The Dissertation Committee for Benjamin Youngjae Cho
certifies that this is the approved version of the following dissertation:

# Main-Memory Near-Data Acceleration with Concurrent Host Access

Committee:

Mattan Erez, Supervisor

Michael Orshansky

Andreas Gerstlauer

Constantine Caramanis

Jonathan Beard

# Main-Memory Near-Data Acceleration with Concurrent Host Access

**by**

**Benjamin Youngjae Cho,**

**DISSERTATION**

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN
May 2021

Dedicated to my beloved wife Suyeon Au, son Ryan Cho,

my parents Hanjin Cho and Jinwook Shin.

# Acknowledgments

First and foremost, I would like to thank my advisor Mattan Erez. Without his warm advice and guidance, I would have not accomplished this long journey to a Ph.D. When I just joined UT, I recall talking to him about my goal throughout my Ph.D. As a passionate (naive) graduate student, I told him that I want to build a whole new memory system that no one has thought about. It probably did not happen but I remember his smile and encouraging me that I can do it and he will help me. It was the moment that I felt that I am not alone in this journey and I can achieve the goal if with him. Honestly, we personally do not have much in common: I use a Macbook and he uses a PC, I use vim and he uses emacs, I like flavored coffee and he doesn't, I like pizzas with barbecue sauce and he hates it, and so on. However, I like the way he writes paper, the way he makes slides, the way he gives presentation, the way he teaches, the way he directly tackles the problem instead of avoiding it, the way he stays calm and quickly strikes out all the todos several hours before the paper deadline, and the way he provides practical advices and helps based on my current situation. As a professor, researcher, and advisor, he is always going to be my wannabe. I am lucky to have him as my advisor and I would like to thank him once again for being patient, inspiring, and supportive all the time.

I thank my dissertation committee members, Jonathan Beard, Constantine Caramanis, Andreas Gerstlauer, and Michale Orshansky, for providing valuable comments to improve my dissertation. I also would like to thank my intern managers and mentors, Amin Farmahini-Farahani (AMD Research), Nuwan Jayasena (AMD Research), Eiman Ebrahimi (Nvidia Research), David Nellans (Nvidia Research), Jiangli Zhu (Micron 3DXP), and Lavanya Subramanian (Facebook AR/VR), for the opportunity to collaborate on interesting projects with them. I learned how to work as a team and how to contribute to the team as a member. I also want to thank John Kim and Minsoo Rhu for encouraging me when I was having a hard time because of paper rejection.

I would like to thank all the former and current LPH members that I had a chance to work with: Michael Sullivan, Ikhwan Lee, Nick Kelly, Jungrae Kim, Dong Wan Kim, Seong-Lyong Gong, Cagri Eryilmaz, Jinsuk Chung, Esha Choukse, Chun-Kai Chang, Majid Jalili, Yongkee Kwon, Sankug Lym, Song Zhang, Haishan Zhu, Tianhao Zheng, Wenqi Yin, Anyesha Ghosh, and Jeageun Jung. I will not forget the moment that we had lunch and coffee together in the group meeting every Friday, spent all of ours times together for numerous paper and project deadlines, and brain-stormed on my idea when I was struggling with some tricky problems. It was my privilege to meet these great people and work together.

Last but not least, I would like to express my deepest gratitude to my family. I thank my wife Suyeon for her strong and warm support for any decisions I made to build my career path, even though she had to quit her

job at Korea and come to the U.S. with me, even though she had to move to different states every summer for my internship, and even though she had to take care of our son by herself because of all kinds of deadlines that I had to meet. I appreciate all those time she had sacrificed for me. I also thank my son Ryan for coming to us as our son after a long time of waiting. His hug always filled me with positive energy and motivations to work harder. I also thank my parents and in-laws for their financial and moral supports. Though they had to be separated from their son, daughter, and grandson, they always encouraged me to achieve my goals and gave full support. I especially thank my dad for always providing great advices whenever I met problems, sharing his experience as a Ph.D. student when he was in my age, and deeply understanding all my concerns.

Benjamin Youngjae Cho

November 2020, Austin, TX

# Main-Memory Near-Data Acceleration with Concurrent Host Access

Publication No. _____

Benjamin Youngjae Cho, Ph.D.
The University of Texas at Austin, 2021

Supervisor: Mattan Erez

Processing-in-memory is attractive for applications that exhibit low temporal locality and low arithmetic intensity. By bringing computation close to data, PIMs utilize proximity to overcome the bandwidth bottleneck of a main memory bus. Unlike discrete accelerators, such as GPUs, PIMs can potentially accelerate within main memory so that the overhead for loading data from main memory to processor/accelerator memories can be saved. There are a set of challenges for realizing processing in the main memory of conventional CPUs, including: (1) mitigating contention/interference between the CPU and PIM as both access the same shared memory devices, and (2) sharing the same address space between the CPU and PIM for efficient in-place acceleration. In this dissertation, I present solutions to these challenges that achieve high PIM performance without significantly affecting CPU performance (up to 2.4%

degradation). Another major contribution is that I identify killer applications that cannot be effectively accelerated with discrete accelerators. I introduce two compelling use cases in the AI domain for the main-memory accelerators where the unique advantage of a PIM over other acceleration schemes can be leveraged.

**Thesis Statement:** *Processing-in-memory is increasingly important because it can process rapidly-growing real-life data with high performance and efficiency. However, with existing approaches, the host CPU and PIM units (PIMs) cannot share and concurrently access the same memory with high performance. My research, on the other hand, details that how the host CPU and PIMs can efficiently share the same memory and both achieve high performance, even when they collaboratively and concurrently process the same data.*

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Processing in memory is attractive for applications that exhibit low temporal locality and low arithmetic intensity. By bringing computation close to data, PIMs utilize proximity to overcome the bandwidth bottleneck of a main memory bus. Despite decades of research, many challenges remain unresolved, and hinders deploying PIMs in conventional systems. PIM-enabled memory devices (PIMs) are not only accelerators but also a part of the main memory of the CPU. In fact, these two different roles of PIMs give both challenges and opportunities. The main challenge is how to effectively manage the inference and contention between the CPU and PIMs when both try to access the same memory at the same time. In an ideal case, by managing the interference and contention properly, both the low memory latency requirement of the CPU and the high memory bandwidth requirement of the PIMs are satisfied together. The main opportunity comes from the fact that the CPU and PIMs share the same memory and both can collaborate on processing the same data with only one instance of the data. This gives performance and capacity advantages over discrete accelerators because no bulk data copy/loading is required prior to the PIM acceleration. This advantage has not been carefully examined by prior work. As a result, it is still unclear what the compelling use

cases for PIMs are, especially those where other types of accelerators cannot replace PIMs. In summary, the main questions that my dissertation answers are as follows: (1) how can a CPU and PIMs share the same memory and both achieve high performance? (2) what are the unique advantages of PIMs over other discrete accelerators and what are the compelling use cases that can maximally exploit those advantages?

## 1.1 High-performance CPU-PIM Concurrent Access

In the first part of my dissertation, I focus on enabling high-performance concurrent access between the CPU and PIMs to the same memory devices, including to shared data. In conventional systems, PIM-enabled memory devices can be accessed not only as a main memory but also as an accelerator. However, applications that use such devices for different purposes simultaneously exhibit different performance requirements: low latency for CPU main memory access and high bandwidth for acceleration. When the CPU and PIMs concurrently access the same memory, meeting those two requirements at the same time requires new mechanisms. Prior work solves this memory sharing problem with coarse-grained (CG) temporal and spatial partitioning of memory between the CPU and PIMs. However, CG temporal partitioning requires the CPU to wait too long if accessing the same device while PIMs access memory, which significantly degrades CPU performance. CG spatial partitioning cannot enable concurrent access to the same data, reserving a large fraction of memory capacity for PIMs. To address the above challenges, I observe that

two capabilities must be supported at the same time: (1) fine-grained access interleaving to fully utilize internal DRAM bandwidth, and (2) coarse-grained PIM operations, where each PIM operation processes many data elements (vectors/matrices), to mitigate contention for channel command bandwidth for sending both CPU memory requests and PIM command packets. I identify and solve new problems to overcome these two challenges: reduced performance due to unnecessary bank conflicts, penalties for interleaving read and write transactions, defining a common data layout, and synchronizing the state of memory controllers. I develop mechanisms that address these challenges and enable the CPU and PIMs to concurrently access memory with low overhead, whether they access the same data or not.

## 1.2 Use Cases for Main-Memory Accelerators

In the second part of my dissertation, I present two compelling use cases for main-memory accelerators in the AI domain where the unique advantage of a PIM over other acceleration schemes are leveraged. Unlike other discrete accelerators, such as GPUs, PIMs already share the same memory device with the CPU. Therefore, the CPU and PIMs can potentially share one instance of large data and individually, or sometimes, collaboratively execute computation. On the other hand, to use discrete accelerators, the input data must be first loaded from main memory to device memories, which incurs performance overhead, and at least two instances of the same data are needed in the system, which incurs a capacity overhead. I focus on this advantage of PIMs and

introduce two different use-case scenarios where PIMs cannot be replaced by other discrete accelerators.

***Case Study 1: CPU-PIM Concurrent Collaboration for Machine Learning Training*** In the first case study, I demonstrate the potential benefit of CPU-PIM collaboration with a machine learning application– logistic regression with stochastic variance reduced gradient (SVRG). In this collaboration scheme, the CPU executes the main training loop by maximally exploiting locality captured by the large CPU caches while PIMs access the entire training data with high memory bandwidth and provide the correction term that helps the main training converge faster to the optimal solution.

**Case Study 2: Accelerating Machine Learning Inference in Datacenter Servers** In the second case study, PIMs are used to accelerate bandwidth-bound deep learning inference tasks within a warehouse-scale server. In recent language and recommendation models, the execution time for fully-connected (FC) layers dominate the overall inference latency. Evaluating FC layers requires a matrix-matrix multiplication (GEMM). Inference queries have tight latency constraints and cannot form a large batch. As a result, the GEMM operations in FC layers are bandwidth bound, which are good target tasks for PIMs. On the other hand, some inference queries that have somewhat relaxed latency goals and can be grouped into large batches and can execute well on the CPU and GPU. In warehouse-scale servers, these requests with different latency goals are colocated and processed together. My

work enables the CPU and PIMs to each concurrently process batches with different sizes that best fit the performance characteristics of each processor by sharing the large weight matrices of FC layers.

## 1.3 Thesis Statement

*Processing-in-memory is increasingly important because it can process rapidly-growing real-life data with high performance and efficiency. However, with existing approaches, the host CPU and PIM units (PIMs) cannot share and concurrently access the same memory with high performance. My research, on the other hand, details that how the host CPU and PIMs can efficiently share the same memory and both achieve high performance, even when they collaboratively and concurrently process the same data.*

## 1.4 Contributions

- I solve the following new challenges in concurrent access to memory from the CPU and PIMs: bank conflicts from CPU accesses curb PIM performance, read/write-turnaround penalties from PIM writes lower CPU performance, and contention on the memory command bandwidth between CPU memory accesses and PIM packet launches either underutilizes PIMs or degrade CPU performance. I reduce bank conflicts with a new bank partitioning architecture that, for the first time, is compatible with both huge pages and any XOR-based sophisticated memory interleaving, which are modern memory management and address mapping

mechanisms. To decrease read/write-turnaround overheads, I throttle PIM writes with two mechanisms: next-rank prediction that delays PIM writes to the rank actively read by the CPU and stochastic issue that throttles PIM writes at a configurable rate.

- I show the potential of collaboratively processing the same data between the CPU and PIMs with a case study. I introduce an important ML algorithm that leverages the fast CPU for its main training loop and the high-BW PIMs for summarization steps that touch the entire dataset. I develop a variant that executes on the PIMs and the CPU in parallel, which provides speedup of 2x.

- I propose StepStone PIM, which enables independent GEMM execution with PIM under complex CPU DRAM address mapping using: (1) address-mapping cognizant GEMM blocking and (2) PIM-side address generation (AGEN) that matches this blocking. My unique AGEN logic improves throughput by, up to 8× and 6.4× on average, compared to naive or CPU-side address generation. I also identify the tradeoffs of PIM designs in three different DRAM hierarchy levels (channel, chip, and bank-group levels) and evaluate their performance with detailed simulation. I show that activating more PIMs for GEMM improves the arithmetic performance but adds overheads for data localization/replication and reduction.

## 1.5 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides background on DRAM devices and processing in memory; Chapter 3 discusses the limitation of existing PIM approaches for enabling PIM in the main memory of conventional CPUs; Chapter 4 presents proposed mechanisms to enable concurrent host access and one case study to leverage concurrent access in machine learning training; Chapter 5 presents challenges and solutions to use PIMs to accelerate bandwidth-bound deep-learning inference tasks. In Chapter 6, I provide a list of limitations/future work to completely enable processing-in-memory in main memory systems.

# Chapter 2

# Background

This chapter provides background knowledge about conventional DRAM devices and the main memory systems of CPUs. I also explain the basic concept of PIM and its advantage over processing with the CPU.

## 2.1 DRAM Devices and Operation

DRAM devices can be categorized into two categories based on their interface types: classical and packetized DRAMs. I first explain how processors interface with these devices and qualitatively analyze their advantages and disadvantages.

**Classical DRAM Devices,** such as DDR3 and DDR4, are attached to a printed circuit board (PCB) and form a dual-inline memory module (DIMM). DIMMs are installed to the DIMM slots on a motherboard and connected to the CPU. Multiple DIMMs can be attached to a single memory channel and only one DIMM can be accessed through the channel. All the DIMMs sharing the same memory channel can be accessed within a same fixed latency regardless of the distance to the CPU. This DIMM-type memory devices increases memory capacity by: (1) mounting more chips to the PCB board, and (2)

installing more DIMMs. However, memory bandwidth does not scale with the number of DRAM chips in the system. Bandwidth is determined only by the number of pins per channel and the data rate of those pins.

Classical DRAM devices are directly controlled by the host CPU's memory controller with low-level DRAM commands. Each memory controller manages the DRAMs that are connected through the connected memory channel. The low-level DRAM commands are issued by the memory controller based on bank and timing states of DRAMs. To access data in DRAM banks, the target row should be first activated with *ACTIVATE* command. Once the row is activated, or opened, data can be read from and written to the row with *READ* and *WRITE* commands, respectively. If another row is already open, the bank must first be closed with the *PRECHARGE* command. Then, the target row can be activated with an ACTIVATE command. To support DRAM operations, memory controllers track bank state and determine which command to issue next. Another important role of the memory controller for classical DRAM devices is that it maintains timing parameters defined by the DRAM specification. The timing parameters include the time intervals between two DRAM commands, such as row-to-row delay (*tRRD*) and column-to-column delay (*tCCD*), or the time window within which no more that $n$ DRAM commands can be issued, such as four activation window (*tFAW*) with $n = 4$. Unless these timing parameters are obeyed, normal DRAM operations are not guaranteed. The timing state also determines when to issue the next command so that the timing rules are not violated.

**Packetized DRAM Devices,** such as hybrid memory cube (HMC), are connected to the CPU memory controller in a point-to-point manner to achieve high link bandwidth. For the convenience of explanation, I use HMC as a representative example of packetized DRAM devices. HMC is composed of one logic die with multiple DRAM dice stacked on top of it. The logic die and DRAM dice are connected with through-silicon vias (TSVs) and those vertically connected banks form a *vault*. On the logic die, *vault controllers* manage the timing and bank state of each vault, as the memory controllers of classical DRAM devices do for each memory channel.

There are two ways of scaling capacity with packetized DRAM devices: (1) stacking more dice, and (2) daisy-chaining the memory stacks. The number of DRAM dice that can be stacked is limited by thermal and power constraints. On the other hand, though daisy-chaining does not have those physical constraints, the average memory latency increases with the number of stacks connected serially.

To access data from the CPU memory controller, command packets are assembled and passed to the directly-connected DRAM stack. The command packets contain information about target address and transaction type (e.g., read or write). If the target address of a certain packet does not belong to the current DRAM stack, then it will be forwarded to other stacks based on some routing policy. Once it reaches the target vault of the target stack, the packet will be transformed into low-level DRAM commands and issued by the corresponding vault controller.

## 2.2 Baseline DRAM Organization

Figure 2.1 illustrates the organization of the baseline main memory system. I focus on the classical DRAM interface due to its high memory capacity and low memory latency, which are the required features for CPU main memory. Memory controllers access data through a *memory channel*, independently from other memory channels. Multiple DIMMs are connected to each memory channel and each DIMM has multiple DRAM devices (or DRAM chips) mounted. The chips on a DIMM operate together by sharing the same address and command bus. This group of chips is called a *rank*. Each chip is composed of multiple *banks*. The banks can operate in parallel but only one bank at a time can be accessed through the internal DRAM bus. Lastly, each bank has multiple rows and columns (which is memory transaction unit). Since the memory controller can access one column at a time, all the DIMMs, ranks, banks, rows, and columns are accessed through the bandwidth reduction point, introduced in Section 2.3, and can be the memory units in Figure 2.2.

## 2.3 Processing in Memory

Figure 2.2a shows the typical way of scaling memory capacity in conventional CPU systems. Memory units (*Mem*s in the figure) can be any of the following: DIMMs, ranks, banks, rows, and columns in our baseline DRAM organization (Figure 2.1). Multiple memory units are connected to a multiplexer and the output of the multiplexer is connected to the CPU. I refer to

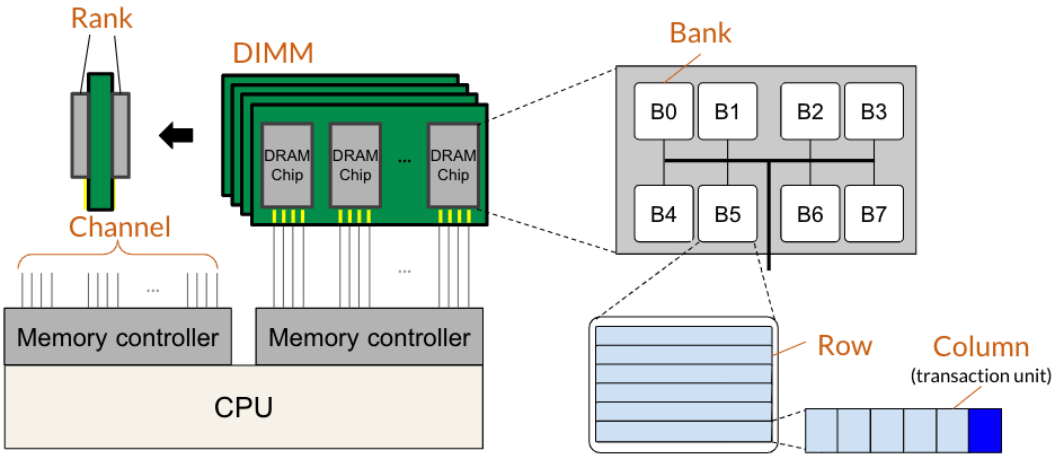
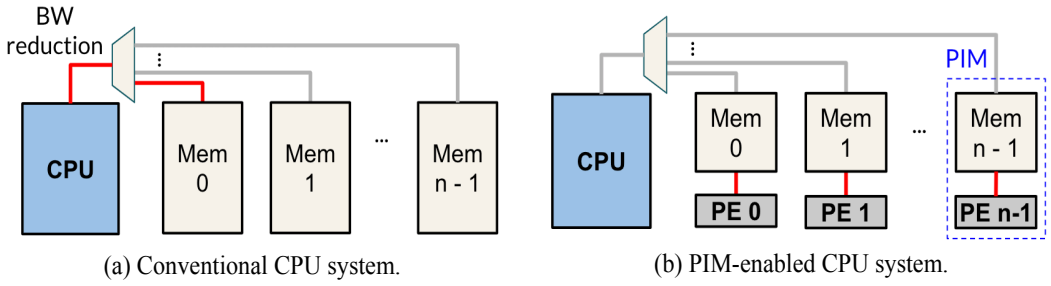



Figure 2.1: Organization of baseline main memory system.

this multiplexer as a *bandwidth reduction point*. There are multiple bandwidth reduction points in the DRAM hierarchy and they are discussed in Section 2.2. The CPU can only access one memory unit at a time and this is mainly because of the pin count limit of the CPU. That is, adding pins to the CPU package is physically limited and, therefore, not all the memory units in the system can be directly connected to the CPU through those pins.

On the other hand, PIM brings computation close to data and this can be done by adding processing elements (PEs) near the memory units. In this way, potentially all the memory units can be accessed by locally located PEs and the aggregated bandwidth can scale with the number of memory units in the system. In addition, since the distance between computation and data becomes shorter, PIMs can benefit from low-energy memory access.




(a) Conventional CPU system. (b) PIM-enabled CPU system.

Figure 2.2: Bandwidth advantage of processing in memory.

## 2.4 Address Translation and Physical-to-DRAM Address Mapping

In conventional CPU systems, user programs access data in the virtual address space, which is partitioned into *pages*. A typical minimum page size is 4 KiB but huge pages of 2 MiB and 1 GiB are supported and frequently used. Each page in the virtual address space is mapped to a physical *frame*. The size of physical address space is equal to the total physical memory capacity. This mapping between virtual pages and physical frames is done by the operating system (OS) at memory allocation time. When the CPU accesses memory with a virtual address, the CPU walks through a page table structure to translate the virtual address into its corresponding physical address. The page offset field is translated to the frame offset field "as is" while the page ID is translated into the frame ID based on the OS-managed page table. The frame ID is set by the OS while the frame offset is not under system control. After address translation, the physical address is passed to the memory controller and is used to access data in memory (or caches).

The memory controller maps a physical address into a DRAM address. A DRAM address is composed of indices to the DRAM organization units: such as channel, rank, bank, row, and column. There are three things to consider for address mapping to minimize memory latency: (1) bank-level parallelism should be maximally exploited to have more outstanding memory requests processed in parallel, (2) locality captured by *row buffer*, which stores the data of a currently accessing row, should be exploited to decrease per-access memory latency, and (3) timing parameters should be considered to decide the interleaving granularity for each DRAM hierarchy level. Considerations (1) and (2) exhibit conflicting requirements, therefore, the interleaving granularity should be optimized for general access patterns. For instance, if banks are interleaved with fine granularity, bank-level parallelism can be maximized yet row-buffer locality cannot be easily exploited. On the other hand, if the interleaving granularity is an entire row, then row-buffer locality can be maximized but bank-level parallelism is lower. For Consideration (3), the timing parameters, such as tRTR and tCCDL, should be considered. The first parameter, tRTR, is the penalty for accessing one rank from another rank back to back. Therefore, to avoid being penalized by this timing parameter, interleaving across ranks should be with coarse granularity. The second parameter, tCCDL, is the long column to column delay for accessing different banks in the same *bank group*. Bank groups are a relatively recent addition to DRAM architectures and trade off density and bus frequency with additional timing constraints. Banks within the same group share some internal buses and thus

require an additional latency penalty when trying to access different banks in the same group vs. different banks across bank groups. To avoid this long delay, physical addresses should be interleaved across bank groups with fine granularity but also interleaved across banks in the same bank group in coarse granularity to maintain high row-buffer locality.

# Chapter 3

# Limitations of Existing PIM Approaches for Main-Memory Acceleration

In this chapter, I describe existing approaches for processing in the main memory of CPUs. Then, I explain their limitations by focusing on the requirements for main memory and accelerators.

## 3.1 Direct Host Control

The first approach for processing in main memory is through direct host control over PIMs [10, 80]. In this approach, the host CPU is responsible for issuing DRAM commands for both the CPU and PIMs. For PIM memory transactions, a new pair of read and write commands are required. Once PIM read commands are issued by a CPU memory controller, the data read from DRAM cells is forwarded to the local PIM unit. For PIM write commands, the internal data path selects the data generated by PEs and write them to the target DRAM cells. In this way, all the commands are issued by the host-side memory controller but the actual data processing is done by the PIMs.

The performance benefits of this approach originate from the fact that (1) CPU caches are not polluted by processing the data with low temporal

locality with PIMs and (2) write bandwidth can be saved for some reduction operations. Note that read bandwidth advantage with this approach is not significant as all the PIM requests are sent by the CPU and read bandwidth is bound by the command bandwidth of each channel. In addition, this command-level PIM control enables CPU-PIM *interoperatility* because the granularity for PIM and CPU memory accesses is matched. This simplifies the required OS and hardware supports for coherence management, virtual-to-physical address translation, and physical-to-DRAM address mapping. However, with this approach, PIM memory bandwidth cannot scale with the number of PIM-enabled memory devices, which is the expected benefit of PIMs. This is because memory bandwidth will be eventually bottlenecked by the command bandwidth available to the memory controllers. The command bandwidth bottleneck will result in low PIM utilization and the bandwidth advantage of PIMs over the CPU cannot be fully exploited with this direct host control approach.

## 3.2 Coarse-Grain Spatial Partitioning (PIM as Another Discrete Accelerator)

The second approach for processing in main memory is with coarse-grain spatial partitioning. Many existing PIM approaches follows this approach and, basically, the CPU and PIMs neither share the same memory devices nor address space. As a result, to enable CPU-PIM collaboration on the same data or to process the data generated by the CPU with PIMs, two

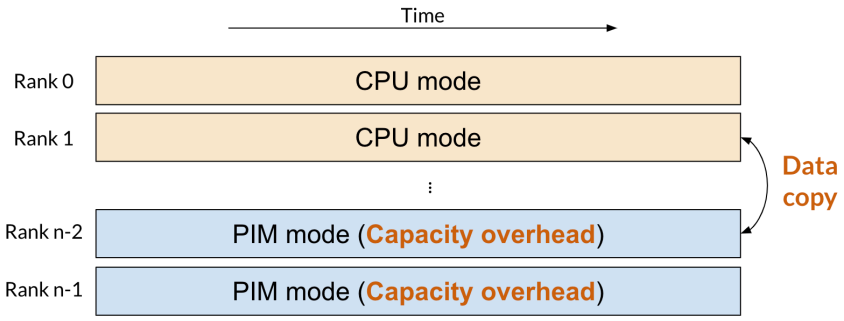



Figure 3.1: A timeline view of coarse-grain spatial partitioning.

instances of the same data are created, one for the CPU and the other for PIMs. This is also true for discrete accelerators, such as GPUs.

As shown in Figure 3.1, the main drawback of this approach is that a large fraction of memory capacity is reserved for the PIMs. Moreover, the CPU and PIMs cannot share the same address space and data copying is always required between two different address spaces, it is unclear what the advantage of this type of PIM is over other discrete accelerators. In fact, discrete accelerators have more relaxed power and area constraints than PIMs, thereby arithmetic performance and the size of on-chip memory can be superior to PIMs. As a result, there is a chance that PIMs with partitioned/discrete memory will remain energy-efficient, low-performance accelerators under heterogeneous systems with various accelerators, which is less attractive considering the much greater potential benefits of the PIM concept.

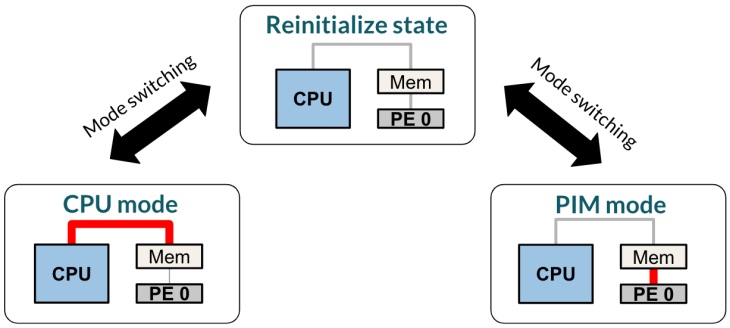



Figure 3.2: Existing mode switching mechanism with state reinitialization.

## 3.3 Coarse-Grain Mode Switching

The third approach for processing in main memory is with coarse-grain temporal mode switching. Unlike direct host control, the PIM units have their own memory controller so that they can access memory independently from the CPU. The ownership for memory is ping-ponged between the CPU and PIMs and only one of them exclusively accesses memory once it temporarily acquires ownership. Since there are two memory controllers managing the same memory (CPU-side and PIM-side), those two memory controllers must synchronize their state (i.e., timing and bank state) before ownership is switched. As shown in Figure 3.2, prior work [44] synchronizes the memory controllers by initializing all the memory banks. That is, the memory controller that currently has the ownership pre-charges, or closes, all the banks and only then hands the ownership to its counterpart. Then, the counterpart starts accessing memory. This method is simple and effective since it does not require communication between memory controllers for state synchronization. However, as shown in Figure 3.3, this approach incurs two overheads hindering the fine-grain own-

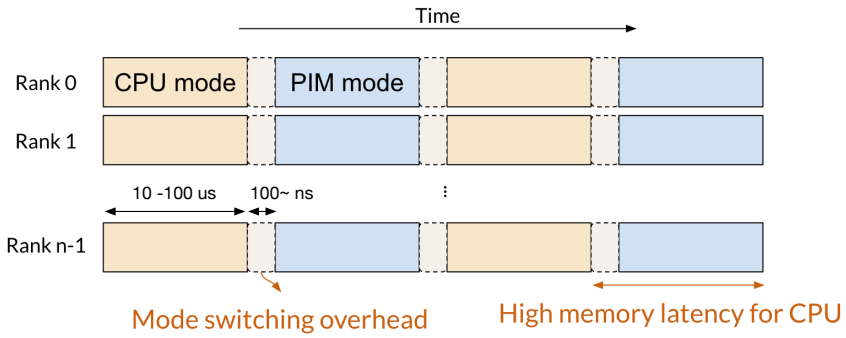



Figure 3.3: A timeline view of coarse-grain mode switching.

ership switching which is necessary for CPU-PIM cooperation: initialization and warmup overheads. Pre-charging the banks for state initialization itself takes time and warmup overheads originating from extra row activations are required. State initialization is especially wasteful when the CPU and PIMs access different banks in between switching. This is because their target rows could remain open and benefit from row-buffer hits if both memory controllers can somehow synchronize their state without initializing. Also, the CPU cannot access memory during the PIM execution mode, resulting in high memory latency.

To amortize these overheads, ownership switching should happen in a coarse-grained manner. As a result, the performance of CPU and PIMs will be directly proportional to the time that they possess ownership. To achieve high PIM performance, the CPU memory transactions are blocked during the PIM access mode and, consequently, increases CPU memory latency, or requires spatial partitioning. On the other hand, to achieve high CPU performance, all the PIM memory transactions should be blocked even though their memory

units remain idle due to the CPU memory access pattern, i.e. accessing at most one memory unit at a time. Therefore, to enable PIM in main memory, we need a better approach to alleviate the strict tradeoff between the CPU and PIM performances.

## 3.4 Concurrent Access: Fine-Grain Access Interleaving to the Same Memory Devices

My approach is to share the same memory devices between the CPU and PIMs and temporally interleave CPU and PIM memory accesses in a fine-grained manner, which I call *concurrent access* (Figure 3.4). Concurrent access should be enabled in a way that achieves low memory latency for the CPU and high memory bandwidth for PIMs, as those are the required features for each to achieve high performance. Concurrent access enables the CPU and PIMs to share the same memory devices, and even the same data, and access them in high performance. This is particularly useful when the size of shared data is extremely large. However, as stated above, there is a set of challenges to overcome for realizing concurrent access: (1) command bandwidth bottleneck caused by sending PIM commands to multiple PIMs connected to each memory channel, (2) contention and interference between the CPU and PIMs, (3) memory controller state synchronization, and (4) sharing address space between the CPU and PIMs for fast in-place acceleration. Especially, for the first two challenges, we need new approach for fine-grain CPU-PIM access interleaving and coarse-grain PIM operations, respectively. I elaborate on each

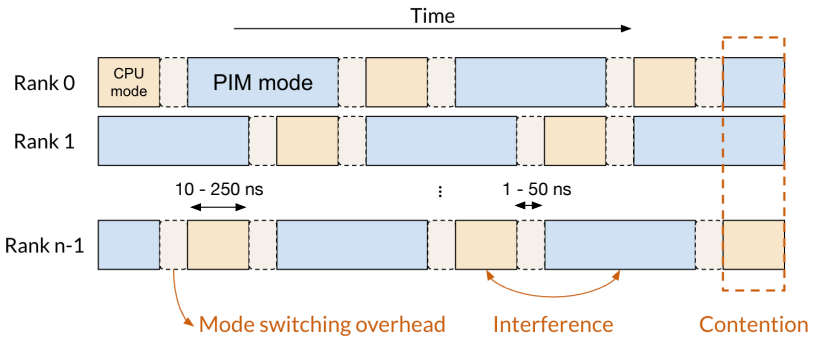



Figure 3.4: A timeline view of the proposed concurrent access mode.

approach below.

***The need for fine-grain access interleaving with opportunistic PIM issue.*** An ideal PIM opportunistically issues PIM memory requests whenever a PIM memory is idle from the perspective of the CPU. This is simple to do in a packetized interface where a memory-side controller schedules all accesses, but is a challenge in a traditional memory interface because the CPU- and PIM-side memory controllers must be synchronized. Prior work proposed dedicating some ranks to PIMs and some to the CPU or coarse-grain temporal interleaving [44, 15]. The former approach cannot enable concurrent access as devices are not shared. The latter results in large performance overhead because it cannot effectively utilize periods where a rank is naturally idle due to the CPU access pattern. Figure 3.5 shows that for a range of multi-core application mixes (methodology in Section 5.3), the majority of idle periods are shorter than 100 cycles with the vast majority under 250 cycles. *Fine-grain access interleaving is therefore necessary.*

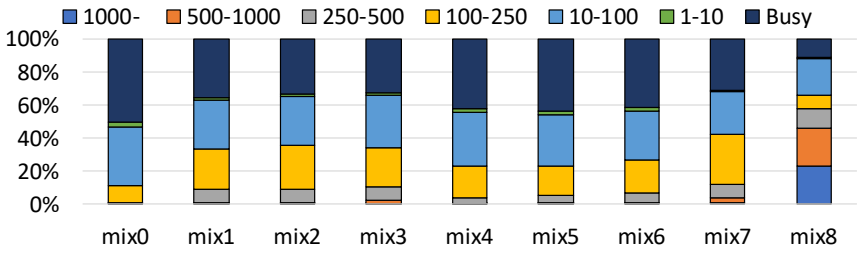



Figure 3.5: Rank idle-time breakdown vs. idleness granularity.

***The need for coarse-grain PIM vector/kernel operations.*** Fine-grain access interleaving is simple if each PIM command only addresses a single cache block region of memory. Such fine-grain PIM operations have indeed been discussed in prior work [10, 9, 93, 107]. One overhead of this fine-grain approach is that of issuing numerous PIM commands, with each requiring a full memory transaction that occupies both the command and data channels to memory. Issuing PIM commands too frequently degrades CPU performance, while infrequent issue underutilizes the PIMs. Coarse-grain PIM vector operations that operate on multiple cache blocks mitigate contention on the channel and improve overall performance. The vector width, $N$, is specified for each PIM instruction. As long as the operands are contiguous in the DRAM address space, one PIM instruction can process numerous data elements without occupying the channel. Coarse-grain PIM operations are therefore desirable, but *introduce the data layout, memory contention, and CPU–PIM synchronization challenges.*

The advantages and disadvantages of existing and my PIM approaches

Table 3.1: Comparison of different PIM approaches

| | Scalable CMD BW | Sharing data | Concurrent exec. |
|---|---|---|---|
| Direct host control | NO | YES | YES |
| CG spatial partitioning | YES | NO | YES |
| CG mode switching | YES | YES | NO |
| Concurrent access (proposed) | **YES** | **YES** | **YES** |

are summarized in Table 3.1.

## 3.5 Processing-In-Memory in Compute-Centric vs. Memory-Centric Systems

Existing PIM research can also be categorized into: compute-centric vs. memory-centric systems. Conventional systems based on the *compute-centric architecture* are organized with CPUs at the center and memory devices attached to them. In these systems, all the data movement and communication go through the CPU and memory devices cannot directly communicate with each other. On the other hand, there are proposals for *memory-centric architectures* [45, 63, 107, 65, 134], where memory devices are connected through a *memory network* and only some of the memory devices are also connected to the processors. Enabling processing-in-memory in each of these system types has different challenges.

Conventional compute-centric systems have been proven to execute diverse applications in high performance. However, in such systems, the bandwidth of memory channels connecting the CPU and memory devices often becomes a performance limiter for some memory-intensive applications.

Processing-in-memory is one potential solution to mitigate this bandwidth bottleneck. Therefore, the target PIM workloads should be ones for which the CPU inherently cannot execute in higher speed than PIMs due to the memory bandwidth bottleneck. In addition, since all the inter-PIM communication is routed through the CPU, the target workloads must exhibit good spatial locality so that inter-PIM communication is rare. If the PIMs cannot do better than the CPU because of the reasons such as locality and communication overhead, the workload can always be executed with the CPU, which adds flexibility to the system. This dissertation focuses on enabling PIMs for this *compute-centric systems*.

On the other hand, enabling PIM in memory-centric systems has similar challenges to distributed systems in general. Data remains stationary in each memory device and compute threads are offloaded via the memory network to where the target data resides. In such systems, data should be laid out such that spatial locality can be maximally exploited so that offloading overhead can be amortized. To port programs to the memory-centric systems, a specific programming model should be used to express data locality. However, the main drawback is that this memory-centric computation can only be effective for certain applications, such as large-scale linked-list traversal.

# Chapter 4

# High-performance CPU-PIM Concurrent Memory Access

In this chapter [1], I address several outstanding issues in the context of PIM-enabled main memory. My focus is on memory that can be concurrently accessed both as a PIM and as a memory. Such memory offers the powerful capability of the PIM and host processor collaboratively processing data without costly data copies. Prior research in this context is limited to fine-grain PIM operations of, at most, cache-line granularity. However, I develop techniques for coarse-grain PIM operations that amortize CPU interactions across processing entire DRAM rows. At the same time, my PIM unit does not block CPU memory access, even when the memory devices are controlled directly by the CPU memory controllers (e.g., a DDRx-like DIMM), which can reduce access latency and ease adoption.

Figure 4.1 illustrates an exemplary PIM architecture, which presents the challenges I address, and is similar to other recently-researched main-memory PIMs [44, 15, 14]. I choose a DIMM-based memory system because

[1] Portions of this chapter have been previously published as [29]. Coauthors of the paper contributed to DRAM power modeling (Yongkee Kwon) and bank-partitioning mechanism (Sangkug Lym).

it offers the high capacity required for a high-end server's main memory. Each DIMM is composed of multiple chips, with one or more DRAM dice stacked on top of a logic die in each chip, using a low-cost commodity 3DS-like approach. Processing elements (PEs) and a memory controller are located on the logic die. Each PE can access memory internally through the PIM memory controller. These local PIM accesses must not conflict with external accesses from the host (e.g., a CPU). A rank that is being accessed by the CPU cannot at the same time serve PIM requests, though the bandwidth of all other ranks in the channel can be used by the PIMs. There is no communication between PEs other than through the CPU. While not identical, recent commercial PIM-enabled memories exhibit similar overall characteristics [38, 104].

Surprisingly, no prior work on PIM-enabled main memory examines the architectural challenges of simultaneous and concurrent access to memory devices from both the CPU and PIMs. I identify and address two key challenges for enabling performance-efficient PIMs in a memory system that supports concurrent access from both a high-performance CPU and the PIMs.

The first challenge is that interleaved accesses may hurt memory performance because they can both decrease row-buffer locality and introduce additional read/write turnaround penalties. The second challenge is that each PIM can process kernels that consume entire arrays, though all the data that a single operation processes must be local to a PE (e.g., a memory chip). Therefore, enabling cooperative processing requires that CPU physical addresses are mapped to memory locations (channel, rank, bank, etc.) in a way that both

achieves high CPU-access performance (through effective and complex interleaving) and maintains PIM locality across all elements of all operands of a kernel. Note that these challenges exist when using either a packetized interface, where the memory-side controller interleaves accesses between PIMs and the CPU, or a traditional CPU-side memory controller that sends explicit low-level memory commands.

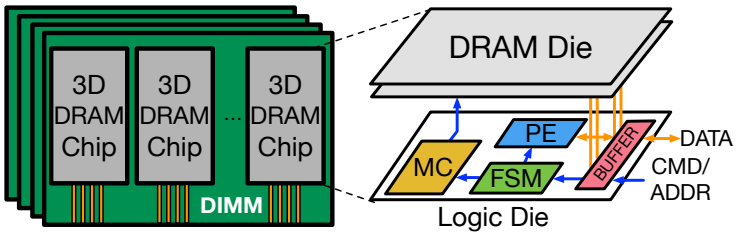



Figure 4.1: Exemplary PIM architecture.

*For the first challenge (managing concurrent access)*, I identify reduced row-buffer locality because of interleaved CPU requests as interfering with PIM performance. In contrast, it is the increased read/write turnaround frequency resulting from PIM writes that mainly interfere with the CPU. I provide two solutions in this context. First, I develop a new bank-partitioning scheme that limits interference to just those memory regions that are shared by the CPU and PIMs, thus enabling colocating CPU-only tasks with tasks that use the PIMs. This new scheme is the first that is compatible with huge pages and also with the advanced memory interleaving functions used in recent processors. Partitioning mitigates interference from the CPU to the PIMs and substantially boosts their performance (by $1.5 - 2\times$).

Second, I control interference on shared ranks by opportunistically issuing PIM memory commands to those ranks that are even briefly not used by the CPU and curb PIM to CPU interference with mechanisms that can throttle PIM requests, either selectively when a conflict is predicted (*next-rank prediction*) or stochastically.

*For the second challenge (PIM operand locality)*, I enable fine-grain collaboration by architecting a new data layout that preserves locality of operands within the distributed PIMs while simultaneously affording parallel accesses by the high-performance CPU. This layout requires minor modifications to the memory controller and utilizes coarse-grain allocations and physical-frame coloring in OS memory allocation. This combination allows large arrays to be shuffled across memory devices (and their associated PIMs) in a coordinated manner such that they remain aligned in each PIM. This is crucial for coarse-grain PIM operations that can achieve higher performance and efficiency than cacheline-oriented fine-grain PIMs (e.g., [10, 80, 64]).

*An additional and important challenge* exists in systems where the CPU maximizes its memory performance by directly controlling memory devices rather than relying on a packetized interface [119, 57]. Adding PIM capabilities requires providing local memory controllers near memory in addition to the CPU ones, which introduces a coordination challenge. I coordinate memory controllers and ensure a consistent view of bank and timing state with only minimal signaling that does not impact performance by replicating the controller finite state machines (FSMs) at both the PIM and CPU sides of

the memory channels. Replicating the FSM requires all PIM accesses to be determined only by the PIM operation (known to the CPU controller) and any CPU memory operations. Thus, no explicit signaling is required from the PIMs back to the CPU. I therefore require that for non-packetized PIMs, each PIM operation has a deterministic access pattern for all its operands (which may be arbitrarily fine-grained).

In this chapter, I introduce *Chopim*, a SW/HW holistic solution that enables concurrent CPU and PIM access to main memory by addressing the challenges above with fine temporal access interleaving to physically-shared memory devices. I perform a detailed evaluation both when the CPU and PIM tasks process different data and when they collaborate on a single application. I demonstrate that Chopim enables high PIM memory throughput (up to 97% of unutilized bandwidth) while maintaining CPU performance. Performance and scalability are better than with prior approaches of partitioning ranks or only allowing coarse-grain temporal interleaving, or with only fine-grain PIM operations.

I demonstrate the potential of CPU and PIM collaboration by studying a machine-learning application (logistic regression with stochastic variance-reduced gradient descent [74]). I map this application to the CPU and PIMs such that the CPU stochastically updates weights in a tight inner loop that utilizes the speculation and locality mechanisms of the CPU while PIMs concurrently compute a correction term across the entire input data that helps the algorithm converge faster. Collaborative and parallel PIM and CPU exe-

cution can speed up this application by 2× compared to CPU-only execution and 1.6× compared to non-concurrent CPU and PIM execution. I then evaluate the impact of colocating such an accelerated application with CPU-only tasks.

In summary, I make the following main contributions:

- I identify new challenges in concurrent access to memory from the CPU and PIMs: bank conflicts from CPU accesses curb PIM performance and read/write-turnaround penalties from PIM writes lower CPU performance.
- I reduce bank conflicts with a new bank partitioning architecture that, for the first time, is compatible with both huge pages and sophisticated memory interleaving.
- To decrease read/write-turnaround overheads, I throttle PIM writes with two mechanisms: *next-rank prediction* delays PIM writes to the rank actively read by the CPU; and *stochastic issue* throttles PIM writes randomly at a configurable rate.
- I develop, also for the first time, a memory data layout that is compatible with both the CPU and PIMs, enabling them to collaboratively process the same data in parallel while maintaining high CPU performance with sophisticated memory address interleaving.

- To show the potential of collaboratively processing the same data, I conduct a case study of an important ML algorithm that leverages the fast CPU for its main training loop and the high-BW PIMs for summarization steps that touch the entire dataset. I develop a variant that executes on the PIMs and CPU in parallel, which increases speedup to 2×.

## 4.1 Background

In this section, I summarize the background and assumptions that are specific to this chapter. More basic and general background can be found in Chapter 2.

***Baseline PIM Architecture.*** My work targets PIMs that are integrated within high-capacity memory modules such that their role as both main memory and as accelerators is balanced. Specifically, my baseline PIM devices are 3D-integrated within DRAM chips on a module (DIMM), similar to 3DS DDR4 [30] yet a logic die is added. DIMMs offer high capacity and predictable memory access. Designs with similar characteristics include on-DIMM PEs [104, 14] and on-chip PEs within banks [38]. Alternatively, PIMs can utilize high-bandwidth devices, such as the hybrid memory cube (HMC) [119] or high bandwidth memory (HBM) [135]. These offer high internal bandwidth but have limited capacity and high cost due to numerous point-to-point connections to memory controllers [15]. HMC provides capacity scaling via a network but this results in high access latency and cost. HBM does not provide such solutions. As a result, HBM devices are better for standalone

accelerators than for main memory.

***Write-to-Read Turnaround Time.*** In general, interleaving read and write DRAM transactions incurs higher latency than issuing the same transaction type back to back. Issuing a read transaction immediately following a write suffers from particularly high penalty. The memory controller issues the write command and loads data to the bus after tCWL cycles. Then, data is transferred for tBL cycles to the DRAM device and written to the cells. The next read command can only be issued after tWTR cycles, which guarantees no conflict on the IO circuits in DRAM. The high penalty stems from the fact that the actual write happens at the end of the transaction whereas a read happens right after it is issued. For this reason, the opposite order, read to write, has lower penalty.

***Coherence.*** Coherence mechanisms between the CPU and PIMs have been studied in prior PIM work [10, 21, 22] and can be used as is with Chopim. I therefore do not focus on coherence in this chapter. In my experiments, I use the existing coherence approach of explicitly and infrequently copying the small amount of data that is not read-only using cache bypassing and memory fences.

***Address Translation for PIM Execution.*** Application use of PIMs requires virtual to physical address translation. Some prior work [65, 62, 45] proposes address translation within PIMs to enable independent PIM execution without CPU assist. This increases both PIM and system complexity. As

| Operations | Description | Operations | Description |
|---|---|---|---|
| AXPBY | $\vec{z} = \alpha\vec{x} + \beta\vec{y}$ | DOT | $c = \vec{x} \cdot \vec{y}$ |
| AXPBYPCZ | $\vec{w} = \alpha\vec{x} + \beta\vec{y} + \gamma\vec{z}$ | NRM2 | $c = \sqrt{\vec{x} \cdot \vec{x}}$ |
| AXPY | $\vec{y} = \alpha\vec{y} + \vec{x}$ | SCAL | $\vec{x} = \alpha\vec{x}$ |
| COPY | $\vec{y} = \vec{x}$ | GEMV | $\vec{y} = A\vec{x}$ |
| XMY | $\vec{z} = \vec{x} \odot \vec{y}$ | | |

Table 4.1: Example PIM operations used in my case-study application. Chopim is not limited to these operations.

an alternative, PIM operations can be constrained to only access data within a physical memory region that is contiguous in the virtual address space. Hence, translation is performed by the CPU when targeting a PIM command at a certain physical address. This has been proposed for both very fine-grain PIM operations within single cache lines [10, 9, 93, 80, 107] and PIM operations within a virtual memory page [114]. In this chapter, I use host-based translation because of its low complexity and only check bounds within the PIMs for protection.

***PIM Workloads.*** I focus on PIM workloads for which the CPU inherently cannot outperform a PIM. These exhibit low temporal locality and low arithmetic intensity and are bottlenecked by peak memory bandwidth. By offloading such operations to the PIM, I mitigate the bandwidth bottleneck by leveraging internal memory module bandwidth. Moreover, these workloads typically require simple logic for computation and integrating such logic within DRAM chips/modules is practical because of the low area and power overhead.

Fundamental linear algebra matrix and vector operations satisfy these criteria. Dense vector and matrix-vector operations, which are prevalent in machine learning primitives, are particularly good candidates because of their deterministic and regular memory access patterns and low arithmetic-intensity. For example, prior work off-loads matrix and vector operations of deep learning workloads to utilize high near-memory BW [79, 46]. Also, Kwon et al. propose to perform element-wise vector reduction operations needed for a deep-learning-based recommendation system to PIMs [89]. In this chapter, I focus on accelerating the dense matrix and vector operations summarized in Table 4.1. I demonstrate and evaluate their use in the SVRG application in Section 4.3. Note that I use these as a concrete example, but my contributions generalize to other PIM operations.

PIM execution of graph processing has also been proposed because graph processing can be bottlenecked by peak memory bandwidth because of low temporal and spatial locality [107, 148, 134, 9, 10]. Prior work either relies on high inter-chip communication to support the irregular access patterns of graph applications, or focuses on fine-grain cache-block oriented PIM operations rather than coarse-grain operations. The former is incompatible with my economic main-memory context and my research offers nothing new if only fine-grain PIM operations are used.

## 4.2 Chopim

I develop Chopim with four main connected goals that push the state of the art: (1) enable fine-grain interleaving of CPU and PIM memory requests to the same physical memory devices while mitigating the impact of their contention; (2) permit the use of coarse-grain PIM operations that process long vector instructions/kernels; (3) simultaneously support the locality needed for PIMs and the sophisticated memory address interleaving required for high CPU performance; and (4) integrate with both a packetized interface and a traditional CPU-controlled DDRx interface. I detail my solutions in this section.

### 4.2.1 Localizing PIM Operands while Distributing CPU Accesses

To execute the N-way PIM vector instructions, all the operands of each PIM instruction must be fully contained in a single rank (single PE). If necessary, data is first copied from other ranks prior to launching a PIM instruction. If the reuse rate of the copied data is low, this copying overhead will dominate the PIM execution time and contention on the memory channel will increase due to the copy commands.

I solve this problem in Chopim by laying out data such that all the operands are localized to each PIM at memory allocation time. Thus, copies are not necessary. This is challenging, however, because the CPU memory controller uses complex address interleaving functions to maximally exploit channel, rank, and bank parallelism for arbitrary CPU access patterns. Hence,

arrays that are contiguous in the CPU physical address space are not contiguous in physical memory and are shuffled across ranks. This challenge is illustrated in the left side of Figure 4.2, where two operands of PIM instruction are shuffled differently across ranks and banks. The layout resulting from my approach is shown at the right of the figure, where arrays (operands) are still shuffled, but both operands follow the same pattern and remain correctly aligned to PIMs without copy operations. Note that alignment is to rank because each rank has one PIM unit in my baseline PIM architecture.

***Data layout across ranks.*** I rely on the PIM runtime and OS to use a combination of coarse-grain memory allocation and coloring to ensure all operands of a PIM instruction are interleaved across ranks the same way and are thus local to a PE. First, the runtime allocate memory for PIM operands such that they are aligned at the granularity of one DRAM row for each bank in the system which I call a *system row* (e.g., 2MiB for a DDR4 1TiB system). For all the address interleaving mechanisms described in the literature ([121, 98, 113, 83]), this ensures that PIM operands are locally aligned, as long as ranks are also kept aligned. To maintain rank alignment, I rely on OS frame coloring to effect rank alignment. I explain this feature below using the Intel Skylake address mapping [121] as a concrete and representative interleaving mapping (Figure 4.3a).

In this mapping, rank and channel addresses are determined partly by the low-order bits that fall into the frame offset field and partly by the high-order bits that fall into the physical frame number (PFN) field. Frame

offsets are kept the same because of the coarse-grain alignment. The OS colors memory allocations such that the PFN bits that determine rank and channel are aligned for a particular color; the specific physical address bits select ranks and channels can be reverse engineered if necessary [121]. The Chopim runtime indicates a *shared color* when it requests memory from the OS and specifies the same color for all operands of an instruction. The runtime can use the same color for many operands to minimize copies needed for alignment. In my baseline system, there are 8 colors and each color corresponds to a 4GiB memory space. Multiple regions can be allocated for the same process. Though I focus on one address mapping here, my approach works with any XOR-based address mapping described in prior work [121, 98] as well.

Note that coarse-grain allocation is simple with the common buddy allocator if allocation granularity is also a system row, and can use optimizations that already exist for huge pages [145, 90, 52]. The fragmentation overheads of coarse allocation are similar to those with huge pages and I find that they are negligible because coarse-grain PIM execution works best when processing long vectors.

***Data layout across DRAM chips.*** In the baseline system, each 4-byte word is striped across multiple chips, whereas in my approach each word is located in a single chip so that PIMs can access words from their local memory. Both the CPU and PIMs can access memory without copying or reformatting data (as required by prior work [44]). Memory blocks still align with cache lines, so this layout change is not visible to software. This layout precludes

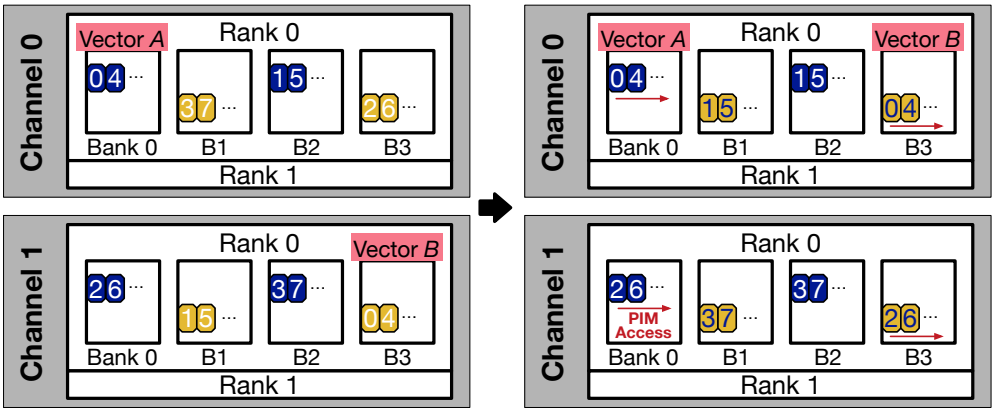
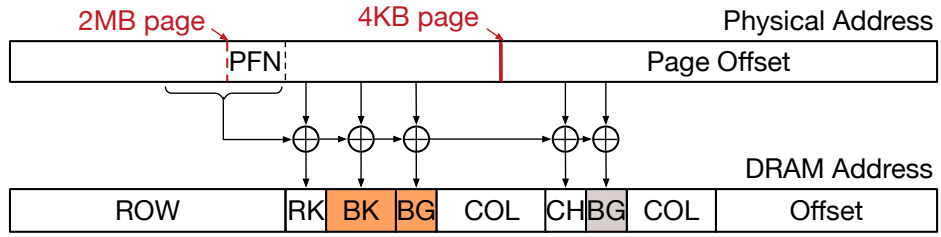



Figure 4.2: Example data layout across ranks for concurrent access of the COPY operation (B[i] = A[i]). With naive data layout (left), elements with the same index are located in different ranks. With my proposed mechanism (right), elements with the same index are co-located. PIMs access contiguous columns starting from the base of each vector.

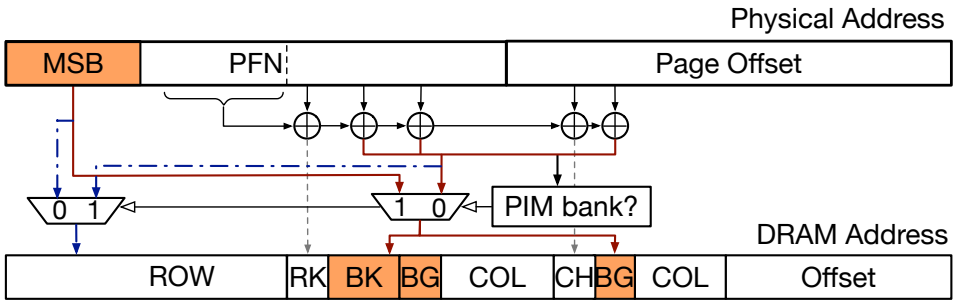
the critical word first optimization from DRAM, but recent work concludes the impact is minimal because the relative latency difference in current memory systems is very small (e.g., [144]). Note that this data layout does not impact the CPU memory controller's ECC computation (e.g., Chip-kill [36]) because ECC protects only bits and not how they are interpreted. For PIM accesses, I rely on in-DRAM ECC with its limited coverage. I do not innovate in this respect and leave this problem for future work.

### 4.2.2 Mitigating Frequent Read/Write Penalties

The basic memory access scheduling policy I use for Chopim is to always prioritize CPU memory requests, yet aggressively leverage unutilized rank bandwidth by issuing PIM requests whenever possible. That is, PIMs




(a) Baseline (Skylake [121])




(b) Proposed (for bank partitioning)

Figure 4.3: Baseline and proposed CPU-side address mapping.

wait when incoming CPU requests are detected, but otherwise always issue their memory requests to maximize their bandwidth utilization and performance. One potential problem is that a PIM request issued in one cycle may delay a CPU request that could have issued in one of the following cycles otherwise.

I find that PIMs infrequently issue row commands (ACT and PRE) due to the streaming nature of target PIM operations. I therefore prioritize CPU memory commands over any PIM row command to the same bank. This has negligible impact on PIM performance in my experiments.

I also find that read transactions of PIMs have only a small impact

on following CPU commands. PIM write transactions, however, can have a large impact on CPU performance because of the read/write-turnaround penalties that they frequently require. While the CPU mitigates turnaround overhead by buffering operations with caches and write buffers [137, 8], the CPU and PIMs may interleave different types of transactions when accessing memory in parallel. I find that PIM writes interleaved with CPU reads degrade performance the most. *As a solution,* I introduce two mechanisms to selectively throttle PIM writes.

My first mechanism throttles the rate of PIM writes by issuing them with a predefined probability. I call this mechanism *stochastic PIM issue*. Before issuing a write transaction, the PIMs both detect if a rank is idle and flip a coin to determine whether to issue the write. By adjusting the coin weight, the performance of the CPU and PIMs can be traded off: higher write-issue probability leads to more frequent turnarounds while a lower probability throttles PIM progress. Deciding how much to throttle PIMs requires analysis or profiling, and I therefore propose a second approach as well.

My second approach does not require tuning, and I empirically find that it works well. In this *next rank prediction* approach, the memory controller inhibits PIM write requests when more CPU read requests are expected; the controller stalls the PIM in lieu of providing a PIM write queue. In a packetized interface, the memory controller schedules both CPU and PIM requests and is thus aware of potential required turnarounds. The traditional memory interface, however, is more challenging as the CPU controller must explicitly

signal the PIM controller to inhibit its write request. This signal must be sent ahead of the regular CPU transaction because of bus delays.

I use a very simple predictor that inhibits PIM write requests in a particular rank when the oldest outstanding CPU memory request to that channel is a read to that same rank. Specifically, the PIM controller examines the target rank of the oldest request in the CPU memory controller transaction queue. Then, it signals to the PIMs in that rank to stall their writes. For now, I assume that this information is communicated over a dedicated pin. My experiments with an FRFCFS [127] memory scheduler at the CPU shows that this simple predictor works well and achieves performance that is comparable to a tuned stochastic issue approach.

### 4.2.3 Partitioning into CPU and Shared Banks

In addition to read/write-turnaround overheads, concurrent access also degrades performance by decreasing DRAM row access locality. When the CPU and PIMs interleave accesses to different rows of the same bank, frequent bank conflicts occur. To avoid this bank contention, I propose using bank partitioning to limit bank interference to only those memory regions that must concurrently share data between the PIMs and the CPU. This is particularly useful in colocation scenarios when only a small subset of CPU tasks utilize the PIMs. However, existing bank partitioning mechanisms [105, 71, 97] are incompatible with both huge pages and with sophisticated DRAM address interleaving schemes.

Bank partitioning relies on the OS to color pages where colors can be assigned to different cores or threads, or in my case, for banks isolated for the CPU and those that could be shared. The OS then maps pages of different color to frames that map to different banks. Unfortunately, advanced physical-to-DRAM address mapping functions and the use of 2MB pages prevent prior bank partitioning schemes from working because the physical frame number (PFN) bits that the OS can control can no longer specify arbitrary bank partitions. Figure 4.3a shows an example of a modern physical address to DRAM address mapping [121]. One color bit in the baseline mapping belongs to the page offset field so prior bank partitioning schemes can, at best, be done at two-bank granularity. More importantly, when huge pages are used (e.g., 2MiB), this baseline mapping cannot be used to partition banks at all.

To overcome this limitation, I propose a new interface that partitions banks into two groups—CPU-reserved and shared banks—with flexible DRAM address mapping and any page size. Specifically, my mechanism only requires that the most significant physical address bits are only used to determine DRAM row address, as is common in recent hash mapping functions, as shown in Figure 4.3b [121].

Without loss of generality, I assume that 2 banks out of 16 banks are reserved for the shared data. First, the OS splits the physical address space for CPU-only and shared memory regions with the CPU-only region occupying the bottom of the address space: $0 - (14 \times (bank\_capacity) - 1)$. The rest of the space (with capacity of 2 banks) is reserved for the shared data and the OS

does not use it for other purposes. This guarantees that the most significant bits (MSBs) of the address of CPU-only region are never b'111. In contrast, addresses in the shared space always have b'111 in their MSBs.

The OS informs the memory controller that it reserves 2 banks (the top-most banks) for the shared memory region. CPU-only memory addresses are mapped to DRAM locations using any hardware mapping function, which is not exposed to software and the OS. The idea is then to remap addresses that initially fall into shared banks into the reserved address space that the CPU is not using. Additional simple logic checks whether the resulting DRAM address bank ID of the initial mapping is reserved for the shared region. If they are not, the DRAM address is used as is. If the DRAM address is initially mapped to one of the reserved banks, the MSBs and the bank bits are swapped. Because the MSBs of a CPU address are never b'1110 or b'1111, the final bank ID will be one of the CPU-only bank IDs. Also, because the bank ID of the initial mapping result is either 14 or 15, the final address is in a row the CPU cannot access with the initial mapping and there is no aliasing. Note that the partitioning decision can be adjusted, but only if all affected memory is first cleared.

### 4.2.4 Tracking Global Memory Controller State

Unlike conventional systems, Chopim also enables an architecture that has two memory controllers (MCs) managing the bank and timing state of each rank. This is the case when the CPU continues to directly manage memory

even when the memory itself is enhanced with PIMs. This requires coordinating rank state information between controllers. Figure 4.4 shows how MCs on both sides of a memory channel track global memory controller state. Information about CPU transactions is easily obtained by the PIM MCs as they can monitor incoming transactions and update the state tables accordingly (left). However, the CPU MC cannot track all PIM transactions due to command bandwidth limits.

To solve this problem, I replicate the finite-state machines (FSMs) of PIMs and place them in the CPU-side PIM controller. When a PIM instruction is launched, the FSMs on both sides are synchronized. I rely on the already-synchronized DDR interface clock for FSM synchronization. Whenever a PIM memory transaction is issued, the CPU-side FSM also updates the state table in the CPU MC without communicating with the PIMs (right). If a CPU transaction blocks PIM transactions in one of the ranks, that transaction will be visible to both FSMs. Replicated FSMs track the PIM write buffer occupancy and detect when the write-buffer draining starts and ends to trigger write throttling. The area and power overhead of replicating FSMs are negligible (40-byte microcode store and 20-byte state registers per rank (i.e., per PIM unit)). *My evaluation uses this approach to enable a DDR4-based PIM-enabled main memory and all my experiments rely on this.*

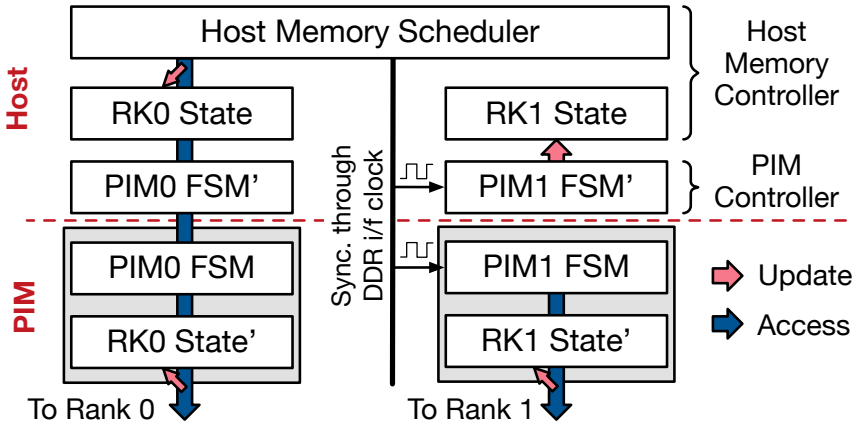



Figure 4.4: Global MC state tracking when the CPU (left) and PIMs (right) issues memory commands. The replicated FSMs are synchronized by using the DDR interface clock.

## 4.3 CPU-PIM Collaboration

In this section, I describe a case study to show the potential of concurrent CPU-PIM execution by collaboratively processing the same data. My case study shows how to partition an ML training task between the CPU and PIMs such that each processor leverages its strengths. As is common to training and many data-processing tasks, the vast majority of shared data is read-only, simplifying parallelism.

I use the machine-learning technique of logistic regression with stochastic variance reduced gradient (SVRG) [74] as my case study. Figure 4.5 shows a simplified version of SVRG and the opportunity for collaboration. The algorithm consists of two main tasks within each outer-loop iteration. First, the entire large input matrix $A$ is *summarized* into a single vector $g$ (see

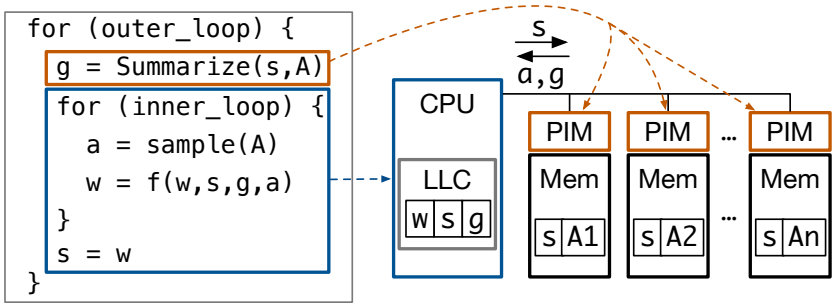



Figure 4.5: Collaboration between CPU and PIMs in SVRG.

Figure 4.7 for pseudocode). This vector is used as a correction term when updating the model in the second task. This second task consists of multiple inner-loop iterations. In each inner-loop iteration the learned model $w$ based on a randomly-sampled vector $a$ from the large input matrix $A$, the correction term $g$, and a stored model $s$, which is updated at the end of the outer-loop iteration.

The first task is an excellent match for the PIMs. The summarization operation is simple, exhibits little reuse, and traverses the entire large input data. In contrast, the second task with its tight inner loop is well suited for the CPU. The CPU can maximally exploit locality captured by its caches while PIMs can leverage their high bandwidth for accessing the entire input data $A$. Note that in SVRG, an *epoch* refers to the number of inner loop iterations.

The main tradeoff in SVRG is as follows. When summarization is done more frequently, the quality of the correction term increases and, consequently, the per-step convergence rate increases. On the other hand, the overhead of summarization also increases when it is performed more frequently, which

offsets the improved convergence rate. Therefore, the *epoch* hyper-parameter, which determines the frequency of summarization, should be carefully selected to optimize this tradeoff.

***Delayed-Update SVRG.*** As Chopim enables concurrent access from the CPU and PIMs, I explore an algorithm change to leverage collaborative parallel processing. Instead of alternating between the summarization and model update tasks, I run them in parallel on the CPU and PIMs. Whenever the PIMs finish computing the correction term, the CPU and PIMs exchange the correction term and the most up-to-date weights before continuing parallel execution. While parallel execution is faster, it results in using stale $s$ and $g$ values from one epoch behind. The main tradeoff in *delayed-update SVRG* is that per-iteration time is improved by overlapping execution, whereas convergence rate per iteration degrades due to the staleness. Similar tradeoffs have been observed in prior work [18, 91, 126, 35]. I later show that delayed-update SVRG can converge in 40% less time than when serializing the two main SVRG tasks.

To avoid races for $s$ and $g$ in this delayed-update SVRG, I maintain private copies of these small variables and use a memory fence that guarantees completion of DRAM writes after the data-exchange step (which the runtime coordinates with polling). Note that I bypass caches when accessing data produced/consumed by PIMs during the data-exchange step. Since $s$ and $g$ are small and copied infrequently, the overheads are small and amortized over numerous PIM computations. Whether delayed updates are used or not, the

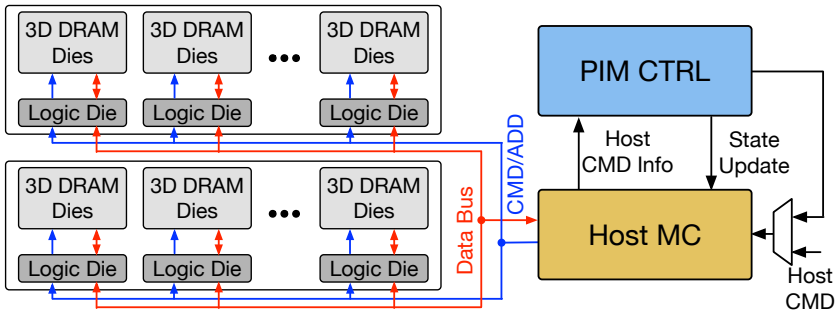



Figure 4.6: Overview of PIM architecture.

CPU and PIMs share the large data, $A$, without copies.

## 4.4 Runtime and API

Chopim is general and helps whenever CPU/PIM concurrent access is needed. To make the explanations and evaluation concrete, I use an exemplary interface design as discussed below and summarized in Figure 4.6. Command and address signals pass through the PIM memory controllers so that they can track CPU rank state. Processing elements (PEs) in the logic die access data by using their local PIM memory controller (Figure 4.1 ). I propose a similar API as other C++ math libraries [128, 69, 66] for the example use case of accelerating linear algebra operations. Figure 4.7 shows example usage of my API for computing the average gradient used in the summarization task of SVRG.

The Chopim runtime system manages memory allocations and launches PIM operations. PIM operations are blocking by default, but can also execute asynchronously. If the programmer calls a PIM operation with operands from

different shared regions (colors), the runtime system inserts appropriate data copies. I envision a just-in-time compiler that can identify such cases and more intelligently allocate memory and regions to minimize copies. For this chapter, I do not implement such a compiler. Instead, programs are written to directly interact with a runtime system that is implemented within the simulator.

PIMs operate directly on DRAM addresses and do not perform address translation. To launch an operation, the runtime (with help from the OS) translates the origin of each operand into a physical address, which is then communicated along with a bound to the PIMs by the PIM controller. The runtime is responsible for splitting a single API call into multiple primitive PIM operations. The PIM operations themselves proceed through each operand with a regular access pattern implemented as microcode in the hardware, which also checks the bound for protection. DRAM addresses are computed by following the same physical-to-DRAM mapping function used by the CPU memory controller.

***Optimizing Load-balance.*** Load imbalance occurs when the CPU does not access ranks uniformly over short periods of time. The AXPY operation (launched repeatedly within the loop shown in Figure 4.7) is short and non-uniform access by the CPU leads to load imbalance among PIMs. A blocking operation waits for *all* PIMs to complete before launching the next AXPY, which reduces performance. My API provides asynchronous launches similar to CUDA streams or OpenMP parallel `for` with a `nowait` clause [33]. Asynchronous launches can overlap AXPY operations from multiple loop iterations.

```
void main () {
  // Memory Allocation
  float alpha, lambda;
  pim::matrix<float> X(n, d, pim::SHARED);
  pim::vector<float> w(d, pim::SHARED);
  pim::vector<float> y(n, pim::SHARED);
  pim::vector<float> v(n, pim::SHARED);
  pim::vector<float> a(d, pim::SHARED);
  pim::vector<float> a_pvt(d, pim::PRIVATE);
  // a_pvt: allocates d elements per PIM rather than
  //        striping the allocation across PIMs

  // Initialization
  ...

  // Average Gradient
  pim::gemv(y, X, w);
  pim::xmy(v, v, y);
  host::sigmoid(v, v);
  pim::xmy(v, v, y);
  pim::scal(v, 1/n);

  // Target for Macro Operation
  parallel_for (int i = 0; i < n; i++) {
    alpha = v[i];
    pim::axpy(a_pvt, alpha, X, i); // a_pvt += alpha *X[i]
  }

  host::reduce(a, a_pvt);
  pim::axpy(a, lambda, w);
}
```

Figure 4.7: Average gradient example code. This code corresponds to *summarization* in SVRG (see Section 4.3).

Any load imbalance is then only apparent when the loop ends. Over such a long time period, load imbalance is much less likely. I implement asynchronous launches using a *macro PIM operation*. An example of a macro operation is shown in the loop of Figure 4.7 and is indicated by the `parallel_for` annotation.

***Launching PIM Operations.*** PIM operations are launched similarly to

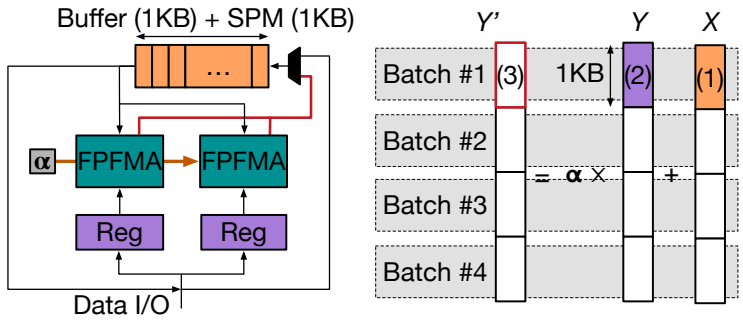



Figure 4.8: PE architecture and execution flow of AXPY.

the architecture of Farmahini et al. [44]. A memory region is reserved for accessing control registers of PIMs. PIM packets access the control registers and launch operations. Each packet is composed of the type of operation, the base addresses of operands, the size of data blocks, and scalar values required for scalar-vector operations. On the CPU side, the *PIM controller* plays two main roles. First, it accepts acceleration requests, issues commands to the PIMs in the different ranks (in a round-robin manner), and notifies software when a request completes. Second, the PIM controller extends the CPU memory controller to coordinate actions between the PIMs and CPU memory controllers and enables concurrent access. It maintains the replicated FSMs using its knowledge of issued PIM operations and the status of the CPU memory controller. The PIM controller is also responsible for throttling specific PIMs if necessary to maintain CPU performance. To specify all the supported PIM operations, the minimum packet size is 32 bytes, requiring 4 write cycles per rank.

***Execution Flow of a Processing Element.*** My exemplary PE is com-

posed of two floating-point fused multiply-add (FPFMA) units, 5 scalar registers (up to 3 operand inputs and 2 for temporary values), a 1KiB buffer for accessing memory, and the 1KiB scratchpad memory. The memory access granularity is 8B per chip and the performance of the two FPFMAs per chip matches this data access rate. PEs may be further optimized to support lower-precision operations or specialized for specific use cases, but I do not explore these in this chapter as I focus on the new capabilities of Chopim rather than PIM in general.

Figure 4.8 shows the execution flow of a PE when executing the AXPY operation. Each vector is partitioned into 1KiB batches, which is the same size as DRAM page size per chip. To maximize bandwidth utilization, the vector $X$ is streamed into the buffer. Then, the PE opens another row, reads two elements (8 bytes) of vector $Y$, and stores them to FP registers. While the next two elements of $Y$ are read, a fused multiply-add (FMA) operation is executed. The result is stored back into the buffer and execution continues such that the read-execute-write operations are pipelined. After the result buffer is filled, the PE either writes results back to memory or to the scratchpad. This flow for one 1KiB batch is repeated over the rest of the batches. This entire process is stored in PE microcode as the AXPY operation. Other operations (coarse or fine grained) are similarly stored and processed from microcode.

***Inter-PE Communication.*** PIMs are only effective when they use memory-side bandwidth to amplify that of the CPU. In the DIMM- and chip-based PIMs, which I target in this chapter, general inter-PE communication is there-

fore equivalent to communicating with the CPU. Communication in applications that match this PIM architecture are primarily needed for replicating data to localize operands or for global reduction operations, which follow local per-PE reductions. In both communication cases, a global view of the data is needed and, therefore, I enable communication only through the CPU. For instance, after the macro operation in Figure 4.7, a global reduction of the PE private copies (`a_pvt`) accumulates the data for the final result (`a`). The reduced result is used by the following PIM operation. Though communicating through the CPU is expensive, my coarse-grained PIM operations amortize infrequent communication overhead. Importantly, since this communication can be done as normal DRAM accesses by the CPU, no change on the memory interface is required.

## 4.5 Methodology

Table 5.2 summarizes my system configuration, DRAM timing parameters, energy components, benchmarks, and machine learning configurations. For bank partitioning, I reserve one bank per rank for the PIMs and the rest for the CPU. I use Ramulator [85] as my baseline DRAM simulator and add the PIM memory controllers and PEs to execute the PIM operations. I modify the memory controller to support the Skylake address mapping [121] and my bank partitioning and data layout schemes. To simulate concurrent CPU accesses, I use gem5 [19] with Ramulator. I choose CPU applications that have various memory intensity from the *SPEC2006* [61] and *SPEC2017* [115]

benchmark suites and form 9 different application mixes with different combinations (Table 5.2). Mix0 and mix8 represent two extreme cases with the highest and lowest memory intensity, respectively. Only mix0 is run with 8 cores to simulate under-provisioned bandwidth while other mixes use 4 cores to simulate a more realistic scenario. For the PIM workloads, I use the DOT and COPY operations to show the impact of extremely low and high write intensity. I use the average gradient kernel (Figure 4.7) to evaluate collaborative execution. The performance impact of other PIM applications falls between DOT and COPY and is well represented by SVRG [74], conjugate gradient (CG) [69] and streamcluster (SC) [122].

For the CPU workloads, I use Simpoint [58] to find representative program phases and run each simulation until the instruction count of the slowest process reaches 200M instructions. If an PIM workload completes while the simulation is still running, it is relaunched so that concurrent access occurs throughout the simulation time. Since the number of instructions simulated is different, I use instructions per cycle (IPC) for CPU performance. To show how well the PIMs utilize bandwidth, I measure bandwidth utilization and compare it with the idealized case where PIMs can utilize all the idle rank bandwidth.

I estimate power with the parameters in Table 5.2. I use CACTI 6.5 [106] for the dynamic and leakage power of the PE buffer. A sensitivity study for PE parameters shows that their impact on power dissipation is negligible. I use CACTI-3DD [26] to estimate the power and energy of

3D-stacked DRAM and CACTI-IO [75] to estimate DIMM power and energy.

## 4.6 Evaluation

I present evaluation results for the various Chopim mechanisms, analyzing: (1) the benefit of coarse-grain PIM operations; (2) how bank partitioning improves PIM performance; (3) how stochastic issue and next-rank prediction mitigate read/write turnarounds; (4) the impact of PIM workload write intensity and load imbalance; (5) how Chopim compares with rank partitioning; (6) the benefits of collaborative and parallel CPU/PIM processing; and (7) energy efficiency. All results rely on the replicated FSM with DDR4.

***Coarse-Grain PIM Operation.*** Figure 4.9 demonstrates how overhead for launching PIM instructions can degrade performance of the CPU and PIMs as rank count increases. To prevent other factors, such as bank conflicts, bank-level parallelism, and load imbalance from affecting performance, I use my BP mechanism, the NRM2 operation (because I can precisely control its granularity), and asynchronous launch. I run the most memory-intensive application mix (mix1) on the CPU. When more cache blocks (CBs) are processed by each PIM instruction, contention between CPU transactions and PIM instruction launches decreases and performance of both improves. In addition, as the number of ranks grows, contention becomes severe because more PIM instructions are necessary to keep all PIMs busy. These results show that my data layout that enables coarse-grain PIM operation is beneficial, especially in concurrent access situation.

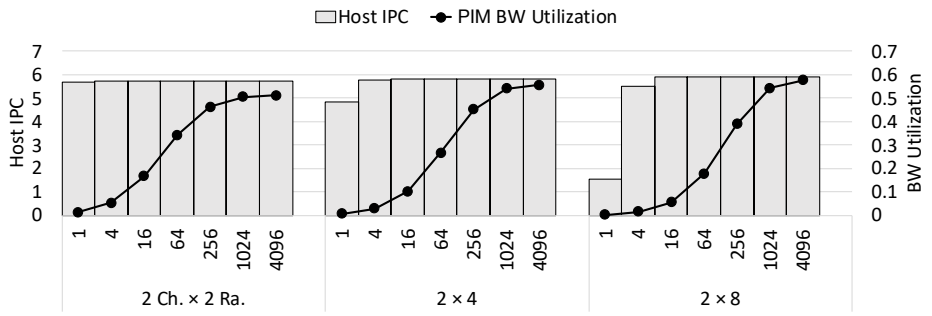



Figure 4.9: Impact of coarse-grain PIM operations. (X-axis: the number of cache blocks accessed per PIM instruction.)

| Takeaway 1: Coarse-grain PIM operations are crucial for mitigating contention on the CPU memory channel. |
|---|

***Impact of Bank Partitioning.*** Figure 4.10 shows performance when banks are shared or partitioned between the CPU and PIMs. I emphasize the impact of write intensity of PIM operations by running the extreme DOT (read intensive) and COPY (write intensive) operations. While not shown, SVRG falls roughly in the middle of this range. I compare each memory access mode with an idealized case where I assume the CPU accesses memory without any contention and PIMs can leverage all the idle rank bandwidth without considering transaction types and other overheads.

Overall, accelerating the read-intensive DOT with concurrent CPU access does not affect CPU performance significantly even with my aggressive approach. However, contention with the shared access mode significantly degrades PIM performance. This is because of the extra bank conflicts caused by interleaving CPU and PIM transactions. On the other hand, accelerating the write-intensive COPY degrades CPU performance. This happens because, in

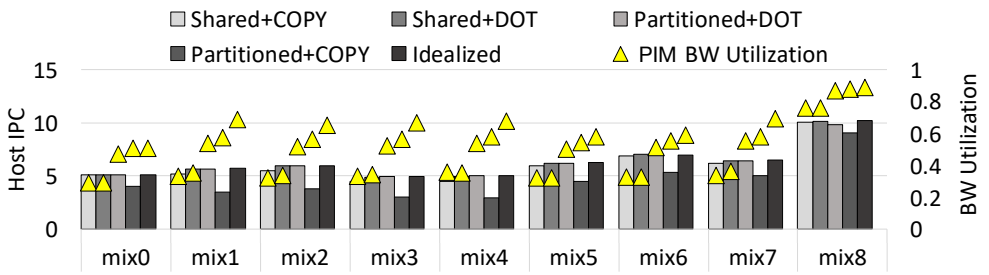



Figure 4.10: Concurrent access to different memory regions.

the write phase of PIMs when the PIM write buffer drains, the CPU reads are blocked while PIMs keep issuing write transactions due to long write-to-read turnaround time. To mitigate this problem, I show the impact of my write throttling mechanisms below. Note that CPU performance of mix0 is the lowest, despite its doubled core count, because contention for LLC increases and memory performance dominates overall performance.

Takeaway 2: Bank partitioning increases row-buffer locality and substantially improves PIM performance, especially for read-intensive PIM operations.

***Mitigating PIM Write Interference.*** Figure 4.11 shows the impact of mechanisms for write-intensive PIM operations. In this experiment, the most write-intensive operation, COPY, is executed by the PIMs and the mechanisms are applied only during the write phase of PIM execution. Stochastic issue is used with two probabilities, 1/4 and 1/16, which clearly shows the CPU-PIM performance tradeoff compared to next-rank prediction.

For stochastic issue, the tradeoff between CPU and PIM performance is clear. If PIMs issue with high probability, CPU performance degrades. The appropriate issue probability can be chosen with heuristics based on CPU

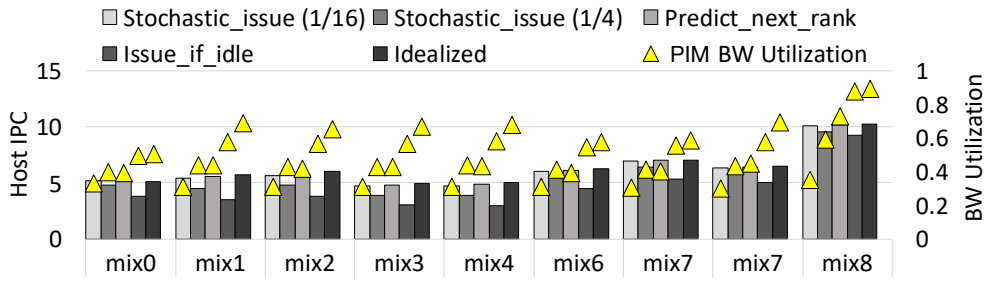



Figure 4.11: Stochastic issue and next-rank prediction impact.

memory intensity though I do not explore this in this chapter. On the other hand, the next-rank prediction mechanism shows slightly better behavior than the stochastic approach. Compared to stochastic issue with probability 1/16, both CPU and PIM performance are higher. Stochastic issue extends the tradeoff range and does not require signaling. I use the robust next-rank prediction approach for the rest of the chapter.

| Takeaway 3: Throttling PIM writes mitigates the large impact of read/write turnaround interference on CPU performance; next-rank prediction is robust and effective while stochastic issue does not require additional signaling. |
|---|

***Impact of Write-Intensity and Input Size.*** Figure 4.12 shows CPU and PIM performance when different types of PIM operations are executed with different input sizes. The CPU application mix with the highest memory intensity (mix1) and the next-rank prediction mechanism is used. In addition, to identify the impact of input size, three different vector sizes are used: small (8KB/rank), medium (128KB/rank), and large (8MB/rank). I evaluate asynchronous launches with the small vector size. I evaluate GEMV with three matrix sizes, where the number of columns is equal to each of the three vector

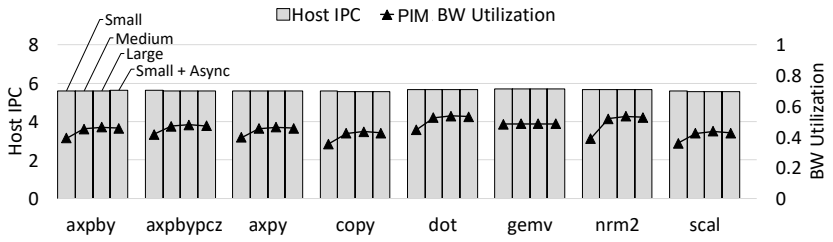



Figure 4.12: Impact of PIM operations and operand size.

sizes and the number of rows fixed at 128.

Overall, performance is inversely related to write intensity, and short execution time per launch results in low PIM performance. The NRM2 operation with the small input has the shortest execution time. Because of its short execution time, NRM2 is highly impacted by the launching overhead and load imbalance caused by concurrent CPU access. On the other hand, GEMV executes longer than other operations and it is impacted less by load imbalance and launching overhead. With the asynchronous launch optimization, the impact of load imbalance decreases and PIM bandwidth increases.

| Takeaway 4: Asynchronous launch mitigates the load imbalance caused by short-duration PIM operations. |
|---|

***Scalability Comparison.*** Figure 4.13 compares Chopim with the performance of rank partitioning (RP). For RP, I assume that ranks are evenly partitioned between the CPU and PIMs. Since read- and write-intensive PIM operations show different trends, I separate those two cases. Other application results (SVRG, CG, and SC) are shown to demonstrate that their performance falls between these two extreme cases. I do not evaluate SVRG with RP be-

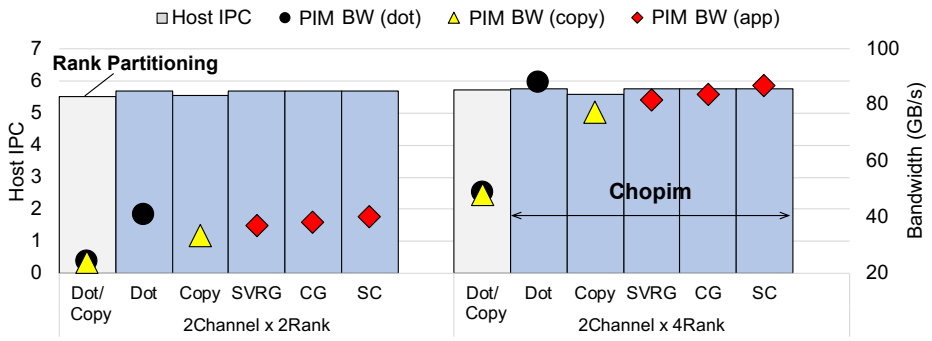



Figure 4.13: Scalability Chopim vs. rank partitioning.

cause it disallows sharing. I use the most memory-intensive mix1 as the CPU workload. The first cluster shows performance when the baseline DRAM system is used. For both the read- and write-intensive PIM workloads, Chopim performs better than rank partitioning. This shows that opportunistically exploiting idle rank bandwidth can be a better option than dedicating ranks for acceleration. The second cluster shows performance when the number of ranks is doubled. Compared to rank partitioning, Chopim shows better performance scalability. While PIM bandwidth with rank partitioning exactly doubles, Chopim more than doubles due to the increased idle time per rank. SVRG results fall between extreme DOT and COPY cases.

| Takeaway 5: Chopim scales better than rank partitioning because short issue opportunities grow with rank count. |
|---|

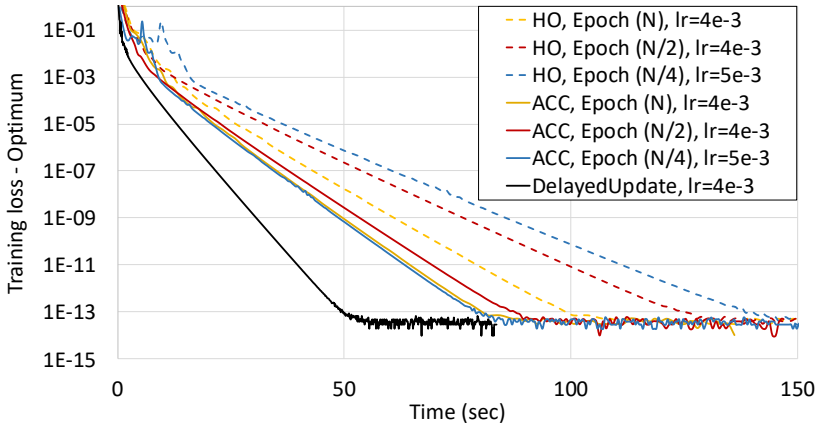
***SVRG Collaboration Benefits.*** Figure 4.14a shows the convergence results with and without PIM (8 PIMs). I use a shared memory region to enable concurrent access to the same data and the next-rank prediction mechanism is used. Compared to the CPU-only case, the optimal epoch size decreases from

$N$ to $N/4$ when PIMs are used. This is because the overhead of summarization decreases relative to the CPU-only case. Furthermore, SVRG with delayed updates gains additional performance demonstrating the benefits made possible by the concurrent CPU and PIM access when each processes the portion of the workload it is best suited for. Though the delayed update approach updates the correction term more frequently, the best performing learning rate is lower than the CPU-PIM ping-ponging approach (denoted as ACC) with epoch $N/4$, which shows the impact of staleness on the delayed update.

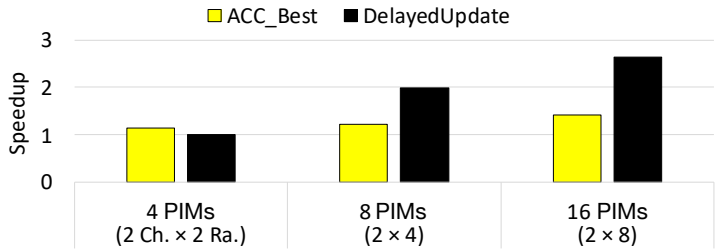
When PIM performance grows by adding PIMs (additional ranks), delayed-update SVRG demonstrates better performance scalability. Figure 4.14b compares the performance of the best-tuned serialized and delayed-update SVRG with that of CPU-only with different number of PIMs. I measure performance as the time it takes the training loss to converge (when it reaches $10_{-13}$ away from optimum). Because more PIMs can calculate the correction term faster, its staleness decreases, consequently, a higher learning rate with faster convergence is possible.

| Takeaway 6: Collaborative CPU-PIM processing on shared data speeds up SVRG logistic regression by 50%. |
|---|

***Memory Power.*** I estimate the power dissipation in the memory system under concurrent access. The theoretical maximum possible power of the memory system is 8W when only the CPU accesses memory. When the most memory-intensive application mixes are executed, the average power is 3.6W. The maximum power of PIMs is 3.7W and is dissipated when the scratchpad




(a) Convergence over time with and without PIM.




(b) PIM speedup scaling (normalized to CPU only).

Figure 4.14: Impact of PIM summarization in SVRG with and without delayed update (HO: CPU-Only, ACC: Accelerated with PIMs, ACC_Best: Best among all ACC options).

memory is maximally used in the average gradient computation. In total, up to 7.3W of power is dissipated in the memory system, which is lower than the maximum possible with CPU-only access. This power efficiency of PIMs comes from the low-energy internal memory accesses and because Chopim minimizes overheads.

| Takeaway 7: Operating multiple ranks for concurrent access does not increase memory power significantly. |
|---|

## 4.7 Related Work

To the best of my knowledge, this is the first work that proposes solutions for processing in memory while enabling the concurrent host and PIM access without data reorganization and in a non-packetized DRAM context. Packetized DRAM, while scalable, may suffer from 2–4x latency longer than DDR-based protocol even under very low or no load [57]. To solve this unique problem, many previous studies have influenced my work.

The study of processing in memory has been conducted in a wide range as the relative cost of data access becomes more and more expensive compared to the computation itself. The nearest place for computation is in DRAM cells [130, 94, 129] or the crossbar cells with emerging technologies [95, 28, 131, 133, 134, 138, 25, 100]. Since the benefit of PIM comes from high bandwidth and low data transfer energy, the benefit becomes larger as computation move closer to memory. However, area and power constraints are significant, restricting adding complex logic. As a result, workloads with simple ALU operations are the main target of these studies.

3D stacked memory devices enable more complex logic on the logic die and still exploit high internal memory bandwidth. Many recent studies are conducted based on this device to accelerate diverse applications [46, 79, 40, 9, 10, 54, 64, 65, 99, 118, 147, 45, 108, 62, 22, 96, 21]. Also, there are proposals about custom memory chip designs that embed processing elements to enable PIM [42, 49]. However, in these proposals, the main memory role of the memory devices has gained less attention compared to the acceleration part.

Some prior work [12, 139, 11, 20] attempts to support the host and PIM access to the same data but only with data reorganization and in a packetized DRAM context. Parrnaik et al. [118] show the potential of concurrently running both the host and PIMs on the same memory. However, they assume an idealized memory system in which there is no contention between PIM and host memory requests. We do not assume this ideal case. The main contributions of Chopim are precisely to provide mechanisms for mitigating interference.

On the other hand, *NDA* [44], Chameleon [15], and MCN DIMM [14] are based on conventional DIMM devices and changes the DRAM design to practically add PEs. Unlike rank partitioning and coarse-grain mode switching used in the prior work, we let host and PEs share ranks to maximize parallelism and partition banks to decrease contention.

## 4.8 Chapter Summary

In this chapter, I introduced solutions to share ranks and enable concurrent access between the host and PIMs. Instead of partitioning memory in coarse-grain manner, both temporally and spatially, I interleave accesses in fine-grain manner to leverage the unutilized rank bandwidth. To maximize bandwidth utilization, Chopim enables coordinating state between the memory controllers of the host and PIMs in low overhead, to reduce extra bank conflicts with bank partitioning, to efficiently block PIM write transactions with stochastic issue and next-rank prediction to mitigate the penalty of read/write turnaround time, and to have one data layout that allows the

host and PIMs to access the same data and realize high performance. My case study also shows that collaborative execution between the host and PIM can provide better performance than using just one of them at a time. Chopim offers insights to practically enable PIM while serving main memory requests in real systems and enables more effective acceleration by eliminating data copies and encouraging tighter host-PIM collaboration.

| System configuration | | |
|---|---|---|
| Processor | 4-core OoO x86 (8 cores for mix0), 4GHz, Fetch/Issue width (8), LSQ (64), ROB (224) | |
| PIM | one PE per chip, 1.2GHz, fully pipelined, write buffer (128) | |
| TLB | I-TLB:64, D-TLB:64, Associativity (4) | |
| L1 | 32KB, Associativity (L1I: 8, L1D: 8), LRU, 12 MSHRs | |
| L2 | 256KB, Associativity (4), LRU, 12 MSHRs | |
| LLC | 8MB, Associativity (16), LRU, 48 MSHRs, Stride prefetcher | |
| DRAM | DDR4, 1.2GHz, 8Gb, x8, 2channels × 2ranks, FR-FCFS, 32-entry RD/WR queue, Open policy, Intel Skylake address mapping [121] | |
| DRAM timing parameters | | |
| tBL=4, tCCDS=4, tCCDL=6, tRTRS=2, tCL=16, tRCD=16, tRP=16, tCWL=12, tRAS=39, tRC=55, tRTP=9, tWTRS=3, tWTRL=9, tWR=18, tRRDS=4, tRRDL=6, tFAW=26 | | |
| Energy Components | | |
| Activate energy: 1.0nJ, PE read/write energy: 11.3pJ/b, CPU read/write energy: 25.7pJ/b, PE FMA: 20pJ/operation, PE buffer dynamic: 20pJ/access, PE buffer leakage power: 11mW (Energy/power of scratchpad memory is same as PE buffer) | | |
| Benchmarks | | MPKI |
| mix0 | mcf_r:lbm_r:omnetpp_r:gemsFDTD bwaves:milc:soplex:leslie3d | H:H:H:H H:M:M:M |
| mix1 | mcf_r:lbm_r:omnetpp_r:gemsFDTD | H:H:H:H |
| mix2 | mcf_r:lbm_r:gemsFDTD:soplex | H:H:H:H |
| mix3 | lbm_r:omnetpp_r:gemsFDTD:soplex | H:H:H:H |
| mix4 | omnetpp_r:gemsFDTD:soplex:milc | H:H:H:M |
| mix5 | gemsFDTD:soplex:milc:bwaves_r | H:H:M:M |
| mix6 | soplex:milc:bwaves_r:leslie3d | H:M:M:M |
| mix7 | milc:bwaves_r:astar:cactusBSSN_r | M:M:M:M |
| mix8 | leslie3d:leela_r:deepsjeng_r:xchange2_r | M:L:L:L |
| PIM Kernels | | |
| PIM basic operations (Table 4.1), SVRG (50K × 3072), CG (16K × 16K), and SC (2M × 128) | | |
| Machine Learning Configurations | | |
| Logistic regression with $\ell 2$-regularization (10-class classification), $\lambda$=1e-3, learning rate=best-tuned, momentum=0.9, dataset=cifar10 (50000 × 3072) | | |

Table 4.2: Evaluation parameters.

# Chapter 5

# Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators

In this chapter [1], I introduce another compelling use case of main-memory accelerators in the AI/ML domain. With the evolution of deep learning (DL), artificial intelligence is being widely used in many internet services. I describe a new approach for reducing the latency of such DL inference tasks by accelerating their fully-connected layers with a *processing in/near memory* (PIM) approach. Park et al. [116] report that for important personalized recommendation and natural language DL inference workloads, a large fraction of DL-related data-center cycles (42%) are spent executing fully-connected (FC) layers in Facebook data centers.

FC layers are executed as matrix-matrix multiplication operations (commonly referred to as *GEMM* kernels) and these GEMMs dominate the overall execution time of some workloads [116, 55]. GEMMs are commonly considered compute rather than bandwidth bound based on decades of scientific-computing and DL training experience. However, I observe that DL inference GEMMs exhibit two unique traits that leave them memory-bandwidth bound

[1] Portions of this chapter have been previously published as

in many cases, and thus amenable to PIM acceleration.

First, DL inference queries require small-batch execution to meet tight latency constraints, leading to very tall/skinny or short/fat activation matrices. Such matrices offer lower locality, increasing the importance of memory bandwidth. Second, some recommender and language models have billions of parameters (across numerous layers) and it is common for multiple models to be colocated on a single node to improve system efficiency and reduce multi-model query latency [56, 77, 151, 111]. As a result, it is common for the larger weight matrices to reside only in main memory, stressing the memory channel when executing on a CPU and often requiring low-bandwidth host-device transfers in systems with accelerators.

My experiments demonstrate that these GEMM operations are in fact bandwidth-bound on both CPU and GPU systems, and describe how they can be accelerated with processing in/near *main* memory (PIM).

I present *StepStone PIM*, which is integrated within the CPU main memory system and solves the dual challenges of utilizing available GEMM locality and sharing data with the CPU under its sophisticated XOR-based DRAM address mapping scheme. Hence, StepStone is an appealing datacenter solution because it: (1) better utilizes bandwidth within the memory system; (2) utilizes locality, enabling high performance and efficiency for datacenter DL inference GEMM operations; (3) does not require additional memory devices or capacity, avoiding the exorbitant cost of additional memory and taking advantage of the already-memory resident matrices; and (4) offloads a low-

performance workload from the CPU, freeing additional execution capacity for colocated tasks.

This unique set of StepStone capabilities is, to the best of my knowledge, not available in any prior PIM architecture and research, including in recent work that targets datacenter DL inference or processing in main memory. While recent work explored PIM-acceleration for datacenter DL inference, it focuses on the embedding layers of DL-inference [89, 77] rather than on the MLP GEMM operations, which require a different approach for exploiting locality. Prior work that considers integrating PIM accelerators within main memory either requires costly data replication to avoid the DRAM address mapping challenge [44, 15, 14] or does not offer the mechanisms to exploit GEMM locality [10, 80, 77, 29].

I choose a straight-forward PIM microarchitecture for StepStone that follows recent research trends. My contributions instead lie with four key innovations. The first is the StepStone PIM GEMM parallelization and execution flow that is cognizant of the XOR-based DRAM address mapping that otherwise break GEMM locality. The second contribution accelerates the localization and reduction operations of the execution flow without consuming CPU core resources. The third contribution enables long-running locality-conserving PIM GEMM kernels with the new StepStone memory-side address generation logic. Long-running kernels relieve PIM pressure on the memory command channel, enabling high-performance colocated CPU tasks.

The fourth contribution is identifying and exploiting a new tradeoff op-

portunity in balancing the performance benefits of parallelization across fine-grained PIM units (PIMs) within DRAM with the data-transfer overheads of the localization/replication and reduction operations necessary for high parallelization. I explore this tradeoff by evaluating channel-, device-, and bank group-level StepStone PIMs.

To summarize my contributions:

- I identify and demonstrate that small-batch GEMM operations of DL datacenter inference workloads are bandwidth bound on CPUs and GPUs, and can hence benefit from PIM-acceleration (Section 5.1).
- I develop the novel StepStone PIM GEMM execution flow that is cognizant of the complex CPU address mapping, thus exploiting GEMM locality and improving performance by $35-55\%$ over a prior PIM architecture that supports complex address mappings [29].
- I accelerate the localization and reduction operations of my new GEMM flow at the CPU memory controller to improve performance by up to an additional 40%.
- I design the novel memory-side StepStone address generator that enables long-running GEMM kernels to minimize command-channel usage, which improves PIM performance by 5.5× when the CPU executes concurrent memory-intensive tasks.

- I identify a new tradeoff opportunity in determining whether to target channel-, device-, or bank group-level PIMs and show benefits of up to 35% in exploiting it.

- I present a detailed StepStone PIM evaluation, including end-to-end performance analysis and conclude that StepStone is an appealing datacenter solution because of its low cost (no additional memory devices or capacity), its potential for lower latency and higher throughput, and its ability to dynamically support the execution of larger-batch and colocated tasks on the CPU.

Combining all my innovative mechanisms, StepStone is able to substantially outperform a CPU when executing GEMM operations on matrices with dimensions typical in datacenter DL inference workloads: (1) StepStone offers 12× lower minimum GEMM latency for these matrices; (2) 77× higher throughput under the strictest latency constraints that correspond to batch-1 on the CPU but if the CPU is allowed 20% additional latency for batch-32 execution, the performance benefit drops to 2.8×; and (3) up to 16× lower end-to-end DL inference latency compared to measured CPU performance.

## 5.1 Motivation and Challenges

***Bandwidth-bound GEMMs.*** Matrix-matrix multiplication (GEMM) is commonly regarded as compute bound. However, I observe that GEMM becomes bandwidth-bound and exhibits low CPU/GPU utilization when both:

(1) one of the two input matrices is much larger than the other (e.g., A is large while B is "tall and skinny") and (2) the large input matrix is in main memory. While rare in traditional linear algebra applications, DL inference tasks in datacenters often meet both conditions.

First, DL inference queries have tight latency constraints that require small batches [116]. The corresponding GEMM operations in fully-connected layers therefore multiply a large weight matrix and a small input matrix. Second, the MLP weights are often only found in main memory because either the total size of the MLP parameters exceeds cache capacity (e.g., in recent language models [23, 39, 124]) and/or multiple models are colocated on a single node [56].

The resulting matrix sizes (Table 5.1) are executed inefficiently on CPUs and GPUs as shown by the roofline analysis presented in Figure 5.1. Each point in the figure corresponds to the performance measured on a 2.7 GHz 28-core Intel Cascade Lake Xeon CPU or an NVIDIA Titan Xp GPU

Table 5.1: Common DL-inference GEMM dimensions.

| | Model | Description | Weights | Batch Size |
|---|---|---|---|---|
| LM | BERT | MLP | 1024×4096 | 1-8 [116] |
| | | MLP | 4096×1024 | |
| | | Projection | 1024×1024 | |
| | GPT2 | MLP | 1600×6400 | |
| | | MLP | 6400×1600 | |
| | | Projection | 1600×1600 | |
| RM | DLRM (RM3) | Bottom MLP | 2560×512 | 1-256 [116] |
| | | Bottom MLP | 512×32 | |
| | | Top MLP | 512×128 | |
| | | Top MLP | 128×1 | |

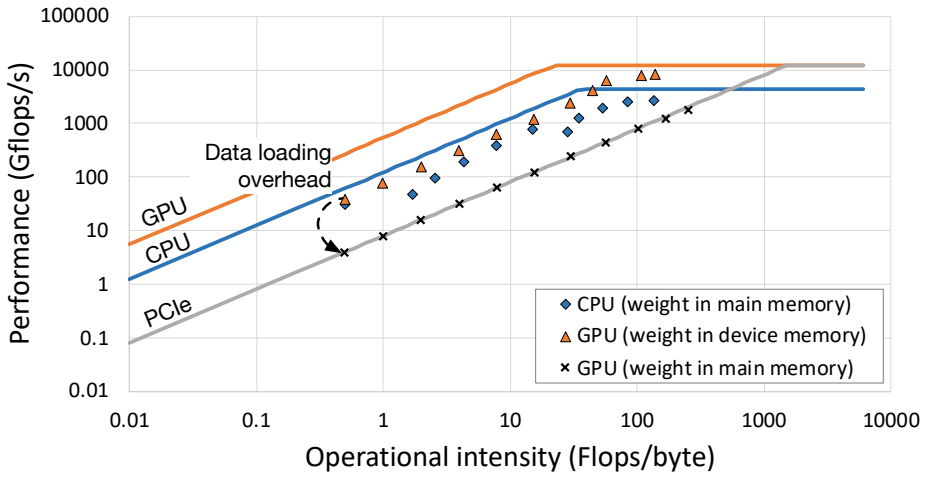



Figure 5.1: CPU (Intel Xeon Platinum 8280) and GPU (NVIDIA Titan XP) roofline modeling when executing bandwidth-bound GEMM operations of a memory-resident $1024 \times 4096$ weight matrix with a $4096 \times N$ matrix; $N$ is swept from $1 - 1024$ in powers of 2 moving from left to right.

when multiplying a memory-resident $1024 \times 4096$ matrix by a cache-resident $4096 \times N$ matrix, where $N$ represents the batch size. The left-most point for each system is when $N = 1$ and each point moving right represents a doubling of $N$. I observe that all three systems are bandwidth bound for inference-appropriate batch sizes ($N \lesssim 32$). Further, for such small batches, GPU performance is lower than the CPU if matrix A is in host memory because of the slow host-device bus.

*I conclude that processing in/near memory (PIM) is appealing for these GEMM operations of datacenter DL-inference workloads.*

**PIM GEMMs with XOR-based DRAM address mapping.** I target systems in which main memory is PIM enabled, implying a shared DRAM ad-

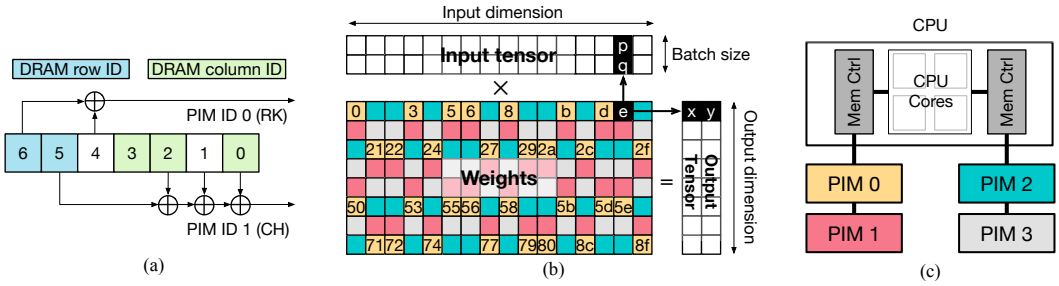



Figure 5.2: An example of bandwidth-bound GEMM operation with PIM and a toy XOR-based address mapping: (a) toy XOR-based physical-to-DRAM address mapping where addresses refer to contiguous row-major matrix elements; (b) layout of an 8 × 16 matrix with colors indicating element→PIM unit mapping; (c) example system with rank-level PIMs.

dress space with the CPU. The CPU relies on sophisticated XOR-based DRAM address mappings to exploit bank and channel parallelism by distributing consecutive cache blocks in the physical address space (PA) across DRAM banks and channels. As a result, matrices that are contiguous in the application virtual space are distributed across PIM units (PIMs) in complex patterns. Effective GEMM execution requires exploiting locality and reuse in matrix blocks, which are challenging to identify.

Figure 5.2 illustrates this challenge for the toy address mapping of Figure 5.2a targeting a system with 4 PIM units (one per rank). Addresses refer to elements of the large matrix shown in Figure 5.2b, which is laid out row major in contiguous memory addresses. Logical blocks of the matrix do not form blocks within each PIM. For example, element `0e` of the weight matrix (marked in black) is mapped to PIM0 and is multiplied by elements `p` and `q` from the input tensor to modify elements `x` and `y` of the output tensor. These

same elements of the input tensor are also needed when reading element `5e` of the weight matrix and the same two output-tensor elements when reading weights `00`, `03`, `05`, `06`, `08`, `0b`, and `0d`. Utilizing this locality requires the PIMs to correctly map between contiguous DRAM addresses within each PIM and the corresponding addresses of the input and output tensors.

Prior approaches to this challenge fall into one of three categories. The first avoids the challenge altogether by maintaining a copy of the data that is stored in a PIM-friendly layout and not accessed by the CPU [59, 79, 89]. This either duplicates substantial data arrays (possibly $>$ 100GiB) [23, 124, 109] or prevents the CPU from assisting with requests that can tolerate higher response latency [55]. Furthermore, a different layout is needed for channel-, device-, and bank group-level PIMs. This either forces even more replicas or prohibits optimizations that dynamically choose the PIM level based on input characteristics (e.g., as in the XLM language model [31]).

The second approach requires the CPU to transfer this correspondence information to the PIMs for each cache block the PIM processes [10, 80]. The CPU sends special DRAM command packets that include operand and opcode information to each PIM unit and all the transactions related to PIM execution are controlled by the host. PIMs are isolated from the address mapping concerns, but performance scalability is poor because: (1) channel bandwidth for sending PIM command packets saturates rapidly, (2) CPU resources are required for address generation, and (3) the frequent PIM command packets severely interfere with CPU memory traffic [29].

The third approach, proposed by Cho et al. [29], aligns long vector PIM operands in memory such that all kernel operands follow the same interleaving pattern after the XOR address mapping. In this way, both the CPU and the vector-oriented PIM can process the same data. However, this vector-oriented approach cannot exploit the GEMM kernel locality. Vector-oriented execution splits the GEMM into multiple matrix-vector (GEMV) operations, requiring a larger number of kernel invocations. The straightforward implementation also requires copies across PIM units to ensure all data is local. The standalone (non main-memory) Newton PIM accelerator [59] also follows this approach. I observe that a different execution flow can be used to block both the input and output matrices to reduce copy overhead. I explain my StepStone PIM GEMM in the following section.

## 5.2 StepStone PIM

*StepStone PIM* enables independent GEMM execution with PIMs under any XOR-based memory-system address mapping. In StepStone PIM, the weight matrix is partitioned and assigned to PIMs based on the underlying address mapping, maintaining internal contiguity and enabling temporal locality when each PIM unit works on its GEMM blocks. From the CPU perspective, the PIMs appear to skip within the address space and only step on those "stones" (i.e. cache blocks) that are physically local to them.

### 5.2.1 StepStone Architecture

The StepStone PIM architecture is, for the most part, a standard PIM. The innovation lies in how I map GEMM operations and in my unique address-generation algorithm, both discussed later in this section. StepStone is comprised of a host-side PIM controller that interfaces with PIM units (PIMs) through the memory channel to control their operation using memory-mapped PIM-side registers. As shown in Figure 5.3a, PIM units (PIMs) can be integrated with: (1) DRAM itself, e.g., at the bank-group level (StepStone-BG); (2) with a memory module, e.g., associated with each DRAM device or buffer chip (StepStone-DV);[2] and/or (3) with a memory channel controller (StepStone-CH). I consider all three integration levels. Note that StepStone-BG accounts for device-level timing parameters such as tRCD and tFAW using control logic at the I/O port of each device.

Each PIM unit (Figure 5.3b) includes SIMD/vector units, scratchpad memory, interfaces, control logic to execute the GEMM kernel command (sub-GEMM to be more precise), and a local-address generation unit. The pipeline is sufficiently deep to hide address generation and access latencies (20 stages in my case). When $N$ (e.g., the batch dimension) is large, performance is bound by the SIMD width. While wide SIMD units are desirable, arithmetic performance must be balanced with area and power constraints.

Following prior work, I aim for $0.15mm^2$ for each StepStone-BG unit [53]

[2]This is a cheap rank-level PIM with no inter-device communication.

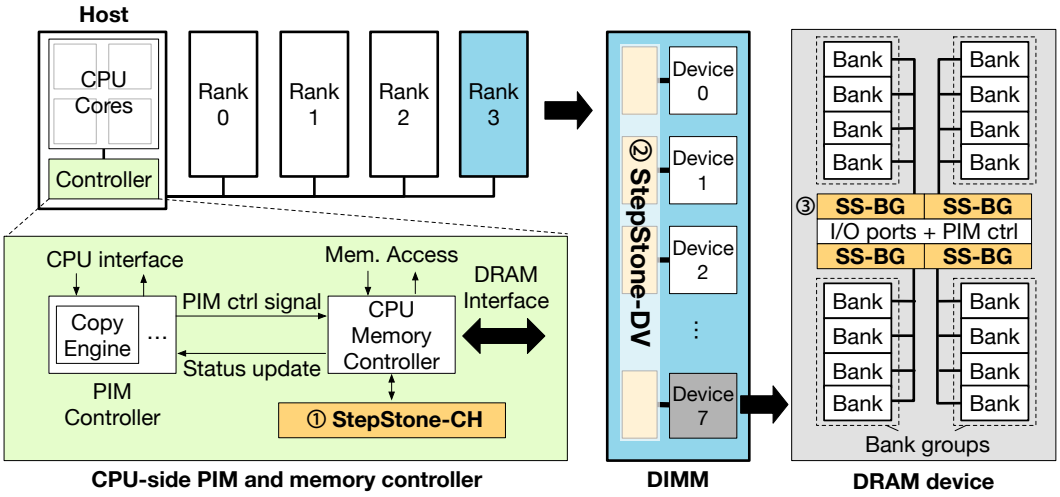
and $1.2mm^2$ for StepStone-DV [15]. I aim for a 2 : 1 ratio between SIMD area and scratchpad area and assume additional logic is of comparable size to the scratchpad. I estimate functional unit and scratchpad area and power at the device-level with the values reported for iPIM [53] and at the device and channel levels following the methodology of Lym et al. [102]. This analysis yields nominal values of 8-wide SIMD with 8KB scratchpad capacity for each StepStone-BG unit (4 PIMs per DRAM device), and 32-wide SIMD with 32KB scratchpad capacity per StepStone-DV PIM unit. For StepStone-CH, I keep the same bandwidth to arithmetic performance ratio as StepStone-DV: 256-wide SIMD units and 256KB scratchpad capacity. I consider all three cases and conclude that StepStone-CH offers the lowest performance and requires the largest die area.

One other component is the replication/reduction unit within the PIM controller, which is used to accelerate the distribution of matrix B and reduction of partial values of C, which are required for the GEMM execution described below.

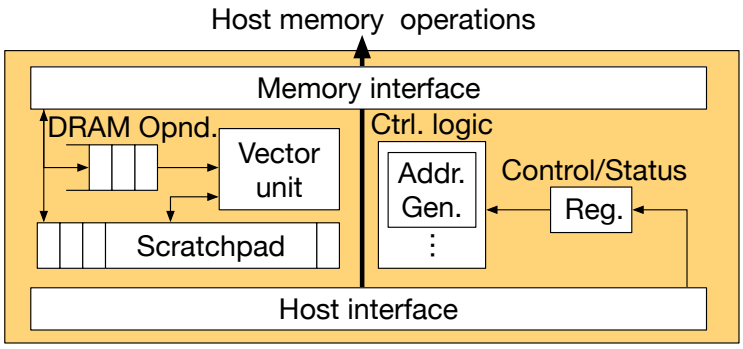
### 5.2.2 StepStone GEMM Execution

GEMM execution starts with the large weight matrix A stored contiguously in the virtual and physical address spaces in row-major order.[3] Therefore, A is distributed across memory devices based on the DRAM address

[3]I assume that the matrix dimensions are powers of two; matrices with non-power-of-two dimensions are either padded or execution is partitioned/serialized into smaller, power-of-two matrices.



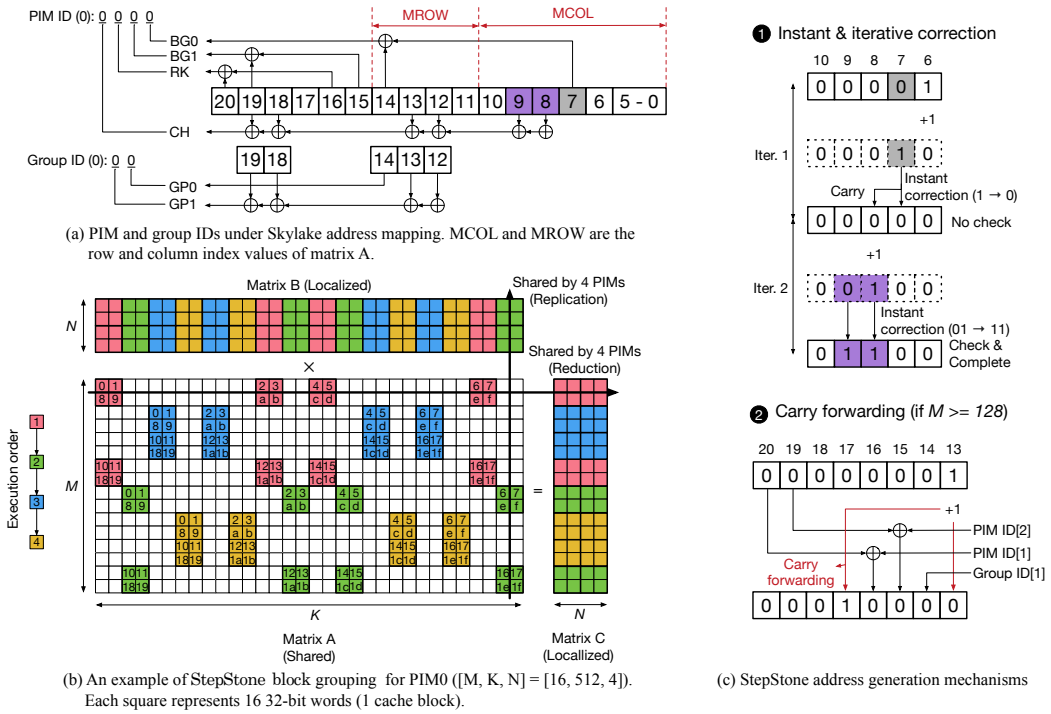

(a) Baseline PIM system.




(b) StepStone PIM architecture.

Figure 5.3: Overview of the StepStone PIM System.

mapping as shown in Figure 5.4 (for the Intel Skylake address mapping [121] on StepStone-BG and depicting only the elements of A that map to PIM0, which I refer to as partition A0). A0 must be multiplied with elements of B and accumulated into elements of C. To maximize parallelism, I first localize private copies of B and C into each PIM unit, also shown in the figure. Localizing data copies the data into a pre-allocated per PIM-unit memory region.




(a) PIM and group IDs under Skylake address mapping. MCOL and MROW are the row and column index values of matrix A.

(b) An example of StepStone block grouping for PIM0 ([M, K, N] = [16, 512, 4]). Each square represents 16 32-bit words (1 cache block).

(c) StepStone address generation mechanisms

Figure 5.4: Overview of GEMM execution with StepStone PIM.

Execution then proceeds with a partial dot product across rows of A0 with columns of B (B is shown transposed in the figure).

However, recall the dual challenges of identifying the corresponding indices in B and C as A0 is advanced while maximizing reuse during execution. I address these challenges grouping together cacheline-sized memory blocks ("cache blocks") into *block groups* that follow the same DRAM address mapping distribution. I note that the grouping depends both on the address mapping and the matrix dimension. Each block group is shaded a different color in Figure 5.4b.

**StepStone locality.** To maximize reuse, each element of B should be multiplied with as many elements of A before overwritten in the buffer. I achieve this by executing one block group at a time: cache blocks within each group across rows reuse elements of C while those along columns reuse elements of B. No reuse exists between groups.

The number of groups required to maximize locality is determined by the number of PIM ID bits that are impacted by addresses within the matrix. For example, the matrix in Figure 5.4 is 16×512 4B words and starts at physical address 0, thus locations within this matrix span the lower 15 address bits. Within these bits, bits 7 and 14 determine one bank-group bit (BG0, which is also PIM ID bit 0) and bits 8, 9, 12, and 13 affect the channel bit (PIM ID3). The other PIM ID bits are fixed for all locations within this matrix. I further note that a group spans entire rows to maximize locality. I therefore exclude address bits associated with matrix columns (MCOL) from defining the group ID bits (GP0 and GP1 in the figure).

**Localizing matrices B and C.** Matrix B is (partially) replicated and localized to the different PIMs before execution begins, and the localized partial-result C matrices are reduced after the GEMM. The replication and reduction, along with data reorganization for spatial locality within each PIM unit are handled by the host and accelerated with a simple DMA engine at the PIM controller. The operation of this engine is illustrated in Figure 5.5 for localizing matrix B for a portion of matrix A that is distributed across PIMs 0, 1, 8, and 9. Matrix B is again represented transposed in the figure and consecutive

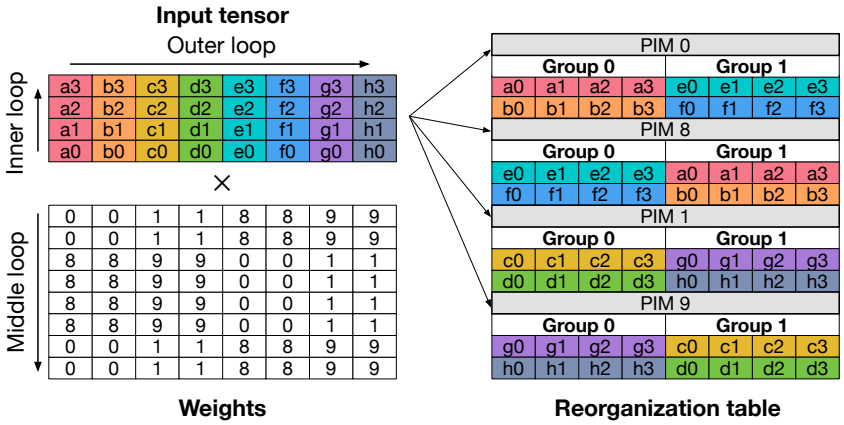



Figure 5.5: Input-matrix Reorganization.

elements in each of its rows appear as columns (e.g., a0 - a3).

During localization, the engine reorganizes the input matrix for each PIM unit such that accesses are sequential during its group-based execution. The outer-most loop iterates over columns of A, localizing the rows of B (appear as columns in $B^T$) needed for each column in the PIMs and block groups it maps to. The PIM and group IDs are computed based on the mappings illustrated in Figure 5.4a. Each cache block of B is read once and then copied to all its relevant PIM-local addresses. Reductions follow a similar execution flow.

### 5.2.3 Overall Execution Flow of StepStone GEMM

The execution flow of complete GEMM is shown in Algorithm 1. After localization, the input matrices are all aligned in the DRAM accessible by each PIM unit and execution proceeds in group order. A slight complication arises when A is very large such that not all elements of B and C that correspond

to a group can be buffered within the scratchpad. In such cases, to still utilize locality, I further block each group. This blocking can be across rows, maximizing reuse of C, and/or across columns. I process blocks of rows first because C offers greater reuse as it is both read and written.

The inner-most GEMM call is coarse-grained for the full StepStone PIM with the mapping-aware AGEN unit, but is split into multiple dot product operations without this innovative AGEN logic. In this way I can isolate the contributions of my algorithm mapping and hardware mechanism when evaluating StepStone PIM compared to prior PIM architectures, like Chopim [29] (I denote my StepStone GEMM flow on Chopim as *enhanced Chopim* or eCHO).

Note that address generation with partitioning is slightly different than as described for unpartitioned groups execution. When crossing different column partitions (groups of columns that partition a row into multiple blocks), address generation must skip over those columns belonging to different partitions. This is simple to do and only requires modifying the address-generation rules to account for group and partition ID.

### 5.2.4 StepStone Address Generation

Within a single cache block, the address is a simple increment, but once the value of a bit that determines the PIM ID is modified, that contiguous physical address must be skipped until the next physical address that maps to the same PIM unit and block group is reached. A simple iterative approach of incrementing the address until the address is again within this same block and

PIM ID, the number of iterations required when the number of PIMs is large introduces bubbles into the execution pipeline and degrades performance.[4]

I propose new *increment-correct-and-check* AGEN logic that skips to the next closest address with the same PIM and group IDs (after the simple increment falls outside the target IDs). I do this by ensuring that those address bits that are XORed to determine each ID bit always maintain their parity. I can thus skip incrementing bits that are lower than the lowest ID-affecting address bit. The AGEN logic iterates over the ID-affecting bits (from LSB to MSB), each time incrementing the next ID-affecting bit and checking whether the PIM and group IDs match their target values.

The number of iterations is limited to the number of ID-affecting bits, but can be further reduced with two additional rules. The first rule applies for adjacent address bits that both affect the same ID bit. When the lower of the two is incremented, the upper must be as well to maintain parity. This can be done directly, saving one iteration. The second rule applies for chains of contiguous address bits that each affect a different ID bit. In this case, when the first is incremented, the carry it propagates will have to corrected in multiple iterations to maintain the parity corresponding to each bit in the chain. Thus, the chain can be skipped with the carry simply directly propagated to the next-higher address bit. These rules are shown in Figure 5.4c. The top part of the figure illustrates the first rule of instantly correcting from `01` to `11`,

[4]I assume that the CPU address mapping is available for PIMs either by reverse engineering, by CPU vendors building the PIMs, or by agreement.

and the bottom part illustrates the second rule of forwarding the carry from bit 13 to 17 since bit 14-16 affect different ID bits.

**Algorithm 1:** Group-based GEMM with partitioning.

```
localize(B);
localize(C);
for rpart in row_partitions do
    buffer_fill(C);
    for grp in block_groups do
        for cpart in col_partitions do
            buffer_fill(B);
            if StepStone then
                StepStone_GEMM;
            end
            else if eCHO then
                for row in cpart do
                    DOT(row);
                end
            end
        end
    end
    buffer_drain(C);
end
reduce(C);
```

### 5.2.5 Optimizations

Multiple optimizations over the basic flow described above are possible, including fusing multiple kernel executions for matrices that are not powers of two, balancing parallel execution with the overheads of localization and reduction, and choosing a PIM level for execution (StepStone-BG vs. StepStone-DV). For brevity, I only discuss the latter two.

**Choosing the PIM level.** Bank-group level StepStone-BG offers the highest potential performance when the GEMM is severely bandwidth bound (very small batches) because it accesses underutilized bandwidth within a DRAM device. An interesting tradeoff emerges as bandwidth constraints decrease with somewhat larger batches. The arithmetic intensity (data resuse per operation) in StepStone scales with the batch size ($N$) up to the SIMD width of each PIM unit. This results in comparable arithmetic execution times for $1 \leq N \leq 16$ in StepStone-BG and for $1 \leq N \leq 32$ in StepStone-DV (though obviously the execution times differ between the two PIM levels). At the same time, the overheads of localization and reduction increase with $N$ and with the number of PIMs (number of block groups).

An optimization opportunity emerges for choosing between bank-group and device level PIMs as a result. The best PIM level depends on the matrix dimensions and the address mapping as these determine the number of block groups, and hence the localization and reduction overheads. I demonstrate this tradeoff in Section 5.4 and generally find that StepStone-BG is best when $N \leq 16$. Note that I do not discuss the algorithm for choosing the PIM level, but note that a simple heuristic that estimates execution times and overheads based on available bandwidth and transferred data volumes works well.

**Small weight matrices.** A similar tradeoff between arithmetic performance and localization and reduction overhead exists when the matrices are relatively small. In such cases, it is preferable to only use a subset of the available PIMs, trading off lower arithmetic performance for reduced overheads. I show that

this optimization can yield a $\sim 25\%$ performance improvement in some cases (Section 5.4.4). Another optimization for relatively small matrix A is that CPU can directly write to and read from PIM's scratchpad memory when the matrix B and C fits in it. This reduces the time to move data between DRAM and the scratchpad memory.

Optimizing execution to only utilize a subset of the PIMs comes with additional considerations when allocating memory for the large distributed input matrix A. Specifically, A must remain contiguous in virtual memory yet be mapped to just a subset of the PEs. Because the address mapping and size of the matrix is known, it is possible to allocated physical memory frames to satisfy this mapping constraint as long as the PIM ID bits that are ignored (for subsetting) are not affected by virtual-address offset bits. In other words, this is possible with base pages only (e.g., 4KB pages). Enforcing the mapping constraints to maintain alignment with the PIMs can be done using the proposed coloring interface introduced by Cho et al. [29] and by modifying the application's memory allocator.

For example, if the goal is to execute on half the PIMs of StepStone-BG with the Skylake mapping, I keep BG0 fixed for the entire physical allocation of A. This is achieved by allocating virtual memory at 32KB granularity rather than the minimum 4KB granularity. Additionally, I must ensure that contiguous virtual addresses remain aligned in the DRAM space and therefore must also ensure that the other PIM ID bits follow a consistent mapping. I do that by coloring those bits in a way that the OS-level frame allocator maintains

alignment, as proposed for Chopim [29].

## 5.3 Methodology

**System configuration.** Table 5.2 summarizes my system configuration and DRAM timing parameters. My DRAM model is based on the DDR4 model of the Ramulator detailed DRAM simulator [85]. I implement StepStone-CH, StepStone-DV, and StepStone-BG PIMs inside Ramulator with detailed pipeline and timing models. I emulate my memory allocator and add an address translation engine into the PIM controller on the CPU (5.2.1); address translation is infrequent (once per coarse-grained PIM command) because contiguous physical regions are allocated for PIM execution. To validate my execution flow, I modify Ramulator to read from and write values to memory and check the final output against pre-calculated results. I also compare all addresses from the StepStone AGEN logic with a pre-generated address trace for each PIM. For all the GEMM operations with StepStone PIM, I assume the input activations reside in the CPU caches and are therefore localized to the active PIMs. In the same way, I assume the final GEMM results are reduced by the CPU.

I use the XOR-based address mappings described in DRAMA [121], acquired by reverse-engineering Intel and Samsung CPUs. To show the impact of address mapping on the same DDR4 model, I modify the address mapping of Exynos, Dual-socket Haswell, Ivy Bridge, and Sandy Bridge based on the randomized method (PAE) proposed by Liu et al. [98]. By default I use Skylake's

address mapping. To measure GEMM performance on real machines, I use an Intel Xeon Platinum 8280 (CPU) with Intel oneDNN [2] and an NVIDIA TitanXP (GPU) with CUTLASS [4].

The area and power of the SIMD units are estimated based on Lym et al. [102] for StepStone-DV and StepStone-CH and iPIM [53] for the SIMD units of StepStone-BG. I use Cacti 6.5 [106] to estimate the area and power of scratchpad memory.

**Workloads.** I choose matrix sizes and aspect ratios to clearly show their impact on performance. By default, I use 1024 × 4096. For end-to-end performance evaluation, I use the 4 different DL models summarized in Table 5.2. Since my goal is to accelerate GEMMs with small batch sizes, I vary the batch size from 1 to 32. I refer to input and output activations as matrix B and C, respectively, and I refer to the largest matrix as weight matrix A.

I demonstrate the benefits of long-running kernels for concurrent CPU and PIM execution in a colocation scenario of the default 1024 × 4096 GEMM with a mix of the mcf, lbm, omnetpp, and gemsFDTD from SPEC CPU 2017 [115]. The CPU applications are modeled with gem5 [19] (4 OOO x86 4GHz cores with fetch/issue width of 8, a 64-deep LSQ, and a 224-entry ROB), similarly to [29].

**Comparisons.** I compare my approach with two existing PIM approaches, PEI [10] and Chopim [29], which are capable of accelerating GEMMs in main memory and leveraging multi-level PIMs with one data layout. To make a fair

comparison, I use my baseline PIM system (Figure 5.3) for all approaches and only vary localization/reduction mechanisms and PIM kernel granularity. For PEI, each PIM instruction is used to process one cache block and all the other operands needed for executing the PIM instruction is written to scratchpad memory by the CPU. I evaluate two versions of Chopim. The baseline naive Chopim (nCHO) follows the GEMV mapping approach (Section 5.1). I also use my newly-developed StepStone flow with an "enahnced" Chopim (eCHO). This eCHO configuration exploits locality as well as StepStone PIM, but requires more frequent kernel calls (Algorithm 1) and does not use accelerated localization and reduction operations.

## 5.4 Evaluation Results

In this section, I demonstrate the throughput and latency benefits of StepStone over either a CPU or GPU, evaluate the impact of address mapping and scratchpad capacity, and analyze the tradeoff between arithmetic performance and overheads as the number of active PIM units (PIMs) is varied.

### 5.4.1 StepStone PIM Performance Benefits

I first compare the performance of StepStone PIM to a 2.7GHz 28-core Intel Xeon Platinum 8280 Cascade Lake CPU with a representative 1024 $\times$ 4096 weight matrix (Figure 5.6). StepStone PIM offers tremendous benefits for latency and throughput targets. When considering the minimum latency batch-1 execution, StepStone-BG with a PIM unit per bank group offers far

Table 5.2: Evaluation parameters.

| PIM configurations | | |
|---|---|---|
| StepStone-BG | 8-width SIMD, 8KB scratchpad (per DRAM device), 1.2GHz | |
| StepStone-DV | 32-width SIMD, 32KB scratchpad (per buffer chip), 1.2GHz | |
| StepStone-CH | 256-width SIMD, 256KB scratchpad (per channel), 1.2GHz | |
| Address mappings | | |
| ID = 0 | Exynos-like address mapping (modified) | |
| 1 | Haswell-like address mapping (modified) | |
| 2 | Ivy Bridge-like address mapping (modified) | |
| 3 | Sandy Bridge-like address mapping (modified) | |
| 4 | Skylake address mapping (baseline) | |
| DRAM timing parameters (DDR4-2400R, 4GB, x8) | | |
| tBL=4, tCCDS=4, tCCDL=6, tRTRS=2, tCL=16, tRCD=16, tRP=16, tCWL=12, tRAS=39, tRC=55, tRTP=9, tWTRS=3, tWTRL=9, tWR=18, tRRDS=4, tRRDL=6, tFAW=26 | | |
| Energy components | | |
| In-device RD/WR (11.3pJ/b), Off-chip RD/WR(25.7pJ/b)s, CH/DV/BG SIMD (11.3,11.3,11.3nJ/op), CH/DV/BG Scratchpad (0.03/0.1/0.3 nJ/access) | | |
| DL inference parameters | | |
| RM | DLRM [109] | RM3, Bottom MLP (2560-512-32), Top MLP (512-128-1), bsz=4 |
| LM | BERT [39] | Text classification (WNLI), 24 transformer blocks, MLP (1024-4096-1024), seq. length=8, bsz=4 #attention heads=16 |
| | GPT2 [124] | Text generation, 48 transformer blocks, MLP (1600-6400-1600), seq. length=8, bsz=4 |
| | XLM [31] | Text generation, 12 transformer blocks, MLP (2048-8192-2048), seq. length=8, bsz=4 |

superior latency: 2.8× better than the device-level StepStone-DV and 12× better than the CPU.

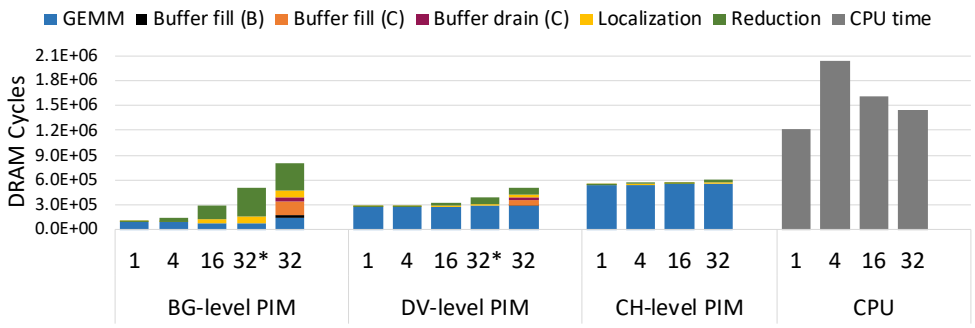



Figure 5.6: GEMM Latency comparison between different PIM options of StepStone PIM and the CPU. The configurations with relaxed area constraints are labeled with * (i.e. enough ALUs and large enough scratchpad memory).

Alternatively, I consider maximum throughput under a latency constraint. When the latency constraint is set to the minimal latency of the CPU (CPU with batch-1), StepStone-DV offers 77× higher throughput (32× more samples at about 40% less time). If I allow a larger-area PIM with larger scratchpads, performance is improved even further to 96×. If I relax the latency constraint and allow the CPU 1.2× more time to complete an inference task, which allows batch-32 on the CPU, the performance benefit of StepStone-DV drops to 3× (3.5× with a larger scratchpad). While I use the highly-optimized Intel OneDNN library on the CPU, the performance I observe falls short of the channel-level StepStone-CH, which can fully utilize the memory-system bandwidth. Still, the finer-grained StepStone-DV (which can be implemented in buffer chips) offers substantially better performance and StepStone-BG offers far lower minimum latency.

**Throughput rooflines.** The throughput benefits of StepStone are also ap-

parent on the roofline plot presented in Figure 5.7, also for a 1024 × 4096 weight matrix. The plot includes the CPU as above, StepStone-BG and -DV (the maximum of the two represents the achieved performance with StepStone), and the performance obtained with an NVIDIA Titan Xp (running CUTLASS) when the model is already present in GPU device memory or when it must first be read from CPU main memory.

In the realistic scenario where GPU memory capacity is too small to accommodate the full recommender system and language models, StepStone exhibits higher throughput (in addition to its latency benefits) at all reasonable batch sizes. In fact, the CPU and GPU offer an advantage only once the batch is 256 samples or greater. Even if the model fits in GPU memory, StepStone offers higher performance for batches of 16 samples or less. The gap between the rooflines and simulated performance of StepStone stems from the localization and reduction overheads.

I emphasize that StepStone PIM achieves this high performance benefits without utilizing CPU or GPU compute resource, such as cores or caches. This implies that the overall system performance can increase even further by colocating additional tasks on the same node.

### 5.4.2 End-to-End Performance

Figure 5.8 compares the inference performance of one recommendation system and three language models with different PIM approaches—PEI, Chopim, and StepStone PIM—to that of a CPU. For the PIM approaches, I

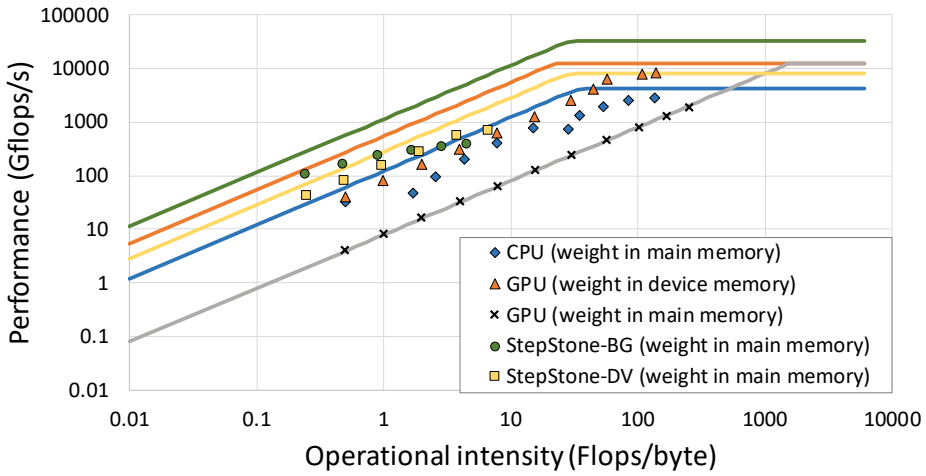



Figure 5.7: Roofline models for CPU, GPU, and StepStone PIMs; measured results are for a 1K × 4K weight matrix for varying batch sizes (the left most point of each system is for batch-1 and the batch is 2× larger for each point moving to the right).

assume the same PIM system (Figure 5.3) and that GEMMs can be executed by either the CPU, device-level (PIM_DV), or BG-level PIMs (PIM_BG); the best performing option is chosen for each GEMM. GEMMs are used for FC and projection layers. All other operations, including concatenation, GELU, softmax, sorting, batched GEMM, and some data reorganization (e.g. stack) operations are executed on the CPU (*CPU_Other*).

I show the performance of two different CPU models: measured on the real system (*CPU*) and idealized CPU performance (*iCPU*). I estimate idealized performance with my StepStone-CH, which maximally utilizes memory channel bandwidth. Overall, the measured results show that the CPU performs poorly on small-matrix GEMMs.

Naive Chopim ($nCHO$) executes GEMMs as multiple GEMV operations, which leads to missed locality opportunities. On the other hand, if Chopim is enhanced with StepStone block grouping ($eCHO$) and divides each GEMM into smaller dot-product operations, it benefits from better PIM buffer locality and the overhead for buffer fill/drain significantly decreases. However, compared to StepStone PIM, eCHO suffers from higher localization/reduction overhead. I evaluate a low-power StepStone PIM mode ($STP*$), where only StepStone-DV is used, and a high-performance mode ($STP$), which selects the best-performing level per GEMM.

The execution time of DLRM is dominated by a single FC layer (92%) and GEMM execution time is long enough to amortize the localization/reduction overheads. This enables Chopim and StepStone PIM to use BG-level PIMs and benefit from their high memory bandwidth. On the other hand, PEI cannot fully utilize BG-level PIMs due to command bandwidth bottleneck and, consequently, using more PIMs with PEI only increases overhead. GPT2 shows a similar trend but the gaps between PEI and StepStone PIM/Chopim are greater due to a larger weight matrix than DLRM. In BERT and XLM, the $N$ dimension is the batch size multiplied by the sequence length after tensor reshaping, offering more efficient GEMM execution. For BERT, $N$ becomes 32 in all FC layers whereas, for XLM, the sequence length starts at 1 and increases by 1 up to the maximum length (8 in my configuration) after each iteration. As a result, XLM utilizes BG-level PIMs when $N$ is small and, later, switches to DV-level PIMs once arithmetic performance saturates and

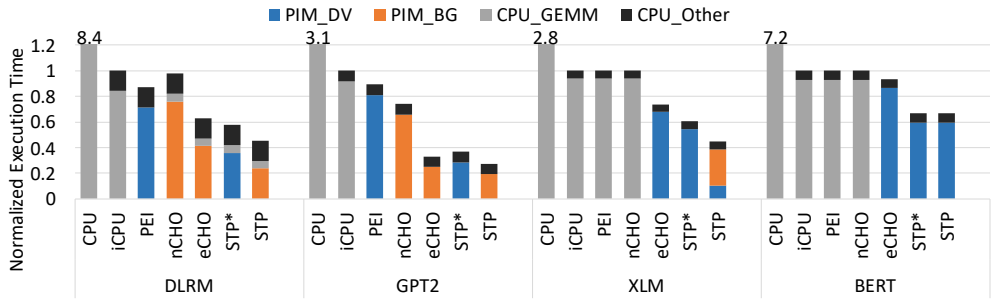


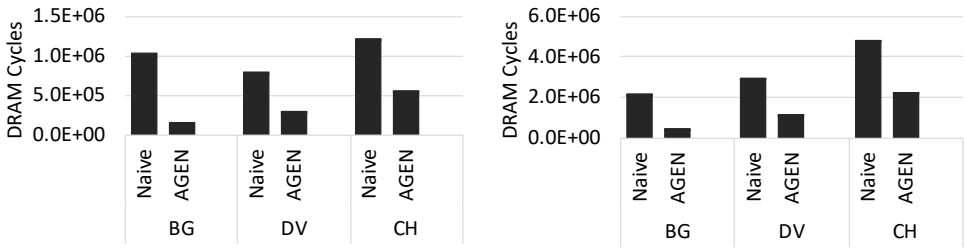

Figure 5.8: End-to-end performance results for various recommendation and language models with the CPU and PIMs.

overheads start to dominate.

Overall, when weight matrices are larger and the batch dimension is smaller, StepStone PIM outperforms other CPU and PIM approaches. Even with somewhat larger batches (e.g., up to $N = 384$ for BERT), StepStone PIM outperforms the CPU by splitting a batch into several batch-32 GEMM operations. For example, StepStone PIM achieves 12× higher performance than the CPU for BERT. Thus, StepStone PIM outperforms the CPU until $N = 12 \times 32 = 384$.

### 5.4.3 Impact of StepStone AGEN

Figure 5.9 shows the performance benefit of StepStone AGEN over the naive approach (explained in Section 5.2.4). Overall, StepStone AGEN outperforms the naive approach by up to 4×, in particular when the number of active PIMs is larger. Intuitively, the naive approach can find the next cache block in a probability of $1/n$, where $n$ is the number active PIMs. For StepStone-BG (Figure 5.9a) there are 16 active PIMs and the performance




(a) m × k = 1024 × 4096 (b) m × k = 2048 × 8192

Figure 5.9: GEMM latency comparison between naive address generator and the proposed StepStone AGEN.

difference between two approaches is 8×. This is mainly because pairs of cache blocks are contiguous in my baseline address mapping, which equates the naive approach with my optimized AGEN. However, when a large gap in the mapping exists, the naive approach requires numerous iterations and requires a large number of cycles to generate the next address. The DRAM burst transfer latency is 4 DRAM cycles and bubbles are introduced any time the next address cannot be generated within that time window. This does not occur with my proposed AGEN and its latency can always be hidden within the pipeline. The difference in performance between the two approaches for this case is apparent for StepStone-DV with a large weight matrix (Figure 5.9b), where the performance gap is 2.5×.

### 5.4.4 Parallelism—Distribution Overhead Tradeoffs

Figure 5.10 shows the GEMM latency comparison between two cases: (1) when all bank group-level PIMs are used and (2) only half of the BG-level PIMs are active. I present bank group-level PIMs tradeoff because I

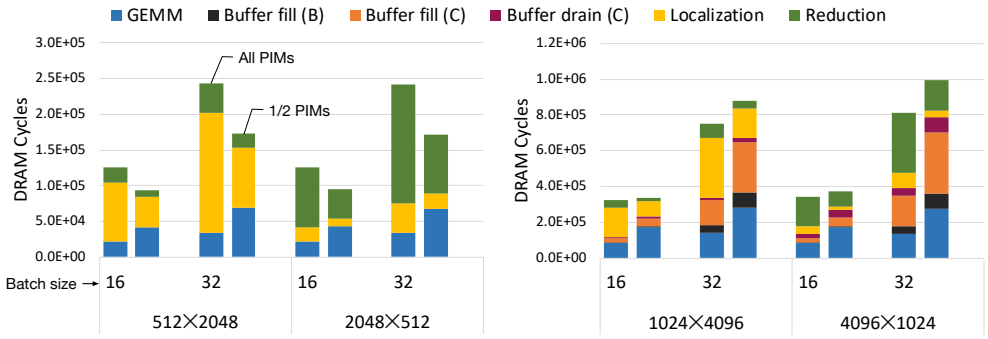



Figure 5.10: Impact of trading off between PIM execution time and replication/reduction overhead.

already discussed tradeoffs with respect to PIM level. When the weight matrix size is small, the fraction of replication and reduction overhead dominates the entire execution time. If I only activate half of the BG-level PIMs the overheads decrease by at most half while arithmetic execution time doubles because parallelism is cut in half. Still, the tradeoff proves beneficial when the matrices are small (left). On the other hand, as the matrix size increases, the fraction of PIM execution time increases as well (right). This is because the PIM execution time quadruples as each dimension size is doubled, whereas the localization/reduction overhead only doubles. Moreover, as the input and output matrices (i.e. activations) grow, they exceed scratchpad capacity. As a result, the fraction of execution time required for buffer fill/drain operations also increases. Even though using fewer PIM units does offer better performance for the larger matrix, it still provides a valid tradeoff option because comparable performance is attainable in some cases while resource usage and power consumption decreases.

### 5.4.5 Impact of Address Mapping

Figure 5.11 shows the execution time of GEMM operations when different address mappings and aspect ratios of the weight matrices are used. To isolate impact to only DRAM address mapping, I set the batch size to 4 such that the input and output matrices fit in the scratchpad memory of all three PIM options. In the bank-group level StepStone-BG, the fraction of localization overhead differs significantly across address mappings when the matrix is short and fat (i.e., $128 \times 8192$). This is because the number of PIMs that share the same input matrix blocks in address mappings 1 and 2 are $2\times$ greater than those with address mappings 3 and 4 and $4\times$ greater than those with address mapping 0. The reason for the low localization overhead with address mapping 0 is that the combination of the address mapping and matrix size interleaves addresses such that matrix columns remain contiguous within each PIM. In contrast, the tall and thin GEMM (i.e., $8192 \times 128$) suffers from high reduction overhead for all address mappings. This is because the CPU address mappings choose fairly fine-grain interleaving across bank groups and channels to maximize bandwidth. StepStone-BG is more sensitive to address mapping and aspect ratio compared to StepStone-DV and -CH, because it distributes work across a larger number of units and the relative overhead of localization and reduction is higher.

Note that address mappings 2 and 3 for StepStone-CH exhibit higher GEMM execution times because these mappings interleave bank groups at a coarser granularity. Hence, bandwidth cannot be maximized because consec-

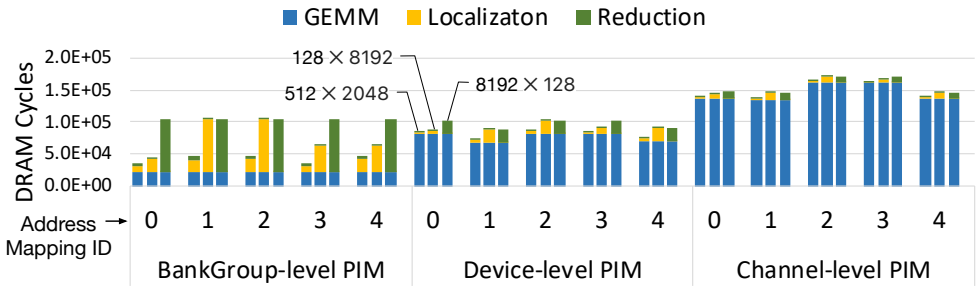



Figure 5.11: Sensitivity to address mapping and aspect ratio of the weight matrix (batch_size = 4).

utive memory accesses are penalized by tCCDL (i.e., column-to-column delay for back-to-back accesses within the same bank group is larger than across bank groups). This demonstrates that timing parameter considerations are also important when deciding the address mappings for PIM-enabled main memory. In theory, the localization and reduction overheads are lower when fewer PIMs share the input and output matrix blocks as common operands. However, this goal of low sharing cannot be realized with a single fixed address mapping because the sharing pattern changes with the matrix size and aspect ratio.

### 5.4.6 Impact of Scratchpad Memory Capacity

Figure 5.12 shows the impact of scratchpad memory capacity on GEMM latency. I analyze StepStone-BG as it has the most stringent area constraint. I search for an optimal allocation across the scratchpad partitioning options between input and output buffer allocations (there are only two buffers so the search converges quickly). I find that interleaving buffer fill/drain operations

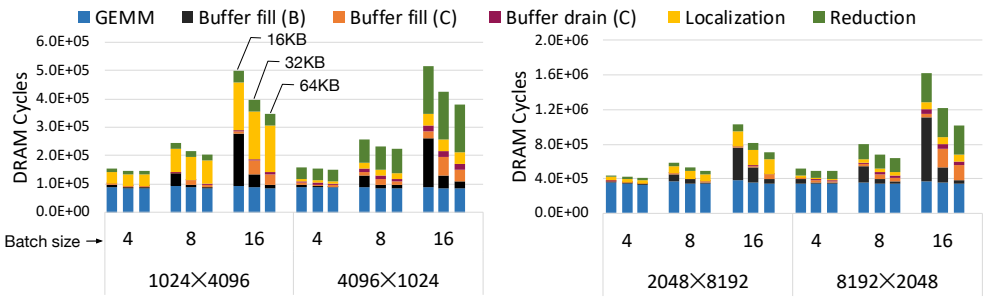



Figure 5.12: GEMM latencies for different matrices and buffer sizes (StepStone-BG).

with arithmetic has negligible impact on GEMM performance. The ability to execute entire kernels limits the benefits of overlapping data transfer with arithmetic and interleaving increases the number of row buffer conflicts, though row-buffer locality remains high.

Larger matrices (right) tends to amortize the buffer fill/drain overheads better than smaller matrices (left). Generally, overhead increases with batch size. Interestingly, the overhead with the $2048 \times 8192$ weight matrix increases at half the rate of other matrix configurations. This is because the number of block groups with this specific weight matrix dimension is half that of the other matrix sizes I evaluate. Consequently, the working set of the input activation (matrix B) per PIM unit is also half that of other matrix configurations. As explained in Section 5.2.2, the number of groups is determined by both the address mapping and matrix dimensions.

### 5.4.7 Impact of Concurrent CPU Access

I expect StepStone PIM to outperform prior PIM architectures, including Chopim, by enabling longer-running GEMM operations that maintain PIM locality. Long-running operations are important when the CPU also executes a memory-intensive workload concurrently with the PIMs, as both the CPU and PIMs contend for limited command channel bandwidth. I evaluate this using the same colocation used by Cho et al. for evaluating Chopim [29], as described in Section 5.3. While the colocated applications are not DL-related, they run readily on gem5 and clearly demonstrate the impact of command channel contention. I isolate the performance benefits to just the StepStone AGEN unit that enables long-running kernels by running the same StepStone GEMM flow on eCHO and StepStone PIM and reporting results corresponding only to GEMM execution.

StepStone PIM outperforms Chopim when the CPU intensively accesses memory concurrently with PIMs (Figure 5.13). As the matrix shape changes from short-fat to tall-thin, each of eCHO kernels accesses fewer cache blocks, resulting in more PIM kernel invocations and greater contention with the CPU for the command channel. As a result, PIM kernel packets are delayed and the PIMs are underutilized. With BG-level PIM, the relative performance of Chopim to StepStone PIM is worse since even more PIMs are underutilized due to the command bandwidth bottleneck. This performance gap will increase as the number of PIMs in each channel increases, increasing the importance of mechanisms that enable long-running kernels.

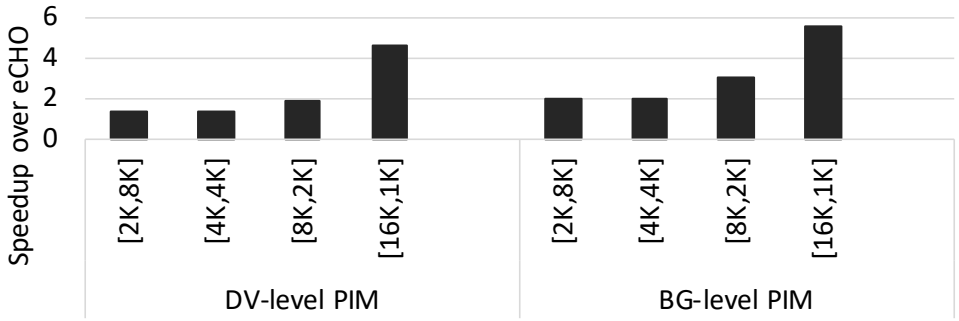



Figure 5.13: Speedup of StepStone PIM (STP) over Chopim enhanced with StepStone block grouping (eCHO) when concurrent CPU access exists. The size of matrices is fixed and its aspect ratio is varied.

### 5.4.8 Power and Energy Analysis

Figure 5.14 shows the power and energy consumption per DRAM device of StepStone-BG and StepStone-DV. As $N$ increases, the relative contribution of arithmetic also increases. However, overall, the power of DRAM access (either within the PIMs or for localization and reduction) dominates the power of the SIMD units. The right side of the figure shows that StepStone-BG is more energy-efficient than StepStone-DV when $N$ is small. The main source of this energy savings is that IO energy is much smaller within a device. However, as $N$ increases, the energy for localization and reduction dominates the energy consumption of arithmetic and StepStone-DV is more efficient. Note that, if power exceeds the delivery/cooling budget for a chip or module, performance can be throttled.

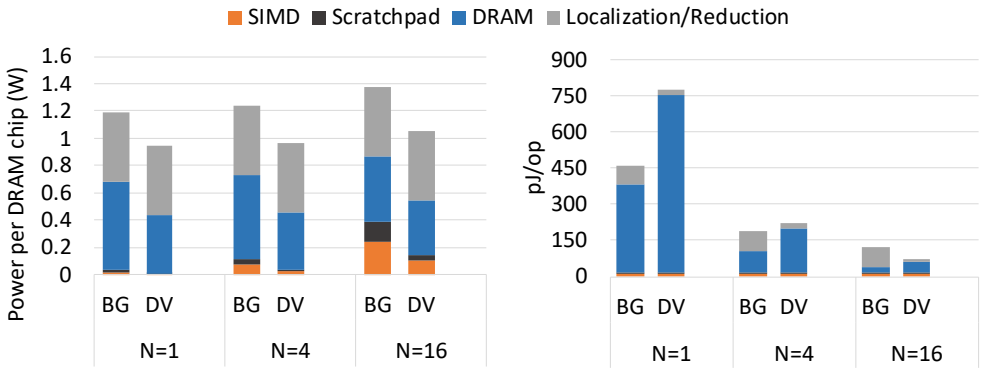



Figure 5.14: Power dissipation per DRAM device (left) and energy consumption per floating-point operation (right) of StepStone-BG and StepStone-DV (weight_matrix = [1024, 4096]).

## 5.5 Related Work

To the best of my knowledge, this is the first work that enables high-performance and CPU-compatible near-memory processing for accelerating bandwidth-bound fully-connected (FC) layers. I describe prior related work below and contrast my contributions for existing approaches.

***Processing in main memory.*** Processing in main memory implies that PIM should play along with the other parts of the system; otherwise, it will have a system-wide impact. Considering this, some researches [10, 80] enable PIM in a fine granularity, such as PIM operations per cache block. This approach can solve the complex address mapping problem. The CPU indicates the next cache block to process with some command packets and PIM processes the cache block. Though this approach can accommodate more applications due to its flexibility, PIM performance will be eventually limited by the command bandwidth. RecNMP [77] mitigates this command bandwidth

bottleneck by sending compound of memory requests but this solution does not scale when there are more than 4 PIMs per channel. Jeong et al. [72] explore PIM-aware data layouts in a multi-channel PIM architecture but do not consider complex address mappings that are commonly used in modern CPUs. Chopim [29] enables coarse-grained PIM kernels under complex address mapping. Though GEMM can be executed with Chopim by multiple GEMV kernel calls, temporal locality cannot be exploited which is crucial for high-performance GEMM operations. Also, Chopim does not provide efficient localization and reduction mechanisms, which incur high overhead for executing GEMMs on PIMs. NDA [44], Chameleon [15], and MCN DIMM [14] are also based on conventional DIMM devices and proposes PIM designs to practically add PEs to main memory.

***GEMM acceleration with PIM.*** The Active Memory Cube (AMC) [139] targets GEMM operations with in-memory processors. AMC considers address interleaving but interleaving is only allowed within the same local PIM memory. This essentially requires partitioning into CPU and PIM memory spaces and data should be copied from one space to another if the processing mode changes. This approach has the same data loading problem as discrete accelerators and does not truly realize PIM potential of sharing the same memory between the CPU and PIM. On the other hand, my solution does not have this limitations and works with any XOR-based address mapping and PIMs in any DRAM hierarchy levels.

***PIM for machine learning.*** PIM for machine learning workloads has

been widely studied. Much of this research targets convolutional neural networks [101, 67, 46, 28, 37, 73, 79, 131, 133], embedding table lookups in recommendation models [77, 89], recurrent neural networks [100], and GAN [103, 125]. In contrast, I target the tall-thin/fat-narrow GEMMs of fully-connected layers in DL inference. Newton [59] also targets fully-connected layers, like StepStone PIM. However, Newton operates as a discrete accelerator that cannot benefit from the advantages of main-memory acceleration described in Section 5.1. More importantly, Newton does not avoid weight copies, does not exploit GEMM locality, cannot trade off parallelization degree overheads with performance benefits, cannot selectively execute at different PIM levels or the CPU to dynamically match changing workload characteristics, and does not support the long-running kernels necessary for concurrent bandwidth-intensive CPU tasks.

## 5.6 Chapter Summary

I identify that small-batch GEMM operations of DL inference workloads are bandwidth bound on CPUs and GPUs and can benefit from PIM acceleration. I introduce *StepStone PIM*, which enables independent PIM GEMM execution under complex CPU DRAM address mapping. The main innovation of StepStone PIM is the address-mapping cognizant GEMM blocking with matching PIM-side address generation. My unique AGEN logic improves throughput compared to naive or host-side address generation. I explore PIM designs in three different DRAM hierarchy levels (channel, chip, and bank-

group levels) and show their tradeoffs with detailed simulation results. I show that activating more PIMs for GEMM improves arithmetic performance but adds overheads for data localization/replication and reduction. I show the benefits of choosing an optimal parallelization with respect to both performance and energy efficiency. I also provide sensitivity analysis for the impact of different address mappings and PIM resource allocation (scratchpad capacity and SIMD width). I conclude that StepStone is an appealing datacenter solution because of: (1) its low cost; (2) its potential for lower latency and higher throughput even when implemented at the buffer-chip level within a DIMM without DRAM device modification; and (3) its locality-optimized high efficiency GEMM execution that frees CPU resources for other tasks.

# Chapter 6

# Dissertation Summary and Future Work

In this dissertation, I solve challenges that hinder the integration of PIM in CPU's main memory and leverage the benefit of CPU-PIM concurrent access to the same memory by introducing several compelling use cases. The challenges include solving the address-space sharing, interference/contention management, and low-overhead state synchronization mechanism. I proposed mechanisms to solve these problems with runtime, OS, and hardware supports. By solving these problems, I enable high-performance CPU-PIM concurrent access, where the CPU and PIMs share the same memory and interleave memory access in a fine-grained manner. Thanks to these mechanisms, the CPU and PIMs are able to collaborate on the same data without any performance and/or capacity overheads. I introduce two compelling use cases in ML/AI domain to exploit the concurrent CPU-PIM collaboration to improve the training and inference speeds, respectively.

I also clarify the limitations of my current work and offer suggestions enabling high-performance PIM in conventional CPU systems.

**Compilation.** In Section 4.4, I introduced PIM APIs that can be used to write programs for PIMs. However, the details about how the programs are

compiled and executed at runtime is excluded. Since concurrent access to the same data is enabled by my research, each API call can be executed by either the CPU or the PIMs. In the worst-case scenario, the program can be run with only the CPU and performance can be improved as more CPU execution can be replaced by PIM acceleration. As I mention in Chapter 5, to achieve end-to-end performance improvement, it is important to consider both expected speedup for the core execution and overheads for aligning operands such that the locality needed for PIM execution is achieved. Therefore, to achieve high end-to-end performance, it is crucial to maximize data reuse opportunities in between PIM operations. If data replication and inter-PIM communication cannot be avoided, they should be at least minimized. This should be accomplished by some compiler and the runtime support, which I leave it as future work.

**Main-memory accelerators for sparse data structures.** Applications with sparse data structures are also potential candidates for PIM acceleration. These applications typically have low data reuse and suffer from the memory bandwidth bottleneck. The main challenge for using PIMs for sparse data structures are twofold: (1) both the metadata and the actual data are required for execution and they should be placed in the same PIM memory; and (2) random access should be limited to within the PIM buffer. For the convenience of explanation, I will use the compressed sparse row (CSR) format and sparse matrix-vector multiplication as an example. In the CSR format, there are two arrays of metadata, `COL_INDEX` and `ROW_INDEX`, and one data array. To process

one element in the data array, both metadata arrays are required to access the corresponding operands in the input and output vector. Therefore, to execute sparse matrix-vector multiplication (SpMV) with PIMs, the metadata should be placed in the same PIM memory as its corresponding data. To always guarantee the colocation of metadata and data regardless of the address mapping being used in the system, this data should be packed into one cache block, since it is the smallest interleaving unit. In addition, though PIMs can stream the matrix data, the access to input and output vectors is quite random. To maximize PIM performance with the PE design and execution flow proposed in this dissertation, only one of the operands should be streamed from DRAM while the other operands should be read from and written to the PIM buffer. To accomplish this, a new matrix blocking based sparse data structure will be required so as to limit the range of random access to the PIM buffer size. In this way, the metadata and data is read from DRAM and input/output operands can be accessed from the PIM buffer.

**Light-weight memory allocation for in-place acceleration.** PIMs can be beneficial for basic operations, such as memory copy, which takes short time per operation but are invoked many times within the system by diverse applications. Since the memory copy is basically moving data from source to destination, re-aligning the source and destination arrays negates any PIM advantage. This means that the source and destination should be already aligned prior to the memory copy operation. To enable this in conventional systems, new interfaces between the user program and memory allocator, and

between the memory allocator and frame allocator should be defined. However, memory allocation is also another frequently executed workload in the system. Therefore, PIM-aware memory allocation routine should not incur high overhead since it affects the performance of other applications.

**Cache coherence mechanism for PIMs.** In Section 4.3, I suggested to maintain coherence between the CPU and PIMs with the combination of data copy, memory fence, and cache bypassing. However, this assumption breaks the cache coherence model in conventional systems. This can be solved by using the PIM controller located near the CPU memory controller. Since the PIM controller knows all the memory transactions happening in the PIMs thanks to the replicated FSMs, it can operate as another node in the coherent system. This can be done by connecting the PIM controller to the coherent bus. If there are any PIM write requests issued, the PIM controller can generate coherence packet and broadcast it through the coherent bus. On the other hand, if there are any reads issued by PIMs to some stale data in DRAM, the PIM controller should pause the PIM, revert back, send the up-to-date data to the PIM, and continue the PIM execution. However, since this is expensive, a better option is to use the existing approach of bulk cache invalidation prior to the PIM execution based on the range of address that PIMs are going to access.

# Bibliography


[1] Jedec ddr4 sdram standard, 2012.

[2] Intel onednn (v1.6.1). In *https://github.com/oneapi-src/oneDNN*. Intel, 2020.

[3] Intel(r) advisor 2020 update 2 (build 606470). Intel, 2020.

[4] Nvidia cutlass (v2.2). In *https://github.com/NVIDIA/cutlass*. NVIDIA, 2020.

[5] nvprof release version 10.0.130 (21). In *https://docs.nvidia.com/cuda/profiler-users-guide/index.html*. NVIDIA, 2020.

[6] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems*, pages 873–881, 2011.

[7] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, et al. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 245–258. ACM, 2017.

[8] Jung Ho Ahn, Mattan Erez, and William J Dally. The design space of data-parallel memory systems. In *SC06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[9] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015.

[10] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 336–348. IEEE, 2015.

[11] Berkin Akin, Franz Franchetti, and James C. Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 131–143, New York, NY, USA, 2015. ACM.

[12] Berkin Akin, Franz Franchetti, and James C Hoe. Hamlet architecture for parallel data reorganization in memory. *IEEE Micro*, 36(1):14–23, Jan 2016.

[13] Mohammad Alian and Nam Sung Kim. Netdimm: Low-latency near-memory network interface architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 699–711. ACM, 2019.

[14] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Wang, Adam Roewer, Thomas McPadden, Oliver OHalloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. Application-transparent near-memory processing architecture with memory channel network,. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[15] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[16] JEDEC Solid State Technology Association et al. Jedec announces support for nvdimm hybrid memory modules. *[Online]. Available from: https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules*, 1, 2016.

[17] Manu Awasthi. Rethinking design metrics for datacenter dram. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 162–163. ACM, 2015.

[18] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[20] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 316–331, New York, NY, USA, 2018. ACM.

[21] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 629–642, New York, NY, USA, 2019. ACM.

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

[23] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[24] Niladrish Chatterjee, Mike OConnor, Donghyuk Lee, Daniel R Johnson, Stephen W Keckler, Minsoo Rhu, and William J Dally. Architecting an energy-efficient dram system for gpus. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 73–84. IEEE, 2017.

[25] Fan Chen, Linghao Song, and Yiran Chen. Regan: A pipelined reram-based accelerator for generative adversarial networks. In *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*, pages 178–183. IEEE, 2018.

[26] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 33–38. IEEE, 2012.

[27] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems.

In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[28] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

[29] Benjamin Y Cho, Jeageun Jung, and Mattan Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. *arXiv preprint arXiv:2012.00158*, 2020.

[30] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. Near data acceleration with concurrent host access. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 818–831. IEEE, 2020.

[31] JS Choi. Next big thing: Ddr4 3ds.

[32] Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. In *Advances in Neural Information Processing Systems*, pages 7059–7069, 2019.

[33] GenZ Consortium et al. Genz core specification. Technical report, Technical Report. GenZ Consortium. http://genzconsortium. org/specifications , 2017.

[34] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, 5(1):46–55, 1998.

[35] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[36] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[37] Timothy J Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, 11:1–23, 1997.

[38] Quan Deng, Youtao Zhang, Minxuan Zhang, and Jun Yang. Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[39] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE, 2019.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[41] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The mondrian data engine. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 639–651. ACM, 2017.

[42] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[43] Duncan G Elliott, Michael Stumm, W Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. Computational ram: Implementing processors in memory. *IEEE Design & Test of Computers*, 16(1):32–41, 1999.

[44] Jason Evans. Scalable memory allocation using jemalloc, 2011.

[45] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung 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*, pages 283–295. IEEE, 2015.

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

[47] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764. ACM, 2017.

[48] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2009.

[49] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. A workload and programming ease driven perspective of processing-in-memory. *arXiv preprint arXiv:1907.12947*, 2019.

[50] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[51] Antonio González, Mateo Valero, Nigel Topham, and Joan M Parcerisa. Eliminating cache conflict misses through xor-based placement functions. In *Proceedings of the 11th international conference on Supercomputing*, pages 76–83. ACM, 1997.

[52] Google. A microbenchmark support library.

[53] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[54] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. ipim: Programmable in-memory image pro-

cessing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2020.

[55] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. In *In the Workshop on Near-Data Processing (WoNDP)(Held in conjunction with MICRO-47.)*, 2014.

[56] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, 2020.

[57] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[58] Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg, Tushar Krishna, and Hyesoon Kim. Performance implications of nocs on 3d-stacked memories: Insights from the hybrid

memory cube. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 99–108. IEEE, 2018.

[59] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[60] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A dram-makers accelerator-in-memory (aim) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[61] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[62] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[63] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. Accelerating linked-list traversal through near-data processing. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, 2016.

[64] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. Hardware implementation and analysis of gen-z protocol for memory-centric architec-

ture. *IEEE Access*, 8:127244–127253, 2020.

[65] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 204–216, June 2016.

[66] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32. IEEE, 2016.

[67] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. High performance smart expression template math libraries. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 367–373. IEEE, 2012.

[68] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 802–815. IEEE, 2019.

[69] Inphi. Introducing lrdimm - a new class of memory modules.

[70] B Jacob, G Guennebaud, et al. Eigen: C++ template library for linear algebra, 2013.

[71] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[72] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[73] Taeyang Jeong, Duheon Choi, Sangwoo Han, and Eui-Young Chung. A study of data layout in multi-channel processing-in-memory architecture. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, pages 134–138, 2018.

[74] Biresh Kumar Joardar, Bing Li, Janardhan Rao Doppa, Hai Li, Partha Pratim Pande, and Krishnendu Chakrabarty. Regent: A heterogeneous reram/gpu-based architecture enabled by noc for training cnns. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 522–527. IEEE, 2019.

[75] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[76] Norman P Jouppi, Andrew B Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. Cacti-io: Cacti with off-chip power-area-timing models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1254–1267, 2015.

[77] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, pages 192–201. IEEE, 1999.

[78] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803. IEEE, 2020.

[79] Dong Wan Kim and Mattan Erez. Relaxfault memory repair. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 645–657. IEEE, 2016.

[80] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Computer Architec-*

*ture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 380–392. IEEE, 2016.

[81] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward standardized near-data processing with unrestricted data placement for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 24. ACM, 2017.

[82] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 145–156. IEEE Press, 2013.

[83] Gwangsun Kim, John Kim, Jung Ho Ahn, and Yongkee Kwon. Memory network: Enabling technology for scalable near-data computing. In *2nd Workshop on Near-Data Processing*, 2014.

[84] Moonsoo Kim, Jungwoo Choi, Hyun Kim, and Hyuk-Jae Lee. An effective dram address remapping for mitigating rowhammer errors. *IEEE Transactions on Computers*, 68(10):1428–1441, 2019.

[85] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379, June 2012.

[86] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016.

[87] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[88] Peter M Kogge. Execube-a new architecture for scaleable mpps. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 77–84. IEEE, 1994.

[89] Peter M Kogge, Jay B Brockman, Thomas Sterling, and Guang Gao. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA*, volume 97. Citeseer, 1997.

[90] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 740–753. ACM, 2019.

[91] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *OSDI*, volume 16, pages 705–721, 2016.

[92] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

[93] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):63, 2016.

[94] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 241–252, Oct 2015.

[95] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301. ACM, 2017.

[96] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[97] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A

heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.

[98] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 367–376. ACM, 2012.

[99] Yuxi Liu, Xia Zhao, Magnus Jahre, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Lieven Eeckhout. Get out of the valley: power-efficient address mapping for gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 166–179. IEEE, 2018.

[100] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245. ACM, 2017.

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

[102] Yun Long, Xueyuan She, and Saibal Mukhopadhyay. Design of reliable dnn accelerator with un-reliable reram. In *2019 Design, Automation*

*& Test in Europe Conference & Exhibition (DATE)*, pages 1769–1774. IEEE, 2019.

[103] Sangkug Lym, Armand Behroozi, Wei Wen, Ge Li, Yongkee Kwon, and Mattan Erez. Mini-batch serialization: Cnn training with inter-layer data reuse. *Proceedings of Machine Learning and Systems*, 1:264–275, 2019.

[104] Haiyu Mao, Mingcong Song, Tao Li, Yuting Dai, and Jiwu Shu. Lergan: A zero-free, low data movement and pim-based gan architecture. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 669–681. IEEE, 2018.

[105] Patrick J Meaney, Lawrence D Curley, Glenn D Gilda, Mark R Hodges, Daniel J Buerkle, Robert D Siegl, and Roger K Dong. The ibm z13 memory subsystem for big data. *IBM Journal of Research and Development*, 59(4/5):4–1, 2015.

[106] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In *Proceedings the IFIP International Conference on Network and Parallel Computing*, 2010.

[107] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

[108] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 457–468. IEEE, 2017.

[109] Ravi Nair, Samuel F Antao, Carlo Bertolli, Pradip Bose, Jose R Brunheroto, Tong Chen, C-Y Cher, Carlos HA Costa, Jun Doi, Constantinos Evangelinos, et al. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17–1, 2015.

[110] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[111] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[112] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

[113] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 41–54, New York, NY, USA, 2017. ACM.

[114] Saeyoung Oh and Jong Kim. Reliable rowhammer attack and mitigation based on reverse engineering memory address mapping algorithms. In *International Workshop on Information Security Applications*, pages 146–158. Springer, 2018.

[115] Mark Oskin, Frederic T Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture*, pages 192–203, 1998.

[116] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[117] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[118] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE micro*, 17(2):34–44, 1997.

[119] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 31–44. ACM, 2016.

[120] J Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, 2011.

[121] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Reverse engineering intel dram addressing and exploitation. *arXiv preprint arXiv:1511.08756*, 2015.

[122] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[123] Jayaprakash Pisharath, Ying Liu, Wei-keng Liao, Alok Choudhary, Gokhan Memik, and Janaki Parhi. Nu-minebench 2.0. Technical report, Technical report, Northwestern University, 2005.

[124] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H Loh. There and back again: Optimizing the interconnect in networks of memory cubes. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 678–690. IEEE, 2017.

[125] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners.

[126] Adnan Siraj Rakin, Shaahin Angizi, Zhezhi He, and Deliang Fan. Pim-tgan: A processing-in-memory accelerator for ternary generative adversarial networks. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 266–273. IEEE, 2018.

[127] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[128] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[129] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.

[130] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast bulk bitwise and and or in dram. *IEEE Computer Architecture Letters*, 14(2):127–131, 2015.

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

[132] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26. IEEE Press, 2016.

[133] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: a near-memory content-aware page-merging architecture. In *Proceedings*

*of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 302–314. ACM, 2017.

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

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

[136] JEDEC Standard. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[137] Harold S Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, Jan 1970.

[138] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: Coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 72–82. ACM, 2010.

[139] Yuliang Sun, Yu Wang, and Huazhong Yang. Energy-efficient sql query exploiting rram-based process-in-memory structure. In *Non-Volatile*

*Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pages 1–6. IEEE, 2017.

[140] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, et al. Data access optimization in a processing-in-memory system. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 6. ACM, 2015.

[141] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 363–374. IEEE, 2010.

[142] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[143] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rmi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing:*

*System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[144] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 344–355. IEEE, 2014.

[145] Doe Hyun Yoon, Min Kyu Jeong, Michael Sullivan, and Mattan Erez. The dynamic granularity memory system. In *Proceedings of ISCA*, 2012.

[146] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[147] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[148] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98. ACM, 2014.

[149] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 544–557. IEEE, 2018.

[150] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 32–41. IEEE, 2000.

[151] Hongzhong Zheng, Jiang Lin, Zhao Zhang, Eugene Gorbatov, Howard David, and Zhichun Zhu. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 210–221. IEEE, 2008.

[152] Haishan Zhu, David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Mattan Erez. Kelp: Qos for accelerated machine learning systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 172–184. IEEE, 2019.

# Index

# Vita

Benjamin Youngjae Cho was born in Wayne, New Jersey on December 17th 1985, the son of Dr. Hanjin Cho and Jinwook Shin. He received his Bachelor of Science degree in Electrical and Electronics Engineering from Yonsei University, Seoul, South Korea. From the same university, he received his Master of Science degree under the guidance of Dr. Won Woo Ro. Then, he joined the doctoral program at the University of Texas at Austin, in the Electrical and Computer Engineering, specifically the Computer Architecture track, under the guidance of Dr. Mattan Erez. During his doctoral study, his research interests are focused on enabling processing in/near memory devices in conventional CPU systems and leveraging computing resources of the CPU and PIMs in parallel. During the course of his doctoral work, he has done internships at AMD research, Nvidia research, Micron, and Facebook AR/VR, working on various challenges in high-performance CPU/GPU memory systems.

Permanent address: bjcho@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.