Figure 4.7: Kernel performance. GPU:GDDR5, DC:GDDR5.

for some applications. This can be explained by the fact that some applications have very small reuse factor of data, which can make inter-DRAM data movement cost prominent as shown in Figure 4.6. *Duality Cache* is integrated into the same memory hierarchy as the host, and thus this data movement cost does not exist, resulting in higher performance.

Some CPU models [88] using integrated on-chip GPU could possibly reduce the data movement cost. However, they have typically 10× smaller compute resources than our baseline server GPU (Titan Xp), while taking more than the half of CPU die area. *Duality Cache* is clearly distinguishable from them by the ability to provide the orders of magnitude higher compute resources with only 3.5% of area cost.

B. Massively Parallel Execution: Compute-intensive kernels enjoy *Duality Cache* 's massive parallelism. Figure 4.9 shows the average number of active Control Blocks (CBs) in the kernels. A CB has 1024 threads and can map several CTAs. Since CBs are independent of each other, this chart indicates the available parallelism of the applications. Each Xeon socket can execute 280 CBs (yellow dash line), thus we have 560 CBs (dark orange line) in total in the baseline dual-socket system. Our GPU has 30 SMs, each can have up to 2 CTAs (Note that this is a register size based calculation; threads in CTAs use GPU cores in a time-multiplexed way).

We can see kernels with a high level of parallelism (e.g. backprop, b+tree, nn, gaussian, gaus-

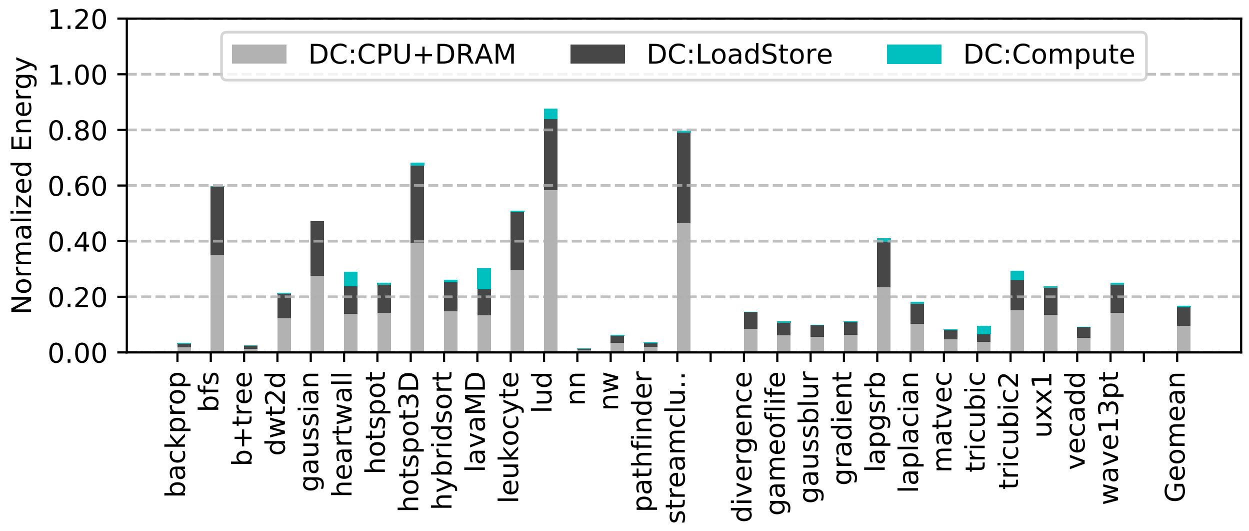


Figure 4.8: Energy efficiency (system).

blur, etc.) significantly reduces execution time in Figure 4.6 as they can harness *Duality Cache* resources. On the other hand, other benchmarks such as `lud`, `nw` and `streamcluster` have limited parallelism available, resulting in large critical compute time in the kernel performance. In Section 4.4.4 we discuss several optimizations we applied to enhance the parallelism beyond the original CUDA programs.

C. Compute / Memory Access Overlap: Some applications show a large compute time portion in the kernel performance, despite enough parallelism (e.g. `lavaMD`). These kernels can successfully hide memory latency with computation. Note few benchmarks show a slowdown (`hotspot3D` and `streamcluster`) with *Duality Cache* because they are memory bandwidth bound. For those, newer memory technology (GDDR5) could help improving performance, as explained shortly (Figure 4.7),

D. Flexible Cache Allocation: While GPU may underutilize / overutilize its memory bandwidth, *Duality Cache* can adjust parallelism and cache allocation size to balance memory bandwidth (Section 4.2.5). Many of the evaluated applications benefit from the cache partitioning. By default, we assign the unused Control Block units as cache, but we changed the allocation size based on the applications' behavior. We will discuss it in Section 4.4.5.

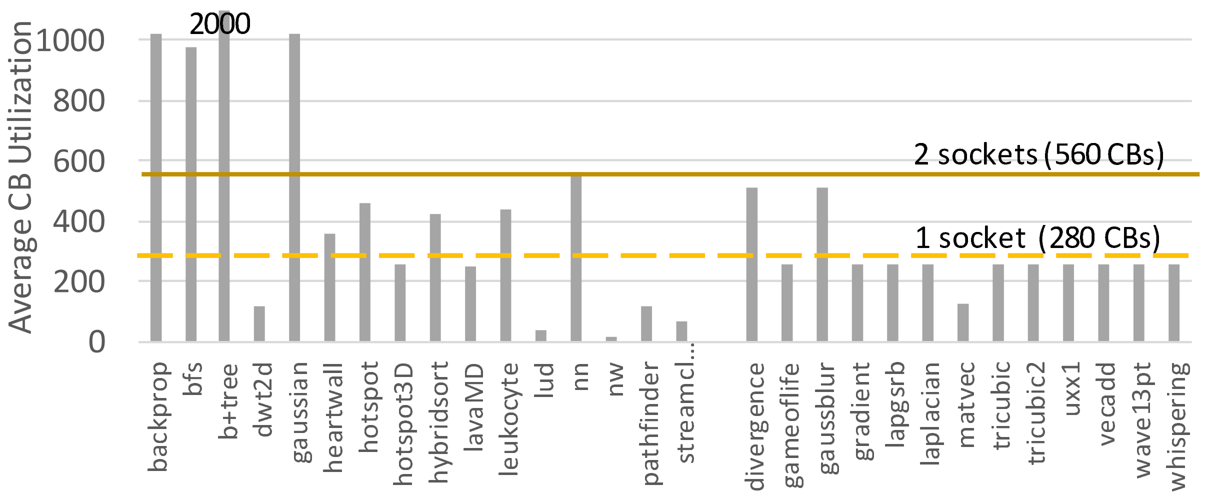


Figure 4.9: Control Block Utilization.

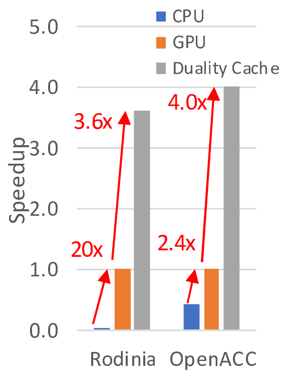


Figure 4.10: Average speedup.

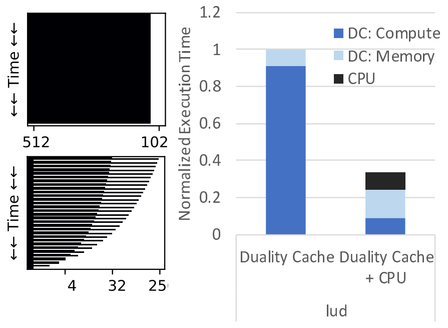


Figure 4.11: Kernel launch patterns (time vs. #CTAs) of `bfs`(top) and `lud`(bottom) and CPU Hybrid execution.

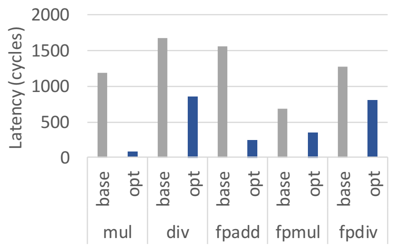


Figure 4.12: Average operation latency.

4.4.3 Performance without Host-Device Transfer

Figure 4.7 presents kernel execution time for *Duality Cache* and GPU. This experimental setup *eliminates* memcpy time from GPU and provides a GDDR5 memory to both *Duality Cache* and GPU. The goal is to compare the raw compute power of both architectures in a bandwidth neutral fashion. The execution time is normalized to that of GPU. We observe a 1.92× average speedup for Rodinia kernels and 2.39× speedup for OpenACC kernels. This speedup comes at a fraction of area cost of the CPU (3.5%), while the GPU server adds a new die of size 471 mm^2.

4.4.4 Deep Dive of Applications

Harnessing Full Potential of *Duality Cache* : Applications can fully exploit *Duality Cache* by exposing large parallelism and reusing data. We notice that many CUDA applications are optimized to GPU architectures, being aware of warp size and CTA size. Generally, programmers write a *tiled* program where each tile owns its sub-problem assigned to a CTA. Internally they use for-loops to iterate over the data for the sub-problem, often incrementing induction variable of a thread by warp size (32) to make warps sweep on the data. This creates a dependency between iterations, despite the absence of actual dependency. This is also driven by the fact that CTA size is limited to 1,024 threads due to the maximum register size of an SM. Although our CB can own 1,024 threads, we can expand CTA size beyond this limit provided there is no local communication between threads, as we discussed in section 4.2. Eliminating local communication is trivial by using atomic operations etc., so we modify some of the source code to unroll the outer for-loops and/or increase the CTA size, as shown in Table 4.3. OpenACC programs can also easily change the CTA size by setting the vector and worker size option in pragma. This optimization provides significant improvement in performance (e.g. 8.5× for leukocyte and 10.2× for heartwall).

Another important factor is data reuse. Given enough threads to fill CBs and existence of shared data, it is recommended to load the data and reuse it using fixed-length for-loops after launching CTAs to fill CBs. By this, we can avoid multiple fetches of shared data by different CTAs, and also take advantage of thread independent variable isolation.

Fine Interleaving of Serial and Parallel Code Using CPU: Since host-device communication cost is non-trivial for GPUs, CUDA programs tend to incorporate serial or nearly serial code with parallel code. Figure 4.11(left) illustrates kernel launch patterns of `bfs` and `lud` by showing number of launched CTAs (x-axis) vs. time (y-axis, advances from top to bottom). Ideal truly parallel kernels have a pattern similar to `bfs`, however, as can be seen, `lud` iteratively launches three kinds of kernels, one of which only contains 32 threads. Taking advantage of *Duality Cache* 's tight integration with CPU, we optimize `lud` to execute these small kernels on the host CPU using OpenMP. Figure 4.11(right) shows the execution time breakdown (normalized to the original

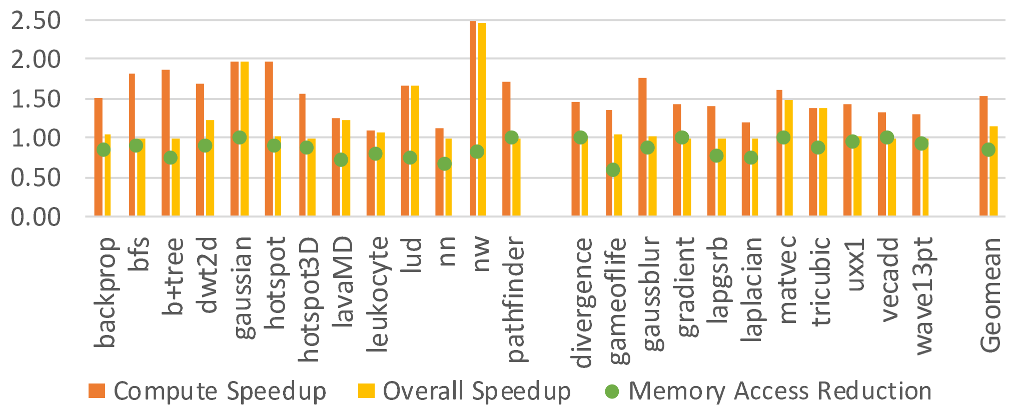


Figure 4.13: Effect of compiler optimizations.

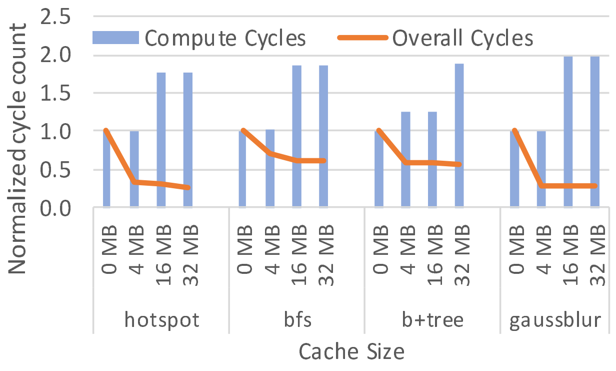


Figure 4.14: Effect of different cache allocation size.

version). We observe the optimized version of `lud` achieves a 2.26× speedup. Since the single operation latency of *Duality Cache* is much higher than CPU, we study CPU is more efficient to execute those small kernels. The same idea applies to OpenACC benchmarks as well.

4.4.5 Impact of Optimizations

Arithmetic Operation Latency: Figure 4.12 shows average arithmetic operation latency before (base) and after (opt) optimizations we present in Section 4.2.1. The operation latency is measured using Rodinia benchmarks. Integer multiplication observes the highest reduction in latency (13× better than the baseline). This is because, in many practical cases, integer multiplication is used to calculate address or some variables based on induction variables, and thus contains many leading zeros which we can skip by our optimization.

Floating point addition in many applications has a small dynamic range. The number of unique ediff found usually has its peak at 1 in the distribution (Section 4.2.1). Overall, optimized fpadd is 6.1× faster than the baseline. The proposed optimizations are not as effective for floating point multiplication and division compared to the correspondent integer operations. This is because the floating point is normalized and has implicit leading 1, which disables the leading zero optimization. However, *Duality Cache* still benefits from skipping iterations of bit 0s.

Cache Allocation: By default, we use unassigned CBs as cache. However, depending on the workload, allocating more cache can improve performance despite sacrificing parallelism. We analyze the source code through static analysis and identify some kernels that can possibly benefit from larger cache size, and adjust the cache allocation. Figure 4.14 illustrates the system performance of different cache allocation size for some representative applications. As in Figure 4.9, these applications have a high level of parallelism and can fill more than half of total CBs. The blue bars show compute cycles, and the orange lines present overall performance including memory access. The largest cache size we allocate is 32MB, which is equivalent to the half of the total CBs in our 2-socket baseline. We normalize the cycle count to that of 0-cache configuration. Although augmented cache allocation roughly doubles the computation due to reduced compute units, overall execution cycles decrease substantially because of improved memory performance contributed by the large caches. This optimization provides 3.54× performance improvement on average for applications with a high level of parallelism (CB utilization ≥ 512).

Compiler Optimization: To assess the effect of our compiler optimization, we compare the execution time of applications using two compilers: our compiler and a simple ISA translator. The simple ISA translator replaces PTX with DC-PTX without any scheduling and PTX optimizations. Since kernels cannot use multiple arrays without appropriate handling of operands between arrays, simple ISA translator uses one array per TB. This, on the other hand, provides 4× more available threads, thus each CB maintains 4K threads, each with 8 32-bit registers. For both compilers, we apply our arithmetic operation latency optimizations.

Figure 4.13 presents the application speedup of our compiler. Compute speedup shows the speedup of the critical computation path, and overall speedup includes memory access latency. The green dots show the reduction in the number of memory access saved by the parallel instruction scheduling. PTX optimizations and our instruction scheduler's efforts to maximize parallelism and to reduce register spilling achieve 1.52× faster computation and 14.3% less memory access. This translates into 1.14× better overall performance.

4.4.6 Energy

Figure 4.8 shows the energy breakdown of the benchmarks. This is a system-to-system comparison; GPU includes energy for both memcpy and kernel. *Duality Cache* energy is normalized to the GPU energy and has a breakdown of CPU+DRAM (including memory controller), load/store instructions, and computation in *Duality Cache* . Because of the reduced execution time, we achieve 5.85× energy efficiency compared to GPU system. One core is active during execution to serve instructions. This makes CPU and DRAM access dominant in energy consumption. The only exception is tricubic, one of compute-intensive kernels, where compute energy accounts for 30.7% of total energy consumption.

4.5 Summary

We present the *Duality Cache* system stack that runs general purpose GPU programs on caches. Enabling in-situ floating point and transcendental functions brings computation capability that can execute SIMT programs. Our compiler introduces optimizations to enhance parallelism and efficiency within the constraints of in-cache computation, and compiles CUDA and OpenACC programs for *Duality Cache* . Our experimental results show the *Duality Cache* architecture improves performance of GPU benchmarks by 3.6× and OpenACC benchmarks by 4.0× over a server class GPU. Re-purposing existing caches provides 72.6× better performance for CPU with only 3.5% of area cost.

CHAPTER 5

Multi-Layer In-Memory Computing

As the memory hierarchy today has combined different memories exploiting the difference in their characteristics(e.g., speed and density), in-memory computing has different advantages and disadvantages depending on the memory substrates. For example, while SRAM can benefit from its fast clock speed, NVMs can store a large amount of data utilizing dense memory cells.

The wide spectrum of memory technologies and their differentiated compute capabilities open up an opportunity to perform in-memory computing in variable memory layers in the hierarchy. The preference of in-memory computing is determined by multiple and intertwined factors, such as reuse patterns, data size, and instruction mix [74, 89]. Considering the multiple options of memories, customizing the location of in-memory computing for specific application domains will yield a significant benefit. This is prominent for applications with runtime workload dynamism, i.e., the performance determinants (e.g. working dataset size) have a broad distribution and are knowable only at runtime. To maximize the potential of multi-layer in-memory computing, determining when and where to execute in the memory hierarchy is a challenge. In this chapter, we introduce a framework that enables MLIMP, multi-layer in-memory processing.

5.1 Challenges and Opportunities

This work addresses several challenges to enable multi-layer in-memory computing. *First*, considering the state-of-the-art, we propose a system design that supports a common programming frontend for multi-layer in-memory computing and a memory allocation scheme that allows in-memory computing to co-exist with traditional memory systems. *Second*, we find Graph Neural Networks (GNNs) [90, 91, 92] entail significant workload dynamism for processing subgraphs. We devise kernel mappings of the critical kernels in GNNs for in-memory computing, paying attention to maximizing its reuse and resource utilization, and use them as a representative case study to show the advantages of multi-layer in-memory computing. We also conduct other case studies for multiprogramming scenarios using data parallel applications studied in the prior work. *Third*, we design a scheduler and a performance predictor that are essential to perform efficient job scheduling and fully utilize the resources in multi-layer in-memory computing. Job scheduling in multi-layer in-memory computing is classified into an NP-hard resource constrained project scheduling problem. Also, memory allocation size has to be adjusted to balance parallelism and per job latency. Based on an analytical scaling model, we develop efficient heuristics to schedule jobs in heterogeneous in-memory systems. Further, to provide an estimation of performance for a specific configuration, we propose a performance predictor based on neural network based regressors. We observe that taking full advantage of multi-layer in-memory computing is *not possible without introducing sophisticated job scheduling*.

5.1.1 GNNs and Dynamism in Workload

We conduct a detailed workload analysis for Graph Neural Networks or GNNs for in-memory computing and show an interesting case study of GNNs that can significantly benefit from multi-layer in-memory computing. This section covers the basic concept of GNNs and our preliminary analysis showing their inherent dynamism in the workload.

5.1.1.1 GNN

GNN [90, 91] is an algorithm applied to a graph $G = (V, E)$, where V is a set of vertices, and E is a set of edges. Each node v has its input feature vector x_v, which is generated in the preprocessing step. Generally, GNN has one to five layers, but the number of layers is arbitrary. Each node

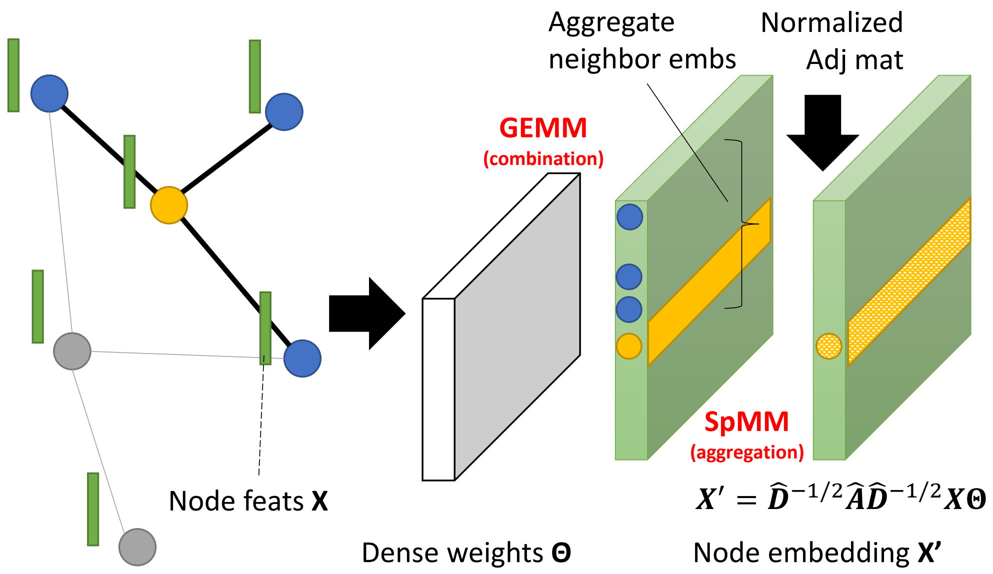


Figure 5.1: Operations in GCN

propagates its node features to its neighbors and updates the features at each layer, using input vector x_v for the first layer. Thus, output node features (also referred to as node embeddings) of the k-th layer include information from k-hop away neighbors.

The GNN training (forward pass) and inference usually contain three main steps: *pre-processing*, *iterative updates*, and *readout*. This initial step, *pre-processing*, is done offline. It generates the initial input feature vectors and graph representations. It is followed by the *iterative update* step, where the feature vectors of each node and edge are iteratively updated at each layer in GNN via an *aggregation-combination* function, as shown in Figure 5.1.

During aggregation, the feature vectors from neighboring nodes and the feature of the node/edge itself are *aggregated* by functions such as mean, max, weighted mean to a single feature vector. This process can be expressed using a matrix representation, $B = \overline{A}X$, where $\overline{A}$ is the normalized adjacency matrix and X is the feature matrix of which row is composed of the feature vector of a node in the graph. Graph Convolutional Network (GCN) uses normalized adjacency matrix $\overline{A} = \hat{D}^{-1/2}A\hat{D}^{-1/2}$, where D is the diagonal matrix such that $\hat{D}_{v,v} = |N(v)|$ and $N(v)$ is the neighbor node set of v. Since A is a sparse matrix and X is a dense matrix, aggregation performs SpMM (sparse matrix matrix multiplication), which is known to have an irregular memory access pattern.

Then the aggregated feature vector will be *combined* to yield a new feature vector. The new feature vector is used as an input for the next step. The combination function is typically composed of a feed-forward network. Thus, the combination step can be described as $X' = \sigma(B\Theta + b)$, where B is the possibly-dense matrix after the aggregation step, Θ is the dense weight matrix, b is the bias, σ is a non-linear activation function (e.g., ReLU). Hence, the main kernel of the combination step is GEMM (general matrix multiplication), which involves intense computation.

Following the iterative update step, the *readout* step outputs node/edge embeddings which are low-dimensional vectors or a graph embedding that summarizes the graph information. These outputs are passed to the downstream prediction tasks. For example, link prediction typically uses an additional MLP based predictor to predict the linkage of a pair of nodes.

When training GNNs, the backward pass follows the explained forward pass. The backward pass preforms backpropagation and optimizes model parameters using gradient descent.

5.1.1.2 Workload Dynamism

State-of-the-art GNN based link prediction framework employs an approach called subgraph learning [93, 94]. It extracts the k-hop neighborhood of nodes and applies GNNs. This approach can also be regarded as mini-batching because only a small subset of nodes are fed to the GNN layers.

Traditional GNN such as Graph Convolutional Network (GCN) [92] and GraphSAGE [95] employ full batching, where a GNN layer accepts the entire graph and learns from them using message-passing like feature communication. In fact, training GNNs on a large-scale graph is challenging due to node dependency and requires lots of memory and time [96]. In this context, mini-batching attracts growing attention for its memory efficiency, fast learning time, scalability to large graphs, and comparable accuracy. OGB [97] reports mini-batching can even outperform traditional GNNs because it can induce regularization effects. It is also worth mentioning that it is possible to perform mini-batching for full-batch GNNs, while the benefit of mini-batching is not as much as GNNs designed for mini-batching. That being said, subgraph learning approaches are

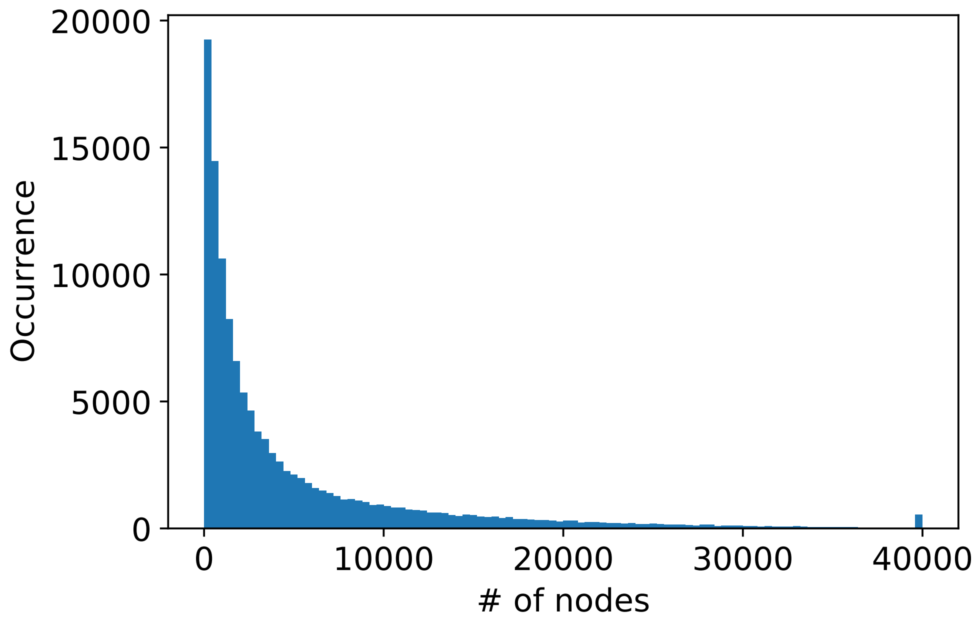


Figure 5.2: Node distribution of 3-hop subgraphs (ogbl-citation2).

widely applicable to many existing GNN frameworks beyond the link prediction tasks.

Mini-batching induces substantial variation in the working dataset size and compute load. The graph size distribution of a real-world graph is shown in Figure 5.2, and we see the variation is also propageted in the processing time. Thus, a monolithic approach using a single type of hardware or memory-centric acceleration (e.g., GPU or In-SRAM only) might be suboptimal, and sophisticated job scheduling is crucial to take the full adgantage of multi-layer in-memory computing.

5.1.2 Motivation

While there is a significant body of research on in-memory computing with individual layers of memory, a framework that integrates multiple computable memories into the memory hierarchy has been lacking. Figure 5.3 shows the relative energy per access, delay, and metrics for calculating available compute parallelism of different memory technologies. The parallelism can be estimated based on available sense amplifiers at the bitline peripherals per unit area. Available parallelism is also dependent on the bit-cell structure and design target (e.g., cache, main memory, or storage DIMMs). For example, while NAND-Flash and DRAM have a small cell size, their available

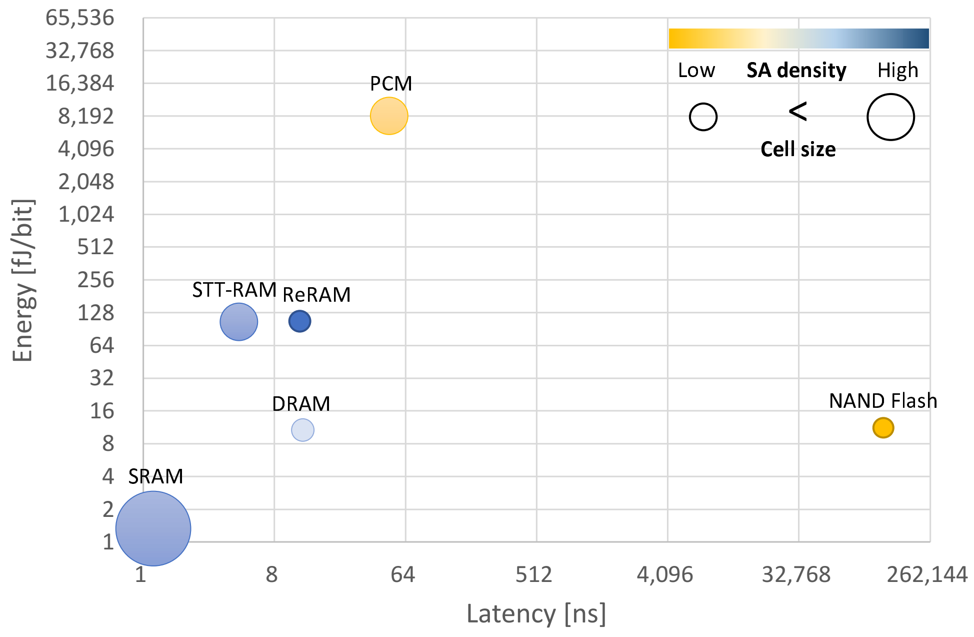


Figure 5.3: Energy, latency, and parallelism characteristics of various memory technologies.

parallelism can be low because a large number of cells in an array share the same set of sensing amplifiers (low SA density).

Computing with Non-Volatile Memories (NVMs) has different trade-offs compared to computing with SRAM or DRAM. Since NVMs are more stable against data corruption, they can support operations involving multiple wordlines. Due to their high density, NVMs can accommodate large datasets which dwarf SRAMs. Higher density also increases data level parallelism of in-place computation. On the other hand, in-memory computing in NVMs (STT-RAM and ReRAM) can be one to two orders of magnitude slower and requires significantly higher energy per bit when compared to SRAMs. Further, NVMs have limited endurance (and high write energy/delay) which curtails the number of writes the memories can reliably sustain. Similarly, DRAMs pose their own unique challenges. Given the wide spectrum of memory technologies and their differentiated compute capabilities, customizing the memory hierarchy for specific application domains may yield significant benefits.

In this work, we will closely look at mini-batching GNN as a systematic case study that can benefit from multi-layer in-memory computing. Mini-batching leads to a considerable variation of working data set size, operation intensity, and memory access locality depending on the input query and sampled subgraphs. From the rich set of workloads from the GNN framework, we aim to figure out efficient job execution strategies to make full use of multi-layer in-memory computing.

5.2 Multi-Layer In-Memory Processing

In this section, we present a system stack and job dispatching strategies for multi-layer in-memory computing. The architecture of multi-layer in-memory computing is presented, then the kernel mapping for important GNN operations is discussed for each memory substrate. Following that, we introduce two important components to enable efficient job planning, i.e., job scheduler and performance predictor.

5.2.1 Architecture Overview

5.2.1.1 Common Programming Interface

Prior work has covered most of the innovation needed to enable in-memory computing in the existing memory hierarchy as described in the previous chapters. To interface with the heterogeneous in-memory computing resource efficiently, we need to design a common programming frontend. While the instruction set architecture (ISA) and the preferred data mapping within an array vary for each memory (e.g., in-SRAM computing performs bit-serial computing on vertically aligned data, while in-ReRAM computing uses bit-parallel computing with multi-level cells), most of the in-memory computing work supports arithmetic operation level abstraction either in their API or ISA. They usually support integer arithmetic operations, and some also support floating points. For wide compatibility with past proposals, we focus on integer operations in this work.

For our baseline system, we use in-SRAM computing based on [89, 1], in-ReRAM computing based on [3, 74], and charge-sharing based in-DRAM computing [4, 98]. Binary bit-serial computing with bit transposed data is employed for in-SRAM and in-DRAM computing. To make peripheral complexity comparable, memory arrays compute a universal operator such as NAND and NOR with the smallest possible cycle counts, and the result and any byproducts (e.g., AND) are fed to extra logic gates at the peripheral to perform the rest of the operations. In-ReRAM computing performs bit-parallel computing with the peripheral shifter and adder, and extra logic such as LUTs is introduced to enable other non-native operations.

Taking the intersection of supported arithmetic operations among the three types of in-memory computing devices, the programming interface supports integer addition, subtraction, multiplication, division, comparison, moves, and simple transcendental functions (e.g. exp2). The arithmetic level abstraction is further expanded into a sequence of micro-operations within controllers or FSMs in each memory [4, 89, 1].

These arithmetic primitives are wrapped by a high-level API. This high-level API provides operation level abstractions such as GEMM and SpMM to work with existing machine learning frameworks. Each kernel is compiled for each memory device, and at the execution time, a suitable in-memory function is chosen from the library by a job scheduler and offloaded to the memories. In this way, architecture-specific optimizations (e.g., VLIW execution of [89]) and algorithm-level optimizations (e.g., kernel execution order) can co-exist.

5.2.1.2 Memory allocation

Memory workspace for in-memory computing is allocated within a scratchpad memory region in each memory to ease the collocation with the existing memory virtualization frameworks. There is a body of work trying to enable private scratchpad memory within the cache and main memory [99, 100]. This is a middle ground approach of two extremes: using all memory space as a scratchpad for in-memory computing (most of the prior work) and completely integrating in-memory computing with the existing memory management system.

Complete integration would enable seamless processing of in-memory operations with minimized data copy and transformation. While there are non-trivial benefits for the complete integration of in-memory computing under the existing memory management system, its cost is also non-trivial. Supporting compute cache lines and other cache lines in a finer-grained manner under the general set-associative cache scheme would lead to a prohibitive cost for guaranteeing data layout and bookkeeping the cache lines for avoiding unexpected cache line replacement, etc. In contrast, the hybrid approach using VLS [99] enables scratchpad memory on a coarse partition of cache (e.g., a single way) with a tiny modification to the cache architecture.

It is possible in the main memory to align data to an exact position of a physical memory array by reverse-engineering the XOR-based address mapping of microarchitectures [101] and by modifying operating systems' memory management system to support finer-grained page coloring [102]. It is, however, comes at the cost of expensive page management (i.e., page numbers for each color have to be extensively searched [102, 103]) and involves a risk of external fragmentation. We thus consider the complete integration of in-memory computing with the current memory virtualization scheme does not provide convincing benefits compared to its cost, but future work can address this issue. The hybrid approach still allows compute regions to co-exist with existing managed memory space in a coarse grained manner, while guaranteeing data layout flexibility that is essential to in-memory computing.

5.2.1.3 Control Granularity

Control granularity defines the number of controllers that process independent instruction streams. If there is only one controller, all arrays (or a group of arrays) process the same instruction as a massive SIMD unit. Multiple controllers can accept different instruction streams in a MIMD manner, similar to a multicore vector processor or a GPU. The control granularity is traded off by the area cost of the controller and wires.

5.2.2 Kernel Mapping

We show an example workflow of GNN in Figure 5.4. In this section, we present the kernel mapping of the representative kernels in GNN, i.e., GEMM and SpMM.

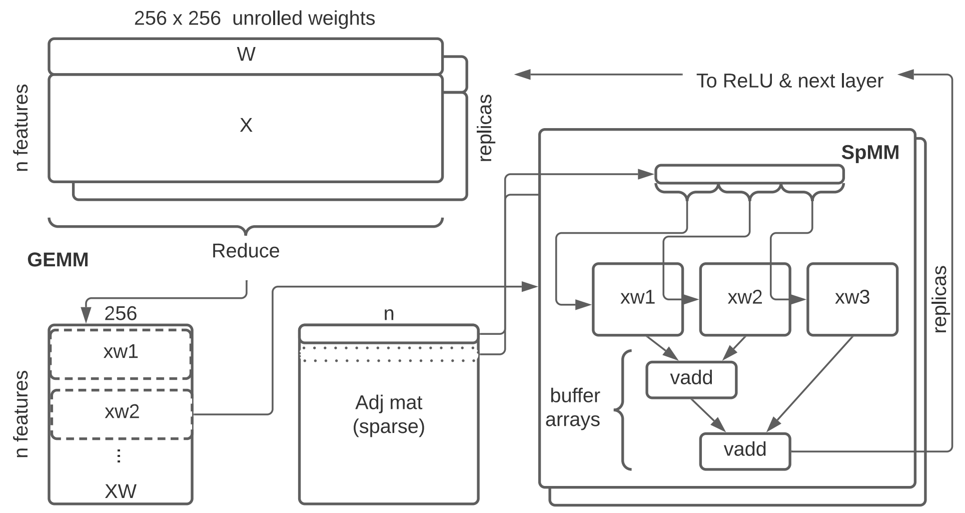


Figure 5.4: GNN workflow in bit-serial architecture.

5.2.2.1 GEMM

General matrix multiplication is a core kernel of many machine learning frameworks. Prior work has proposed efficient GEMM operations in memory [3, 2, 14, 1]. For example, in-ReRAM computing can perform vector-matrix multiplication using analog multi-operand MAC computation. The weights are stored in memory and reused across different inputs. The compute-efficient data mapping of weights varies according to the memory. For example, in-ReRAM computing generally employs a natural 2D mapping of the weight matrix. Each value can use multiple memory cells to improve precision [3].

Bit-serial computing does not generally support multi-operand operations. Thus, it is crucial to exploit parallelism in architecture efficiently. For example, Neural Cache [1] unrolls input activation of CNN for each sliding window and duplicates it for each output channel. We take a similar approach for GEMM. The weight matrix is serialized to a vector representation and stored in the topmost register of each SIMD slot. The input feature vector is duplicated for each column of the weight matrix and stored in a SIMD slot with a corresponding weight multiplicand. In this way, all multiplication operations can be done in parallel for each input feature vector. Then, reduction operations are performed to make sums to complete dot-product operations. A memory array can

Algorithm 2 Sparse Matrix Multi-Vector Multiplication (SpMM).

```
Input: CSR A[M][N], float B[N][K]
Output: float C[M][K]
1: for $i = 0$ to $A.rows - 1$ do
2:     for $j = A.rowptr[i]$ to $A.rowptr[i+1] - 1$ do
3:         for $k = 0$ to $K - 1$ do
4:             $C[i][k] += A.values[j] \times B[A.colidx[j]][k]$
5:         end for
6:     end for
7: end for
```

have multiple input feature vectors. The weights can also be replicated to fully utilize the available memory space.

5.2.2.2 SpMM

Sparse Matrix Matrix Multiplication (SpMM) computes the multiplication of sparse matrix A and dense matrix B. Its algorithm is shown in Algorithm 2. For GNN, SpMM is used to perform message passing between nodes, using the (sparse) adjacency matrix as A and dense node features or embeddings as B. Generic SpMM performs dot-product as the *combination* operation of GNNs, but the reduction operator (sum in this case) can be replaced with other operators, such as max or mean.

In-memory computing in general is less efficient for sparse computation. The main reason for this is the random scattered access patterns of the workload. While in-memory computing generally requires operands to be arranged in a designated location to perform computation and exploit parallelism, this scheme cannot be directly applied to a compressed storage format of a sparse matrix. To expose the computation models of in-memory computing, sparse matrices need to be decompressed to the dense matrix format, reinserting null elements eliminated by the compression. Existing work on in-memory graph processing accelerator [104] also performs decompression of the CSR format to perform sparse matrix-vector multiplication (SpMV), which many graph algorithms can be transformed into.

Decompression leads to the following inefficiencies. First, it triggers data movement in the memory, paying the cost for bandwidth, energy, and cell write count for additional write cycles. Second, because of the sparsity, computation density per array becomes low, undermining the dense compute capability of a highly parallel in-memory architecture. Third, while there is a locality in edge access (e.g., identifying adjacent vertices in the CSR format), access patterns of vertices are generally irregular, causing only a subset of decompressed data to be utilized, given a general vertex-centric scheme is used. This not only reduces computation density in a decompressed matrix, but also results in repetitive decompression to access vertices in the dense format across different timeframes. GraphR [104] converts a submatrix of compressed adjacency matrix to a dense format and load it to a ReRAM crossbar. While submatrix with all zeros can be skipped, sparsity in the input graph can still cause the above-mentioned computation density issues.

For these reasons, we store the dense matrix B in the memory array to bypass the computation density issues related to the sparse format. B is partitioned into horizontal slices and stored into arrays. Then, a corresponding vertical slice of the sparse matrix A is loaded from the main memory and processed row-by-row. If A is a binary adjacency matrix, arrays will perform a series of vector additions of some rows of B, using non-zero column indices of A as indices to look up the B rows. If A has non-binary values (e.g. edge weights), arrays will instead perform a dot-product computation using the A value as the multiplicand. Buffer arrays are utilized to temporarily store and accumulate the partial sum vector from multiple arrays, playing a similar role as a reduction tree.

As illustrated in Figure 5.5, there are several reuse patterns for SpMM [105], and our approach is classified into B-stationary. B-stationary maximizes the reuse of the dense B matrix, which is ideal for mini-batching GNNs because it is known that all the node features (stored as B) are reused several times while processing the batch. Exploiting the predetermined knowledge of feature reuse, in-memory computing can fully utilize the locality of node feature access. As in [105], we adopt Densified CSR (DCSR) format [106] for providing a vertical slice of A for memory and compute performance. DCSR can be precomputed or dynamically generated using efficient near-memory hardware [105]. On the other hand, B-stationary requires an atomic update of the results given

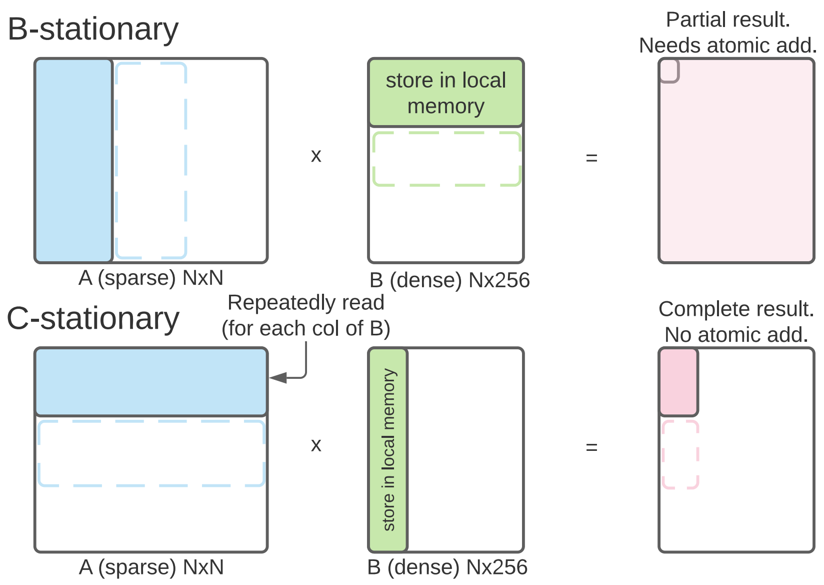


Figure 5.5: Data reuse patterns of SpMM.

multiple processor entities are working on different matrix slices that add partial sums to the same output elements. This atomic update can be done with a memory side atomic module with paying approximately half of the bandwidth cost to perform both load and store.

Alternatively, one can minimize this atomic update cost by employing a C-stationary approach. C-stationary computes the complete output in C, reusing the local memory for storing the partial sums. C-stationary loads complete columns (a vertical slice) of B to the memory arrays, and for each slice, the full A is accessed to compute a vertical slice of C. While B-stationary fetches A and B only once and updates C multiple times, C-stationary accesses B and C once and fetches the full A multiple times.

C-stationary is a common straightforward approach for GPU and CPU [106, 105], while it is less efficient for in-memory processors. From the viewpoint of memory access, the size of A is usually smaller than the size of B and C. Therefore, loading A multiple times (C-stationary) seems more efficient. However, given the skewed non-zero distribution of real word graphs, it is observed that not many slice pairs of A and B contribute to one output position [105], thus the cost of atomic add is not significant [105]. This is in contrast to C-stationary where multi-loading A is most likely unavoidable. B-stationary also provides an opportunity for efficient vector processing. In contrast, C-stationary needs to perform lengthy reduction operations with a lot of null entries to make a complete output. We observe B-stationary achieves 4.3x better memory latency performance and 42x better compute performance (obgl-collab dataset [97]). It is certain that the gap between B- and C-stationary becomes small as A becomes denser, but with the solid advantages of B-stationary in in-memory computing, we adopt B-stationary as our baseline SpMM kernel mapping approach.

We also replicate the B slices within the memory allocation, reducing the slice size accordingly. This is to leverage the input row (i.e., rows of A) parallelism. Input row parallelism can be exploited by performing multiple reductions or dot-product operations for different rows of A. While a single large B slice can reduce memory roundtrips and replication cost, it is difficult to leverage input row parallelism without replicas because each array has to sequence and handle multiple rows in flight. More than one A rows can concurrently access the same rows of B on the same memory array, and each must keep track of the destination buffer arrays etc. to perform the final reduction. B replication can exploit the input row parallelism by processing different A rows on each replica, each with its own buffer. Since the input row parallelism is easy to find, we find that having a few replicas can help achieve a good performance scaling.

5.2.2.3 GNN kernels

As illustrated in Figure 5.4, GNN layers are mainly composed of GEMM and SpMM, which form the building block of the aggregation and combination kernels. While it also contains other operations such as activation functions (e.g., ReLU), they take insignificant time and are thus executed in the host processor.

The initial step of mini-batching GNN is the mini-batch generation. Mini-batches can be precomputed or dynamically generated using a data generator process or a remote graph server. We assume the mini-batch data is precomputed [94], but since the workload is similar to breadth first search (BFS), it can be efficiently integrated with any near-memory computing enabled memory systems [107, 48, 108].

By default, a mini-batch contains a subgraph for each query input. However, one can choose to generate a concatenated subgraph that has a union of all nodes in the adjacency matrix. This approach relinquishes the opportunity of mapping each small subgraph to different memories (covered in the following subsections); however, it is useful if the graph has a high degree of connectivity and the intersection of nodes in each k-hop subgraph is large. In such a case, we have a good chance of reusing node features across different query inputs, thus putting all node features in one memory can maximize this reuse while minimizing the redundant memory access to the same node. We observe the large connectivity in some of the graphs in the nature domain in OGB [97] and take this approach. For all the other graphs, we generate subgraphs for each input, as they result in better performance due to the opportunity of subgraph level load distribution.

The SpMM and GEMM kernels are modularized, which means the kernels do not depend on each other and can be executed in any order. Since different GNN approach has different preference or restriction of the execution order, we can flexibly change the order based on the performance and the GNN algorithm itself. In this work, we adopt GCN [92], which prefers GEMM followed by SpMM.

5.2.2.4 Data Parallel Applications

Data parallel application explored in IMP [74] has large SIMD vectorizability exploiting the data-parallel nature of the workload. We extract its compute kernels and compile them for in-SRAM and in-ReRAM processors. Each program is compiled for each memory, and at the execution time, we choose the best in-memory processor based on the scheduler output.

5.2.3 Scheduler

The scheduler receives a batch of jobs, allocates memory resources in an appropriate in-memory device for each job, and schedules jobs in the batch in a right order. The job scheduler targets to maximize the utilization of the available memory resources and minimize the batch processing time. This job scheduler is the key enabler of efficient heterogeneous execution on a multi-layer in-memory computing system. In this section, we will discuss the challenges of resource constraint job scheduling and our approach to solving them using heuristics.

5.2.3.1 Scheduling Strategies and Resource Constrained Project Scheduling Problem (RCPSP)

Conventional computers perform job scheduling in their operating system (OSs). Among many job scheduling approaches for OSs, one of the simplest and best approaches to minimize waiting time is the shorted job first (SJF) approach [109]. SJF takes a table of processes, their arrival time, and their expected execution time (burst time), and tries to minimize the average amount of time that each process has to wait by scheduling the process with the smallest burst time first. Multilevel feedback queue, widely employed scheduler algorithm by UNIX OSs, tries to approximate SJF without the prior knowledge of oracle execution time, also adding sophistication for various job types (e.g., interactive or IO heavy jobs, etc.) [109].

We notice that the existing job scheduling approaches are not directly applicable to in-memory computing. First, while OS job scheduling targets a homogeneous CPU architecture, we need to choose from a variety of in-memory processors. While there exist OS job schedulers for heterogeneous cores (e.g., big.LITTLE architecture [110]), in-memory computing is additionally required to determine the allocation size of the memory. Execution time is also largely dependent on memory properties and jobs; thus, simple scaling techniques (e.g., big cores are about x times faster than small cores) cannot be applied. These three factors, i.e. memory type, allocation size, and difficulty of estimating execution time, make it difficult to apply OS's job scheduler for in-memory computing.

In fact, the job scheduling problem of this set-up is categorized into NP-hard Resource Constrained Project Scheduling Problem (RCPSP) [111]. As illustrated in Figure 5.6, scheduler has to choose the right resource amount (and resource type) for a job, as well as its execution order. Multiple jobs can be executed at a time if the busy resource amount does not exceed the limit at any time. While there is a rich body of work from the operations study community [111, 112, 113], there is no known golden solution to RCPSP, so the problem needs to be approached on a case-by-

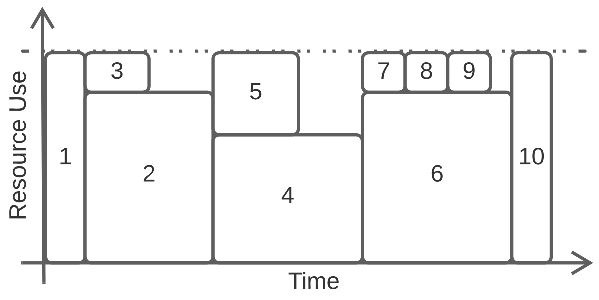


Figure 5.6: Resource Constrained Project Scheduling Problem (RCPSP). Multi-layer in-memory computing has another dimension for resource (memory) type.

case basis [114].

Common approaches to solving project scheduling problems can be classified into three: Johnson's rule [115], dispatching rule, and meta-heuristics. Johnson's rule [115] can provide an optimal schedule only for specific problem set-ups (e.g., up to two processors with fixed resources); its restriction makes its scope very narrow and difficult to apply for other problems. Dispatching rule includes FIFO, SJF, and others that define the dispatching order, and meta-heuristics include taboo search, annealing, etc. Dispatching rules are simple and good at locally optimized job scheduling (e.g., scheduling within a queue), but it finds difficulty in finding globally optimized schedules. This is particularly important when there are multiple processing entities(i.e., memories) with different job queues. Meta-heuristics takes a long time to converge. Considering the limited scheduling time, our approach is based on the efficient dispatching approach, which is assisted by a custom-made heuristic to make globally optimized scheduling.

5.2.3.2 Baseline Schedulers

For our baseline, we use the longest job first (LJF) scheduling. LJF scheduling tends to increase work in progress while making short jobs late. This is ideal for minimizing the batch process time. Short jobs being late does not matter because all jobs in the batch need to wait for the completion of batch processing before moving to the next step. Rather, increasing in-flight jobs is important because it improves resource utilization. Moreover, if jobs in one processor end earlier, LJF makes it easier to load balance by filling the gap with smaller jobs left in the queue.

Each job in an incoming batch is first processed by a performance predictor (covered in the next section) to calculate the estimated execution time for each in-memory processor. The baseline scheduling does not adjust the memory allocation size. The predictor predicts the execution time assuming a fixed allocation size M_{mem}/P where M_{mem} is the available memory allocation size of memory mem and P is the number of parallel jobs. Then, the jobs are pushed in a queue in descending order according to the shortest execution time among all available memories. Whenever a spot is available, a job item is dequeued and scheduled to the available memory, giving priority to spots in the best performing memory.

We also design a multi-queue LJF where each memory has its own queue and jobs are enqueued to the queue of the best performing memory. This approach reduces the chance of inter-memory job balancing but also the chance of allocating jobs to suboptimal memories.

5.2.3.3 Resource Scaling and Allocation Size Criteria

The baseline LJF relinquishes the opportunity of adjustable memory allocation. A larger allocation can be useful for a problem that prefers a large memory allocation to deal with large working set data for efficiency. To seize this opportunity, it is important to know the best memory allocation size for a job. This requires an understanding of the relationship of the allocation size with the execution time. Since this relationship is program and problem-specific, we will make a performance model trainable and adjustable for each problem.

Our performance model is composed of two parts, load model and compute model, and the expected process time t for job x with allocation size z is calculated as the sum of the latency from the two models:

$$t(x, z) = t_{ld}(x, z) + t_{cmpt}(x, z). \tag{5.1}$$

The load latency t_{ld} is calculated based on the load cost and the replication cost. For each load iteration, a working dataset (e.g., horizontal slices of the dense B matrix for SpMM) is loaded from the main memory $t_{memld}(x)$ and optionally replicated across different control regions (Section 5.2.1.3) to exploit the data level parallelism. The number of replications is calculated by $z/a_{repunit}$, where $a_{repunit}$ is a unit allocation for one replica (user-defined parameter). If the whole working set does not fit in the allocated memory, the number of load iterations $n_{lditer}(x) = x[\text{datasize}]/a_{repunit}$ becomes larger than 1. Thus, we have

$$t_{ld}(x, z) = n_{lditer}(x) \times (t_{memld}(x) + t_{replica}(x)(z/a_{repunit})). \tag{5.2}$$

The compute model assumes the *scale free* property [116] of resource size and performance. With a scale parameter α and a shape parameter β, the scale-free model is described as $f(x) = (\alpha/x)^{\beta}$ [116]. Given perfect scaling ($\beta = 1$), the model dictates that $n\times$ larger x always results in $n\times$ smaller $f(x)$, regardless of x. Since parallel part of applications can achieve close-to-linear performance scaling w.r.t the number of processors regardless of the baseline processor configuration, assuming no memory bottlenecks, this model fits well for in-memory computing. The parallelization cost can be modeled by setting the shape parameter β to less than 1 (empirically obtained).

The baseline performance is provided by the performance predictor covered in the following subsection. This provides the unit performance $t_{unit}(x)$ under the unit allocation $a_{unit} = M_{mem}/P$. Therefore, we have

$$t_{cmpt}(x, z) = t_{unit}(x) \left(\frac{a_{unit}}{z}\right)^{\beta}. \tag{5.3}$$

We observe the scale-free compute model fits well to a variety of problems. For example, SpMM in OGB sees a median R^2 of 0.998. Although there is a small deviation in the performance model for the combination of small-sized jobs and large memory allocation due to imperfect job balancing issues (e.g., there is not enough parallelism to exploit all allocated resources), it less affects the overall performance because such jobs typically take a very small time and because it is unlikely for small jobs to get large resource from a scheduler.

Now, we need to make a suggestion on the allocation size. The minimum of the performance curve can be analytically calculated by taking the partial derivative of $t(x, z)$ w.r.t z. It

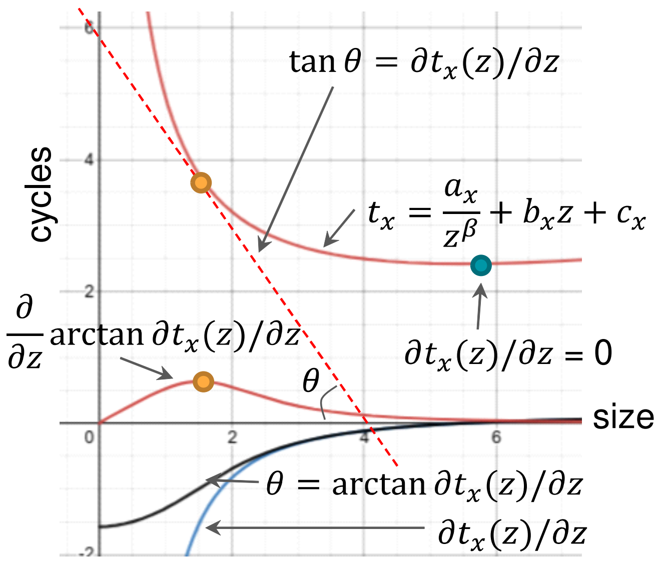


Figure 5.7: Analytical model for allocation suggestions.

is straightforward to calculate the derivative considering the fact that $t(x, z)$ can be written as $t_x(z) = a_x/z^\beta + b_x z + c_x$. The coefficients a_x, b_x, c_x are fixed given a problem x. We find that using $z_{\max}$ that gives minimal t_x (i.e., the blue dot in Figure 5.7) does not provide good performance because $z_{\max}$ tends to overprovision the resource up to the point that the immediate asymptotic increase of z does not significantly contribute to the performance t.

We thus use the angle θ of the tangent line of t_x. The angle of the tangent line of $t_x(z)$ is given by $\theta = \arctan \partial t_x(z)/\partial z$. The performance improvement per small increase of z is projected to the angle speed of θ. Unlike t_x, θ fits within the half-open interval $[-\pi/2, k)$, where k is the asymptote of θ. By taking the largest infinitesimal change of θ, the resource-efficient allocation size z' is obtained (the orange dot in Figure 5.7). This can be calculated by taking the maximum of the derivative of θ:

$$z' = \arg\max \frac{\partial}{\partial z}\left(\arctan\left(\frac{\partial}{\partial z} t_x(z)\right)\right). \tag{5.4}$$

The derivative of the function after argmax in Equation 5.4, needed for analytically calculating z', is in fact an equation as simple as quadratic formula plus a single exponentiation to a real number. Therefore, calculating z' will never be a bottleneck.

Based on the suggested allocation size, we will make a final decision on allocation size and

Algorithm 3 Inter-Queue Adjustment.

```
Input: $queues$: Map[mem, queue], $t_{mem}$: $x \rightarrow t_{mem\_unit}(x)$
 1: for up to $N$ times do
 2:     $\bar{t}$ = {$mem$ : get_mean($queues[mem]$) foreach $mem$}
 3:     Get $max\_mem, max\_mean$
 4:     Get $min\_mem, min\_mean$
 5:     if difference($max\_mean$, $min\_mean$) > $\epsilon$ then
 6:         $migr\_cand = \arg\min_{x \in queues[max\_mem]} t_{min\_mem}(x)$
 7:         Migrate $migr\_cand$ from $queues[max\_mem]$ to $queues[min\_mem]$ if $\bar{t}$ improves
    else break
 8:     else
 9:         break
10:     end if
11: end for
12: return $queues$
```

dispatching order using the following scheduling approaches.

5.2.3.4 Adaptive Scheduling

To balance the execution time of the multi-queue LJF scheduling, we introduce *inter-queue adjustment* (Algorithm 3). The goal of the inter-queue adjustment is to balance the mean execution time between queues. For each iteration, it calculates the mean processing time of the job items in each queue. If the maximum difference of the mean times is larger than the acceptable gap ϵ, it migrates $migr\_cand$, the job with the smallest execution time (when executed in min_mem), from the max_mem queue to min_mem queue. This is repeated until the mean time difference is below ϵ or migration no longer contributes to improvement in job balancing. After successful inter-queue adjustment, proper resource distribution leads to an execution time close to the mean.

Adaptive scheduling dispatches jobs in the queue in a greedy fashion. Whenever there are available resources that can run a job with its requested allocation, it runs the job, giving priority to larger jobs. If there are any remainder resources not allocated by the prior procedure, the scheduler calculates the expected completion time for each awaiting job in the queue and dispatches jobs if they can finish earlier than the completion of jobs in flight using the remainder resources.

5.2.3.5 Global Scheduling

Adaptive scheduling can flexibly adjust the dispatching order even if there is a gap between the estimated execution time and the actual time. However, it is challenging to fully utilize the resources due to scheduling bubbles. Bubbles are introduced when a small remainder allocation cannot be utilized by any waiting jobs.

The global scheduler adjusts the allocation size in each queue to fully utilize the resources and generates a complete job dispatching schedule beforehand. Instead of directly using the recommended resource allocation, the global scheduler further adjusts the allocation size using the *intra-queue adjustment* algorithm in Algorithm 4. The objective of the intra-queue adjustment is to balance the time of long jobs, which can take longer than the mean execution time, by trading the resources from the smaller jobs in the queue.

For each queue, the intra-queue adjustment finds the largest and smallest job, and if the largest job takes more time than the mean, it calculates the allocation size necessary to achieve the mean execution time. The difference is migrated from the smallest job's allocation, as long as minimum resources are left. It repeats this process until all jobs can finish within the mean execution time. In a rare case with a large discrepancy in the job size distribution, the longest job cannot achieve the mean even setting the minimum allocation to the other jobs.

We observe that global scheduling can achieve better performance under the circumstances where the predicted execution time is precise because of better resource utilization and fewer bubbles. Thus, the choice of adaptive or global scheduler will be determined by the accuracy of the performance predictor.

5.2.4 Performance Prediction

The performance predictor predicts the expected execution time of a job. Compute time (excluding the data loading time) of most of the in-memory workload studied before can be deterministically calculated at the compile time. This applies to GEMM and many data-flow applications. On the other hand, the execution time of SpMM is dependent on the adjacency matrix of the subgraph.

Algorithm 4 Intra-queue Adjustment.

```
1: for each $queues$ do
2:     $z = z_0$
3:     for up to $N$ times do
4:         Sort $queue$ based on $t(x, z(x))$
5:         Get $max\_x, max\_t$
6:         Get $min\_x, min\_t$
7:         if difference($max\_t$, $mean\_t$) $> \epsilon$ then
8:             $swap\_cnt = t^{-1}_{max\_x}(mean\_t) - z(max\_x)$
9:             $swap\_cnt = \max(swap\_cnt, 1)$
10:            break if $swap\_cnt == 0$
11:            Migrate $swap\_cnt$ of resources from $min\_x$ to $max\_x$
12:        else
13:            break
14:        end if
15:    end for
16: end for
17: return $queues$
```

This is because the in-memory device also serves as a memory for storing features, and its access patterns are dependent on the input adjacency matrix. Here, each access is followed by a vector MAC operation. While the cycles for MAC operation can be deterministic, we do not know how many MACs are triggered. This requires a complete scan of the input, which is impossible at the compile time, and even at the execution time, performing a full scan of the adjacency matrix for cycle estimation becomes costly.

Job size per allocation unit can be used as a proxy to estimate the execution time for such workloads. For SpMM, the job size within a given allocation and control range can be calculated from the number of non-zero partial rows (prows) of width w in the adjacency matrix. Prows are rows in vertical strips and non-zero prows are such rows with at least one non-zero element. Let $H_w(x)$ be a function that return the number of non-zero prows of a subgraph x of width w, then the average amount of jobs per allocation (translated into w) can be calculated from $nnz(x)/H_w(x)$.

Figure 5.8 shows the memory preference t_{SRAM}/t_{ReRAM} for different jobs using $nnz(x)/H_w(x)$ as the metric. We can see that ReRAM outperforms when the job size per allocation is large, i.e., the access is likely to be localized and there are lots of opportunities to perform the multi-operand

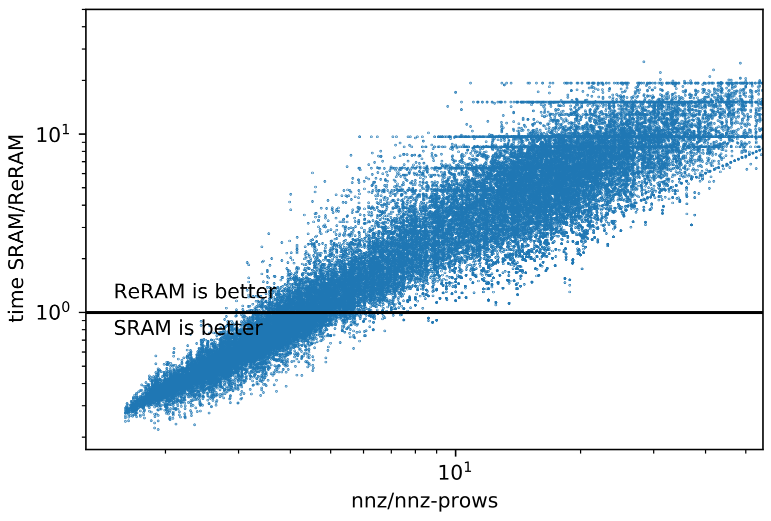


Figure 5.8: t_{SRAM}/t_{ReRAM} vs. $nnz(x)/H_{128}(x)$.

reduction in an array. This trend is reasonable because ReRAM has a larger register capacity per array and can perform a multi-operand dot product operation. While $nnz(x)/H_w(x)$ can roughly classify the memory preference, there are a lot of borderline jobs that are classified incorrectly.

We thereby use MLP based regressor to give a better classification for this non-linear classification task. We use a regressor to generate an estimated time for each memory. A similar approach is adopted in a prior work which used MLP regressor and classifier to make the best selection of matrix permutation for SpMM [117]. Job size and performance can be correlated with a set of subgraph metadata, given the subgraphs are generated from the same *mother* graph, based on the *scale-free* property of the real-world graphs. We use an MLP regressor to learn cycle counts and H_w from the graph metadata. It is trained for each mother graph, taking w, the dimension of a submatrix, and nnz as the input from the training subgraphs. It has two hidden layers with 16 and 8 nodes. While MLP regressors are simple (even compared to [117]), the cycle count predictor can achieve good accuracy (e.g., R^2 score of 0.995 and RMSE of 22% of the mean cycles for ogbl-citation2 in SRAM).

We notice that using more hidden nodes/layers in MLP does not significantly contribute to the performance. Random forest based solutions such as XGBoost [118] regressor can achieve up to

Table 5.1: Dataset details.

Dataset	#Vertex	Input/hidden feature	#Edges	Raw datasize	Min. req. memory
ogbl-collab	235,868	128/256	1,285,465	293M	5GB
ogbl-citation	2,927,963	128/256	30,561,187	3.8G	40GB
ogbl-ppa	576,289	58/256	30,326,273	340M	2GB
ogbl-ddi	4,267	- /256	1,334,889	9.5M	2GB
ogbn-products	2,449,029	100/256	61,859,140	3.4G	33GB

2x better accuracy (RMSE), while requiring significantly more computation and parameter storage cost compared to MLP.

5.3 Methodology

Benchmarks: We use graphs from Open Graph Benchmarks (OGB) [97] for our GNN study and use three Graph Convolutional Kernel (GCN) [92] layers. GNN input features and weights are trained for 16bit fixed-point precision with an additional feed-forward network, which only results in a slight accuracy degradation $< 1\%$. All GNN workloads are built on top of PyTorch framework and PyTorch Geometric (PyG) libraries that are compiled for both CPU and GPU. Subgraphs are generated by PyG's neighbor sampler. We use the autograd profiler of PyTorch and NVIDIA's NVVP and PyProf profilers to generate the execution trace and profiling results on the native machines. We use a batch size of 64. Due to the limitation on the simulation time, we sampled a random 10 batches (640 queries in total) for the simulation.

In addition to the GNN applications, we also use the data-parallel applications in IMP [74]. We compare the kernel execution time of each application. The kernels are compiled for target machine configurations of SRAM and ReRAM. The machine configuration of ReRAM is taken from IMP [74] and that of SRAM is taken from Duality Cache [89]. For both targets, the latency of the compute kernels can be calculated deterministically.

Table 5.2: MLIMP configurations

	Array			SIMD ALUs			MAC throughput / ALU		
	Dimension	# arrays	MB/mm^2	MHz	#ALUs/array	#ALUs	ops/cycle (2ops)	Mops/s (2ops)	Mops/s (4ops)
SRAM	256 x 256	5,120	2.5	2,500	256	1.31 M	0.00331	8.278	2.070
DRAM	8 KB x 8,192	1,024	0.6	300	65,536	67.1 M	0.00066	0.199	0.050
ReRAM	128 x 128 x 2 (bit/cell)	86,016	58.48	20	16	1.37 M	0.12500	2.500	2.500

Performance and Power Models: We develop an event-driven simulator with timing models from IMP [74] for in-ReRAM computing and Duality Cache [89] for in-SRAM computing. We use parameters from Ambit [4] for bit-serial in-DRAM computing. The execution trace from the autograd profiler is replayed in the simulator, and the actual input data is regenerated to perform the timing simulation in each module. Load and store bandwidth for the main memory communication is simulated using Ramulator [87] integrated in our simulator. The data transfer bandwidth between CPU and GPU for baseline GPU execution is recalculated using the actual bandwidth of the PCIe channels measured by CUDA Toolkit to bypass PyTorch's bottlenecks. Predictor latency is measured by a C implementation of the regressor models.

The power parameters for in-memory computing are taken from the prior work [74, 89, 4]. Power and energy for CPU and DRAM activity are measured by profiling microbenchmarks using Intel Rapl interface. We use NVIDIA nvprof to measure GPU power.

5.4 Results

5.4.1 Configurations Studied

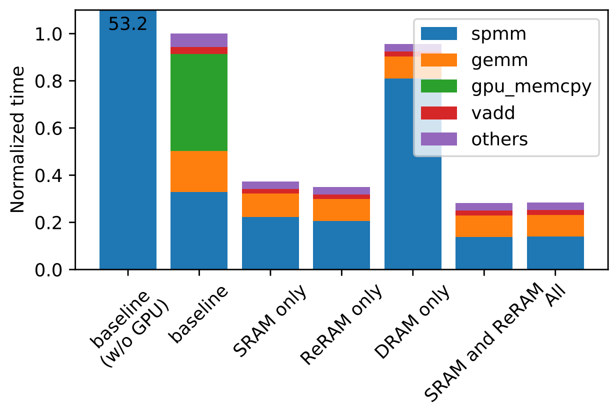
In this section, we evaluate the proposed multi-layer in-memory computing (MLIMP) comparing it to the GPU baseline. Our baseline is composed of a dual-socket Xeon E5-2697 v3 (64GB DDR4) server and NVIDIA Titan XP (12GB GDDR5) GPU. The system configuration for MLIMP in

Table 5.2. We assume 336 MB ReRAM accelerator chip (scaled down from [74]). It has a similar area as the on-chip cache of a dual-socket CPU server. We use half of total SRAMs for in-cache computing allocation because reserving an SRAM portion for general caches is beneficial for both CPU processes and in-cache computing as suggested in [89]. In this configuration, SRAM and ReRAM have a similar number of SIMD ALUs. For in-DRAM computing, we assume DDR4-2400 memory with 4 channels, 1 rank, 16 chips, and 16 banks, supporting bank-level in-memory operations. Each baseline in-memory processor can handle up to 8 outstanding jobs at a time.

5.4.2 GNN Performance

5.4.2.1 Kernel Performance

In this section, we discuss the performance of GNN applications. The execution time breakdown for the three major kernels, i.e. GEMM, SpMM, and vector add (Vadd), is shown in Figure 5.9, assuming different in-memory devices are activated for acceleration. We use the ogbl-citation2 dataset, but a similar trend is observed in most of the real-world graphs that we tested. Compared with CPU, the compute kernels are significantly accelerated by GPU, while GPU execution incurs additional data transfer costs for transferring submatrices and input features to GPU. This data transfer is unavoidable when dealing with large graph data. In-memory computing can bypass the memcpy bottleneck by tight integration with the host memory hierarchy, although memory access time in each kernel sees a slight increase due to narrower DDR4 memory technology. From the different mixture of in-memory computing devices, we can see the SpMM kernel is dominating for all scenarios, while we see the smallest execution time in “SRAM and ReRAM” and “All”. The execution time for SpMM is extracted and compared in Figure 5.10. SRAM and ReRAM result in a similar kernel performance because they have a similar SIMD width and an average MAC throughput per SIMD slot considering the multi-operand operations. In-DRAM SpMM observes worse performance for SpMM due to the smaller SA density (Figure 5.3) and available array-level parallelism. While DRAMs have a large array width, their SIMD slots cannot be fully utilized by GNNs of a small feature vector size.



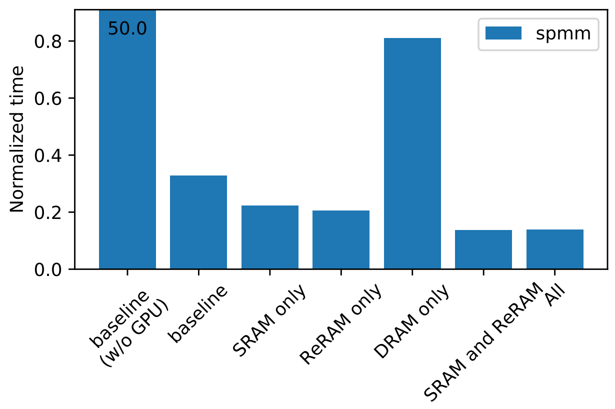


Figure 5.9: Kernel performance (ogbl-citation2). Figure 5.10: SpMM performance (ogbl-citation2).

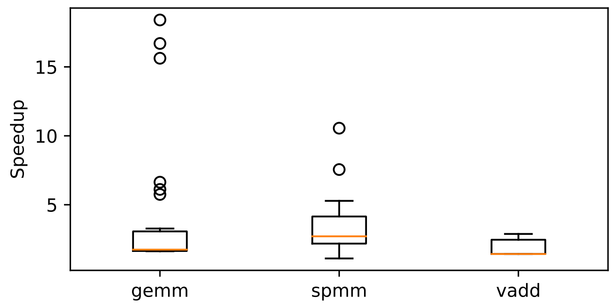


Figure 5.11: Kernel speedups (ogbl-citation2).

As discussed in detail in Section 5.4.2.3, this is 78% of the theoretical best, which is an oracle case with a perfect job balancing across the memories.

The box chart of the distribution of the kernel speedup is shown in Figure 5.11. It assumes the hybrid execution of in-SRAM and in-ReRAM computing. We observe the average speedup of 4.07× for GEMM, 3.40× for SpMM, and 1.82× for Vadd. The massive parallelism of in-memory computing contributes to the speedup of compute-intensive kernels such as GEMM and Vadd. SpMM additionally benefits from internal reuse of input features and input parallel execution.

5.4.2.2 Application Performance

Figure 5.12 shows the application time breakdown for different input graphs, normalized to the baseline GPU execution. While most of the graphs have the power-law distribution, ogbl-ppa and

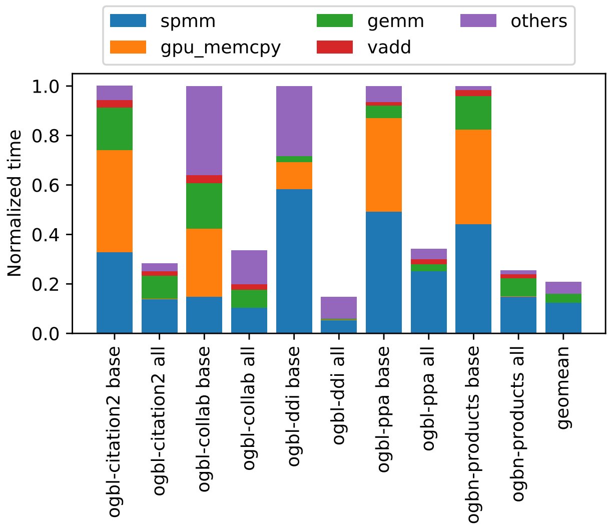


Figure 5.12: Application performance.

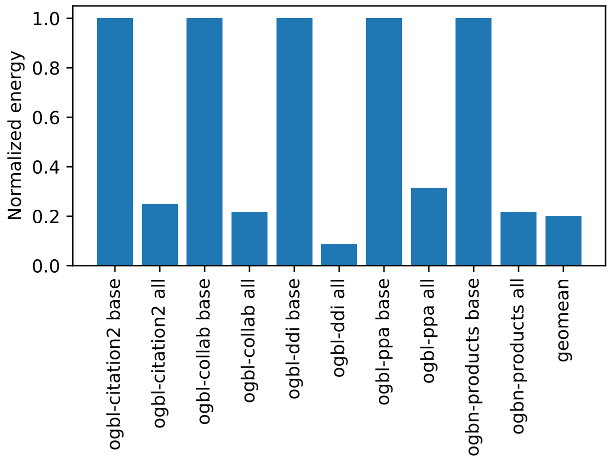


Figure 5.13: Application energy.

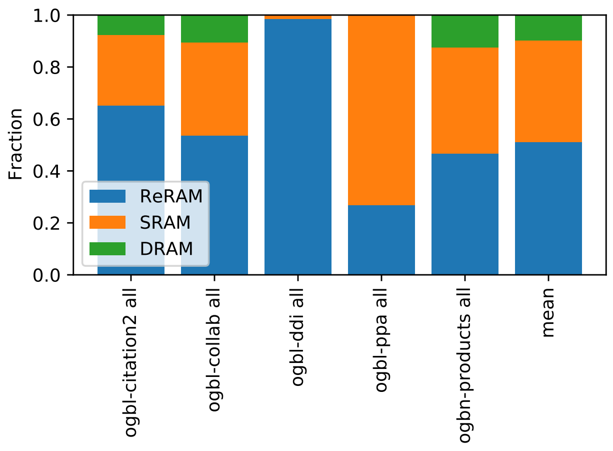


Figure 5.14: Allocation breakdown (execution time weighted).

ogbl-ddi, which represent graphs in nature domains such as biological networks and molecular graphs, tend to have a large average node degree. Thus, the subgraphs are densely connected and there are a lot of overlaps between different subgraphs. In such a case, we do not use the subgraph approach for each input but reuse the same input graph for all input queries in the batch (Section 5.2.2.3).

We observe drastic speedup in the memcpy time and SpMM kernel for most of the input graphs. The speedup of GEMM is moderate compared to other kernels. This is because the majority of GEMM time is spent for data communication to fetch the input features, and we do not benefit

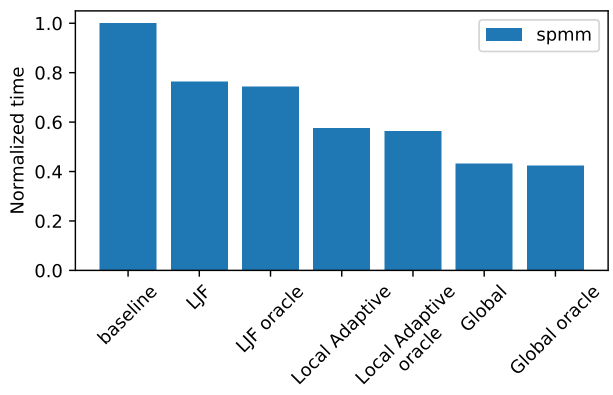


Figure 5.15: Scheduler performance (ogbl-citation2).

from overlapped execution because the compute time is smaller than the data communication time due to the massively parallel execution. Overall we achieve 4.80× geomean speedup for the graphs we evaluated.

Figure 5.13 shows the energy consumption of the GNN applications. Because of the reduced data transfer, which generally takes more energy than computation in the conventional CPU and GPU architecture, we achieve a greater energy benefit from in-memory computing. On average, we achieve 5.02× better energy efficiency for multi-layer in-memory computing.

Figure 5.14 shows the breakdown of the memory allocation in multi-layer in-memory computing using all three memories. It is weighted by the execution time in each allocation. It is observed that our job scheduler can dispatch the jobs to different in-memory devices to maximize efficiency.

5.4.2.3 Scheduler and Predictor Performance

The performance of our job scheduler and performance predictor is illustrated in Figure 5.15. The results are based on the ogbl-citation2 dataset and use different job schedulers presented in Section 5.2.4. We use an oracle predictor, which returns the accurate cycle counts of a job in each memory, and our MLP regressor based predictor. We compare the execution time for SpMM.

We notice that the local adaptive scheduler slightly decreases the execution time compared to the global scheduler. This is mainly because of the bubbles caused by small fragmented resources that were not scheduled to any of the awaiting jobs in the queue. On the other hand, global

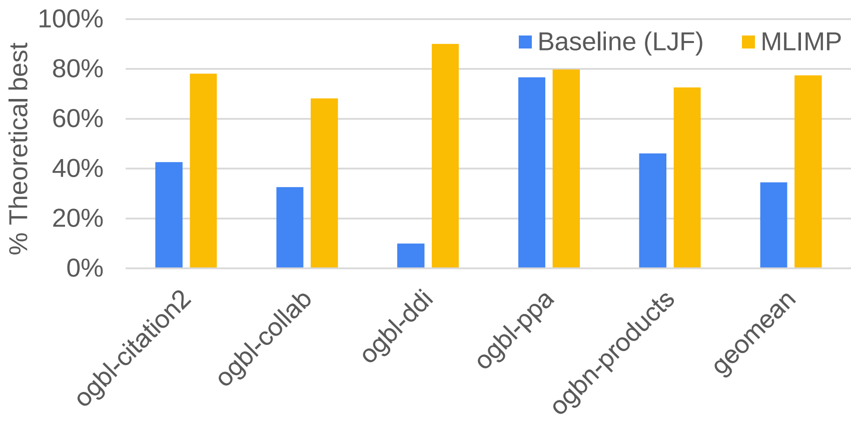


Figure 5.16: Performance compared to theoretical best (perfect job balancing).

scheduling results in the best performance, providing a highly balanced job schedule across different in-memory devices. We also notice our MLP regressor based performance predictor provides reasonably good performance estimates, and the scheduler performance gap between the oracle predictor and ours is trivial (less than 1%). The accuracy of the performance predictor also contributes to the global scheduler outperforming the others.

We conduct a stress test of the schedulers to measure the tolerance to impreciseness of the predictor with an artificial dataset that follows Pareto (scale-free) distribution. We observe the adaptive scheduler results in better performance with added Gaussian noise of $\sigma > 0.39$ on average. In such a case, the global scheduler sees relatively large tail latency for the delayed job items, whereas the adaptive scheduler can automatically adjust by itself which more than amortizes the bubble overhead. The error tolerance of the global scheduler becomes low if batch size is small (threshold $\sigma = 0.25$ for a batch size of 16).

Figure 5.16 compares the performance of our approach with the theoretical best performance, which assumes the perfect job balancing among the memories. The best case performance is calculated by making a sum of the throughput of each in-memory processor. The baseline assumes the same server configuration as MLIMP, but uses naive LJF scheduling to schedule jobs. We observe that the scheduling approach of MLIMP achieves 77% of the best on average, while the naive baseline barely achieves 34%. It is notable that naive scheduling approach is likely to result

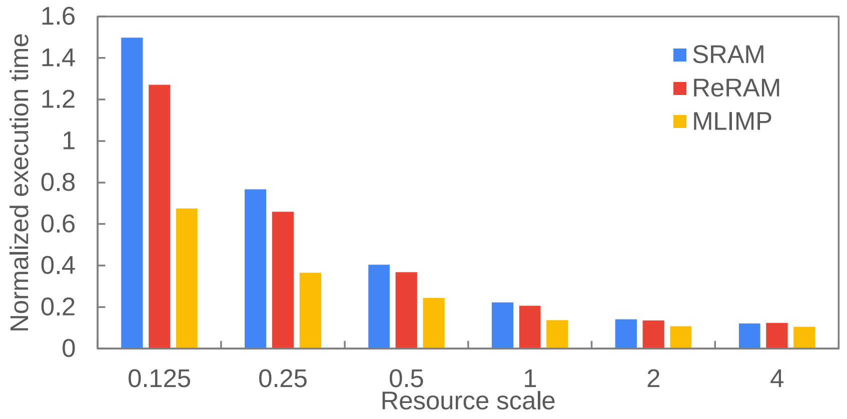


Figure 5.17: Resource scaling.

in the single processor performance of the best in-memory processor, and further performance improvement can only be made by introducing an intelligent job scheduling approach.

5.4.2.4 Resource Scaling

To study the sensitivity of computing resources in multi-layer in-memory computing, we vary the total allocation size for in-memory computing and plot the execution time for the SpMM kernel in Figure 5.17. We notice that the execution time scales well with the allocation size, although the gain diminishes as the total resource becomes large because the bottleneck shifts to data communication with the main memory. We plot the resource scaling for multi-layer in-memory computing as MLILP, using in-SRAM and in-ReRAM computing. This configuration can be compared with the SRAM or ReRAM bars on the right (e.g., ReRAM + SRAM vs. 2xReRAM). It is observed that in many cases multi-layer in-memory computing provides comparable performance to the 2× resource configuration of the better memory (ReRAM in this case).

5.4.3 Multiprogramming

In this section, we evaluate the data parallel applications in multi-layer in-memory computing using several multiprogramming scenarios. We first present the kernel execution time of in-SRAM and in-ReRAM computing in Figure 5.18. Half of the tested applications prefer ReRAM and the

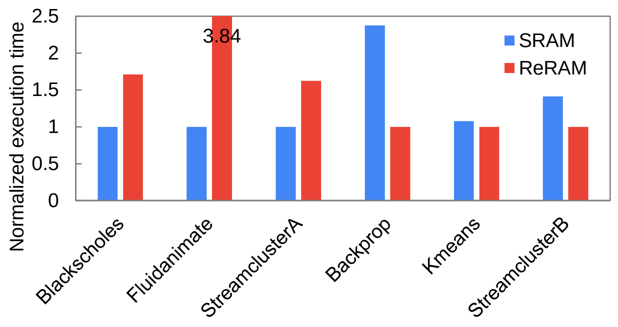


Figure 5.18: Performance of single IMP app execution.

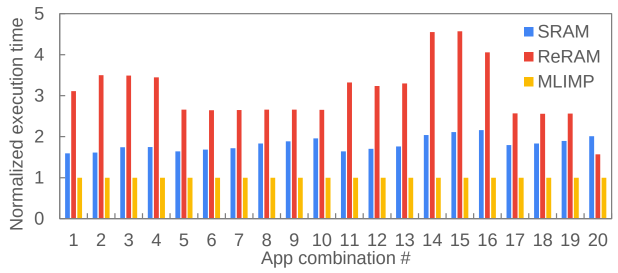


Figure 5.19: Performance of multiple IMP app execution.

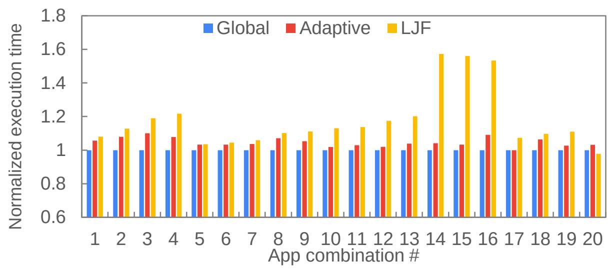


Figure 5.20: Scheduling comparison for IMP apps.

rest SRAM. The preference of in-memory devices depends on many factors as we discussed, while working data set size and instruction mix are some of the dominating factors.

We then assume scenarios of launching multiple programs from the program set. In this experiment, we launch three programs, thus we have ${}_6C_3 = 20$ different scenarios. We compare the execution time in Figure 5.19. The execution time is normalized to the multi-layer in-memory computing configuration. While the preferred memory depends on the type of programs launched, multi-layer in-memory computing can schedule jobs to minimize the latency while balancing the load, resulting in the best performance. Assuming the baseline can pick the best in-memory device for each combinations, we achieve 1.79× better performance. We also compare the performance of different scheduling approaches in Figure 5.20. Because the execution time of the compute kernels here can be calculated deterministically, the global scheduler which can perform both local and global adjustment achieve the best performance for almost all scenarios.

We conclude that a system with multi-layer in-memory computing can benefit from the rich parallel in-memory processing resources and reduced data transfer. Moreover, a good job scheduling and performance estimation allow such a system to exploit the heterogeneous characteristics of different in-memory devices for jobs with dynamisms and substantial variation.

5.5 Summary

We propose multi-layer in-memory computing that runs various computing kernels with workload dynamism in variable layers in the in-memory computing enabled memory hierarchy. By introducing a job scheduler and a performance predictor, GNN inference jobs, which show significant variation in the working dataset and reuse patterns, are well mapped to appropriate memories. Our multi-layer in-memory computing approaches provide performance advantages to multiprogramming scenarios for general data-parallel applications compiled for multiple in-memory device targets. Our experimental results show that multi-layer in-memory computing improves the performance of various GNN inference tasks in OGB by $4.80\times$ over server class GPU. Re-purposing the existing memory hierarchy for multi-layer in-memory computing provides $5.02\times$ better energy efficiency.

CHAPTER 6

Conclusion

The amount of data produced by individuals and corporations is growing explosively. The performance and efficiency of applications dealing with a large amount of data are dependent critically on efficient access and processing of data. In addition, the communication cost of data between storage and processors is enormously high compared to the cost of actual computation that happens in the processors.

In-memory computing directly addresses the inevitable cost of the traditional computation paradigm. However, these architectures are often limited in their application scope or programmability. Our multi-faceted approach enables general data-parallel acceleration in the layers of compute capable memory hierarchy, providing programmable interfaces to utilize the new architecture. We first propose a ReRAM based in-memory accelerator that exploits analog computation of ReRAM arrays. Dense ReRAM array can be used as next generation main memory, and to minimize the hardware cost for data communication, we employ wide-SIMD and VLIW execution using TensorFlow as the programming frontend. Second, we propose an SRAM cache based in-memory accelerator to generalize its bit-line computing for data-parallel programs. We extend SIMT execution model and programming model with VLIW instruction scheduling to balance programmability and hardware complexity. Lastly, we propose a multi-layer in-memory computation stack and a framework that determines the appropriate level of memory hierarchy and method of in-memory computing. Together, these components unlock massive compute capabilities and energy efficiency of in-memory computing for general data-parallel applications.

Today, we have several products and prototypes that demonstrate the advantages of memory-centric computing. UPMEM [119] realizes the idea of near-DRAM computation on a production chip, and HBM-PIM [120] integrates programmable computing units into the DRAM dies near the memory banks. Samsung's SmartSSD computational storage drive [121] combines SSD and Xilinx's FPGA with a fast private data path between them, enabling efficient parallel computation at the SSD. These near-memory computing approaches bring considerable benefits with great adaptability to currently available memory technologies. However, the hidden gems of computing with memory are only discoverable if compute logic and memory are deeply fused as is done in in-memory computing. The savings of near-memory computing increases with an increase in the proximity of the PIM logic to the memory arrays, while there are implementation challenges to make them closer. Likewise, reduced communication overhead of near-memory computing is often traded off by reduced computation throughput due to less performant cores or limited area for custom logic, providing insufficient parallelism and throughput. For these reasons, there will be a higher potential in in-memory computing. While there are a few prototypes and products for in-memory computing, such as Mythic's Flash based analog matrix processor [122], challenges in programming for in-memory computing devices have limited their application targets.

This dissertation studies three key enablers of general data-parallel computing in the heterogeneous system with in-memory computing: enhanced arithmetic operations, parallel programming models with compilers, and parallel execution models. The limited compute capability of in-memory computing can be extended for various arithmetic operations and operation precision by introducing throughput optimized parallel algorithms. The programming model and execution model are designed to fully expose the parallelism in the architecture. In particular, VLIW model combined with SIMD is useful for various in-memory processing because it can effectively exploit TLP and ILP with simplified hardware. The tightly coupled register processor entities of in-memory computing can also flexibly transform themselves for ideal VLIW cores for different optimization targets. Abundant register resources in dense memory arrays can be leveraged for maximizing parallelism and compute bandwidth, while register usage can also be minimized for small SRAM arrays optimizing for latency. A compiler is an important component of the system stack that can expose the compute resource of in-memory computing for a broad range of applications.

Multi-layer in-memory computing is studied for applications with workload dynamism. It is an interesting case study that data-parallel applications and dynamic datasets have different processor preferences and can be intelligently scheduled for in-memory computing combined with general memory hierarchy for better overall performance. Likewise, prior work shows that applications with random memory access chained by small address calculations in between can benefit from multi-layer near-memory computing. We envision future computing systems will increase processor heterogeneity with in- and near-memory computing. It will not just be an additive component to today's computing system, but can also influence and change the traditional microarchitecture design and the system stack. Future work can explore how this transformation of the existing system stack will take place and how we will be able to fully exploit all kinds of heterogeneous resources in a system for various applications without much complexity.

BIBLIOGRAPHY

[1] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396, June 2018.

[2] P. Chi, S. Li, and C. Xu, "PRIME : A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *IEEE International Symposium on Computer Architecture*, pp. 27–39, IEEE, 6 2016.

[3] A. Shafiee, A. Nag, N. Muralimanohar, and R. Balasubramonian, "ISAAC : A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 14–26, 6 2016.

[4] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 273–287, IEEE, 2017.

[5] NVIDIA, "Parallel thread execution isa." `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, 2018.

[6] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, Sept 1959.

[7] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, spring joint computer conference*, pp. 379–385, ACM, 1971.

[8] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 481–492, Feb 2017.

[9] 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 Journal of Solid-State Circuits*, vol. 51, pp. 1009–1021, April 2016.

[10] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family," *J. Solid-State Circuits*, 2013.

[11] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” *Nature*, vol. 453, no. 7191, p. 80, 2008.

[12] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, and W. Lu, “A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications,” *Nano Letters*, vol. 12, no. 1, pp. 389–395, 2011.

[13] 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.

[14] L. Song, X. Qian, H. Li, and Y. Chen, “Pipelayer: A pipelined reram-based accelerator for deep learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, Feb 2017.

[15] M. N. Bojnordi and E. Ipek, “Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning,” *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–13, 2016.

[16] C. Yakopcic and T. M. Taha, “Energy efficient perceptron pattern recognition using segmented memristor crossbar arrays,” in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–8, IEEE, 2013.

[17] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[18] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, “‘memristive’switches enable ‘stateful’logic operations via material implication,” *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[19] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, “Beyond von neumann—logic operations in passive crossbar arrays alongside memory operations,” *Nanotechnology*, vol. 23, no. 30, p. 305205, 2012.

[20] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, “The programmable logic-in-memory (plim) computer,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 427–432, Ieee, 2016.

[21] M. Soeken, S. Shirinzadeh, L. Gaetano, A. Rolf, and G. D. Micheli, “An MIG-based Compiler for Programmable Logic-in-Memory Architectures,” *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, vol. 1, 2016.

[22] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, “ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting,” *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 782–787, mar 2017.

[23] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “Memristor-based material implication (imply) logic: Design principles and methodologies,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.

[24] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “Magic—memristor-aided logic,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[25] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, “In-memory intelligence,” *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[26] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “Scope: A stochastic computing engine for dram-based in-situ accelerator.,” in *MICRO*, pp. 696–709, 2018.

[27] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, *DRAM circuit design: fundamental and high-speed topics*, vol. 13. John Wiley & Sons, 2007.

[28] F. Gao, G. Tziantzioulis, and D. Wentzlaff, “Computedram: In-memory compute using off-the-shelf drams,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 100–113, 2019.

[29] X. Xin, Y. Zhang, and J. Yang, “Roc: Dram-based processing with reduced operation cycles,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[30] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *Micro, IEEE*, 1997.

[31] D. G. Elliott, W. M. Snelgrove, and M. Stumm, “Computational ram: A memory-simd hybrid and its application to dsp,” in *Custom Integrated Circuits Conference*, vol. 30, pp. 1–30, 1992.

[32] M. Oskin, F. T. Chong, T. Sherwood, M. Oskin, F. T. Chong, and T. Sherwood, “Active Pages: A Computation Model for Intelligent Memory,” *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 192–203, 1998.

[33] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, “Programming the flexram parallel intelligent memory system,” *SIGPLAN Not.*, vol. 38, pp. 49–60, June 2003.

[34] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, “A low cost, multithreaded processing-in-memory system,” in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI ’04, (New York, NY, USA), pp. 16–22, ACM, 2004.

[35] H. S. Stone, “A logic-in-memory computer,” *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 73–78, 1970.

[36] W. H. Kautz, “arrays,” *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 719–727, 1969.

[37] C.-C. Yang and S.-S. Yau, “A cutpoint cellular associative memory,” *IEEE Transactions on Electronic Computers*, no. 4, pp. 522–528, 1966.

[38] W. H. Kautz, “A cellular threshold array,” *IEEE Transactions on Electronic Computers*, no. 5, pp. 680–682, 1967.

[39] H. Consortium *et al.*, “Hybrid memory cube specification 2.1,” *Retrieved from hybridmemorycube. org*, 2013.

[40] J. Standard, “High bandwidth memory (hbm) dram,” *JESD235*, 2013.

[41] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “Top-pim: Throughput-oriented programmable processing in memory,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC ’14, 2014.

[42] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[43] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, “A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing,” in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, 2013.

[44] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46.

[45] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads,” in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[46] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA ’15, 2015.

[47] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory,” in *Proceedings of ISCA*, vol. 43, 2016.

[48] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.

[49] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *2012 symposium on VLSI technology (VLSIT)*, pp. 87–88, IEEE, 2012.

[50] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 41–54, IEEE, 2017.

[51] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, J. Shlens, B. Steiner, I. Sutskever, P. Tucker, V. Vanhoucke, V. Vasudevan, O. Vinyals, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2015.

[52] 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," *Nanotechnology*, vol. 20, no. 42, p. 425204, 2009.

[53] M. Hu, R. S. Williams, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, and J. J. Yang, "Dot-product engine for neuromorphic computing," in *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, (New York, New York, USA), pp. 1–6, ACM Press, 2016.

[54] J. Sandrini, M. Barlas, M. Thammasack, T. Demirci, M. De Marchi, D. Sacchetto, P.-E. Gaillardon, G. De Micheli, and Y. Leblebici, "Co-Design of ReRAM Passive Crossbar Arrays Integrated in 180 nm CMOS Technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, pp. 339–351, 9 2016.

[55] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Muraoka, S. Mitani, S. Fujii, K. Katayama, M. Iijima, T. Mikawa, T. Ninomiya, R. Miyanaga, Y. Kawashima, K. Tsuji, A. Himeno, T. Okada, R. Azuma, K. Shimakawa, H. Sugaya, T. Takagi, R. Yasuhara, K. Horiba, H. Kumigashira, and M. Oshima, "Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism," in *2008 IEEE International Electron Devices Meeting*, pp. 1–4, IEEE, 12 2008.

[56] M. Ercegovac, J.-M. Muller, and A. Tisserand, "Simple Seed Architectures for Reciprocal and Square Root Reciprocal,"

[57] M. Cornea, J. Harrison, C. Iordache, B. Norin, and S. Story, "Divide, square root, and remainder algorithms for the ia-64 architecture," *Open Source for Numerics, Intel Corporation*, 2000.

[58] J. Harrison, T. Kubaska, S. Story, *et al.*, "The computation of transcendental functions on the ia-64 architecture," in *Intel Technology Journal*, Citeseer, 1999.

[59] J. R. Ellis, *Bulldog: A Compiler for VLSI Architectures.* Cambridge, MA, USA: MIT Press, 1986.

[60] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The multiflow trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, May 1993.

[61] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 141–150, ACM, 2006.

[62] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, no. January, pp. 72–81, 2008.

[63] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[64] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge, "Scaling towards kilo-core processors with asymmetric high-radix topologies," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 496–507, IEEE, 2013.

[65] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, "A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 468–469, IEEE, 2 2013.

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

[67] P.-Y. Chen, D. Kadetotad, Z. Xu, A. Mohanty, B. Lin, J. Ye, S. Vrudhula, J.-s. Seo, Y. Cao, and S. Yu, "Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 854–859, IEEE, 2015.

[68] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 86–96, IEEE, 2013.

[69] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das, "Re-nuca: A practical nuca architecture for reram based last-level caches," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 576–585, May 2016.

[70] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010.

[71] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct 2009.

[72] C. Xu, "Part III : Emerging Nonvolatile Memories Emerging Non-volatile Memory," 2014.

[73] Intel, "x87 and sse floating point assists in ia-32: Flush-to-zero (ftz) and denormals-are-zero (daz)." https://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-deno 2008.

[74] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 1–14, ACM, 2018.

[75] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 191–200, ACM, 1998.

[76] MathWorks, "Compute square root using cordic." https://www.mathworks.com/help/fixedpoint/examples/compute-square-root-using-cordic.html, 2018.

[77] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 353–364, ACM, 2010.

[78] R. compiler infrastructure, "Rose compiler." http://rosecompiler.org/, 2018.

[79] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 516–523, IEEE, nov 2013.

[80] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 15–24, ACM, 2012.

[81] PathScale, "Performance test suite for openacc compiler, intel mic, patus and single-core cpu." https://github.com/pathscale/OpenACC-benchmarks, 2013.

[82] A. Tabuchi, M. Nakao, and M. Sato, "A Source-to-Source OpenACC Compiler for CUDA," in *Euro-Par 2013: Parallel Processing Workshops* (D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, eds.), (Berlin, Heidelberg), pp. 178–187, Springer Berlin Heidelberg, 2014.

[83] NVIDIA, "Cuda toolkit." https://developer.nvidia.com/cuda-toolkit, 2018.

[84] NVIDIA, "Pgi compilers & tools." https://www.pgroup.com/, 2018.

[85] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge, "Scaling towards kilo-core processors with asymmetric high-radix topologies," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 496–507, Feb 2013.

[86] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.

[87] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator.," *Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[88] Intel, "Intel processor graphics." https://software.intel.com/en-us/articles/intel-graphics-developers-guides, 2018.

[89] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 397–410, ACM, 2019.

[90] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[91] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[92] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[93] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in Neural Information Processing Systems*, vol. 31, pp. 5165–5175, 2018.

[94] M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin, "Revisiting graph neural networks for link prediction," *arXiv preprint arXiv:2010.16103*, 2020.

[95] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

[96] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

[97] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[98] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 100–113, 2019.

[99] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," tech. rep., Technical Report No. UCB/EECS-2009-131, 2009.

[100] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache It Too," in *ISCA'15*, pp. 707–719, 2015.

[101] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 565–581, USENIX Association, Aug. 2016.

[102] B. Y. Cho, J. Jung, and M. Erez, "Accelerating bandwidth-bound deep learning inference with main-memory accelerators," *CoRR*, vol. abs/2012.00158, 2020.

[103] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near data acceleration with concurrent host access," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 818–831, 2020.

[104] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, IEEE, 2018.

[105] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17, 2019.

[106] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, U. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, (New York, NY, USA), p. 66–79, Association for Computing Machinery, 2018.

[107] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pp. 457–468, IEEE, 2017.

[108] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in

2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 544–557, IEEE, 2018.

[109] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Operating systems: Three easy pieces, chapter 5," *Arpaci-Dusseau Books*, vol. 1.

[110] Arm Ltd., "Arm big.LITTLE."

[111] R. Kolisch and S. Hartmann, *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pp. 147–178. Boston, MA: Springer US, 1999.

[112] L. Özdamar and G. Ulusoy, "A survey on the resource-constrained project scheduling problem," *IIE transactions*, vol. 27, no. 5, pp. 574–586, 1995.

[113] R. Kolisch, "Efficient priority rules for the resource-constrained project scheduling problem," *Journal of Operations Management*, vol. 14, no. 3, pp. 179–192, 1996.

[114] K. S. Naphade, S. D. Wu, and R. H. Storer, "Problem space search algorithms for resource-constrained project scheduling," *Annals of operations research*, vol. 70, pp. 307–326, 1997.

[115] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954.

[116] B. C. Arnold, "Pareto distribution," *Wiley StatsRef: Statistics Reference Online*, pp. 1–10, 2014.

[117] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations for efficient spmm on gpu architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 48–58, IEEE, 2021.

[118] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

[119] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24, IEEE Computer Society, 2019.

[120] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, *et al.*, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 350–352, IEEE, 2021.

[121] "Samsung smartssd computational storage drive." https://web.archive.org/web/20210118224545/https://samsungsemiconductor-us.com/smartssd/index.html. Accessed: 2021-03-01.

[122] M. Demler, “Mythic multiplies in a flash,” *Analog In-Memory Computing Eliminates DRAM Read/Write Cylcles, The Linley Group Microprocessor report*, 2018.