UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Architecture Supports and Optimizations for Memory-Centric Processing System

Permalink
https://escholarship.org/uc/item/921045hk

Author
gu, peng

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Santa Barbara

Architecture Supports and Optimizations for Memory-Centric Processing System

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Peng Gu

Committee in charge:

Professor Yuan Xie, Chair
Professor Yufei Ding
Professor Li-C Wang
Professor Zheng Zhang

March 2021

The Dissertation of Peng Gu is approved.

Professor Yufei Ding

Professor Li-C Wang

Professor Zheng Zhang

Professor Yuan Xie, Committee Chair

January 2021

Architecture Supports and Optimizations for Memory-Centric Processing System

Copyright © 2021

by

Peng Gu

This dissertation is dedicated to my family and friends who understand and encourage me. Thank you for supporting me to explore this amazing world and pursue my dream.

Acknowledgements

First of all, I would like to express my sincere appreciation for my advisor, Prof. Yuan Xie. As a passionate and visionary researcher, he inspires me to expand my knowledge domain, to explore the unknown, and to pursue the valuable yet challenging targets which could make real impacts. He also taught me the way to collaborate and make contributions in the research community. Words are pale to say all my gratitude.

Many thanks to Prof. Yufei Ding, Prof. Li-C Wang, and Prof. Zheng Zhang for serving in my dissertation committee and giving valuable feedback to my qualify exam and this dissertation.

I would like to express my thanks to Prof. Yu Wang, Prof. Rong Luo, and Boxun Li, who provided me with great opportunities and guidance related to computer architecture during my bachelor study in Tsinghua University; Dr. Shuangchen Li, who helped me develop the skills in memory subsystem and foster the ability to conduct independent research when I was a junior graduate student. I would also like to thank Dr. Dimin Niu, Dr. Krishna Malladi, Benjamin S. Lim, Dr. Hongzhong Zheng, Dr. Cong Xu, and Dr. Naveen Muralimanohar for the great mentorship during my interships in the industry. I would like to especially thank my collaborator and roommate Xinfeng Xie, who provided me with valuable help and suggestions in the research projects.

I sincerely thank coauthors of my research papers, including but not limited to Dr. Dylan Stow, Dr. Eren Kursun, Wenqin Huangfu, Dr. Prashansa Mukim, Alvin Oliver Glova, Tianqi Tang, Dr. Itir Akgun, Dr. Mingyu Yan, Liu Liu, Dr. Xing Hu, Abanti Basak, Dr. Guoyang Chen, Dr. Weifeng Zhang, Andrew Chang, and Russell Barnes.

Last but not least, I would like to thank Xiling Yin, who gives me encouragement and inspiration for the past four years. Also, I want to thank my parents and grandparents for the caring and support, whose words and actions motivate me to be a better man.

Curriculum Vitæ

Peng Gu

Education

2021	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2018	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2015	B.S. in Electrical Engineering, Tsinghua University

Publications

[1]. Xinfeng Xie, Zheng Liang, **Peng Gu**, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator" Proc. International Symposium on High-Performance Computer Architecture (HPCA), 2021.

[2]. **Peng Gu**, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture" Proc. International Symposium on Computer Architecture (ISCA), 2020.

[3]. **Peng Gu**, Xinfeng Xie, Shuangchen Li, Krishna T. Malladi, Dimin Niu, Hongzhong Zheng, and Yuan Xie. "DLUX: a LUT-based Near-Bank Accelerator for Data Center Deep Learning Training Workloads" Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.

[4]. **Peng Gu**, Benjamin Lim, Wenqin Huangfu, Krishna T. Malladi, Andrew Chang, Yuan Xie. "NMTSim: Transaction-Command based Simulator for New Memory Technology Devices" Proc. Computer Architecture Letters (CAL), 2020.

[5]. Xinfeng Xie, Xing Hu, **Peng Gu**, Shuangchen Li, Yu Ji, and Yuan Xie. "NNBench-X: A Benchmarking Methodology for Neural Network Accelerator Designs" ransactions on Architecture and Code Optimization (TACO), 2020.

[6]. Wenqin Huangfu, Krishna T. Malladi, Shuangchen Li, **Peng Gu**, and Yuan Xie. "NEST: DIMM based Near-Data-Processing Accelerator for K-mer Counting" Proc. International Conference On Computer Aided Design (ICCAD), 2020.

[7]. Xinfeng Xie, Xing Hu, **Peng Gu**, Shuangchen Li, Yu Ji, and Yuan Xie. "NNBench-X: Benchmarking and Understanding Neural Network Workloads for Accelerator Designs" Proc. Computer Architecture Letters (CAL), 2019.

[8]. Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, **Peng Gu**, and Yuan Xie. "MEDAL: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm" Proc. International Symposium on Microarchitecture (MICRO), 2019.

[9]. Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, **Peng Gu**, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. "Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach" Proc. International Symposium on Microarchitecture (MICRO), 2019.

[10]. **Peng Gu**, Dylan Stow, Prashansa Mukim, Shuangchen Li, and Yuan Xie. "Cost-efficient 3D Integration to Hinder Reverse Engineering During and After Manufacturing" Proc. Asian Hardware Oriented Security and Trust Symposium (AsianHOST), 2018.

[11]. Jaya Dofe, **Peng Gu**, Dylan Stow, Qiaoyan Yu, Eren Kursun, and Yuan Xie. "Security Threats and Countermeasures in Three-Dimensional Integrated Circuits" Proc. Great Lakes Symposium on VLSI (GLSVLSI), 2017.

[12]. **Peng Gu**, Dylan Stow, Russell Barnes, Eren Kursun, and Yuan Xie. "Thermal-aware 3D Design for Side-channel Information Leakage" Proc. International Conference on Computer Design (ICCD), 2016.

[13]. **Peng Gu**, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Eren Kursun, and Yuan Xie. "Leveraging 3D Integration Technologies to Improve Hardware Security: Opportunities and Challenge" Proc. Great Lakes Symposium on VLSI (GLSVLSI), 2016.

[14]. Shuangchen Li, Liu Liu, **Peng Gu**, Cong Xu, and Yuan Xie. "NVSim-CAM: A Circuit-Level Simulator for Emerging Nonvolatile Memory based Content-Addressable Memory" Proc. International Conference On Computer Aided Design (ICCAD), 2016.

[15]. Dylan Stow, Itir Akgun, Russell Barnes, **Peng Gu**, and Yuan Xie. "Cost Analysis and Cost-Driven IP Reuse Methodology for SoC design Based on 2.5D/3D Integration" Proc. International Conference On Computer Aided Design (ICCAD), 2016.

[16]. Lixue Xia, Boxun Li, Tianqi Tang, **Peng Gu**, Xiling Yin, Wenqin Huangfu, Pai-yu Chen, Shimeng Yu, Yu Cao, YuWang, Yuan Xie, and Huangzhong Yang. "MNSIM: Simulation Platform for Memristor-based Neuromorphic Computing System" Proc. Design Automation and Test in Europe (DATE), 2016.

Abstract

Architecture Supports and Optimizations for Memory-Centric Processing System

by

Peng Gu

For the past two decades, the scaling of main memory lags behind the advancement of computation in aspects of bandwidth and capacity. First, conventional compute-centric architecture faces challenges to scale memory bandwidth due to the limitation of off-chip interconnect resources and the energy-inefficiency of long distance data movement. Also, the emerging big data workloads have increasing demand for higher memory capacity, which cannot be satisfied by traditional DRAM technology scaling.

To address these challenges, this dissertation focuses on exploring memory-centric architectures and design optimizations for higher memory bandwidth and larger memory capacity. Three categories of memory-centric designs have been researched. The analog process-in-memory architecture merges computation logics inside memory arrays. It employs the in-situ computing capabilities of resistive memory arrays to eliminate data movements and benefits from massive data parallelism. The digital process-near-memory architecture integrates computation units near memory arrays. The near-memory lightweight components can utilize abundant bandwidth of the internal memory arrays while the optimizations maintain hardware programmability. The enhanced memory design develops a simulation framework for emerging non-volatile memory technologies, which can greatly boost the memory capacity. Using both emerging non-volatile memory and 3D stacking memory technologies, this dissertation investigates four architectures and one simulation framework, covering a wide spectrum of application domains including deep learning, image processing, and high-performance parallel computing.

Contents

Curriculum Vitae **vi**

Abstract **viii**

1 Introduction **1**
1.1 Motivations 4
1.2 Challenges and Opportunities 6
1.3 Contributions 8

2 Backgrounds and Related Work **10**
2.1 Backgrounds on Main Memory Technologies 10
2.2 Backgrounds on Main Memory Architecture 13
2.3 Related Work on Process-In-Memory Systems 14
2.4 Related Work on Process-Near-Memory Systems 15
2.5 Related Work on Enhanced Memory 17

3 QUANTMEC: Quantized Deep Neural Network on Memristive Crossbar with Dynamic Precision **19**
3.1 Motivation 21
3.2 Architecture Design 23
3.3 Software Support 28
3.4 Experiment 33
3.5 Summary 45

4 DLUX: a LUT-based Near-Bank Accelerator for Data Center Deep Learning Training Workloads **46**
4.1 Motivation 48
4.2 Architecture Design 52
4.3 Software Support 59
4.4 Experiment 65
4.5 Summary 75

5 iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture **76**
5.1 Motivation 78
5.2 Architecture Design 80
5.3 Software Support 87
5.4 Experiment 89
5.5 Summary 99

6 MPU: Towards Bandwidth-abundant SIMT Processor via Near-bank Computing **100**
6.1 Motivation 102
6.2 Architecture Design 104
6.3 Software Support 112
6.4 Experiment 114
6.5 Summary 122

7 NMTSim: Transaction-Command based Simulator for New Memory Technology Devices **124**
7.1 Simulator Design 126
7.2 Optimization 130
7.3 Experiment 134
7.4 Summary 137

8 Summary **138**

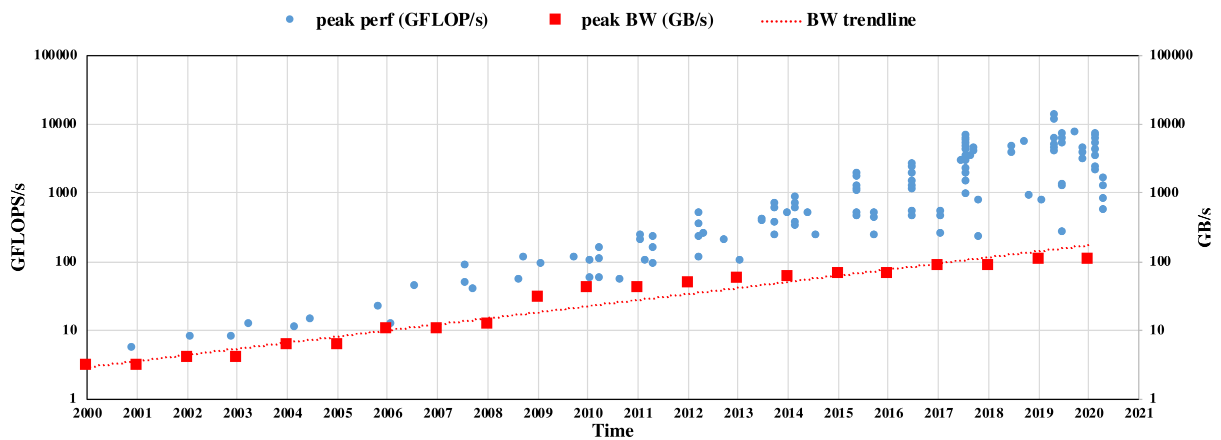
Bibliography **141**

Chapter 1

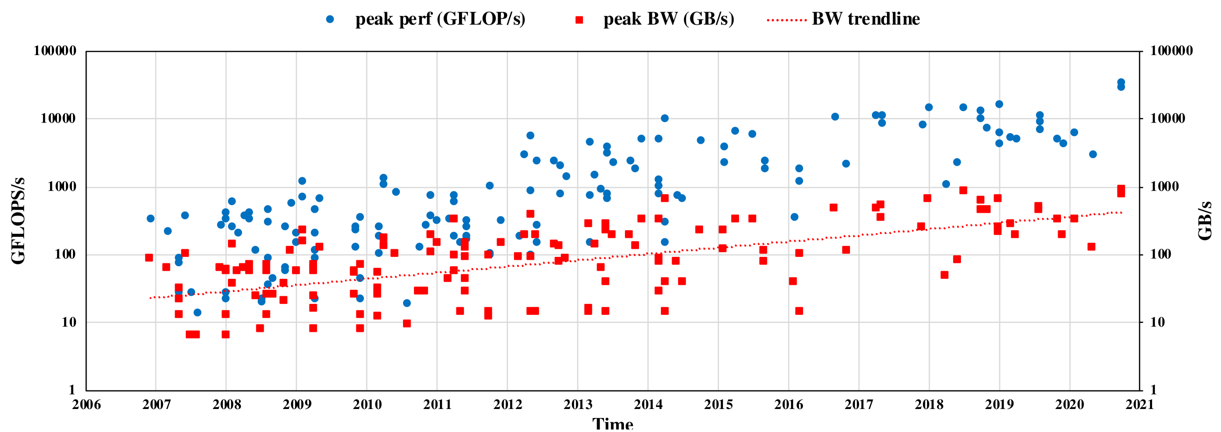
Introduction

Nowadays, the ever-increasing number of connected intelligent devices [1] and the booming of emerging big-data applications such as deep learning [2] generate a growing volume of data, demanding high-performance information processing systems. The development of such systems relies on improving the performance of both computation units and memory modules. On the one hand, with the slowdown of the Moore's law [3] and the chip power density rising significantly [4] over the past two decades, it becomes increasingly difficult to boost the performance of data processing systems by enhancing clock speed and transistor density. For example, from the computation's perspective, the maximum clock frequency for commodity CPUs has improved less than 1.5× since 2004 [5], and the transistor count growth rate per year drops from 40% near 2010 to around 25% in the following years for intel chips [6]. From the memory's perspective, the maximum data rate per I/O pin for the incoming DDR5 protocol is only 3× compared with DDR3 protocol which was first released in 2007, and the internal clock frequency remains almost the same. DRAM transistor count growth rate per year drops from 45% around early 2000s to around 25% in 2016 [6]. On the other hand, the invention of parallel computing techniques such as multi-core [7] architectures has significantly boosted the throughput

of the computation units. However, the development of memory bandwidth lags significantly behind the computation throughput for the past two decades, hitting a "memory bandwidth wall" [8]. Also, as the number of core count continuously increases per processor and the main memory capacity scaling slows down, we will encounter "memory capacity wall" [9].



(a) CPU's trend.



(b) GPU's trend.

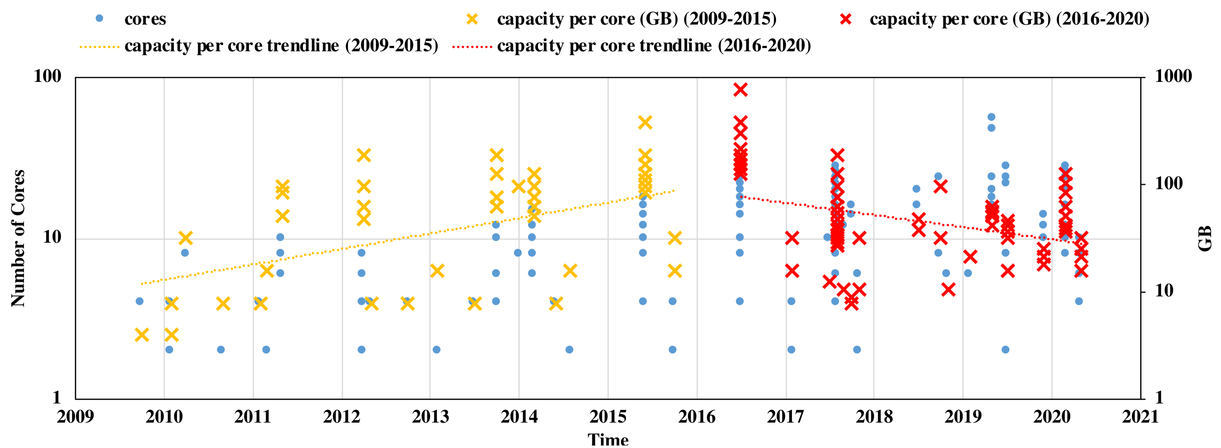
Figure 1.1: Trend of peak single precision performance and peak memory bandwidth.

The "memory bandwidth wall" issue exists in both CPUs and GPUs. To clearly demonstrate this issue, Fig. 1.1a shows the peak single precision floating point throughput per socket[1] and the peak memory bandwidth per socket[2] from 2000 to 2020 based on

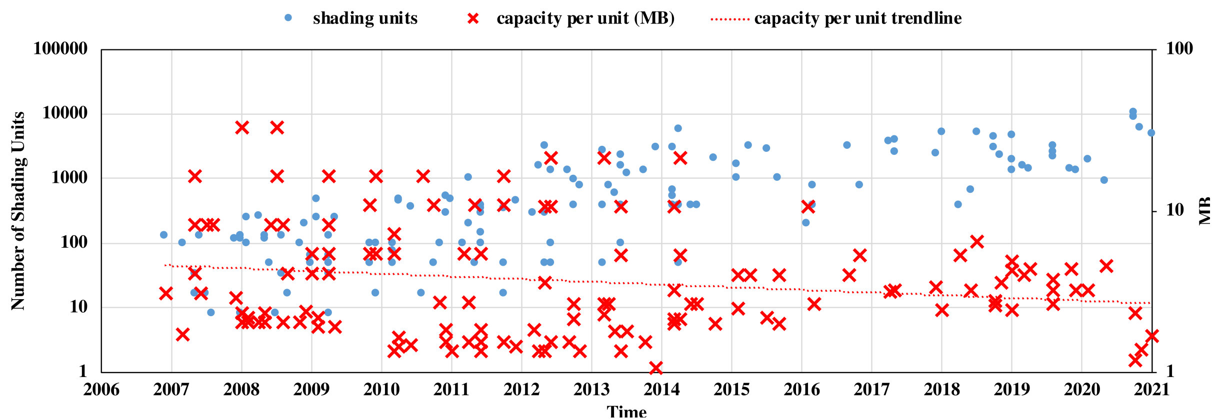
[1] https://www.cpu-world.com/info/charts.html

[2] https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend

the intel desktop and server CPUs. It is observed that over the past two decades the computation throughput is increased by 2431× , meanwhile the DDR based memory bandwidth has only improved 56×. Fig. 1.1b shows the peak single precision floating point throughput per GPU and the peak memory bandwidth per GPU from 2006 to 2020 based on the NVIDIA desktop GPUs[3]. While the computation throughput per GPU boosts by 2477×, the memory bandwidth per GPU has only increased by 119×.



(a) CPU's trend.



(b) GPU's trend.

Figure 1.2: Trend of core count and main memory capacity per core.

Using the same data source, we also illustrate the "memory capacity wall" issue for both CPUs and GPUs. Fig. 1.2a shows the core count per socket and the main memory

[3] https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

capacity per core from 2009 to 2020 based on the intel desktop and server CPUs. We can observe that the memory capacity per core grows every year before 2016, but declines after that. This is caused by the slowdown of DRAM technology scaling and limited increase of memory channels per CPU socket, as well as the steady increase of core count per CPU. This issue is more serious for throughput processors like the GPU, where the number of processing cores (e.g. shading units) per processor grows every year at a higher rate in comparison with CPU. Fig. 1.2b shows the shading units per GPU and the main memory capacity per unit from 2006 to 2020 based on the NVIDIA desktop GPUs. The trend of decreasing memory capacity per unit is apparent with the increase of shading units per GPU.

1.1 Motivations

The cause for the "memory bandwidth wall" issue has both hardware and software implications. It is also deeply impacted by the conventional compute-centric architecture where computation units and memory modules are designed and manufactured separately, and the interconnect buses between them should be scaled to match their performance. The "memory capacity wall" issue is mainly caused by the slowdown of the traditional DRAM technology. The following paragraphs will briefly introduce these problems, and motivate memory-centric architecture for further performance scaling.

From the hardware perspective, the continuous scaling of memory bandwidth has both performance and power challenges. First, the number of data pins for commodity DDR memory buses has remained 64 I/Os per channel for decades, and the number of memory channels per CPU socket is limited (up to 6 channels [10]). Scaling the number of I/Os per CPU is constrained by the limited I/O pins per package. Recently proposed high-bandwidth memory [11] incorporates 3D-stacked memory cubes and the processor

in the same package, greatly increases the number of data I/Os to 1024 per memory cube. However, these I/Os already consumes 18.8% area of each 3D layer [12], posing a serious challenge for increasing the I/O number. Second, the clock frequency of the memory bus has improved little for the past decade. Increasing the memory bus frequency is hindered by both the signal integrity problem and the increased power consumption. In addition to the performance bottleneck, the power efficiency of the system will also degrade as more data is moved around. Previous work has shown that a single memory access is orders more expensive in energy than performing a floating point computation [13].

From the application perspective, the emergence of big data workloads such as deep learning [14] and bioinformatics [15] have a great demand for both memory capacity and bandwidth, which is insatiable for conventional compute-centric architecture. A great number of these applications exhibit irregular memory access patterns, low spatial and temporal locality, low computational density and large working set. For compute-centric architecture, these features make it inappropriate to effectively utilize on-chip memory resources, incurring a large amount of off-chip memory traffic which is bound by memory bandwidth. For example, the DNA seeding algorithm used in bioinformatics involves a large number of fine-grained random memory accesses and can cause up to 93.24% last-level cache misses for the CPU [15]. In deep learning training tasks, a lot of operators have low arithmetic density and low temporal locality, thus becoming memory-bound on the GPU [14]. Even worse, as the deep learning model becomes more complex, the memory footprint is continuously increasing, requiring more main memory capacity.

Memory-centric architecture [16] either integrates lightweight computation units closer to the memory modules, or flattens the memory hierarchy to embrace more memory capacity for computation units. For the former case, as computation units can directly access internal memory buses, this architecture can provide improved memory bandwidth and better memory latency, further reducing the energy for memory accesses. For

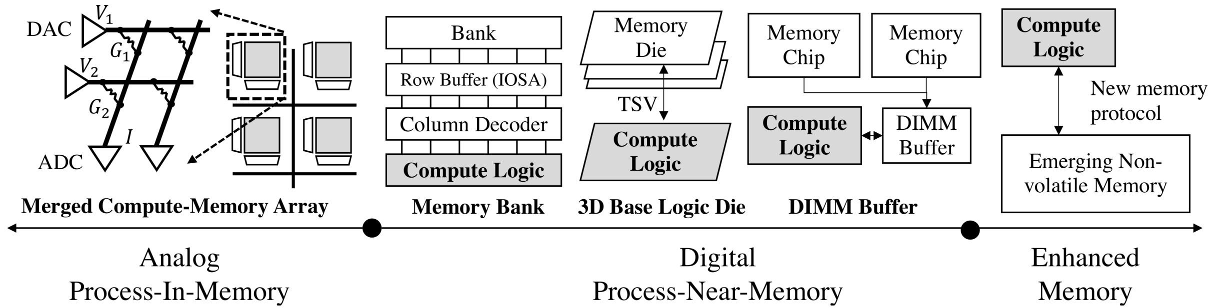


Figure 1.3: Classification of memory-centric architectures.

the latter case, as new memory technologies can increase the density of main memory systems, it is promising to augment the main memory capacity of the computing system. Thus, memory-centric architecture is a promising approach to solve the "memory bandwidth wall" and the "memory capacity wall".

1.2 Challenges and Opportunities

Previous memory-centric architectures can be classified into three categories according to the relationship between computation units and memory units as shown in in Fig. 1.3. From right to left, the computation units are more closely integrated with memory arrays and can have better memory bandwidth and latency, while the overhead of integrating computation units also increases. The trade-offs among different memory-centric architectures can lead to different designs for various application domains.

Analog Process-In-Memory. The first architecture is referred to as analog process-in-memory architecture. This architecture merges computation units inside the memory array by utilizing the memory cells attached to the same bitline to perform simple operations in analog domain. For example, crossbar circuits with emerging non-volatile memory devices such as Resistive-Random-Access-Memory (RRAM) can perform matrix multiplications using the voltage-current relationship [17]. There are two salient

opportunities of this architecture. First, by enabling memory cells to perform computations, tremendous bitline-level parallelism can be obtained to speedup embarrassingly parallel programs. For example, the previously mentioned crossbar circuits can store the weights of one matrix in the RRAM cells and use the input voltages to represents the weights of the other matrix, which accomplish $O(n^2)$ operations in unit time. Second, since computations are performed in-place, this architecture completely eliminates the data movements of one input operand, which also means the read bandwidth of that input operand achieves theoretical maximal number as long as there is no remote memory accesses involved.

However, two significant **challenges** exist for this architecture. First, the analog computing nature used in this architecture introduces noise and errors. How to deal with these errors and adapt mapping applications to this structure to tolerate these errors become important questions. Second, the modification of peripheral circuit of the memory array incurs significant area overhead and the complex peripheral logic could even add more energy consumption. For example, the analog-to-digital converters (ADC) used in the crossbar array takes majority of the area and energy consumption.

Digital Computation. The second architecture is referred to as digital process-near-memory architecture. This architecture integrates simple digital computation logics near the memory arrays and do not require modifications of the memory circuits as compared with the process-in-memory architecture. For example, previous work have explored integrating lightweight arithmetic units near each memory bank [18], placing simple low-power cores on the base logic die of the 3D stack [19], and adding simple processing units on the memory DIMM [15]. There are two **opportunities** associated with this architecture. First, the near-memory integration of computation logics is more feasible than the previous approach using existing technologies. Also, since the computation is performed

in digital domain, it introduces far less errors than the previous approach. Second, this architecture enables different tradeoffs between cost and performance depending on the locations where the computation units are integrated.

However, there are two obvious **challenges** associated with this architecture. First, the logic units added in the memory die using memory process incurs larger area overhead the the CMOS process. It is important to reduce this overhead while maintain the advantage of near-data processing. Second, previous architecture designs in this domain usually focus on certain applications without much programmability. It is necessary to enable better programmability of this architecture for a wider application domain to amortize the cost of hardware design and manufacturing.

Enhanced Memory. The third architecture is referred to as enhanced memory architecture. This architecture usually only requires changing the host's memory controller to support new memory protocols for novel memory technologies. The opportunity is that all hardware components are available for use.

However, its main challenge is that how to design a universal solution for various kinds of emerging technologies with varied performance characteristics. It is necessary to explore effective architecture solutions to support a wide range of novel memory technologies while achieving optimal performance.

1.3 Contributions

The **goal** of this dissertation is to study the analog process-in-memory, the digital process-near-memory, and the enhanced memory architectures and explore optimizations for them. The key challenges we would like to address is the lightweight computation units design for the previous two architectures and the flexible supports for various memory

for the third scheme.

Specifically, we have one analog process-in-memory design. Chapter 3 introduces QUANTMEC, which is a highly optimized memristor crossbar architecture to implement quantized deep neural network to support flexible quantization configurations and reduce peripheral circuit overheads.

We also have three digital process-near-memory design. Chapter 4 presents DLUX, which is a near-bank architecture for deep neural network training acceleration with both high performance and low hardware overhead. Chapter 5 introduces iPIM, which is 3D-stacking near-bank architecture for image processing applications. By using a decoupled control-execution architecture, iPIM supports programmability with small area overhead per DRAM die. Chapter 6 discusses MPU, which is a near-bank SIMT processor using a hybrid pipeline with an instruction offloading mechanism. By integrating lightweight hardware components on the DRAM die, MPU achieves a small area overhead for general purpose processing.

In addition, we have one enhanced memory design. Chapter 7 introduces NMTSim, which is a transaction-command based and cycle accurate simulator for new memory technologies, where a command issue optimization and an early notification functionality for transaction-command are incorporated.

As a conclusion, by utilizing the rich bandwidth provided with in-memory and near-memory architectures, and by exploiting the high density of emerging non-volatile memory technologies, we propose novel memory-centric architecture designs and optimizations that significantly improve the performance for memory-bound workloads. The techniques in this thesis highlight the contributions to strike a balance between memory-centric processing overhead and domain programmability, and create an infrastructure to enable architecture explorations in emerging memory technologies.

Chapter 2

Backgrounds and Related Work

In this chapter, we first discuss the fundamental knowledge about the main memory from both technology and architecture perspectives. Then, we survey the related work for process-in-memory, process-near-memory, and enhanced memory systems to highlight the uniqueness of the architectures proposed in this dissertation.

2.1 Backgrounds on Main Memory Technologies

This section will briefly introduce the device and circuit level backgrounds on DRAM and emerging NVM technologies.

Dynamic Random-Access Memory (DRAM) Technology. DRAM technology is widely used for commodity main memory modules with a wide range of application scenarios, such as embedded devices (LPDDR [20]), desktop and workstation (DDR [21]), and graphics and high-performance computing (GDDR [22] and HBM [11]). It uses the amount of electric charge in a capacitor to represent the stored information. An access transistor is contained in a DRAM memory cell to charge and discharge the capacitor,

performing memory accesses. Thus, DRAM has some unique properties due to its charge-based storage methodology. First, the reading of a DRAM cell is based on voltage-sensing, where the access transistor will connect the target DRAM cell with a local bitline, and the sense amplifier will distinguish the change of the voltage level in the bitline over a period of time. Second, a DRAM read is destructive where the charge is lost after accessing the memory cell, so a precharge operation is required to restore the previously accessed memory cell. Third, the DRAM is volatile, which requires power to maintain the stored information. Last but not least, since the charge in DRAM cells leaks off over time, DRAM requires peripheral circuits to periodically refresh the memory cells, which rewrites the memory cells and restore them to the original charge level.

Fabricating logic components in the DRAM process is possible but suffers from extra power and area overhead. The reason is that the DRAM process which optimizes for the memory density and the low charge leakage is incompatible with the CMOS process which optimizes for the transistor speed. Also, the DRAM process usually has much fewer metal layers (e.g. 3 metal layers) as compared with the CMOS process (e.g. 12 metal layers), so the routing overhead is larger to fabricate logic components in the DRAM process. Thus, it is extremely important to design lightweight logic when using the DRAM process.

Emerging Non-volatile Memory (NVM) Technology. Emerging NVM technology has been researched extensively for the past decades as a promising approach to replace DRAM based main memory, with pioneering commercial products [23] and novel standards [24] to support them coming out these days. It uses resistive cells which have different physical working mechanisms to represent the stored information. Some of the promising device candidates include PCM [25] which uses phase change material to archive two device states, STT-MRAM [26] which uses the magnetization direction of

the magnetic tunnel junction (MTJ) to represent states, and RRAM [27] devices which rely on metal-oxide conductive filaments to record states. Usually an access transistor is required to connect the cell with a bitline to reducing sensing noise and write disturbance. The emerging NVM has some salient features compared with traditional NVM technologies such as Flash [28]. First, emerging NVM has much better performance than the Flash, although with sacrifice to storage density. Second, emerging NVM are byte-addressable, which can be used as the main memory as compared with Flash which has block access granularity.

Due to the differences in fundamental device operating mechanisms, emerging NVM technologies have some unique features as compared with DRAM. First, emerging NVM devices uses current sensing instead of voltage sensing. This enables these devices to perform analog computations by manipulating the voltage-current relationships. Second, emerging NVM devices have asymmetric read and write performance, where the write latency is much worse than the read latency because of the need to change the physical states of the devices. Second, reading a cell will not disrupt its state, and emerging NVM is non-volatile, which means the data persists even when there is no power, and no refresh is required to keep its state. This feature motivates persistent memory, which has similar performance with DRAM, and data persistence as compared with Flash memory. Third, NVM devices are more compact than DRAM devices, which can provides more memory density and boost memory capacity. NVM devices also support multi-level cells, where a device can store multi-bit information.

Despite of the salient features of NVM, it has several significant drawbacks. First, like traditional NVM devices, emerging NVM devices can endure limited number of writes, which is a serious issue for write-intensive workloads such as deep learning training. Therefore, emerging NVM devices are more suitable for workloads with small write traffic such as deep learning inference tasks. Second, the write latency and the write power of

NVM devices are much worse than DRAM, causing another disadvantages for write-intensive benchmarks. Third, there exist various NVM media candidates with drastic different performance characteristics and management policies. Thus, a flexible and high-performance protocol is required to efficiently support various emerging NVM medias.

2.2 Backgrounds on Main Memory Architecture

This section will briefly introduce modern memory organizations and memory controller designs to serve memory access requests.

Main Memory Organizations. The main memory is organized in a hierarchical way and connected to the host processor through memory buses. When moving up from this hierarchy, fewer global data buses are available and these buses are shared among different memory entities, so the memory bandwidth decreases and the memory access latency increases as a result of bus conflicts. The lowest level of this hierarchy are memory arrays which are consisted of memory cells interleaved by local bitlines and wordlines, driven by local decoders and sense amplifiers. DRAM and emerging NVM memories have different circuit architectures for this level due to different operating mechanism. However, once the data is fetched into local row buffers of the array, the following data movement paths have little difference between these two memories. For better scalability, the local arrays are connected horizontally by global wordlines to form subarrays, and subarray row buffers are connected vertically by global bitlines to the global row buffer, forming a memory bank. Memory banks can be grouped together and connected to global data I/Os, forming a memory chip. Depending on different memory configurations, memory banks can be organized in very different ways. Commodity DRAM modules such as DDR memory organize memory banks into 2D memory chips, and the memory chips

are mounted on DIMM slots and connected to the host via off-chip memory buses which has 64 data I/O pins. To increase memory bandwidth with better power-efficiency, high-bandwidth memory adopts 3D die stacking architecture, where memory banks are first grouped in a 3D layer, and then connected to the base die by through silicon vias (TSVs). The base logic die then connects with the host processor using passive interposer in the same package, which allows 1024 data I/Os per 3D memory stack.

Memory Controller Basics. Memory controller is a bridge between the host processor and the memory modules. It receives the memory transaction requests sent by the host, translates them into device-specific commands, buffers and reorders these requests according to the device's timing constraints and performance optimization policies, and serve the memory requests when resources are available. It also helps perform some device specific management policies, such as refresh for DRAM devices and wear-leveling for emerging NVM devices.

2.3 Related Work on Process-In-Memory Systems

RRAM is an emerging non-volatile memory device that has the advantages of small cell size ($4F^2$), low operation energy and non-volatility [27]. RRAM crossbar structure can be used to accomplish vector-matrix multiplication by feeding voltage inputs to the rows of the crossbar and summing the current of each column of the crossbar. It can reduce the computation complexity of matrix-matrix multiplication from $O(n^2)$ to $O(1)$, thus significantly accelerating matrix multiplication [29]. Since matrix multiplication is the major operation and the bottleneck of neural network applications, memristive crossbars are suitable for implementation of neural network applications.

Previous work has explored how to use memristive crossbars for implementing full-

precision and fix-precision neural networks. ISAAC [30] has designed a full-fledged convolutional neural network engine with pipelined designs. Analog-to-digital converters (ADC) are the main area and power overhead of ISAAC. It also lacks the flexibility to support more types of activation functions. Another work, PRIME [31], proposes to use RRAM crossbar both as a memory and a computing engine. It maximizes the reuse of peripheral circuits so as to reduce the area and power overhead. PRIME also includes high ADC overhead. Tang et al. [32] propose an RRAM crossbar-based accelerator for binary neural networks in forward process. Although it has achieved significant improvement compared with CMOS design, it lacks the flexibility to support more diverse computations in a neural network considering the algorithms are evolving rapidly. It only supports binary mode but no full-precision mode, and the ADCs are all fixed to 4 bits resolution.

As a summary, all of the above mentioned hardware implementations lack flexibility to support more network types or neural network operations. Bit precision is also fixed once set.

2.4 Related Work on Process-Near-Memory Systems

Early Logic-in-DRAM Work. The early studies mostly adopt logic-in-DRAM style, where an entire core, including control logic, registers, arithmetic units, and data paths, is integrated inside a DRAM chip. The motivation is to exploit the internal bandwidth provided by the memory array inside the DRAM chip and reduce the data movement energy cost by tightly integrate logic components besides memory cells. However, these early logic-in-DRAM studies are not commercially successful due to the following reasons. First, the Moore's law continued scaling at that time, and the multi-core architecture design was a more cost-efficient approach for performance scaling. Second, the logic-in-

DRAM design has technology challenges of implementing logic in the DRAM process.

Based on the observation that the row buffer bandwidth is much larger than the DRAM chip-level bandwidth, EXECUBE [33] integrated CPU cores directly besides DRAM banks. The DRAM memory is partitioned into independent memory partitions, one per CPU. The cores can operate in: (1) MIMD mode, where each CPU obtains its own instructions from its own memory partition; (2) SIMD mode, where instructions can be sent (via the SIMD Broadcast Bus) from outside the chip directly into each CPU's Instruction Register. Computational RAM [34] utilized internal memory bandwidth by pitch-matching simple processing elements to memory columns (simple ALUs before column decoder). Computational RAM can function either as a conventional memory chip or as a SIMD computer. It also discussed alternatives for Process-Near-Memory: (1) a RISC processor connected after column decoders, and (2) non-pitch-matched processing elements connected by wiring. Terasys [35] proposes SIMD-style Process-Near-Memory logic: a standard 4-bit memory augmented with a single-bit ALU controlling each column of memory. Operations on data parallel operands are conveyed as memory writes through the Terasys interface board to PIM memory and cause the single bit ALUs to perform the specified operations at the specified row address across all the columns of the memory.

3D-stacked Processing Work. To overcome the bottleneck of memory bandwidth, the 3D-stacking processing-in-memory (3D-PIM) architecture provides a promising solution. This architecture embraces higher memory bandwidth by integrating compute-logic nearer to memory.

The first kind of 3D-PIM designs, process-on-base-die solution [36, 37, 38], places compute-logic on the base logic die to utilize the cube-internal Through-Silicon-Via (TSV) bandwidth, and demonstrates bandwidth advantages over GPU. To further un-

leash the bank-level bandwidth of 3D-PIM, the near-bank solution is proposed [39, 40, 41], which closely integrates compute-logic to each bank in the DRAM dies. It can provide around 10× peak bandwidth improvement compared with the previous solution since compute-logic directly accesses the local bank without going through limited TSVs.

AXRAM [40] proposes to integrate lightweight approximate multiply-accumulate (MAC) units inside DRAM dies (GDDR5), and offloading neural computation portion of the threads from GPU to in-DRAM accelerator. It proposes detailed mechanisms to offload computation to in-DRAM engines: When the warp reaches the in-DRAM code region, it instructs the streaming processors to send an initiation request to the memory controller (MC) and goes to halting mode, just like a context switch. MC periodically polls the DRAM memory-mapped mode register MR0 to determine if the computation has finished.

However, these solutions lacks programmability and focuses on a narrow application scenario. It is necessary to come up with more general-purpose and programmable designs to amortize hardware design and manufacturing costs.

2.5 Related Work on Enhanced Memory

To mitigate the impact of non-deterministic media access latencies in new memory technology devices, a recently proposed NVDIMM (Non-Volatile Dual In-line Memory Module) standard, NVDIMM-P [24], uses novel out-of-order transaction commands.

However, existing memory simulators are either unable to support the novel transaction features in NVDIMM-P, or confined to a limited media scope. Previous DRAM simulators, including DRAMSim2 [42], NVMain2 [43], and Ramulator [44], only employ deterministic DDR timing protocols. Significant modification efforts are required to add handshaking and transaction handling logic in the complex scheduling unit of memory

controller. Also, the passive memory module needs to add extensive new functionalities to become a media controller that can independently process host-issued commands. Previous NVDIMM simulators (e.g., FlashDIMMSim [28]) are customized for traditional flash media with block-granularity. While emerging NVMs have demonstrated better performance and byte-addressability, it is more promising to explore them as memory media for NVDIMM.

Chapter 3

QUANTMEC: Quantized Deep Neural Network on Memristive Crossbar with Dynamic Precision

This chapter focuses on accelerating various quantized deep neural network (DNN) applications using the memristive crossbar array and proposing solutions to reduce peripheral area overhead and support more primitive operations. There are three major challenges. First, quantized DNNs usually exhibit different precision requirements for various layers and workloads to fully exploit the performance benefit and energy improvement, thus requiring flexible architecture supports which are lacking in previous accelerators [30, 31]. Second, previous accelerators employs full-precision analog-to-digital converters (ADC), which introduces large area and energy overhead. Third, very few types of activation functions are supported by previous work, excluding their applications from many state-of-the-art deep neural networks.

In this chapter, we present a novel architecture named QUANTMEC with memristor crossbars that are highly customized for quantized DNN. First, QUANTMEC supports

configurable precisions for weights and activations in different layers as specified by the applications. The design adopts a bit-serial input interface and transfers different bit positions in the same weight vector to its corresponding columns in the crossbars. Layer by layer, the input and weight numerical precisions can be extended or reduced with great flexibility and mapped onto the hardware. Second, in order to support many common non-linear operations, we designed universal approximators based on dot-product engines for general function approximation. We also redesigned a general purpose pooling engine that supports both max pooling and average pooling with dynamic size. Third, to reduce ADC power and area overheads, some memristor crossbars are configured with full-resolution ADCs, and others are configured with reduced-resolution ADCs. Significant bits of the output values are computed with high precision crossbars, and low-order bits of the output values can be processed with lower-precision crossbars to trade for power savings. QUANTMEC supports dynamic precision output of intermediate results of quantized DNN layers and is more energy efficient due to reduced precise hardware in comparison with the state-of-the-art [31] [30]. Experimental results of AlexNet [45] on ImageNet [46] and VGG [47] on Cifar-10 dataset [48] show that QUANTMEC is able to achieve 22.36% energy savings without accuracy loss and 27.95% energy savings with $< 3.8\%$ accuracy loss compared with full-precision implementations.

The contributions of this chapter are summarized as follows:

- We propose a highly optimized memristor crossbar architecture to implement quantized DNN which supports flexible configurations.
- We propose a hybrid-precision ADC design to reduce its power and area overhead. The memristor crossbars in the proposed architecture are divided into high-precision and reduced-precision configurations with trade-offs between accuracy and energy efficiency. DNN applications that tolerate noise in low-order bits could

be computed with less precise memristor crossbars for energy savings. For models requiring high precision, we provide an encoding scheme to configure the reduced-precision crossbars for high-precision computation, offering great flexibility for application mapping.

- The architecture supports many non-linear functions at the tile level with a crossbar-based universal approximation engine. Design space explorations are carried out to study the impact of bit precision and hidden layers on the approximation accuracy.

3.1 Motivation

3.1.1 Hardware Acceleration of QDNNs

Pioneering work using application-specific circuits to implement QDNN demonstrates better performance and energy-efficiency results compared to general purpose processing units like GPU or CPU. These ASICs can be classified by whether they use emerging technologies such as resistive RAM, in-situ computation, or the standard digital CMOS technology. YodaNN [49] is a system-on-chip design based on CMOS technology that is optimized for BinaryConnect [50] CNNs. It has multiple banks for filters and images, and has compute units for sum-of-product and channel summation. In order to improve the energy-efficiency, it adopts a latch-based standard memory cell design which adapts the supply voltage of the architecture according to the performance requirements of the application. The structure of this design is still based on a von Neumann architecture, so the memory wall challenge cannot be eliminated. Second, it adopts a full-precision adder tree structure for summing the results, which is not energy efficient compared to some approximate computing schemes since the quantized network can utilize an approximate computing scheme for more power savings. Tang et al. [32] propose an RRAM crossbar-

based accelerator for binary neural networks in forward process. This design discusses the matrix splitting problem and the pipeline implementation. Although it has achieved significant improvement compared with CMOS design, it lacks the flexibility to support more diverse computations in a neural network considering the algorithms are evolving rapidly. It only supports binary mode but no full-precision mode, and the ADCs are all fixed to 4 bits resolution. Also, it only has one type of activation function, which is not sufficient according to DoReFa-Net [51]. Moreover, the design adopts a signal splitting method where the negative and positive weights are represented using 2 crossbars respectively. In comparison, a two's complement fixed-point representation can be used to stand for positive and negative values, which saves area by using 1 crossbar. Yu et al. [52] recently fabricated a 16 MB chip to implement a Binary Neural Network. The solution demonstrates a simple network that could fit into a single crossbar, with all inputs as 1 bit. It is a good demonstration of Neural Network on Crossbar, but it is not very applicable for general use. In more complex benchmarks, which are used by the majority of researchers, the inputs are multi-bit and the the size of a single layer is too large to fit into one crossbar. Besides, this design uses fixed ADC resolution which either incurs large area and power overhead, or accuracy will be degraded.

3.1.2 Computing With Memristive Crossbar

RRAM is an emerging non-volatile memory device that has the advantages of small cell size ($4F^2$), low operation energy and non-volatility [27]. RRAM crossbar structure can be used to accomplish vector-matrix multiplication by feeding voltage inputs to the rows of the crossbar and summing the current of each column of the crossbar. It can reduce the computation complexity of matrix-matrix multiplication from O(n^2) to O(1), thus significantly accelerating matrix multiplication [29]. Since matrix multiplication

is the major operation and the bottleneck of neural network applications, memristive crossbars are suitable for implementation of Neural Network applications.

Previous work has explored how to use memristive crossbars for implementing full-precision neural networks. ISAAC [30] has designed a full-fledged convolutional neural network engine with pipelined designs. ADCs are the main area and power overhead of ISAAC, and by adopting a quantized neural network, there is chance that ADC overhead could be reduced [32]. Also, ISAAC does not support a Batch Normalization layer which has been proven to be necessary [53] and used in the majority of the mainstream CNN models. It also lacks the flexibility to support more types of activation functions. Another work, PRIME [31], proposes to use RRAM crossbar both as a memory and a computing engine. It maximizes the reuse of peripheral circuits so as to reduce the area and power overhead. PRIME also includes high ADC overhead that could be alleviated through the use of quantized neural networks.

All of the above mentioned hardware implementations lack flexibility to support more network types or neural network operations. Bit precision is also fixed once set. This work proposes to utilize the intrinsic fault-tolerant properties of quantized neural networks to radically reduce the precision of intermediate results without loss of final network precision. Also, this work adopts a universal approximator to approximate the various computing operations used in the network, thus allowing for more flexibility. Finally, this network could be configured as both a full-precision mode and a reduced-precision mode, allowing for more design choices for the network.

3.2 Architecture Design

The design overview of QUANTMEC is shown in Fig. 3.1(a). First, the deployed model for inference provides detailed input configuration information to the QUANT-

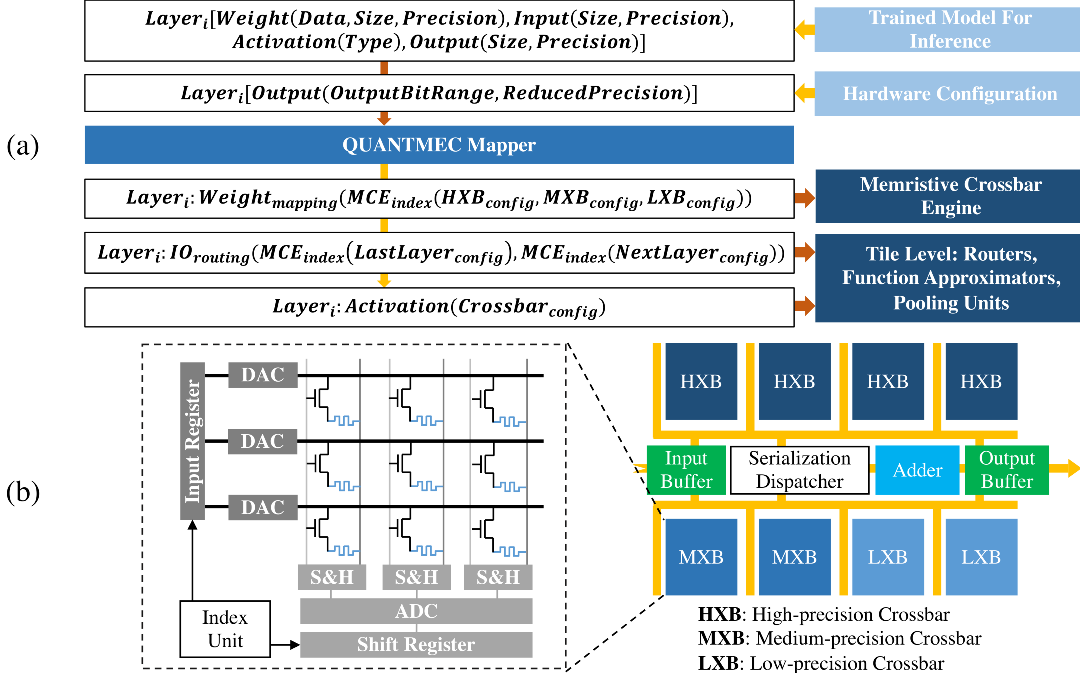


Figure 3.1: (a) Overall software hardware mapping flow for QUANTMEC. (b) Memristive Crossbar Engine design

MEC crossbar hardware after the QDNN model is fully trained. The configuration information includes the input, weight, output, and activation configurations of each layer. The application should also provide extra hardware configuration to configure the corresponding output precision for different output bit ranges. Then, to implement the input QDNN model, the QUANTMEC hardware mapper will generate corresponding mapping data for the Memristive Crossbar Engine (MCE), including the precision configuration for weights, input/output routing configurations, and crossbar configurations for each layer's activation function.

The overview of QUANTMEC architecture design is illustrated in Fig. 3.1(b). A QUANTMEC chip consists of several tiles connected by an on-chip interconnection network to support large QDNNs which cannot be allocated in a single tile. Each tile contains buffers for storing intermediate results, a number of memristive crossbar engines (MCE) and some analog approximate computing engines. These components are

connected through shared buses. Some auxiliary functional engines are also incorporated at the tile level, including shift-and-add units and bit-serial dispatchers. Every MCE has several crossbars with ADCs and analog summing units connected with shared buses. Each analog approximate computing engine has sigmoid units implemented by the crossbar as well as max-pooling units and quantization units. The details of MCE and tile designs will be discussed later in following sections.

3.2.1 Memristive Crossbar Engine

The array level design is shown in the left dotted box of Fig. 3.1 (b). The array is a 128x128 1T1R crossbar with 2 bit precision in each cell. Each column of the array stores the two-bit position of the corresponding weight, and the input is a bit-serialized input. To perform the computation, the input is fed into the input register and the DAC will transfer the input to the two-bit input of the crossbar array. The current is summed over the same column and buffered in the sampling and hold circuit. The analog value stored in the sampling and hold unit is transferred to a digital voltage value by the output ADC. The output DAC is time-multiplexed to be used among all the output bit-lines in the same array, so one crossbar will only have one output ADC. Because each output's weight depends on the input and the bit position of the weight, an index unit will indicate what the bit position of the output is. The bit position of the output is fed into the shift register and shift the corresponding output value by the DAC.

3.2.2 Tile Design

The tile-level design is shown on the right of Fig. 3.2. The data is stored in the input buffer and will be sent to the available MCEs for bit-serialized vector multiplication. For some large vectors that cannot fit in a single MCE, several MCEs will be used to

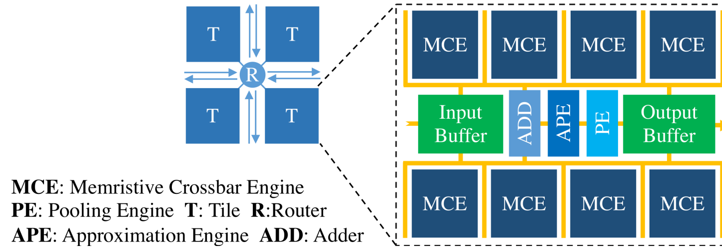


Figure 3.2: Tile level design

produce the partial results, which will then be sent to the adder for summation. After the partial results are summed in the adder, the output is fed into an approximation engine. The approximation engine acts as a non-linear activation function and will be introduced in Sec. 3.2.3. After this stage, the output could be fed into pooling engine for pooling operations or could go directly into the output buffer.

3.2.3 Function Approximator

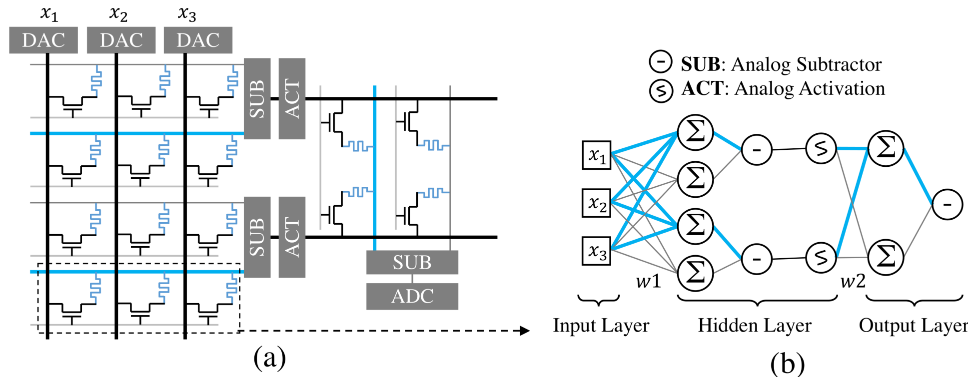


Figure 3.3: Approximation Engine (APE) design

The function approximator design is described in Fig. 3.3. The function approximator acts as a non-linear activation function after the convolutional layers or fully connected layers. Previous work (ISAAC [30] and PRIME [31]) all assumed fixed sigmoid functions for activation functions. While previous work has shown that the type of different activation functions is important for the precision of the neural network, it is also important

to support the flexibility of user-defined activation functions.

$$y = g(x) \approx \sum_{j}^{m} f(\sum_{i}^{n} x_i \times (w1^{+}_{i,j} - w1^{-}_{i,j})) \times (w2^{+}_{j} - w2^{-}_{j}) \tag{3.1}$$

Previous work has proposed to use memristive crossbar for function approximation. The universal approximation theorem [54] has proven that a three layer feed-forward neural network with a hidden layer and non-linear activation function is able to approximate any non-linear function to any given precision. Supposing the function to be approximated is $y = g(x)$, Equation 3.1 shows the input-output mapping of the neural network where $f()$ is a non-linear activation function. The circuit implementation of this neural network is shown in Fig. 3.3 (a) and all the corresponding connections and components are demonstrated in Fig. 3.3. Unlike the two's complement representation of Memristive Crossbar Engine (MCE), to support both positive and negative weights, the original weights are divided into two parts which are both positive and could be stored in RRAM, as shown by the blue and grey lines in network weights $w1$ and $w2$ in the bottom of Fig. 3.3. The subtraction of partial inner product is done by an analog subtractor, and the output is fed into an analog activation unit. The analog activation unit is a current-input-voltage-output circuit composed of several transistors and it resembles a sigmoid function. The output layer uses a linear activation unit without any additional components after subtraction. This design is power efficient and fast since it accomplishes all computation in the analog domain and takes advantages of the vector inner product of the dense memristive crossbar.

In order to incorporate this function approximator into QUANTMEC, we need flexible support for digital and analog interfaces, and we must also consider the effect of the limited precision RRAM. To simplify the input interface and to support dynamic input length, the input number is represented in two's complement and will be fed into the

parallel 1-bit DAC inputs in one cycle. The cells in the columns without input bits will be set to highest resistance state so that they will not interfere with other input columns. To support dynamic output resolution, an ADC which can be configured from 1 to 8 bits is used so that output resolution can be traded for speed and power savings [55]. The influence of limited RRAM precision and crossbar size on the approximation accuracy for different non-linear functions will be discussed in Sec. 3.4.7.

3.3 Software Support

3.3.1 Flexible Bit Precision for Quantized Neural Network

The precision of the weights and the input for the network can be adjusted. The computation of both the convolutional layers and fully connected layers can be decomposed into a vector inner products, so here we can analyze how QUANTMEC could support flexible bit-precision for the vector inner products. We assume that the input vector $\bar{X}$ and the weight vector $\bar{W}$ have length N, that is $\bar{X} = (x_1, x_2...x_N)$ and $\bar{W} = (w_1, w_2...w_N)$. We assume the crossbar has size $R \times R$. This paper will set $R = 128$ since previous work has reported the successful fabrication of 128×128 1T1R array with 2 bit precision cell. For deep neural networks like VGG [47], most of the inner products involved have length $N > R$, so a single vector needs to be divided into $M = \lceil N/R \rceil$ sub-vectors and the partial results will be summed up to get the final result. For cases where $N < R$, a single crossbar could contain the whole vector, so this case is trivial to discuss. After the above mentioned discussion, we will analyze the vector multiplication done on a single crossbar where $\bar{X'} = (x_1, x_2...x_R)$ and $\bar{W'} = (w_1, w_2...w_R)$. We assume the partial vector inner product result to be $\bar{Y'}$.

We use fixed-point representation here and assume the input has p bits of precision

with p_f fraction bits and the weight has q bits precision with q_f fraction bits. For a single product:

$$\begin{aligned} x_i \times w_i &= (\sum_k^p x_{i,k} \times 2^{k-p_f}) \times (\sum_v^q w_{i,v} \times 2^{v-q_f}) \\ &= 2^{-(p_f+q_f)} \times \sum_k^p \sum_v^q x_{i,k} \times w_{i,v} \times 2^{k+v} \end{aligned} \tag{3.2}$$

We could replace this equation into the partial inner product equation to get the following expression:

$$\begin{aligned} \bar{Y'} &= \bar{X'} \cdot \bar{W'} = \sum_i^R x_i \times w_i \\ &= 2^{-(p_f+q_f)} \times \sum_k^p \sum_v^q (\underline{\sum_i^R x_{i,k} \times w_{i,v}} \times 2^{k+v}) \\ &= \gamma \times \sum_k^p \sum_v^{q/m} (\underline{\sum_i^R \{x_{i,k} \times (\sum_{h=0}^{m-1} w_{i,mv-h} \times 2^{m-h})\}} \\ &\times 2^{m(v-1)+k}) \end{aligned} \tag{3.3}$$

In the above equation, the underlined terms calculate one inner product of a crossbar column and one input vector, and $2^{-(p_f+q_f)}$ is represented as γ for convenience. The result will be shifted $k+v-(p_f+q_f)$ if the RRAM has 1 bit precision or $m(v-1)+k-(p_f+q_f)$ bits if the RRAM has m bits precision by the Shift Register in Fig. 3.1 (b). The indices k, v, p_f, q_f, m will be kept in the Index Unit in Fig. 3.1 (b) and fed into the Shift Register in each computing cycle. Using single-bit RRAM or multi-bit RRAM has different power, area overhead and performance characteristics and will be discussed in detail in Sec. 3.4.2.

3.3.2 Mapper for Hybrid ADC Scheme

Previous work on ADC interfaces to crossbar computing engines either adopts a fixed high-resolution ADC (ISAAC) or a dynamic-resolution ADC with $1 \sim P_o$ bits (PRIME). A fixed-resolution ADC will introduce considerable area and power overhead in ISAAC's IMA level design. Although dynamic-resolution ADC in PRIME permits flexible output precision, all the outputs support the same dynamic precision range, which results in less savings in power and area. We adopt a different method in this paper to further reduce ADC overhead. For each crossbar, the output has a fixed-resolution ADC, but different crossbars have different ADC precision ranges. Since each ADC is only dedicated to one precision range, the ADC is much simpler than the full-range dynamic-resolution ADC in PRIME and could save more area and power when doing computation. For example, each MCE in our design has 4 high-precision crossbars (HXB), 2 medium-precision crossbars (MXB), and 2 low-precision crossbars (LXB) as shown in Fig. 3.1 (b). Since we assume $R = 128$ and 2-bit precision cell in this paper, the HXB's ADC will have 8 bits resolution. We set the resolution of MXB's ADC to be 6 bits and the resolution of LXB's ADC to be 1 bit.

According to the above equation, different output bit positions will be computed independently. For a fixed $k + v$ value, we could arrange the corresponding bit position of weight vectors in the same crossbar using the same resolution ADC. The basic idea is to use low-resolution ADC for computing the less significant bits in the output and the high-resolution ADC to compute the more significant bits. We set two thresholds for the resolution level:

- If $k + v \geqslant t_2$, then the corresponding output bit must be computed using a high-resolution ADC.
- If $t_2 > k + v \geqslant t_1$, then it could be computed using a medium-resolution ADC.

- If $t_1 > k + v$, then it could be computed using a low-resolution ADC.

The algorithm to determine which bit position in the weight vector to be sent to which type of crossbar is described in Algorithm 1.

Algorithm 1: Mapping Weight Bits to Crossbars with Different ADC Resolution

Input: Threshold Value: T_1, T_2, Input Precision p, Weight Precision q
Output: Sets of Bit Positions in the Weights for Crossbars with Different ADC Precision

for $v = 1 : q$ **do**
  if $v < T_1$ **then**
    if $v + p < T_1$ **then**
      Bit Position v is assigned to LXB
    else if $v + p < T_2$ **then**
      Bit Position v is assigned to MXB and LXB
    else
      Bit Position v is assigned to HXB, MXB, and LXB
  else if $T_1 \leq v < T_2$ **then**
    if $v + p < T_2$ **then**
      Bit Position v is assigned to MXB
    else
      Bit Position v is assigned to HXB and MXB
  else
    Bit Position v is assigned to HXB

For some medium- or high-order bit positions in the weight vector, this algorithm will duplicate one bit position to several crossbars with different ADC resolutions. This kind of redundancy is acceptable because there are abundant high-density and small footprint memristive crossbars in each tile so the redundancy will introduce low area overhead. On the other hand, this redundancy allows some low-order bits in the output to be calculated using a low-resolution ADC, which could save power. This duplication also allows parallelism where different bit positions in the input vector are calculated with the same bit positions in the weight vector, which could accelerate computation. The redundancy, power savings and performance improvement will be discussed in the evaluation section.

This architecture also provides the opportunity to operate in a full spectrum of precisions. In our design, the full-resolution ADC is 8 bits, the medium-resolution is 6 bits and the low-resolution ADC is 1 bit. A special encoding scheme is proposed so that the effective ADC range could be covered from 1 bit to 8 bits. The idea is to duplicate one row to N rows and the corresponding input is also duplicated. Thus if an output will lose K bits of resolution originally, after encoding the inputs and weights, the effective loss in resolution will be reduced to $K - log_2(N)$ bits. For example, by encoding 1 bit position using 4 crossbar rows, the crossbar with a 6-bit ADC could improve 2 bits of resolution in the output. The trade-off is that a crossbar will support fewer rows (a shorter length of vectors in the inner product) for improved precision, which will be discussed in the evaluation section.

This flexibility to configure low-precision crossbars as high-precision ones is very important because in some circumstances, the low-order bits of outputs that could suffer from a loss in precision are very confined. For example, for outputs with a 32 bit-width, only 2 low-order bits are allowed to be computed with precision loss. The crossbars with low-resolution ADCs could be set to operate in improved-precision mode to avoid hardware under-utilization. In the case of binary neural network, since the inputs and weights are both confined to $+1$ or -1, which are 01 and 11 in 2s complement representation respectively, applying reduced-precision output on the inner product will introduce unacceptable loss in precision. In this circumstance the whole network is computed in full-precision mode enabled by the flexibility of the above scheme. For further optimization, since the last bit of 01 and 11 is 1, the LSB of the output of the product will always be 1 and is not required to be computed.

3.4 Experiment

3.4.1 Setup

The hardware architecture of QUANTMEC is based on ISAAC [30] but makes some changes compared to the previous architecture. At the array level, one QUANTMEC Memristive Crossbar Engine (MCE) has 8 crossbars, among which 4 crossbars are configured with 8-bit (high-resolution) ADCs, 2 crossbars are configured with 4-bit (medium-resolution) ADCs and the last 2 crossbars are configured with 1-bit (low-resolution) ADCs. This combination is chosen according to the design space exploration in Section 3.4.4 and Section 3.4.5. The memristive crossbar chooses the size of 128×128 and each cell has 2-bit precision according to the results in Section 3.4.2. For each crossbar there is a shift-adder. The input register has a size of 2KB and the output register has a size of 256B.

As for the tile level, there are 12 MCEs, 1 Pooling Engine (PE) and 1 Approximation Engine (APE). The tile has 1 eDRAM buffer with $64KB$ capacity, 2 banks and $256b$ bus width for input data. The output register is configured with $3KB$ capacity. 4 tiles will share one router with 32-bit flit size and 8 ports. The buses that connect eDRAM to each MCE use 384 parallel wires to satisfy intra-tile traffic bandwidth. A HyperTransport serial link model is used for off-chip links, similar like the one in ISAAC.

For buffers and all on-chip interconnects, CACTI 6.5 [56] is used for modeling energy and area at 32 nm. The 1T1R memristive crossbar's parameters are derived from a fabricated chip [57]. The shift-and-add circuits in MCE and the adders and comparators in the PE reference the design of DaDianNao [58]. The analog sigmoid circuit in the APE is adapted from a compact design [59]. The analog subtractor is based on a simple operational amplifier and simulated in FreePDK [60]. Also, the Serialization Dispatcher in MCE and the MUX and DEMUX in PE is designed using FreePDK. All of the above

	Convolution	FC
vgg1	3x3,64(1)-3x3,128(1)-3x3,256(2)-3x3,512(2)-3x3,512(2)	4096(2)-1000(1)
vgg2	3x3,64(2)-3x3,128(2)-3x3,256(2)-1x1,256(1)-3x3,512(2)-1x1,512(1)-3x3,512(2)-3x3,512(2)-3x3,512(2)-1x1,512(1)	
vgg3	3x3,64(2)-3x3,128(2)-3x3,256(3)-3x3,512(3)-3x3,512(3)	
vgg4	3x3,64(2)-3x3,128(2)-3x3,256(4)-3x3,512(4)-3x3,512(4)	
msra1	7x7,96(1)-3x3,256(5)-3x3,512(5)-3x3,512(5)	
msra2	7x7,96(1)-3x3,256(6)-3x3,512(6)-3x3,512(6)	
msra3	7x7,96(1)-3x3,384(6)-3x3,768(6)-3x3,896(6)	
alexnet	12x12,96(1)-5x5,256(1)-3x3,384(2)-3x3,256(1)	
resnet	7x7,64(1)-3x3,64(6)-3x3,128(8)-3x3,256(12)-3x3,512(6)	1000(1)
deepface	11x11,32(1)-9x9,16(2)-7x7,16(1)-5x5,16(1)	4096(1)-4030(1)

Table 3.1: The benchmarks and network architectures

components are scaled to 32 nm.

For apples-to-apples comparison, the ADC components adapt the same design and scaling methodology as ISAAC. A successive approximation ADC [61] with $1.2Gbps$ sampling rate and 8-bit resolution is used as the baseline analysis. The design has four major components: a vref(voltage reference) buffer, memory, clock, and a capacitive DAC. ADCs of different resolutions but with the same style as the baseline design are derived by scaling the power/area of the vref buffer, memory, and clock linearly, and the power/area of the capacitive DAC exponentially [62]. The 1-bit ADC adopts a simple sense amplifier design, and its area and energy is acquired from NVSIM [63]. Since each input channel for the crossbar uses 1-bit DAC for input, the DAC in this design uses an area and power efficient capacitive DAC model [62].

At the tile level, the neural network models are manually mapped to each MCE to achieve the throughput and resource balance between each layer. The routing will ensure any data conflicts are avoided in tile and MCE computation. The intra-layer computation follows a parallel architecture design, thus enabling maximum throughput in a balanced way. For inter-layer traffic, the computation follows a dataflow model where the computation of the next layer will start once enough, but not all, data is generated by

the previous layer. Since different bit positions in weight vectors are already configured in crossbar during inference, the Serialization Dispatcher will be responsible for recording the index of bit position of inputs to each memristive crossbar and give them to shift-and-add units for results accumulation. Thus, given fixed input and weight precision and neural network models, the dataflow and resource consumption is deterministic and can be calculated.

Benchmarks: We use five benchmarks to simulate the energy, area and performance of the QUANTMEC design using state-of-the-art CNNs and DNNs as summarized in Table 3.4.1. For convolutional layers, the number in the front indicates kernel size and the number of kernels, respectively. For fully connected layers (FC), the number in the front indicates the output size. For both layer types, the number in the parenthese represents the repeating instances of such layers. We choose AlexNet (ILSVRC 2012) [45], four versions of Oxford VGG (ILSVRC 2014) [47], three versions of MSRA (ILSVRC 2014) [64] and ResNet (ILSVRC 2015) [65] for testing CNN applications. For DNN applications, we choose DeepFace [66]. The testing dataset uses ILSVRC 2012 [46] and cifar-10 [48].

3.4.2 Analysis of RRAM Bit Precision and Crossbar Size

Since high-resolution ADC have high area and power overheads, RRAM precision and crossbar size are restricted by the bit resolution of the ADC. Assuming the same output ADC resolution, there is a trade-off between RRAM precision and crossbar size with the relationship described in ISAAC [30]. On one hand, increasing crossbar size will increase the number of inputs and outputs, thus boosting the throughput of MCE. On the other hand, increasing the RRAM precision to m bits will increase the storage density for a crossbar array and allow one input to a crossbar to generate multiple bits of final output

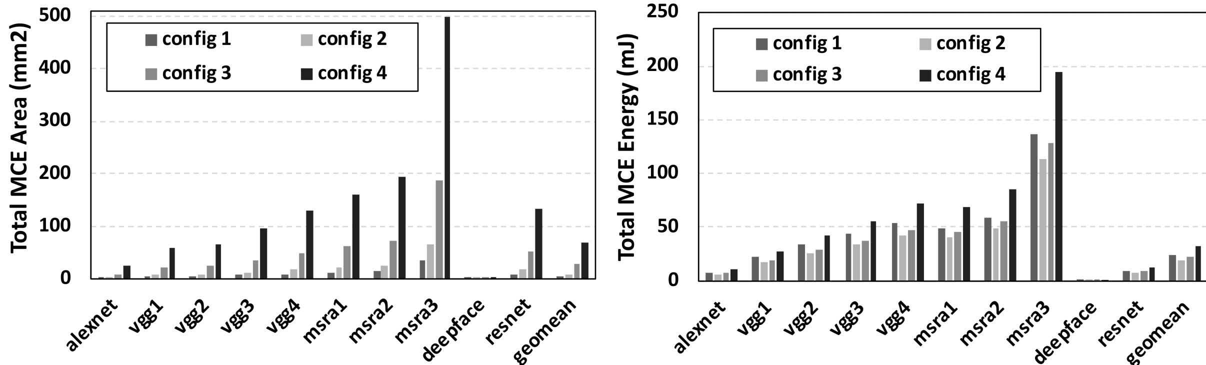


Figure 3.4: MCE area/energy consumption under different configurations of RRAM precision and crossbar size

values. To explore the trade-off, we use four configurations summarized in Table 3.4.2. Since the input is bit-serialized, the crossbar with width=R and W-bit RRAM precision then needs $log2(R)+W$ of bits ADC resolution. We use the bit-encoding scheme adopted in ISAAC to further reduce one bit from output ADC, so all configurations need 8-bit ADC resolution.

configuration	1	2	3	4
crossbar size	256	128	64	32
RRAM bit(s)	1	2	3	4

Table 3.2: Configuration of RRAM and crossbar

We use the benchmarks described in Section 3.4.1 and calculate the total area of MCEs needed to map all the parameters of one network model. We then calculate the energy consumption of MCEs for computing one input for the given model. According to Fig. 3.4, when RRAM precision increases and crossbar size decreases, the total MCE area increases exponentially. Also observed in Fig. 3.4, the total MCE energy consumption will drop initially, then increase when RRAM precision rises and crossbar size drops. The reasons are threefold. First of all, since the multiplication of kernel size and the number of input channels are very large in most layers, a large vector inner product needs to be disassembled and offloaded to smaller crossbars. Using a larger crossbar size will reduce

the number of crossbars needed for a long vector. Thus, an inner-product will go through fewer AD/DA conversions in a computation, which is the main overhead in the MCE. Second, due to the same output ADC resolution, the reduction of precision in an RRAM cell could be traded for a larger crossbar size, thus the number of weight bits a crossbar could store does not decrease. Third, 1T1R crossbars only contribute to a small part of the total MCE area and energy consumption, so increasing the crossbar size will not deteriorate much of the overall performance of MCE. Therefore, in the optimal design we adopt crossbar size=128 and RRAM precision=2 bit with full-resolution output ADC resolution=8 bits.

3.4.3 Impact of Reduced Precision ADC on Accuracy

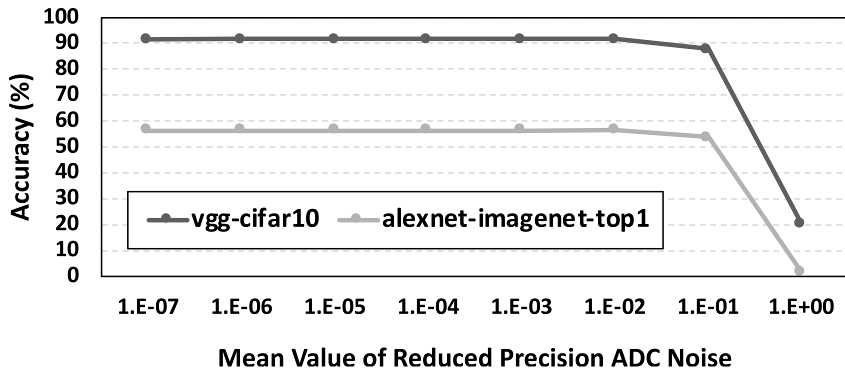


Figure 3.5: Reduced precision ADC's impact on recognition accuracy

We use AlexNet to test the top-1 recognition accuracy of ImageNet 2012 and VGG to test recognition accuracy of cifar-10. The input and weights are first quantized to 16-bit fixed-point representation. According to our simulation results in Sec. 3.4.4, the approximate scheme will generate noise in a Gaussian distribution with a mean value of relative error from $1 * e^{-11}$ to $1 * e^{7}$ and $stdv = 0.1 \cdot mean$. Thus, noise with Gaussian distribution is inserted into each channel of the output in each convolutional and linear layer and increases gradually to observe the decrease in recognition accuracy. According

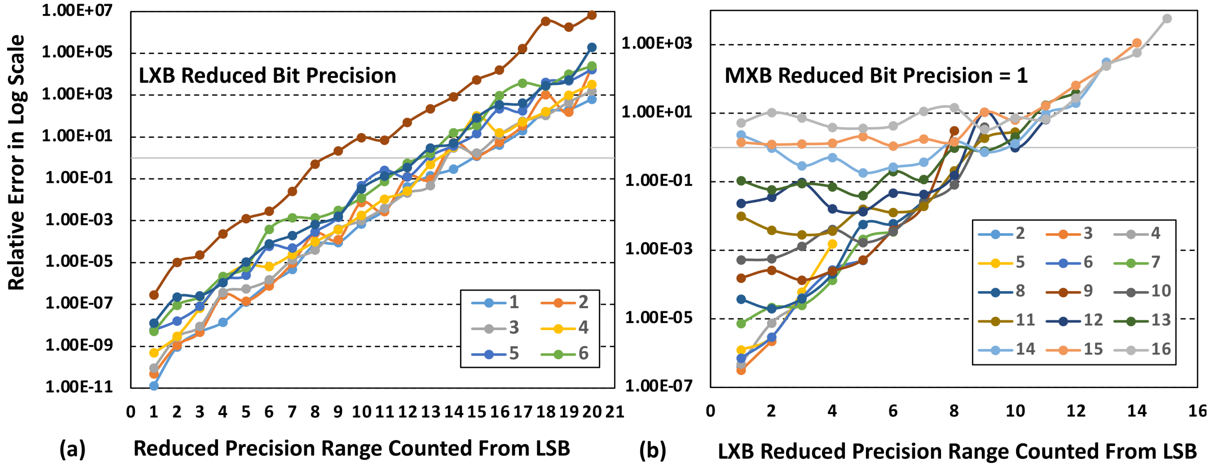


Figure 3.6: Inner product precision loss using one/two threshold level(s)

to Fig. 3.5, the accuracy will not drop until $mean_relative_error(mre) = 1e^{-2}$. When mre increases to $1e^{-1}$, there is a slight drop in accuracy ($< 3.8\%$). Beyond that value, the accuracy drops sharply. Thus, in the following analysis, we give two configurations with no loss in accuracy and a small loss ($< 3.8\%$) in accuracy.

3.4.4 Analysis of Number of Threshold Levels

In the proposed approximate computing scheme, for the same vector inner product, an output bit position with index (*not output value*) under certain threshold value could be computed using reduced-resolution ADC to save energy and area. Here we explore two schemes with different threshold levels. In the one-threshold scheme, the output bit position index less than T1 is computed using LXB and the higher bit position is computed using HXB. The two-threshold scheme is based on the previous scheme: the output bit position index less than T2 and larger than T1 is computed using MXB, and the higher bit position (>T2) is computed using HXB. To explore the accuracy drop of inner products using different schemes and different configurations, we traverse the design space as shown in Fig. 3.6. The relative error is computed by dividing the error

between the approximated result and the accurate result by the accurate result. For one configuration, a Monte Carlo simulation is run with the given distribution of the input, and an average value is collected.

Threshold Levels	1						
Reduced Resolution	1	2	3	4	5	6	7
No Loss	11	11	11	10	9	9	9
Loss<3.8%	12	12	12	12	10	10	10

Table 3.3: Best reduced resolution range with one threshold

Threshold Levels	2				
Reduced Resolution	1	2	3	4	5
No Loss	11-4	11-6	11-6	11-4	10-6
Loss<3.8%	13-5	13-7	13-5	12-5	12-4

Table 3.4: Best reduced-resolution range with two thresholds

In the one-threshold scheme (Fig. 3.6(a)), the horizontal axis shows the reduced resolution range counted from LSB (T1), and different lines represent how many bits are reduced from the full-resolution ADC output. For example, when reduced bit resolution=2, that crossbar uses $8 - 2 = 6$ bits ADC for the output. It is observed that when the reduced resolution range is increased or the output resolution of the ADC is reduced, the relative error for the inner product will rise gradually. In the two-threshold scheme (Fig. 3.6(b)$\sim$(f)), different figures represent different values of reduced bits of resolution for MXB. In each figure, the horizontal axis shows the reduced resolution range counted from LSB (T1). Different lines represent reduced resolution range counted from MXB (T2). The output resolution of LXB ADC is set to 1 bit. It is observed that different combinations of T1 and T2 may generate similar relative error values which could be explored for the best configurations of energy consumption, replication overhead, and throughput, as discussed later.

The optimal reduced resolution ranges (MXB reduced resolution bits in two thresh-

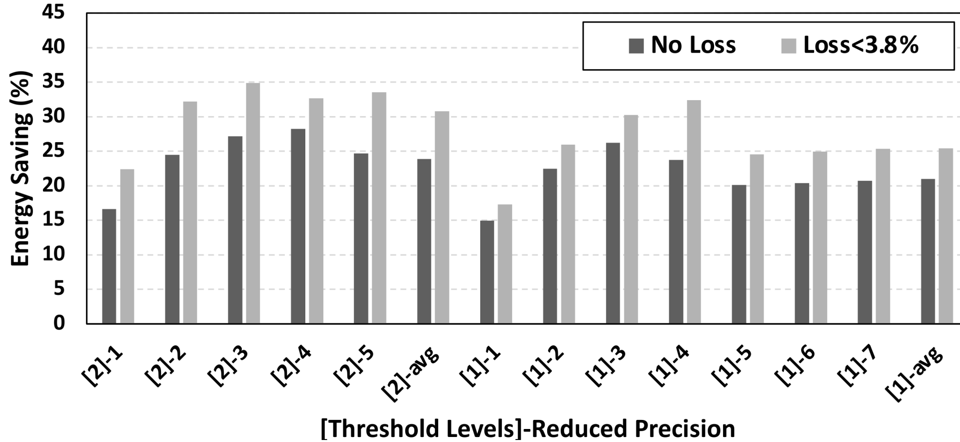


Figure 3.7: Power savings of different settings of reduced-resolution ADCs compared with full-resolution mode. The energy is calculated in inference mode by classifying one input image using the VGG model on cifar-10 dataset.

old mode) under different configurations that achieve the least energy consumption are summarized in Table 3.4.4 and Table 3.4.4. In Table 3.4.4, the first number in the pair represents the MXB reduced resolution range and the second number in the pair represents the LXB reduced resolution range. Using these optimal configurations, the energy needed for processing one input image for the VGG model is calculated and compared with the energy used in full-precision modes. The resulting energy savings are plotted in Fig. 3.7. On average, using two threshold level ($[2]-avg$) achieves more energy savings compared with using only one threshold levels ($[1]-avg$). The reason is that two threshold levels allows the medium reduced resolution range (MXB) to be further extended to reduced energy despite little shrink in low reduced resolution range (LXB). Within each threshold scheme, there is a trade-off between reduced bit resolution and the range for reduced bits for energy savings. To achieve the same relative error, reducing bit resolution will result in the reduction of the range for reduced bits, thus lowering or offsetting the energy savings. Thus, to achieve the maximal energy savings, the optimal reduced bit resolution for MXB is 3 bits in two-threshold mode and the optimal reduced resolution for LXB is 4 bits in one-threshold mode. Moreover, when loss in accuracy ($< 3.8\%$) is

acceptable, 2% ∼ 9% more energy savings could be gained compared with configuration with no precision loss.

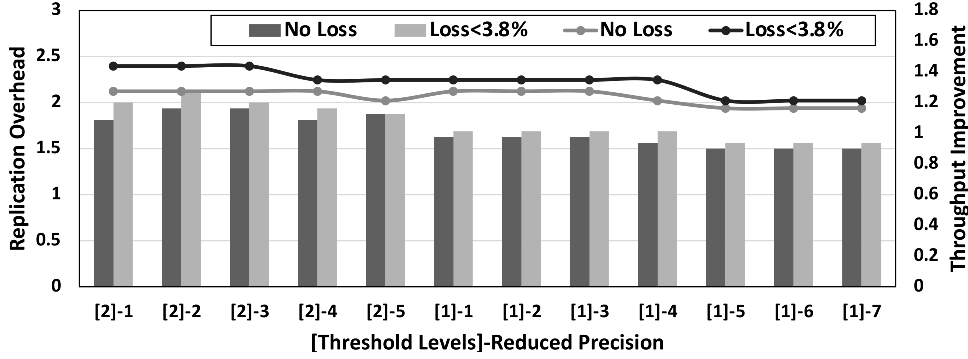


Figure 3.8: Trade-off between replication overhead and throughput improvement among different settings of reduced-resolution ADCs compared with full-resolution mode.

We further compare the replication overhead and the throughput improvement for different configurations discussed above. The replication overhead is calculated by dividing the number of columns used in one weight vector in reduced-precision mode, by its counterpart in full-precision mode. Because columns in HXB must be replicated to MXB and LXB while in reduced-precision mode, replication overhead is always more than one. The throughput improvement is calculated by dividing the rounds of computations (one round is defined as one input bit position computing with one weight bit position) in reduced-precision mode by its counterpart in full-precision mode. As observed in Fig. 3.8, higher replication overhead will result in better throughput improvement. Thus, configuration with acceptable loss ($< 3.8\%$) has slightly larger replication overhead and better throughput improvement compared to the configuration without loss in accuracy.

3.4.5 Analysis of Operation Mode

In order to deal with circumstances where applications require high accuracy and thus more crossbars for high-precision computing, we propose an encoding scheme to use low-

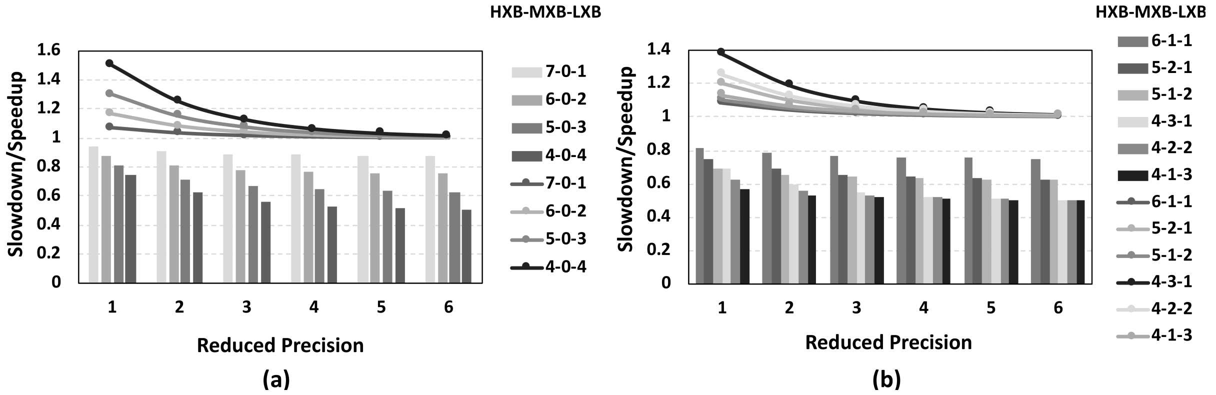


Figure 3.9: Using (a) one threshold and (b) two thresholds for reduced resolution ADC.

precision crossbars (MXBs and LXBs) for high-precision computation. The encoding will replicate one row in high-precision mode into multiple rows in low-precision modes. For example, in MXBs with 2 bits of loss in output resolution, one row will be replicated to $2^2 = 4$ rows to achieve a full-resolution output. We explore the throughput speedup ratio of the proposed scheme compared with the scheme where MXBs and LXBs are simply switched off, as well as the slowdown ratio compared with the scheme when all crossbars are HXBs. The results of different threshold modes are shown in Fig. 3.9. The first, second and third number in the legend shows the number of HXBs, MXBs and LXBs in one MCE. As observed, reducing the number of HXBs results in a large decrease in slowdown ratio but a slight increase in speedup ratio. This indicates that for designs that often need to run high-precision applications, it is better to configure more HXBs.

3.4.6 Area and Power Decomposition

The design principle of QUANTMEC is to reduce the percentage of area and power overheads of ADCs in the MCE level without reducing application accuracy and maximally improving the performance. The power and area comparisons of QUANTMEC with ISAAC at the memristive crossbar engine (MCE) level are summarized in Table 3.4.6.

	QUANTMEC		ISAAC	
	area (mm2)	power (mW)	area (mm2)	power (mW)
Total	0.0089	26.89	0.0131	33.89
ADC	60.54%	33.47%	73.17%	47.21%
DAC	1.91%	14.88%	1.30%	11.80%
S+H	0.45%	37.19%	0.30%	29.51%
Crossbar	2.24%	8.93%	1.52%	7.08%
S+A	2.70%	0.07%	1.83%	0.06%
IR	23.54%	4.61%	16.01%	3.66%
OR	8.63%	0.86%	5.87%	0.68%

Table 3.5: Area/Power decomposition of QUANTMEC and ISAAC

QUANTMEC achieves $\sim$ 32.0% area savings and $\sim$ 20.7% power savings in comparison with ISAAC. The reason is because QUANTMEC adopts a hybrid-ADC scheme, which significantly reduces ADC overhead. QUANTMEC reduces $\sim$ 12.6% ADC area percentage and $\sim$ 13.74% ADC power percentage at the MCE level compared to ISAAC. It is noticed that QUANTMEC adopts the same crossbar size and RRAM cell precision of ISSAC. Also, due to the hybrid precision scheme, the same weight will be duplicated to both high precision crossbars and reduced precision crossbars. Thus, the throughput of QUANTMEC will also improve due to replication of weights.

3.4.7 Design Space Exploration of Approximation Engine

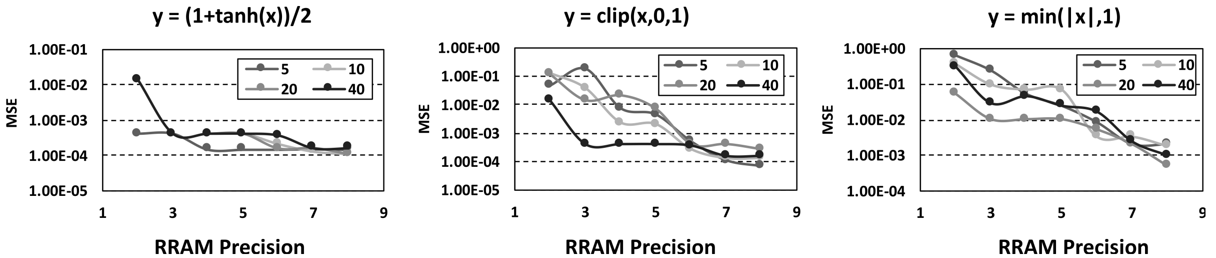


Figure 3.10: Memristive Crossbar Approximator design space exploration

To verify that this approximation scheme could work under limited crossbar size and RRAM precision, the change of approximation accuracy is explored for the target

function by varying the number of hidden layer neurons and RRAM precision. The target approximation function is chosen according to DoReFa-Net and the corresponding design space exploration is demonstrated in Fig. 3.10. The approximation precision is expressed as mean squared error between the ideal output and the approximate output. Here we limit the input precision to 16 bits, as shown to be feasible in deep neural network by previous work [67]. First, we fixed the hidden layer size, which is restricted by the crossbar size. Then, we train the network with varying RRAM precisions. Since the the fractional part of the fixed point representation in each RRAM is undetermined, we choose the best fraction part result as the approximation accuracy of a fixed RRAM precision. The maximum RRAM precision is set to 8 bits per cell [68].

According to the design space exploration, nonlinear functions which possess similar behaviors as the sigmoid circuit used in the function approximator in Fig. 3.3 can be approximated with great accuracy using a small network size and low RRAM precision. For example, the sigmoid function $y = 1/(1 + exp(-x))$ can be approximated using 5 hidden neurons and 2 bits of precision per cell with $mse = 2.15e^{-4}$. For other networks, certain combinations of network size and RRAM precision could result in ideal approximation accuracy. Generally, for the same network size, the approximation accuracy will improve as RRAM precision increases. However, the accuracy will not always improve as network size increases because a larger network size will result in overfitting and unstable behavior in reduction of numerical precision. Since we adopt the 1T1R structure for the memristive crossbar, the crossbar size is not a limiting factor. Thus we could reduce the RRAM bit precision by using an acceptable crossbar size while still satisfy the approximation accuracy. Using Fig. 3.10, we could do a Pareto optimization to find the point approaching the bottom-left corner. For example, $y = (1 + tanh(x))/2$ could be approximated with $mse = 4.17e^{-4}$ with 5 hidden neurons and 2 bits of precision per cell which is acceptable for neural network applications. In order to gain higher precision

with $mse = 1.09e^{-4}$, 10 hidden neurons and 8-bit precision per cell need to be used, which induces much more hardware than the previous example. The design space exploration could be carried out to find the optimal balance between hardware complexity and approximation accuracy.

3.5 Summary

In this chapter, we introduce a precision-configurable memristive crossbar engine for dot-product operations, which are the most compute-intensive and memory-demanding operations. Taking advantage of the fault-tolerant properties of the QDNN, we further reduce power and area overhead and support more flexibility in the hardware by making the following contributions: (i) We propose a scheme to reduce the intermediate precision of the output of the dot-product engine without reducing the final accuracy of the network. (ii) We propose an architecture that could support dynamic precision for both the inputs and weights of the network (iii) We apply the universal approximation theorem to support any non-linear function used in the QDNN, and discuss the influence of the precision of the weights on the results. Our experimental results demonstrate that the proposed approximate architecture can greatly reduce ADC overhead and improve throughput. It achieves 22.36% energy savings without accuracy loss and 27.95% energy savings with $< 3.8\%$ accuracy loss on state-of-the art DNN models.

Chapter 4

DLUX: a LUT-based Near-Bank Accelerator for Data Center Deep Learning Training Workloads

This chapter aims to solve the challenge of accelerating memory bandwidth bound deep neural network (DNN) training workloads in the data center, using the 3D stacking processing-in-memory (3D-PIM) architecture [69, 70, 71, 72, 73] with a near-bank design. This near-bank architecture enables computation resources and memory bandwidth to scale-up synergistically with increasing 3D stacking layers, significantly reduces the in-cube data movement, and allows the DNN training logic to fully utilize bank-level bandwidth without changing the DRAM timing. Nevertheless, the low-area overhead hardware design in memory process [74] and the associated software mapping and scheduling techniques remain key challenges to be addressed. From the hardware perspective, a lightweight floating point (FP) unit design is required, and the expensive cache needs to be replaced without performance penalty. From the software perspective, efficient algorithms need to be invented to increase the utilization of the proposed FP

unit, and novel mapping and scheduling schemes are required to hide the latency and allow flexible data layouts.

In this chapter, we propose a low hardware overhead near-bank 3D-PIM architecture, DLUX, for DNN training acceleration providing high FP performance. We propose both the hardware design and the software mapping/scheduling techniques to solve the challenges mentioned above. From the hardware perspective, to support efficient FP arithmetic with low area overhead, we propose to use an in-DRAM lookup table (LUT), which trades memory capacity for computing performance. In order to reduce the lookup overhead, we adapt a hierarchical LUT structure, where the full LUT table is stored in a DRAM bank but the LUT entries are cached in a small buffer in each bank's peripheral. To reduce the cache overhead, we use a simple scratchpad memory buffer for data reuse, and a transformation unit to assist flexible data layouts. From the software aspect, we solve the mapping and scheduling problems from both the intra-layer phase and the inter-layer phase. During the intra-layer phase, to reduce LUT fetching overhead, LUT entries in the LUT buffer are reused a number of times before reloading. Then, to achieve high concurrency and low data movement among banks, the input data parallelism and the intermediate result stationary scheme are used. During the inter-layer phase, transparent and low overhead techniques are invented to ensure input-output layout consistency and forward-backward layout transpose.

The specific contributions of this chapter are listed as follows.

- We propose a near-bank architecture, DLUX, for DNN training acceleration with both high performance and low hardware overhead.
- We demonstrate the DLUX design, with the highlight of the in-DRAM hierarchical LUT for high-performance FP computing, efficient communication using shared data bus and lightweight support for data transformation.

- We present the DLUX software design. The intra-layer mapping/scheduling improves utilization, concurrency while minimizing data movement, and the inter-layer data transformation ensures layout consistency in dataflow processing.

- We evaluate DLUX and compare it with the Tesla V100 GPU. The results shows DLUX provides on average 6.3× end-to-end speedup and 42× energy-efficiency improvement on representative data center training workloads.

4.1 Motivation

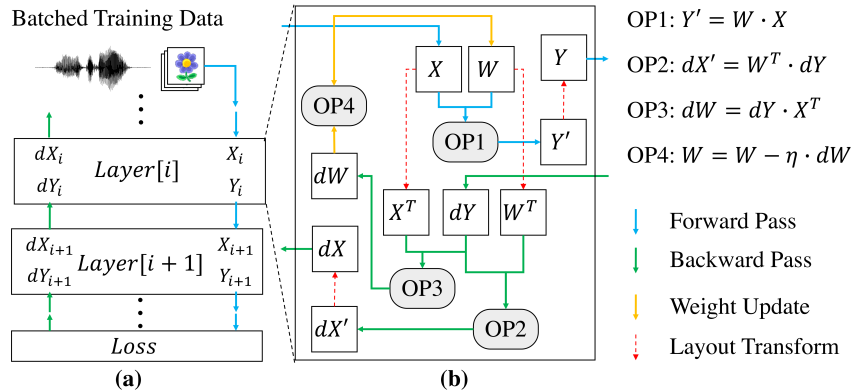


Figure 4.1: **Typical data flow and operations in DNN training. (a) Inter-layer data flow. (b) Intra-layer data flow for a fully-connected layer.**

The DNN training data flow The DNN training process is very memory demanding, and can be abstracted as a data flow graph shown in Fig.4.1, represented as (a) inter-layer and (b) intra-layer data flow. For the inter-layer data flow, each iteration will feed the input data with a certain batchsize into the network, propagate the intermediate results of each layer (Y_i, Y_{i+1}), calculate the loss at the final layer, and then back-propagate the gradient (dX_i, dX_{i+1}) through all layers. For the intra-layer data flow, the backward pass $(OP2, 3, 4)$ requires more computation than the forward pass $(OP1)$. Even worse,

Application	Type	BatchSize	DataSet	Notation
Recommendation	MLP	1000	MovieLens	Recommendation [75]
Speech Recognition	RNN	32	TIMIT	DeepSpeech [76]
Translation	Attention	4096	WMT	Transformer [77]
Text Summarization	RNN	4	Gigaword	TextSum [78]
Sentence Encoder	RNN	128	BookCorpus	SkipThoughts [79]
Compression	RNN	4	Kodak	EntropyCoder [80]

Table 4.1: **Benchmark setting.**

it needs to store the intermediate results (X), which significantly increases the memory overhead. Also, the layout transformation is required, since the layouts of the input and output tensor need to be consistent (Y_i and X_{i+1}), and the backward pass requires transposed format (X^T, W^T).

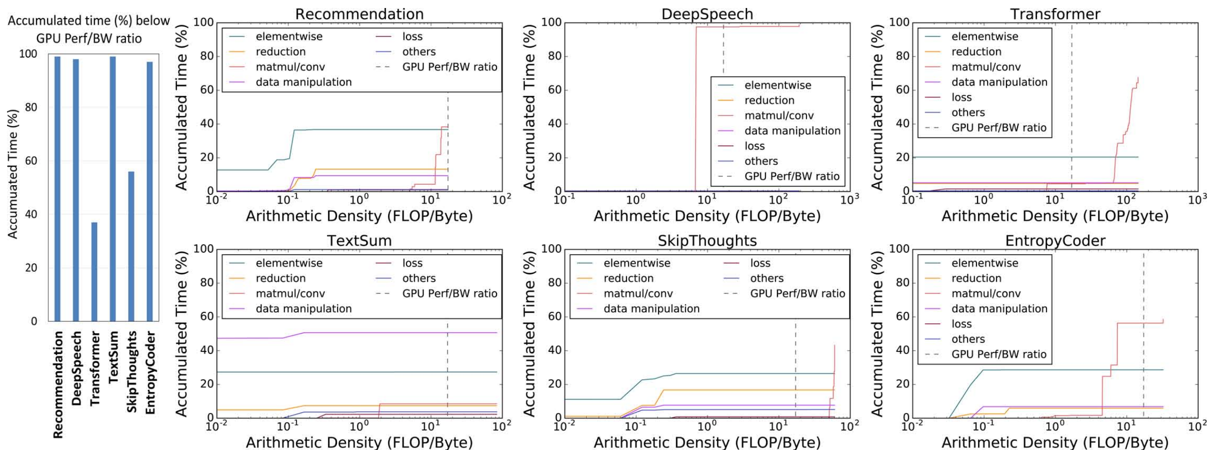


Figure 4.2: **Left: the percentage of time for operations that have lower arithmetic density than GPU performance/bandwidth ratio. Right: accumulated percentage of time distribution according to arithmetic density for different classes of operations.**

A case study for DNN training workloads We conduct a detailed profiling of representative workloads in Table.4.1. We use tensorflow [81] to record the computation instructions, the memory access count, and the execution time of every operations in the benchmarks. First, to analyze the memory-bound behavior, we accumulate the execution time of operations with arithmetic density lower than GPU performance/bandwidth

ratio ($17.5 FLOP/Byte$) as shown on the left of Fig.4.2. Second, to understand the performance bottleneck for each class of operations, we derive the trend of accumulated percentage of time as arithmetic density increases for each class on the right of Fig.4.2. To plot a data point with arithmetic density $X_{density}$ in the trend line, we first sum the execution time of all operations of that class whose arithmetic density is lower than $X_{density}$. Then, we divide the added time by the total execution time of that application to calculate the accumulated percentage of time for that data point. For the results, we first find that these workloads spend a large amount of time ($40\% \sim 100\%$) executing bandwidth-bound operations on GPU. More detailed analysis shows that except for General Matrix Multiplication ($GEMM$), other kernels exhibit memory bound behaviors (plateau occurs left of the GPU Perf/BW ratio according to the arithmetic density distribution). For $GEMM$ kernels, significant portion of time also shows memory bound behaviours. We also discover that most of the training execution time ($> 95\%$) is dominated by 4 categories of kernels: $GEMM$, *Elementwise* (e.g., elementwise addition), *Reduction* (e.g., tensor reduction along one axis), and $DataManipulation$ (e.g., tensor concatenation).

3D Stacking Memory Though with different interfaces, the architectures in various 3D memory standards are similar. One memory *cube* contains a base logic die and multiple stacking DRAM dies. The logic die carries the control circuit and the physical layer (PHY). In one DRAM die, the *subarray* is the minimal DRAM cell array with the dedicated local decoder and the sense amplifier. A group of subarrays that share the global data lines form a *bank*. A bank has a *row buffer* to support data burst and provides spatial locality. A group of banks in the same DRAM die forms a *bank group* and are linked by shared data buses. Several bank groups in different DRAM dies are connected with Through Silicon Vias (TSVs) to controllers in the base logic die, forming a vertical

vault. Note that DLUX is based on general 3D memory and is adoptable for both High Bandwidth Memory (HBM) [11] and Hybrid Memory Cube (HMC) [82] architecture.

Area Overhead in the DRAM die The 3D-PIM's performance challenge is aggravated by the requirement for area-expensive FP units (1.6× larger than an integer unit for 32-bit Multiply–Accumulate (MAC) in 45nm [83, 84]). To alleviate the bandwidth bottleneck of the base die logic integration, a near-bank design places logic inside the DRAM die. However, building complex logic inside the DRAM die results in significant area overhead due to specialized DRAM technology (as high as 80% [74] overhead). In Sec.4.2, DLUX overcomes this area-constrained performance challenge by using a configurable DRAM-based LUT as the extra computing resource.

Slow DRAM-based LUT Using the DRAM as a LUT for computing is non-trivial for achieving high performance. Different from sub-ns fast SRAM-based LUT, DRAM row access latency is tRC (e.g., 48ns [11]). With every DRAM bank serving as a LUT, the extra performance gain from LUTs of a typical 8-die memory cube is marginal 0.01TFLOPS. In Sec.4.2.1, DLUX overcomes this slow LUT challenge by introducing a buffer for the LUT and the software scheduling method to improve its data reuse.

Data Movement A compute-centric accelerator employs customized on-chip interconnect network for efficient inter-node data communication. However, PIM is based on the 3D memory, which only has shared buses for interconnecting. Although the shared data buses are wide thanks to 3D integration, operations requiring inter-bank communication will incur long latency due to shared bus data congestion. In Sec.4.2.2, DLUX leverages this shared data bus structure for bankgroup-local and vault-local data movement.

Requirement for Layout Transformation As shown in Fig.4.1 (b), since the input-output layout needs to be consistent between DNN layers, and the forward-backward layout needs to be properly transposed, potential data movement introduced by the layout transformation is time and energy consuming. Two unique features of PIM make this problem more challenging. First, unlike unified memory abstraction for compute-centric architecture, PIM adopts a distributed memory model, where data are partitioned so that the majority of the data feeding for compute units come from local banks. Improper data partitioning will introduce unnecessary inter-bank data movement, such as all-to-all broadcasting during data transpose. Second, PIM lacks a complex cache design which can be effective in hiding memory latency and allowing flexible memory access patterns. If data placement in the local bank results in poor spatial locality in row buffer, frequent activations of different rows will result in inefficient intra-bank data movement. DLUX first adds light weight logic in Sec.4.2.3, and then uses layout transformation, locality-aware mapping, and PIM-friendly partial transpose format in Sec.4.3.2 to address this challenge.

4.2 Architecture Design

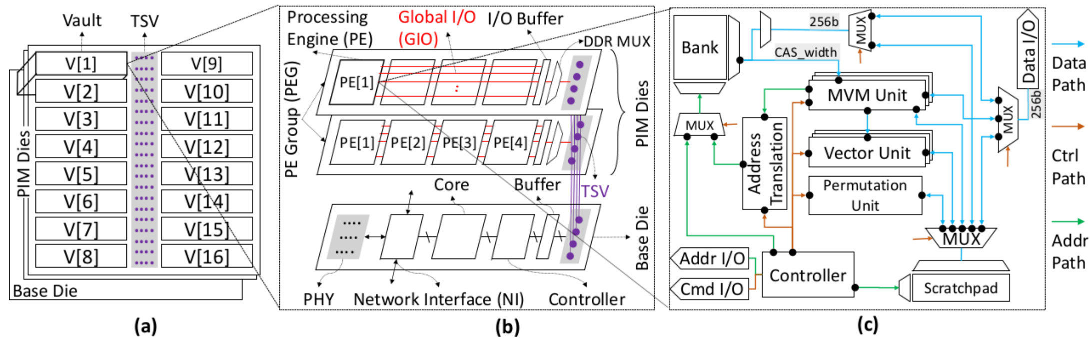


Figure 4.3: **DLUX architecture overview: (a) a cube, (b) a vault, and (c) a processing engine.**

The DLUX design highlights the near-bank architecture with efficient supports for the LUT-based FP operations, the hierarchical shared data buses for efficient data communication, and the lightweight hardware for layout transformation. To fulfill these features, DLUX adopts a scalable architecture composed of cubes, vaults, processing engine groups (PEGs), and processing engines (PEs), as shown in Fig.4.3. A DLUX cube consists of 8 processing-in-memory (PIM) memory dies on the top of a base logic die, connected by TSVs. Each cube is further divided into 16 vertical vaults, each of which owns $64b$ TSVs spreading across 8 layers, as shown in Fig.4.3(a). Fig.4.3(b) zooms into a vault. The base logic die of a vault contains a network interface (NI) for inter-vaults and inter-cube data communication, a simple programmable ARM core to provide friendly user interface and issue DLUX instructions, a hardwired controller, and a buffer. For the other PIM dies in the vault, each die contains one PEG, which executes kernels and communicates with other PEGs in the same vault through the shared TSVs. A PEG contains 4 PEs. They are all connected to a $256b$ global I/O (GIO) bus.

Fig.4.3(c) further zooms into a PE. The PE employs the near-bank architecture, in which computing logics are in the bank peripheral region, without any modification to the memory array. Such architecture fully exploits the high bank-level bandwidth, while remaining manufacture friendly [40, 85, 41].

In the bank peripheral region, we design computing units, control units, data paths, and a scratchpad memory. The computing units include matrix-vector-multiply (MVM) units, vector units, and a permutation unit. The control units include a controller and an address translator. The data paths include the links connecting (1) the computing units and the scratchpad memory; (2) the scratchpad memory and the bank/GIO; (3) the bank and the MVM units/GIO. We double the data bus CAS_width and the bank-level row buffer to reduce communication overhead between the bank and other units. The scratchpad memory supports both the computation units and the communication and

layout transformation operations.

4.2.1 Computation Support

LUT-based Multiplication The key idea is to increase the memory-side FP performance with a limited area budget by leveraging part of the DRAM memory for computing, i.e., using DRAM as a LUT for the FP arithmetic implementation. However, there are two challenges: (1) the exponential memory capacity demand, and (2) the DRAM's long latency.

To overcome the large LUT capacity challenge, instead of looking up the whole FP32-MAC operation directly (2^{98}Byte memory), we only lookup the most area consuming part of the arithmetic. We find that a FP32-MUL's area is 1.8× larger than that of a FP32-ADD [86]. Inside the FP32-MUL, the significand MUL contributes $\sim 87\%$ of the total area. Therefore, we only use LUT for the significand MUL, while implementing other parts with digital logic circuits. Furthermore, even in the 23b significand MUL, we only conduct LUT for the partial product of a $12b \times 4b$ MUL, while adding partial product adders in digital circuits. To overcome DRAM's long latency challenge, we propose a hierarchical and buffered LUT architecture. We first lookup the first operand from the DRAM bank, and store the partial LUT results in the faster SRAM based buffers. Then, we lookup the second operand from the faster buffers. DLUX scheduling will optimize data reuse from the SRAM buffer (i.e., the first operand stays unchanged). Note that the SRAM buffer only stores all possible results given a known first operand, so it is much more efficient than the all-logic FP multiplier.

Fig.4.4 shows the detailed design. As illustrated, we only use the LUT for partial product results in the significand MUL, i.e., Sig-A×Sig-B. Others including the addition of partial product in the significand MUL, the exponent addition, and the normalization

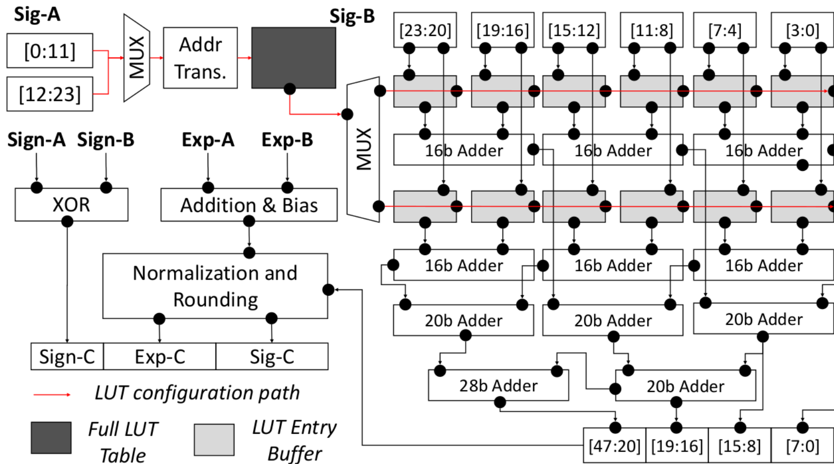


Figure 4.4: **Hierarchical lookup table based FP32 multiplier design.**

etc. are implemented with digital logic circuits. All the computations shown in Fig.4.4 happen in the DRAM layer in each PE, so there will not be inter-layer data movement. The hierarchical LUT architecture has the full LUT table stored in the dense but slow DRAM bank, while having the lightweight but fast SRAM buffers allocated outside of each bank. The first operand (Sig-A) is used as the row address of the DRAM bank (after a simple pre-loaded address translation) to fetch one entry to the SRAM LUT buffer, and the second operand (Sig-B) is used as the column address of the LUT buffer to get a partial product result. These partial results are then summed up by the digital adders.

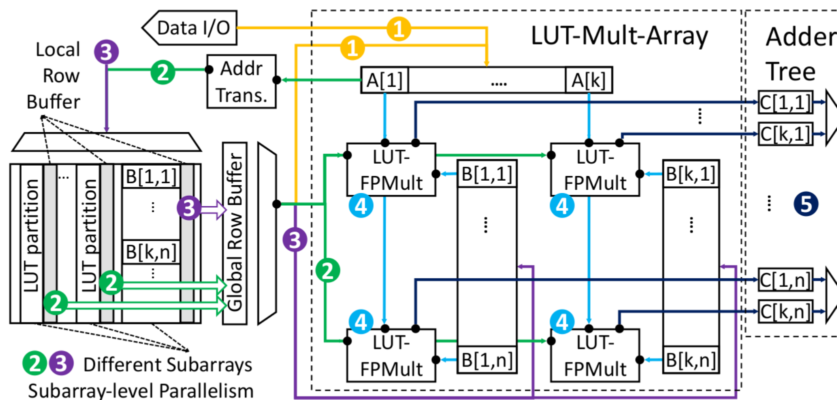


Figure 4.5: **Matrix-vector-multiply (MVM) unit design.**

Matrix-Vector-Multiply (MVM) We build the MVM unit using the lookup table based floating point multiplier (LUT-FPMult), while exploiting the faster LUT buffer, i.e., maximizing the reuse of the first operand. The design and the working flow is shown in Fig.4.5. We denote the input vector as $A[1:k]$ and the input matrix as $B[1:k, 1:n]$. We assign A as the first operand to lookup the full LUT table inside DRAM (like Sig-A in Fig.4.4) and B as the second operand to lookup the LUT buffer. The MVM working flow is divided into five steps: ❶ $A[k:1]$ is fetched from the local bank or the broadcast data from the GIO, to the LUT index buffer. ❷ Values from the LUT index buffer are translated to fetch the corresponding entries of the full lookup table in the DRAM bank. The results are stored in the LUT buffers in each LUT-FPMult. ❸ B matrix is fetched from local bank to the input vector buffers. ❹ Each $B[i, j]$ decodes the LUT buffer to complete the FP32-MUL computing. ❺ The results will be summed ($\sum_{i=1}^{k} A[i] \cdot B[i, j]$) by the adder tree. Such working flow maximizes the LUT buffer locality. Each LUT buffer result is reused n times, since each $A[i]$ will be multiplied with all values in vector $B[i, :]$. We further optimize the performance by hiding latency during data fetching. We place the full LUT table and B matrix data into different subarrays of the same bank, and employ subarray-level parallelism [87] to overlap the time of ❷ and ❸.

Vectorized Computing and SFU The vector unit contains an array of FP32-ADD, FP32-MUL, and other simple logical units implemented by digital logic circuits. It has two functions. First, it accumulates and updates the partial results stored in the scratch-pad memory with new output values from the MVM units. Second, it performs simple elementwise operations, including addition and logical operations (e.g., min) for DNN layers such as ReLu. To support more complex non-linear functions, we add one SFU (Special Function Unit) per PE. This unit enables frequently used non-linear activation functions in training, such as *sigmoid* and *tanh*.

Permutation Unit The permutation unit consists of multiple cyclic-shift FIFOs. The input is fed column-wise to each FIFO, and the output is fetched row-wise where one FIFO contributes one element in the row. The data transposition is scheduled after the completion of a layer and is overlapped with the computation of the next layer.

4.2.2 Communication Support

An efficient communication scheme is critical for DLUX's performance and energy efficiency. From the hardware perspective, DLUX adopts a distributed memory model owing to its PIM nature, where accesses to non-local addresses will introduce expensive inter-PE data movement. From the application perspective, many important DNN kernels incur significant communication traffic due to the requirement for the scattering (e.g., data-sharing) and the gathering (e.g., reduction) computation patterns.

To meet these requirements, we propose to fully exploit and reuse the already existing hierarchically shared bus architecture in 3D memory for data movement. First, 4 PEs in the same PEG share $256b$ GIO, which is used for the inter-PE communication. Second, 8 PEGs in the same vault share $64b$ TSVs, which are used for the inter-PEG communication. Third, 16 vaults communicate through the on-chip network, which is used for the inter-vault communication. Last, each cube has a full-duplex serial link with peak bandwidth $80GB/s$ [88], which is used for the inter-cube communication.

Fig.4.6 illustrates an example of using the hierarchical interconnects to perform a tensor reduction operation. Fig.4.6 (a) shows the reduction pattern among four tensors, where each hardware unit contains one tensor. For load balancing, each tensor is partitioned into four parts with equal size marked as $p1$ to $p4$, and each unit is responsible for reduction of a single part among the four tensors. For example, $Unit1$ will collect all $p1$ parts from the other three units and perform local accumulation. Fig.4.6 (b)-(d) shows

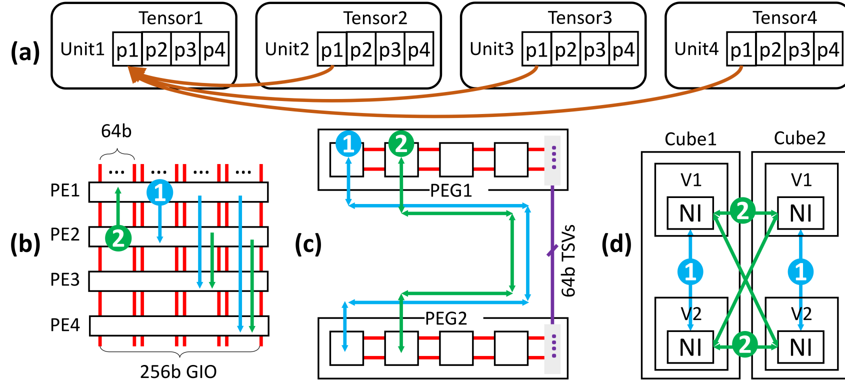


Figure 4.6: **Hierarchical interconnects to perform reduction operations: (a) reduction pattern, (b) inter-PE reduction, (c) inter-PEG reduction, (d) inter-vault and inter-cube reduction.**

the cases when the reduction is at PE-level, PEG-level, and vault/cube-level. All cases need four passes for the 4-tensor reduction. The blue and the green arrows show the examples of the first two passes. In Fig.4.6 (b), the PE-level reduction, each PE sends corresponding data to other PEs through the shared 256*b* GIO, one after another. In Fig.4.6 (c), the inter-PEG reduction, the shared TSV bus is used for data movement. In Fig.4.6 (d), since data paths involved in high-level nodes (e.g., inter-cube) always have lower bandwidth and higher data movement costs than those in the low-level nodes (e.g., inter-vault), low-level data communication is always granted with a higher priority.

4.2.3 Layout Transformation Support

Customized memory layout is important when attempting to increase PE's performance and utilization. Lacking complex cache hierarchy, DLUX's performance heavily relies on the memory coalescing and spatial locality in the bank row buffer. To guarantee the layout for each data structure during every phase of the data flow, data transformations are needed, which is efficiently supported by permutation unit (Sec.4.2.1) in each PE. First, the input data is read to scratchpad memory, and streamed into the permutation unit using a row major order. Second, the column reorder operation is performed

by selecting a certain cyclic first-in-first-out (FIFO) queue to write each input row. For matrix transpose, the cyclic FIFO is selected from left to right. Third, the row reorder operation is performed by cyclically moving the FIFO, so that elements in the same row can be shuffled. Lastly, all FIFOs will output its elements in sequence, and each time a permuted row is acquired and written to scratchpad memory.

4.3 Software Support

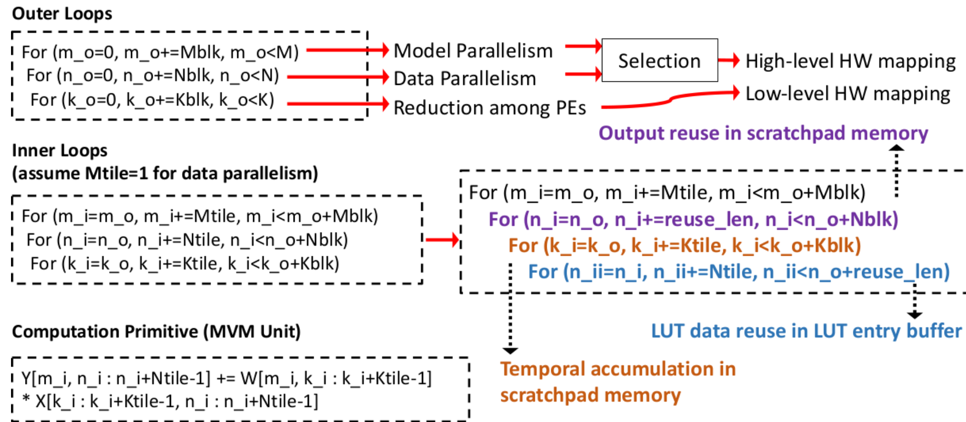


Figure 4.7: **GEMM mapping scheme loop formulation.**

4.3.1 Intra-layer Partitioning and Scheduling

We first focus on the most important general matrix multiplication (GEMM) kernel, which is followed by the description of supporting other kernels, with a detailed example of the batch normalization (BN) kernel.

General Matrix Multiplication (GEMM)

The GEMM kernel is the most important kernel, because it is used to compute many major DNN layers, such as the fully-connected layer, the recurrent layer, and the convolu-

tional layer [89], and hence is the time-dominating kernel for some DNN tasks ($40\%-90\%$ as shown in Fig.4.2). Comprehensive optimizations on mapping and scheduling are introduced as follows.

Problem Formulation with Nested Loop GEMM can be formulated as a three-level nested loop. We denote the input matrix as X of size $K \cdot N$, the weight matrix as W of size $M \cdot K$, and the output matrix as Y of size $M \cdot N$. We apply two-level loop tiling to the original loop nest, as shown on the left side of Fig.4.7 (a). For the first tiling, we denote it as the outer loops, which partition the M, N, and K dimensions into $Mblk$, $Nblk$, and $Kblk$, respectively. We also denote tiles from these outer loops as *blocks*, to distinguish from these in the second tiling. For the second tiling, we denote it as the inner loops, and it further partitions the $Mblk$, $Nblk$, and $Kblk$ dimensions into $Mtile$, $Ntile$, and $Ktile$, respectively. The loop body is the computation primitive calculating the matrix multiplication of $W_{Mtile \cdot Ktile} \times X_{Ktile \cdot Ntile}$.

Outer Loop Optimization: Spatial Partition to Increase Concurrency We use the outer loop tiling for spatial partition, i.e., we partition the input (X) or weight (W) matrix into blocks and distribute them to different PEs.

First, we explain how to partition, i.e., to determine the value of $Mblk$, $Nblk$, and $Kblk$. It is a 3-step decision. Step-1: we decide to partition either M or N, i.e., setting either $Mblk = M$ or $Nblk = N$. We do not partition both because we want to minimize the data movement. Allowing only one partition means that only one input matrix (either X or W) is required to be broadcast across all spatial partitions. The other input matrix with the spatial parallelism can be stationary in their local banks during the entire process of training, so that the data movement is minimized. We choose the larger one from M and N to partition in order to maximize spatial concurrency. Step-2:

we determine the value of $Nblk$ (or $Mblk$ if selecting M to partition in Step-1). This partition is not straightforward. On the one hand, we want $Nblk$ (or $Mblk$) as small as possible, in order to maximize spatial parallelism. On the other hand, we want the $Nblk$ to be large enough to fully utilized the hardware, e.g., scratchpad memory, in a PE , so that each block runs faster. The overall performance is then an overall consideration of both the parallelism and the single block performance. We solve the problem by building an analytical model with the objective of maximizing the overall performance. Step-3: we determine the K partition, i.e., the value of $Kblk$. We find the maximal $Kblk$ that ensures every PE has at least one block to work on.

Second, we explain the mapping. The partition of M or N is mapped to the cube and then to the vault, which is referred to as high-level hardware mapping. The partition of K is mapped to PEG and PE, as the low-level hardware mapping. Each block from the tiling of the outer loop is mapped to one PE.

Inner Loop Optimization: Temporal Scheduling to Increase Reuse The inner loop tiling further partitions the input matrix block into tiles to ensure all the tiles run on the same PE but in different time frame.

First, we explain the partition, i.e., the selection of $Mtile$, $Ntile$, and $Ktile$. All these parameters are fixed according to the configuration. We assign the input tile size to be the same as the input tensor size of an MVM unit. Therefore, $Mtile$ is set to 1 since MVM takes 1-D vector as the input. $Ntile$ and $Ktile$ are set as the weight and the height of the MVM unit.

Second, we describe the scheduling. In order to improve temporal data reuse, we need to fully exploit the scratchpad memory. For this reason, we apply another level of tiling inside the inner loop for the N dimension, as shown in the right side of Fig.4.7. We assign this tile size, $reuse_len$, according to the capacity of the scratchpad memory. The loop

order represents the scheduling, i.e., which tile runs first. We apply loop permutation so that we can first compute the tiles within the same *reuse_len*, which means all their results can fit in the scratchpad. The second loop we schedule is the K dimension with the purpose of applying output stationary, so that all the results can be accumulated by only accessing the scratchpad memory, eliminating the use of the slow DRAM. The third loop in the schedule is the N-loop and the last one is the M-loop. We set the M-loop last because we want the vector input of the MVM unit remain unchanged (see Sec.4.2.1 for detail reasons) so that LUT loading overhead is reduced and the LUT latency is hidden by buffer LUTs.

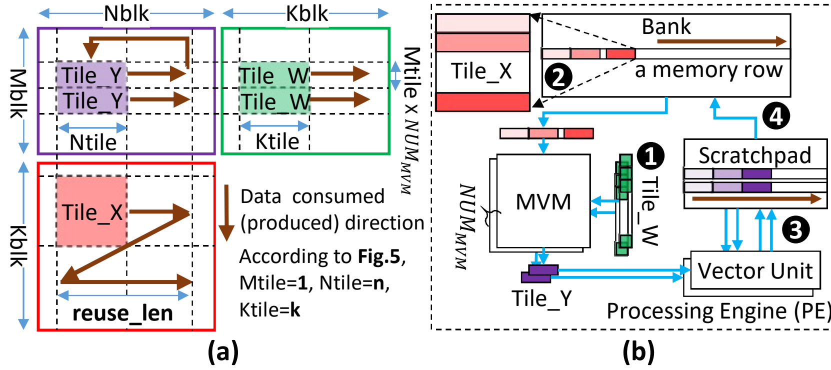


Figure 4.8: **Inner loop mapping. (a) Data partitioning within a PE. (b) Computation flow and temporal data reuse scheme based on (a).**

Finally, we describe an example shown in Fig.4.8 (a). The brown arrows indicate the running order of the tiles, i.e., the scheduling. Fig.4.8 (b) further explains the working flow. In Step-❶, we read a vector of *Tile_W* in from the GIO. The vector initializes the LUT buffer and will remain there until every related computation is done. In Step-❷, we read the matrix *Tile_X* from the local DRAM bank. Note that we optimize the data layout in order to maximize DRAM row buffer hit when access X by *Tile_x*. In Step-❸, we get the result vector *Tile_Y* from the MVM unit and apply accumulations (if necessary) with previous partial results stored in the scratchpad memory using the

vector unit. The accumulated result vector is then updated and stored to the scratchpad memory. In Step-❹, when K-loop ends and we get the final result, we write it back to the DRAM bank.

Other Kernels

DLUX can also efficiently supports other kernels, including activation, elementwise, and batch normalization. Activation kernels (e.g., $ReLu$) are fused with the GEMM kernel, supported by vector units. Elementwise kernels are also supported by vector units, but require reading data from the local bank (with potential support of permutation unit) into the scratchpad memory before computing. For batch normalization, the mean, variance, and normalization are calculated using the MVM unit and Vector unit, with the help of hierarchical interconnects to perform reduction and broadcasting, and SFU to perform square root and inversion. For Resnet50's FusedBatchnorm, DLUX shows 13.13× speedup an 97.14% energy savings over one V100 GPU.

4.3.2 Inter-layer Layout Transformation

Although intra-layer mapping can achieve maximal performance and minimal data movement, inter-layer operation efficiency can be challenging for DLUX due to specialized layout requirements of each DNN layer, and the distributed memory model intrinsic to PIM architecture. We solve that issue by maintaining the input-output layout consistency and the forward-backward layout consistency.

Input-Output Layout Consistency

The coalesced data layout is the key to improve the spatial and temporal reuse. To maintain the coalesced layout during the layer-by-layer computation, we need to keep

each layer's input and output data layout consistent, considering both the matrix layout transposing and the data partition.

The matrix layout transposing is heavily demanded in the training process, as shown in Fig.4.1. Before writing back the result of the previous layer, we apply the matrix transpose if necessary, so that the next layer can use the previous result as it is. The transposing is done on-the-fly and locally by the permutation unit in each PE. Here, the data transformation can be local since we only require the conversion between row-major data fetching to column-major data fetching without involving inter-PE traffic. Therefore, there is insignificant data movement energy overhead and the latency can be hidden by layer computations.

We also keep the input/output data layout consistent in terms of data partition across PEs. The tensor reduction in Fig.4.6 (a) is a good example. The four input tensors are partitioned across four units (PE), so we want to partition and store the output tensor also in four units, accordingly. Otherwise, if we apply a different data partition for the result tensor, e.g., storing the whole tensor into one unit without partition, the next layer which takes the result tensor as the input can only use one unit for computing, leaving the other three underutilized. To keep the data partition consistent, we control the data communication destination addresses like the cases in Fig.4.6 (b)-(d) when storing the data.

Forward-Backward Layout Transpose

Layout transpose of the same tensor is required for the backward pass. A naive transpose scheme will introduce all-to-all broadcasting traffic.

We propose a partial transpose layout which involves no inter-PE data movement. First, instead of moving data blocks from forward mapping PE to backward mapping PE, we only change the index of the block stored in local PE. For example, given a PE

associated with block index (i, j) in the forward pass, the block index will be transposed to (j, i). Then, the PE-level data transpose is performed by the permutation unit introduced in Sec.4.2.3, where continuous row coalescing becomes continuous column coalescing. Here, coalescing means data is stored in adjacent memory locations in DRAM so access locality is maximal. This transpose operation can be overlapped with other operations. Also, as the same input data tile is read multiple times from the local bank during one inner loop calculation, it is only transposed once for the last round, which incurs very low time overhead.

4.4 Experiment

4.4.1 Experimental Setup

Workload For end-to-end workloads analysis, we use 6 representative benchmarks as shown in Table.4.1. These benchmarks are selected from MLperf [90], Fathom [91] and NNBench-X [92], and are trained using their default configurations.

Hardware Configuration DLUX adopts 3D-stacking memory configuration similar to previous HMC-based accelerator [69], but additionally, DLUX has the near-bank computing design. The detailed configuration, hardware latency, and energy settings are shown in Table.4.2. In addition, the CAS_width per bank for internal data reading is set as 2048 bits, the total number of TSVs per cube is 1024, and the inter-cube communication uses 4 full-duplex Serializer/Deserializer (SERDES) links with maximum $30Gb/s$ per link bandwidth.

Evaluation Methods We develop an in-house simulator adapted from ramulator [44] based on the key architecture and timing parameters in Table.4.2. DLUX is designed to

run at a clock frequency of $1GHz$ under $22nm$ technology node. We use cacti-3DD [93] to evaluate the inter-PE interconnects, TSVs, and the 3D DRAM bank access latency and energy. The energy, performance, and area of the scratchpad memory and the SRAM LUT is also simulated by cacti-3DD. The base logic die and the SERDES energy is set based on previous near data processing work [94]. For an FP unit evaluation in the MVM unit and the vector unit, we use an open-source tool [86] to generate VHDL code for various $FP32$ arithmetic and logic units. We also implement RTL codes for the DLUX controller, address translation unit, and permutation unit. These hardware components are synthesized by design compiler considering DRAM process overhead [40] to derive performance, power, and area results. For the GPU evaluation, the baseline DNN training performance is derived from Tensorflow profiling tool and $nvprof$, and the power information is sampled using *nvidia-smi*. When measuring a kernel, we disable all other tasks on the target GPU to isolate the power consumption number. Also, we acquire the steady state of the power consumption for the kernel by running it multiple times and sample the plateau area to ensure accuracy. Because ScaleDeep [95] simulator is not open-source, we optimistically estimate its performance using an analytical model based on the roofline model [96], by using its peak computing throughput (*comp*), peak memory bandwidth (*bw*), and average hardware utilization (*util*) provided in their paper. For each operator in a given benchmark, we use its arithmetic density to determine whether it is compute-bound or memory-bound in the roofline model. The operator information contained in a computation graph is generated by Tensorflow profiling. For a compute-bound operator, we divide its total operations by the sustained computing throughput ($comp \times util$) to get its execution time. For a memory-bound operator, we divide its total memory footprint by the sustained memory bandwidth ($bw \times util$) to get its execution time. The end-to-end time of a benchmark on ScaleDeep is the accumulation of all its operator time.

Parameter Names	Configuration
Cubes / Vaults_cube / PEGs_vault	1 / 16 / 8
PEs_PEG / MVMs / Ktile / Ntile	4 / 2 / 4 / 4
Bank / RowBuffer / Scratchpad (Kb)	131072 / 16 / 32
tCK/tRCD/tCCD/tRTP/tRP/tRAS (ns)	1 / 14 / 4 / 4 / 15 / 33
RD,WR/PRE/ACT/Spad (nJ/access)	0.224 / 0.507 / 0.521 / 0.005
MVM / VU / PU (pJ/access)	139.9 / 3.6 / 64.3
interPE/TSV/SERDES (pJ/bit)	0.017 / 4.64 / 4.50

Table 4.2: **DLUX hardware configuration parameters.**

4.4.2 Performance, Bandwidth, and Area

Performance Analysis For fair comparison assuming a single compute-node, in the end-to-end analysis, we choose one DLUX cube with 8 PIM layers. **DLUX-1** represents naively implementing FP32 logic without LUT optimization, so the number of FP units is halved under the same area constraint. **DLUX-2** represents removing cache without applying the layout transformation scheme, so extra latency is added due to increased row activation number for sub-optimal access locality. **DLUX-3** incorporates both LUT optimization and layout transformation scheme. One Nvidia V100 [97] GPU is used as the performance baseline, and one ScaleDeep [95] FcLayer chip is used as the state-of-the-art DNN training accelerator baseline. During the simulation process, we acquire the entire computation graph of the workload, including kernel types, input/output shapes, and total execution time on a GPU. Then, we evaluate the currently supported kernels on ScaleDeep and DLUX (more than 98% of total execution time for most benchmarks) and assume the non-supported kernels have the same execution time as GPU.

Fig.4.9 shows the end-to-end execution speedup for 6 representative data center DNN training workloads. We observe that on average, one optimized DLUX-3 cube achieves 6.3× speedup compared to a single V100 GPU, and 2.4× speedup compared to a single ScaleDeep chip. We further investigate the time breakdown, and find DLUX accelerates $GEMM$, *Elementwise*, *Reduction*, and $DataManipulation$ kernels w.r.t GPU at 5.8×,

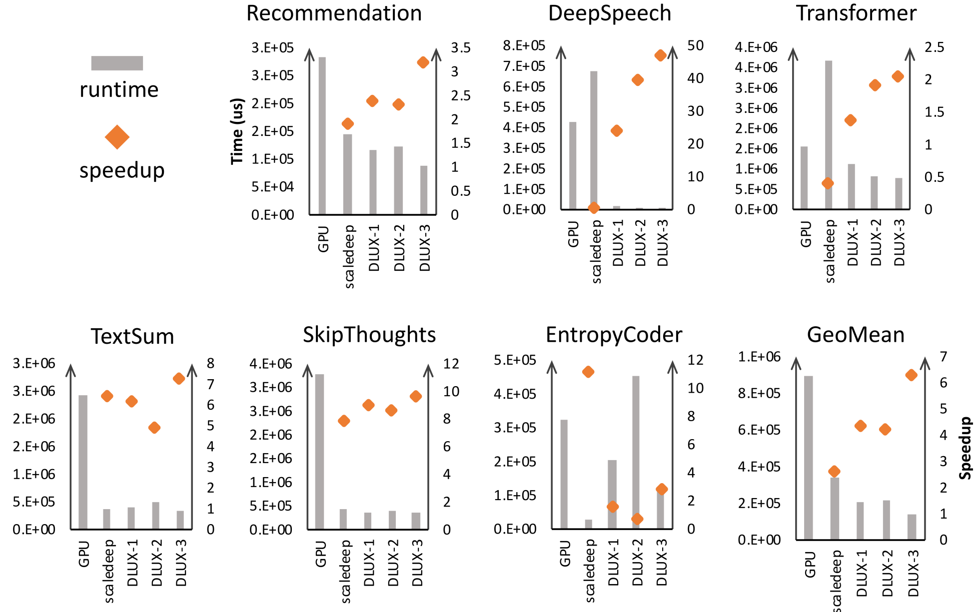


Figure 4.9: **End-to-end execution time and speedup comparison for benchmarks in Table.4.1.**

26.5×, 3.3×, and 14.8× respectively. DLUX can provide significant performance gain for typical memory-bound kernels ($Elementwise$ and $DataMani-pulation$) due to the abundant memory bandwidth provided by near-bank architecture. DLUX can also provide decent speedup for compute-intensive ($GEMM$) and communication intensive kernels ($Reduction$). For $GEMM$, the speedup is attributed to both the high peak FP performance and the high utilization enabled by the proposed software mapping and scheduling techniques. For $Reduction$, the performance roots from the utilization of hierarchical data buses in the 3D memory for efficient data communication.

However, naively implementing near-bank architecture may result in sub-optimal results. From Fig.4.9 we can observe that DLUX-3 can achieve 1.43× and 1.50× speedup compared with DLUX-1 and DLUX-2, respectively.

We also observe the variation of speedup numbers for different workloads when compared to ScaleDeep, since each workload is a mix of different types of kernels, and each type of kernel takes up varying portions of total execution time and shows various

arithmetic density values. For DeepSpeech and Transformer, DLUX obtains significant speedup on $GEMM$ (98% and 65% of total time respectively) with good data reuse and parallelism. For EntropyCoder, DLUX suffers from $GEMM$ (60% of total execution time) with inferior LUT data reuse. For other workloads, since there is not a single time-dominating operation (less than 50% of total execution time), DLUX provides slightly better speedup w.r.t. ScaleDeep due to a mix of operations (1.3× on $GEMM$, 7× on $Elementwise$, 1.1× on $Reduction$, 0.7× on $DataManipulation$).

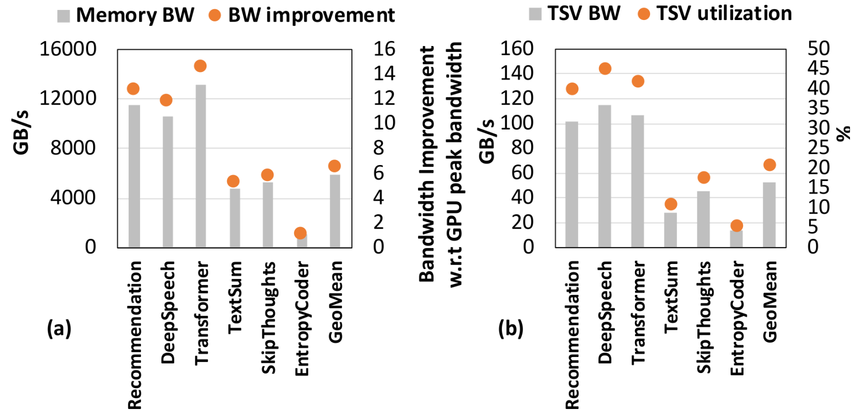


Figure 4.10: **(a) Memory bandwidth and improvement w.r.t. GPU. (b) TSV bandwidth and utilization.**

Bandwidth Analysis We profile the total memory bandwidth and TSV traffic of DLUX and show the results in Fig.4.10. Fig.4.10 (a) shows that on average, one DLUX cube can achieve $6TB/s$ sustained memory bandwidth, which is $\sim 6.6\times$ of one V100 GPU bandwidth. The significantly high bank-level bandwidth justifies the adoption of near-bank architecture, which benefits performance for memory bound kernels and reduces data movement for memory intensive kernels. This also proves the necessity of integrating computation logic in the memory die, since by putting the computation units on base logic die, only $256GB/s$ peak memory bandwidth can be achieved, assuming 1 cube. Fig.4.10 (b) shows that on average, one DLUX cube can achieve $53GB/s$ sustained TSV

bandwidth, which equals to 21% utilization assuming $256GB/s$ peak TSV bandwidth. The high TSV bandwidth for Recommendation, DeepSpeech, and Transformer proves the importance of 3D stacking architecture, since different PEs inside the same vault need to use TSVs.

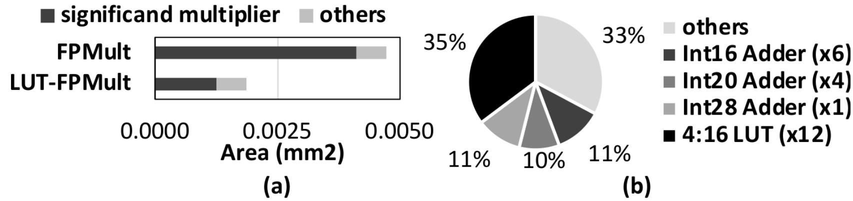


Figure 4.11: **LUT-FPMult area analysis**

Area Analysis To achieve high performance for compute intensive kernels ($GEMM$), we need a high performance LUT-FPMult. Currently, we equip 32 FP multipliers (FPMult) per PE, and the proposed LUT-FPMult significantly reduces this area overhead by 60.6% as shown in Fig.4.11. (a) shows that, by replacing area-consuming significand MUL by the proposed LUT-based design, the total overhead is greatly reduced. (b) shows the area breakdown of the LUT-FPMult design, where LUTs takes 35.2% of total area. The FPMult uses an autogenerated VHDL-code multiplier from Flopoco [86]. The LUT-FPMult uses the same VHDL-code, and replaces expensive mantisa-multiplier ($\sim$ 87% of FPMult) with lightweight SRAM-based LUT array ($\sim$ 26.5% of FPMult). We use CACTI-3DD [93] to estimate SRAM LUT's area, performance, and power.

Using the hardware configuration from Table.4.2, the area breakdown of a PE is demonstrated in Table.4.3. The total area overhead of a PE (34.02%) is normalized to a DRAM bank ($1.22mm^2$) under $20nm$ technology node [12]. Here, we assume DRAM process with 3 metal layers will double the area overhead of bank peripheral logic compared to a normal CMOS process with 8 metal layers [40]. LUT can help reduce 14.27% total overhead of peripheral, since LUT can help reduce the normal FPMult overhead of

Component	Area (mm^2)	Overhead (%)
MVM Unit (x2)	0.2440	20.02
Vector Unit (x2)	0.1172	9.62
Permutation Unit (x1)	0.0040	0.33
SFU (x1)	0.0140	1.15
Scratchpad (x1)	0.0260	2.12
Full LUT (x1)	0.0095	0.78
Total	0.4146	34.02

Table 4.3: **The area of components in a PE.**

MVM Units (from 35.06% to 20.02%), but only introduce a small extra full LUT overhead (0.78%). Using this area overhead number and assuming one 3D cube has the area footprint of $96mm^2$ [12], we estimate the total silicon footprint of one DLUX cube (8 layers) to be $\sim 998mm^2$. In contrast, one V100 GPU has one processor die with $815mm^2$ and four HBM stacks (4 layers), and the total silicon footprint is $2351mm^2$. Comparison with a V100 GPU, one DLUX cube has 65% less silicon footprint.

4.4.3 Energy Analysis

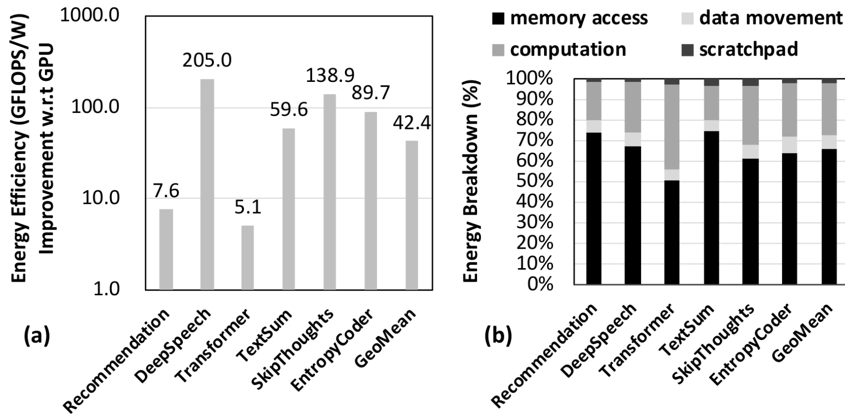


Figure 4.12: **(a) Energy-efficiency improvement. (b) Energy breakdown.**

From Fig.4.12 (a), we observe that one DLUX cube on average improves energy-efficiency (GFLOPS/W) by 42.2× compared with one V100 GPU. To further understand

the improvement, we profile the detailed energy consumption numbers and plotted them in Fig.4.12 (b). The memory access energy includes bank activation/precharge and access energy. The data movement energy includes inter-PE, inter-PEG, inter-vault, and inter-cube energy. The computation energy includes the energy consumption of MVM units, vector units, SFU, and permutation unit. The scratchpad energy covers all access energy to scratchpad memory. The energy of the controller and the address translation unit is very small ($< 1\%$), so it is ignored in the analysis. On average, memory access, data movement, computation, and scratchpad access consumes 66.1%, 6.3%, 25.2%, and 2.4% of total energy, respectively. Since the benchmarks we evaluate are memory-intensive, and DLUX has optimization for data movement reduction, it is expected that memory access and computation dominate the energy consumption (over 91.3%).

DLUX can provide significant energy savings for all benchmarks, since DLUX can greatly reduce data movement, which is the major energy overhead for compute-centric architectures [13]. This is shown by the small percentage of energy spent on data movement (6.3%) for DLUX. This small energy consumption number is first attributed to the reduction of data movement of near-bank architecture, since the majority of the data is streamed from the local bank. Second, DLUX spatial partitioning and mapping method guarantees minimum data sharing between different PEs, and DLUX scheduling techniques ensure that local data reuse is very high for a PE, so that a remote memory transfer from other PEs is infrequent. The consistent input-output layout assumptions and PIM friendly forward-backward transpose format further reduces inter-layer data movement.

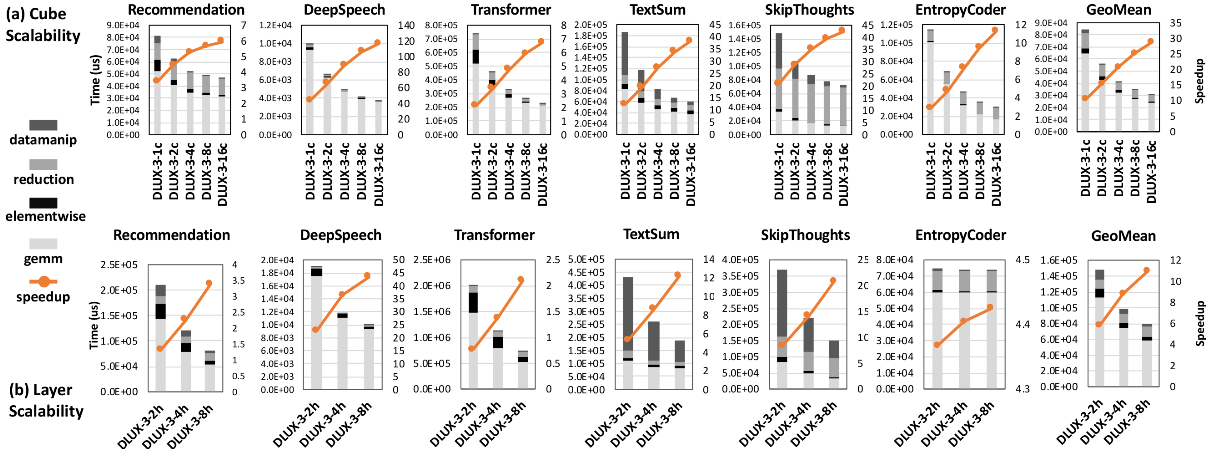


Figure 4.13: **(a) Scalability analysis w.r.t the number of cubes. (b) Scalability analysis w.r.t the number of layers.**

4.4.4 Scalability Study

Scaling Cube and Layer Number We study the scalability of the proposed software partitioning and mapping techniques by conducting a weak scaling analysis, where each workload remains unchanged but the number of hardware components are scaled. As shown in Fig.4.13, we first keep the total number of layers per cube unchanged, while scaling the total cube number from 1 (DLUX-3-1c) to 16 (DLUX-3-16c) in (a), and then we keep the total number of cubes unchanged, while scaling the number of layers per cube from 2 (DLUX-3-2h) to 8 (DLUX-3-8h) in (b). For each evaluated configuration, we also plot the breakdown of the execution time into four categories of computation kernels as well as the speedup ratio w.r.t. one V100 GPU.

For the cube scaling results, on average, by doubling the total cube number, a 1.30× scaling ratio can be achieved. Further analysis shows that *GEMM*, *Elementwise*, *Reduction*, and *Datamanipulation* kernels can achieve 1.28×, 1.50×, 1.23×, and 1.84× scaling ratio, respectively. For all the benchmarks, we observe that, when we scale the cube number, the performance bottleneck will gradually become either *GEMM* or *Reduction*, which all have low scaling ratio. The reason is that *Elementwise* and

Datamanipulation kernels have abundant parallelism and simple data access and communication patterns, so it is relatively simple to scale them by adding more cubes. For *Reduction* kernels, depending on whether the reduction dimension is distributed among different cubes, different scaling ratios can be achieved due to the data dependency in reduction process. For $GEMM$ kernels, the non-ideal (ideal scaling ratio should be 2) scaling ratio is attributed to the tradeoff between the spatial utilization and the temporal utilization. The current partitioning scheme increases spatial utilization by distributing input data evenly across all vaults, so that an increasing vault number will decrease the local data reuse per vaults, harming temporal utilization. The N dimension is partitioned among all vaults, where the bank in each vault gets an $Nblk$ partition. Since the LUT data reuse depends on the tiling of the $Nblk$ dimension, for $GEMM$ kernels with small N size, the larger the total cube number is, the smaller the effective data reuse will be. This explains that for $GEMM$ kernels with larger N, near optimal scaling ratio is achieved, and other kernels with smaller N achieves sub-optimal scaling ratio.

For the layer scaling results, by doubling the number of layers per cube, on average, a 1.23× scaling ratio can be achieved for all the evaluated benchmarks. Further analysis shows that $GEMM$, *Elementwise*, *Reduction*, and *Datamanipulation* kernels can achieve 1.24×, 1.36×, 0.98×, and 1.54× scaling ratio, respectively. For $GEMM$ and *Reduction* kernels, the scaling is non-ideal since data reduction overhead exists. Taking $GEMM$ kernels for example, the K dimension is partitioned among all PEs in the same vaults, where the bank in each vault gets a $Kblk$ partition. By adding more layers, reduction operations will increase. However, in the shared hierarchical bus, TSVs among layers are shared resources. Adding more layers will reduce the workloads per PE, as well as add total reduction latency in the final step. Although spatial parallelism is achieved by adding more layers, the inter-PEG reduction time will take over the PE local computation time if K is small and layer number increases. This explains why adding more

layers will not help reduce the $GEMM$ execution time for EntropyCoder.

4.5 Summary

In this chapter, we present both the hardware architecture and the software mapping and scheduling techniques for DLUX, a 3D-stacking LUT-based near-bank accelerator for DNN training. We address the area-constrained performance challenge by leveraging DRAM LUT for computation. We design a hierarchical lookup-table for high performance and low overhead FP computing. Then, to enable efficient communications, the hierarchical data buses are utilized to perform high bandwidth data broadcasting operations. Beside the hardware, we also propose mapping and scheduling techniques to further improve the spatial and temporal utilization of DLUX. In addition, transparent and low overhead techniques are invented to ensure the input-output layout consistency and forward-backward layout transposition. We finally evaluate DLUX on representative deep learning training tasks and compare the results with Tesla V100 GPU. Area analysis shows that DLUX can reduce overhead by 60% against direct implementation of FP32 unit. Compared with Tesla V100 GPU, end-to-end evaluation shows that DLUX provides on average 6.3× speedup and 42× energy efficiency improvement.

Chapter 5

iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture

This chapter focuses on methods to speedup image processing workloads which require high memory bandwidth by using the 3D-stacking processing-in-memory (3D-PIM) near-bank solution. The near-bank design [39, 40, 41] closely integrates compute-logic to each bank in the DRAM dies. It can provide around 10× peak bandwidth improvement compared with the previous process-on-base-die solution [36, 37, 38], since compute-logic directly accesses the local bank without going through limited through-silicon-vias (TSVs). Although near-bank architecture has great potential for accelerating image processing applications, there are still several challenges. First, heterogeneous image processing pipelines exhibit various computation and memory patterns, thus requiring programmable hardware support. However, directly attaching control cores to each DRAM bank introduces large area overhead [33, 98, 99], so it is challenging to design a lightweight architecture supporting diverse image processing pipelines. Second, the design of instruc-

tion set architecture (ISA) needs to be concise yet powerful because it needs to avoid complex hardware support while enabling flexible computation, data movement, and control flow operations at the same time. Third, end-to-end compilation support for this accelerator requires easy programming interfaces to enable the efficient mapping of various image processing pipelines to the near-bank architecture, as well as backend optimizations to fully exploit the hardware potentials.

In this chapter, we present a programmable image processing accelerator (iPIM) and an end-to-end compilation flow based on Halide [100] to efficiently map applications onto our accelerator. First, iPIM uses a decoupled control-execution architecture to integrate a control core under the tight area constraint. Specifically, the control core is placed on the base logic die of the 3D-stack, while lightweight computation units and several small buffers are attached to each memory bank in DRAM dies. During the execution of instructions, the control core broadcasts instructions to all associated banks using TSVs, and all computation units conduct parallel execution in lockstep. Second, we design Single-Instruction-Multiple-Bank (SIMB) ISA for the proposed near-bank accelerator. The SIMB ISA supports SIMD computation which utilizes the bank's high I/O width ($128b$), flexible data movement within the near-bank memory hierarchy, control flow instructions that enable index calculation, and synchronization primitives for communication. Third, we develop an end-to-end compilation flow with new Halide schedules for iPIM. This compilation flow extends the frontend of Halide for supporting these new schedules and includes a backend with optimizations for iPIM including register allocation, instruction reordering, and memory order enforcement to reduce resource conflict, exploit instruction-level parallelism, and optimize DRAM row-buffer locality, respectively.

The contributions of this chapter are summarized as follows:

- We design a standalone programmable accelerator, iPIM, using 3D-stacking near-bank architecture for image processing applications. By using a decoupled control-execution architecture, iPIM supports programmability with small area overhead per DRAM die ($\sim$ 10.71%).

- We propose SIMB (Single-Instruction-Multiple-Bank) ISA which enables flexible computation, data access, and communication patterns to support various pipeline stages in image processing applications.

- We develop an end-to-end compilation flow based on Halide with novel iPIM schedules and various iPIM backend optimizations including register allocation, instruction reordering, and memory-order enforcement.

- Evaluation results of representative image processing benchmarks, including single stage and heterogeneous multi-stage pipelines, show that iPIM design together with backend optimizations can achieve 11.02× speedup and 79.49% energy saving on average over an NVIDIA Tesla V100 GPU. The backend optimizations improve 3.19× performance compared with the naïve baseline.

5.1 Motivation

First, we find that memory-bandwidth is the performance bottleneck for GPU, which is the current state-of-the-art image processing accelerator [101]. We conduct a detailed profiling of representative benchmarks (Table.5.2) using Halide framework [100] and DIV8K [102] dataset on an NVIDIA Tesla V100 GPU [103]. The measured total DRAM bandwidth, DRAM utilization, and ALU (both FP32 and INT32) utilization are shown in Fig.5.1(a). We observe that these benchmarks exhibit DRAM bandwidth-bound behavior by achieving 57.55% DRAM utilization (518GB/s bandwidth) and 3.43% ALU

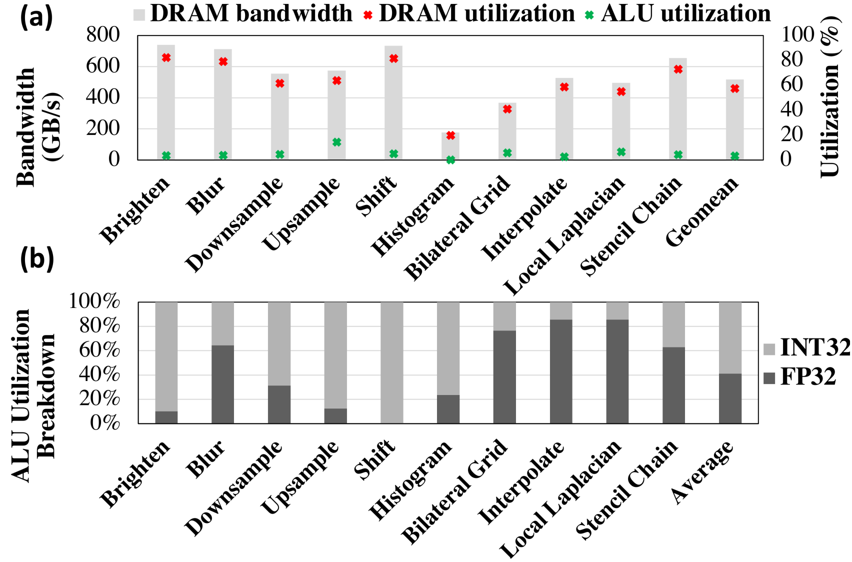


Figure 5.1: GPU profiling results for image processing workloads (Table.5.2).

utilization on average. We also note that the memory and ALU utilization are both low for Histogram benchmark, which results from that Histogram involves value-dependent computations and the Halide schedule for GPU cannot achieve ideal performance.

Second, we observe that multi-stage benchmarks (the last 4 in Fig.5.1), which are optimized by Halide pipeline fusion, show little performance improvement compared with single-stage benchmarks (the first 6 in Fig.5.1). The ALU utilization only increases from 2.85% to 4.53%. Also, the DRAM utilization is merely reduced from 58.80% to 55.73%, which is still significantly higher than the ALU utilization. We conclude that Halide compiler optimizations cannot change the memory-bound behavior of image processing applications on GPU, motivating an accelerator providing more memory bandwidth.

Third, we find that index calculation, which is an important part of programmability support for flexible memory access patterns, consumes a large portion of total ALU utilization for image processing workloads. For the current profiling, index calculation uses INT32 data type and algorithm-related computation uses FP32 data type. The breakdown of the ALU utilization is shown in Fig.5.1(b). We observe that on average

index calculation takes 58.71% of total ALU utilization, and index calculation dominates the total ALU utilization (> 60%) for 5 out of 10 benchmarks. The index calculation ratio is high because image processing requires frequent translations from 2D image to 1D memory space [104]. This motivates us to enable architecture support for index calculation in iPIM.

5.2 Architecture Design

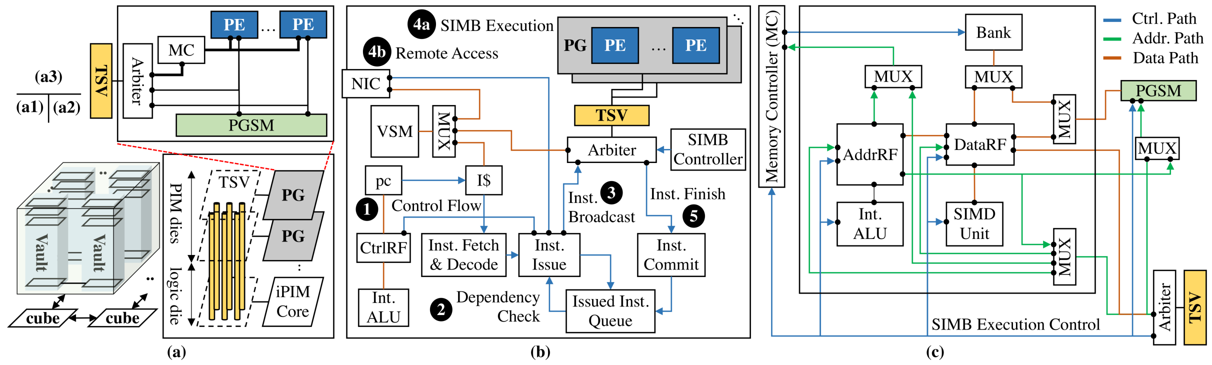


Figure 5.2: iPIM control-execution decoupled 3D-stacking microarchitecture: (a1) 3D-stacking cubes. (a2) A vault. (a3) A Process Group (PG). (b) Components inside an iPIM control core on the base logic die. (c) Components inside a Process Engine (PE) on the PIM dies.

First, we introduce the microarchitecture overview in Sec.5.2.1. Second, we describe iPIM's decoupled control-execution scheme in Sec. 5.2.2. Then, we explain the instruction set architecture design in Sec. 5.2.3. Next, we discuss the remote memory access mechanism and present the method for inter-vault synchronization in Sec.5.2.4. In the end, we detail the functionalities of iPIM's hardware components in Sec.5.2.5.

5.2.1 Microarchitecture Overview

In general, iPIM uses the 3D-stacking near-bank architecture with a top-down hierarchy of *cube*, *vault*, *process group*, and *process engine* as illustrated in Fig.5.2(a). First,

Category	Instruction	Description	Operands
computation	**comp**	SIMD computation (mode:vector-vector,scalar-vector) +FP/INT arithmetic (add,subtract,multiply,mac) +logical arithmetic (shift,and,or,xor,crop-lsb,crop-msb)	comp,op,mode,dst_drf, src1_drf,src2_drf,vec_mask,simb_mask
index calculation	**calc_arf**	memory address calculation (INT only)	calc_arf,op,dst_arf,src1_arf,src2_arf,simb_mask
intra-vault data movement	**st/ld_rf**	store(/load) data to(/from) the bank from(/to) the DataRF	st/ld_rf,dram_addr,drf_addr,simb_mask
	st/ld_pgsm	store(/load) data to(/from) the bank from(/to) the PGSM	st/ld_pgsm,dram_addr,pgsm_addr,simb_mask
	rd/wr_pgsm	read(/write) data from(/to) the PGSM to(/from) the DataRF	rd/wr_pgsm,pgsm_addr,drf_addr,simb_mask
	rd/wr_vsm	read(/write) data from(/to) the VSM to(/from) the DataRF	rd/wr_vsm,vsm_addr,drf_addr,simb_mask
	mov_drf/arf	move data from(/to) DataRF to(/from) AddrRF	mov_drf/arf,arf_addr,drf_addr,simb_mask
	seti_vsm	set immediate value to a VSM location	seti_vsm,vsm_addr,imm
	reset	reset a DataRF entry to zero	reset,drf_addr,simb_mask
inter-vault data movement	**req**	request data from a remote vault to the local vault	req,dst_chip_id,dst_vault_id,dst_pg_id,dst_pe_id, dst_dram_addr,src_vsm_addr
control flow	**jump/cjump**	jump/conditional jump	jump/cjump,(cond),crf_addr
	calc_crf	control flow data calculation (INT only)	calc_crf,op,dst_crf,src1_crf,src2_crf
	seti_crf	set immediate value to a CtrlRF location	seti_crf,crf_addr,imm
synchronization	**sync**	inter-vault synchronization	sync,phase_id

Table 5.1: iPIM's Single-Instruction-Multiple-Bank (SIMB) Instruction Set Architecture

iPIM consists of multiple cubes (Fig.5.2(a1)) interconnected by SERDES links similar to HMC [82]. Second, one cube is horizontally partitioned into multiple vaults (usually 16 per cube) connected by an on-chip network. Each vault (Fig.5.2(a2)) spans multiple 3D-stacking layers, including several process-in-memory (PIM) dies (usually 4 to 8 per vault) and one base logic die. The inter-layer communication is realized by Through-Silicon-Vias (TSVs, usually 64 per vault), which are high-bandwidth vertical interconnects that link each layer to the base logic die. The base logic die of each vault contains one iPIM control core (Fig.5.2(b)), which is the basic unit to execute an iPIM program. Next, one PIM die of each vault contains one process group (PG) (Fig.5.2(a3)), which further consists of many process engines and a shared process group scratchpad memory (PGSM). Last but not least, each process engine (PE) (Fig.5.2(c)) employs near-bank architecture, where compute-logic and lightweight buffers are integrated with a DRAM bank. Especially, each PE adds an address register file and an integer ALU to efficiently support index calculations which are important for image processing (Fig.5.1(b)).

Based on this microarchitecture, iPIM decouples the control, which happens on the *base logic die*, from the massive bank-level parallel execution, which happens on the *PIM dies* (Sec.5.2.2). In addition, we design SIMB ISA to support various computation and

memory access patterns in image processing, and efficiently move data among the iPIM hierarchy (PE-level, PG-level, vault-level, or cube-level) (Sec.5.2.3).

5.2.2 Decoupled Control-Execution Architecture

iPIM uses a novel decoupled control-execution design to reduce the overhead of the control core by placing it on the *base logic die*, and allows the parallel execution of processing engines on the *PIM dies* to benefit from the abundant bank-level bandwidth. For the control core, the design principle is to keep the hardware simple and rely on compiler optimizations (Sec.5.3) to realize high performance. Therefore, iPIM uses a pipelined, single-issue, and in-order core, where the data hazard is eliminated when an instruction is issued, so the hardware needs no complex forwarding logic. For the execution part, the SIMB ISA (Sec.5.2.3) can exploit massive bank-level parallelism by programming the bits of $simb_mask$.

Next, we introduce the detailed pipeline execution of iPIM in Fig.5.2(b) (with related instructions in Table.5.1) as follows. ❶ Depending on the program counter (pc), an instruction will be fetched from the instruction cache ($I\$$) and decoded. pc can be updated from control register file (CtrlRF) using $jump/cjump$, and $calc_crf, seti_crf$ are used to calculate control flow values. ❷ The decoded instruction will be checked against instructions in the Issued Inst Queue. If true/anti/output data dependency is found, the instruction will stall with a pipeline bubble inserted. Once the instruction is issued, it is added to the Issued Inst Queue until retirement. ❸ The issued instruction is broadcast by SIMB controller to each PE according to the $simb_mask$, or sent to a vault-level unit for execution (e.g. $seti_vsm$). If the instruction involves remote vault access, it is dispatched to the network interface controller (NIC). ❹ (a) For the vault-local SIMB execution, each PE will check the corresponding bit in $simb_mask$ and proceed execution

or stay idle. (b) For the remote vault access, the request will be translated into packets and traverse the on-chip network or off-chip links. ❺ The SIMB instruction executes in lock-step, and an instruction retires only if all bits in the *simb_mask* are cleared. Each time a PE finishes an instruction, the SIMB controller will clear its execution bit. After an instruction finishes, it is committed by popping the corresponding entry from the Issued Inst Queue. This also clears data dependency for later instructions.

As a conclusion, this architecture not only enables lightweight programmability to control heterogeneous pipeline stages (*base logic die*) but also supports parallel execution to provide abundant memory bandwidth for data-intensive image processing operations (*PIM dies*).

5.2.3 Single-Instruction-Multiple-Bank (SIMB) ISA

To exploit the data-parallelism in image processing, we propose a Single-Instruction-Multiple-Bank (SIMB) ISA to expose bank-level parallelism as detailed in Table.5.1. From a high-level overview, this ISA resembles a RISC-like SIMD ISA that enables bank-parallel computation as well as efficient memory access as detailed below.

For the computation, we highlight the support for SIMB and SIMD execution. To enable SIMB, each SIMB-capable instruction has a *simb_mask* field, which is a boolean vector indicating whether the corresponding PE should execute this instruction or not. For example, in a vault with 8 PGs where each PG has 4 PEs, the *simb_mask* should be a $32b$ boolean vector. To enable SIMD, each computation and data movement instruction operates on a vector of FP32/INT32 elements. The vector length is chosen to be 4 to match the local bank's interface ($128b$ per access) and TSV's data transfer width ($128b$ per cycle), so the internal bandwidth is fully utilized. For each vault, control signals and data signals share the same physical TSVs through time multiplexing, which is realized

by the arbiter in Fig.5.2. Therefore, there is no additional TSV area cost for control signals to each PE.

For memory access, we emphasize the support for data movement and memory indexing. To enable data movement, SIMB ISA contains different instructions to realize customized data flow along the memory hierarchy. To support flexible indexing, SIMB ISA contains index calculation instructions and allows communication between the address register file and the data register file to enable data-dependent computation.

More detailed explanations about SIMB ISA are as follows:

The computation instruction (*comp*) supports vector-vector(/-scalar) operations specified by the *mode* field. The *vec_mask* indicates which positions in the vector are valid for computation. The *op* defines the operation to be performed.

The index calculation instruction ($calc_arf$) supports parallel address calculations among PEs, so each PE can have independent memory access patterns. To allow different PEs inside a vault to operate on different addresses in the SIMB fashion, indirect addressing is supported for the bank (*dram_addr*), PGSM (*pgsm_addr*), and VSM (*vsm_addr*) addresses. When indirect address mode is used, the corresponding address field will first index into the address register file in each PE, and then the fetched address will be used to index the target memory component. This can satisfy the need for flexible 2D memory access patterns in image processing.

The data movement instructions (intra-vault/inter-vault) are classified into two types. The first type involves DRAM bank access ($st/ld_rf, st/ld_pgsm$ for local vault access, and *req* for remote vault access). The second type includes data movement along the memory hierarchy within a vault.

The control flow instructions support control flow (*jump/cjump*) and related calculations ($calc_crf, seti_crf$). These enable iPIM programs to have dynamic behaviors to support various computation patterns in image processing.

The synchronization instruction (*sync*) allows different vaults to synchronize computation stages according to a *phase_id*. Sec.5.2.4 contains a detailed example.

5.2.4 Remote Access and Synchronization

iPIM supports data access from a remote vault by implementing an asynchronous request instruction (*req*). First, the local vault needs to provide the remote vault's memory address and issues a *req* to local vault's NIC. Then, the remote vault adds this request to the DRAM request queue of the corresponding PE. Next, the accessed data is temporarily buffered in the remote vault's VSM and sent back to the local vault after inter-vault link traversal. The communication interface guarantees delivery, so no acknowledgment is required.

iPIM realizes synchronization among different vaults through a lock-step synchronization instruction (*sync*), which acts as a barrier to block all instructions after this *sync*. The synchronization relies on a centralized master-slave protocol, where a selected vault is designated as the master vault and all other slave vaults are coordinated. For a vault, a synchronization point is reached only if all instructions before that *sync* finish execution. Then, the slave vault will signal the master vault, after which the master vault will update a global synchronization status vector. After the global synchronization point is reached, the master vault will broadcast a *proceed_phase* message to all slave vaults, and all vaults will commit the *sync* instruction and proceed execution phases.

5.2.5 Hardware Components and Usages

This section introduces important information regarding the hardware components in iPIM as follows:

Data/Address Register File (DataRF/AddrRF): Both the DataRF and AddrRF em-

ploy multi-port architecture to avoid resource hazards during execution. The DataRF has a vector interface ($128b$) that aligns with the bank's width. To accommodate the scalar interface ($32b$) of AddrRF, a multiplexer is added. In addition, AddrRF locations $A0$-$A3$ are reserved to store PE's $peID$, $pgID$, $vaultID$, and $chipID$, respectively.

Process Group Scratchpad Memory (PGSM): PGSM is used for data sharing among PEs in a PG. To access another PE's memory, a simple way is to generate a ld_pgsm from the source PE followed by a rd_pgsm to the destination PE with the same PGSM address. To enable parallel PE access, PGSM allocates individual ports for each PE and employs multi-bank architecture. (Fig.5.2(a3)). Each PE has a separate read port and a write port into PGSM so that data loading to PGSM can be overlapped with PGSM access. Also, PGSM has a 2D memory abstraction for image processing applications.

Vault Scratchpad Memory (VSM): VSM has three functionalities. First, VSM is used for data sharing among PEs in a vault. To access another PG's memory, a possible solution is to generate a ld_rf and a wr_vsm to write the data to a VSM location, and use a rd_vsm to bring the data to local PE. Note that TSVs are shared among PGs, so VSM has only one data port for TSVs. Second, VSM temporarily buffers the data for remote vault access. Third, VSM acts as the instruction memory that accepts computation offloading from a host.

In-DRAM Memory Controller: iPIM integrates a lightweight memory controller that serves the banks inside each PG (Fig.5.2(c)). The memory controller contains a memory request queue, a DRAM command buffer, DRAM command translation and issuing logic, a counter to record last DRAM command issuing cycle, a DRAM status register, and an open row address register. Currently the memory controller supports two page policies (open/close page) and two DRAM scheduling policies (FCFS,FR-FCFS) [105]. It also schedules DRAM refresh commands according to t_{REFI} and t_{RFC} timing parameters similar to AxRAM [40].

On/off-chip Network: iPIM adopts a 2D mesh topology for both the on-chip and off-chip network. Each router assumes Input-Queued (IQ) microarchitecture and implements the $X - Y$ routing algorithm. Also, simple flow control and channel allocation policies [106] are used.

5.3 Software Support

This section details the design of an end-to-end compilation flow based on Halide for iPIM hardware. First, we introduce the programming interface of iPIM in Sec.5.3.1, which includes the design of new schedules for iPIM. Second, we explain the compilation flow in Sec.5.3.2 including the extension of Halide front-end compilation passes and our customized backend for iPIM.

5.3.1 Programming Interface

To support various image processing applications composed of heterogeneous pipelines on iPIM, we use Halide as the programming language because of its success in this application domain. Our front-end support for Halide eases the burden of programmers from two perspectives. First, the image processing algorithm written in Halide does not have to be changed for iPIM because Halide decouples the algorithm from its schedules. Second, we develop customized schedules to provide an easy-to-use high-level abstraction for indicating workload partition and data sharing among PEs in iPIM. Thus the workload partition and data sharing are optimized automatically by our end-to-end compilation flow according to these high-level schedules without programmers' involvement.

We develop customized schedule primitives to efficiently exploit hardware characteristics on iPIM hardware. In particular, we extend Halide with two new schedule primitives, *ipim_tile*() and *load_pgsm*(), for distributing data into different banks and utilizing the

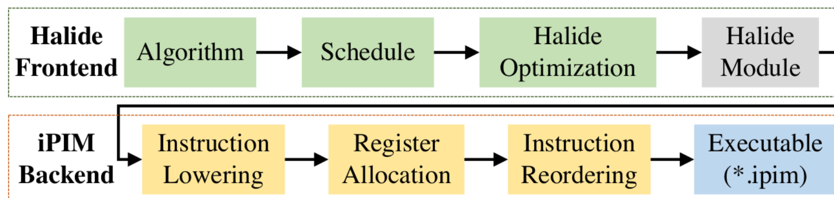


Figure 5.3: The end-to-end compilation flow of iPIM.

scratchpad of a processing-group. The first customized schedule for iPIM, *ipim_tile*(), specifies the dimensions of image data to be partitioned and distributed across the hierarchy of iPIM. This schedule also indicates the distribution of these image tiles across all PEs. The second customized schedule for iPIM, *load_pgsm*(), indicates the usage of shared scratchpad memory at the PG level. By supporting this schedule, data sharing across adjacent image tiles can happen at the PG level.

In addition to our customized schedules for data partition and sharing on iPIM, we leverage existing Halide schedules to specify the fusion of pipelines and the vectorization of computation on iPIM. During code generation, each *compute_root*() implies a kernel function reading input data from and writing output results to DRAM banks. We also exploit the vectorization schedule ($vectorize(xi, 4)$) supported by Halide for iPIM because our ISA includes SIMD instructions. Specifically, we exploit the compilation pass of vectorization in Halide frontend aligning data to improve the utilization of SIMD units in iPIM.

5.3.2 Compilation Flow

As shown in Fig.5.3, we develop an end-to-end compilation flow to support an automatic transformation from a Halide algorithm with customized iPIM schedules to a hardware executable program on iPIM. We develop the frontend code transformation to support our iPIM schedules and the backend instruction optimizations to improve the

performance of generated programs. Our backend optimizations have unique challenges due to our novel near-bank architecture from two perspectives. First, because of the simple in-order control core design, our register allocation phase needs to prevent data hazards due to register contention. Thus, this phase aims to span virtual registers into different physical registers to avoid such data hazards instead of minimizing the number of allocated registers in the typical register allocation phase. Second, our instruction reorder phase needs to optimize row buffer locality when exploiting the instruction-level parallelism (ILP) because of the timing characteristics of DRAM banks. Thus we add new virtual dependencies to enhance the row-buffer locality which is critical to the performance of programs. In summary, our end-to-end compilation flow takes advantage of customized schedules to generate programs exploiting iPIM hardware features, such as PGSM, and our backend optimizations further improve the performance of the programs.

5.4 Experiment

5.4.1 Experimental Setup

Benchmark and Dataset Selection. As detailed in Table.5.2, we use a set of single-stage and multi-stage benchmarks for an in-depth and comprehensive analysis. The single-stage benchmarks cover a wide range of computation and memory patterns in important image processing operations [109], such as elementwise, stencil, reduction, gather, shift, and other data-dependent operations. With them, we are able to provide isolated in-depth analysis for each image processing operation. The multi-stage benchmarks, which are widely used in image processing programming languages [100, 110, 111, 112], on the other hand, contain heterogeneous pipeline stages that require the support of programmability. We use DIV8K [102] dataset, which contains over 1500 images covering

Table 5.2: Image Processing Benchmark Setting.

Category	Benchmark	Description
Single-stage Benchmarks	**Image Brighten**	out(x,y)=$\alpha\cdot$ in(x,y)
	Gaussian Blur	blur_x(x,y)=(in(x,y)+in(x+1,y)+in(x+2,y))/3 blur_y(x,y)=(blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3
	Downsample	d(x,y)=(in(2x-1,y)+in(2x, y)·2+in(2x+1,y))/4 out(x,y)=(d(x,2y-1)+d(x,2y)·2+d(x,2y+1))/4
	Upsample	u(x,y)=(in(x/2,y)+in((x+1)/2,y))/2 out(x,y)=(u(x,y/2)+u(x,(y+1)/2))/2
	Shift	out(x,y)=in(x-4,y-4)
	Histogram	RDom r(0,in.width(),0,in.height()) histogram(in(r.x,r.y))+=1
Multi-stage Benchmarks	**Bilateral Grid**	It uses the bilateral grid filter to smooth images with edges preserved (4 pipeline stages) [107]
	Interpolate	It interpolates pixel values using a pyramid of low-resolution samples (12 pipeline stages) [100]
	Local Laplacian	It tone-maps an image and enhances its local contrast using a multi-scale method (23 pipeline stages)[108]
	Stencil Chain	It is composed of a chain of stencil computations (32 pipeline stages) [100]

diverse scene contents with 8K (7680×4320) resolution for all the evaluated benchmarks. The choice of a high-resolution dataset is to reflect the application trend on workstations and data-center that deep learning training, medical image processing, and geographical information system require higher image quality.

Hardware Configuration. iPIM assumes 3D-stacking memory configuration similar to previous near-bank accelerators [39, 40, 41] without changing DRAM's core timing. We list the detailed hardware configuration, latency values, energy consumption, and DRAM settings in Table.5.3. We also consider important timing parameters to limit power (t_{RRDS}=4, t_{RRDL}=6, t_{FAW}=16). iPIM contains 8 iPIM cubes (total $\sim 850mm^2$) to compare with a Tesla V100 GPU card [97] with 4 HBM stacks (total $\sim 1199mm^2$), where one HBM stack consumes $\sim 96mm^2$ footprint [12].

Simulation Methodology. We develop a cycle-accurate simulator extended from

Table 5.3: iPIM hardware configuration parameters.

Parameter Names	Configuration
Cubes/Vaults/PGs/PEs/InstQueue/DRAMReqQueue	8/16/8/4/64/16
SIMD_len / CAS_width / link_width (SERDES)	4/128b/4
Bank / AddrRF / DataRF / PGSM / VSM (Byte)	16M/256/1K/8K/256K
tCK / tRCD / tCCD / tRTP / tRP / tRAS (ns)	1/14/2/4/14/33
tADDRRF / tDATARF / tPGSM / tVSM (ns)	1/1/1/1
tADD(SUB) / tMUL / tMAC / tLOGIC (ns)	4/5/8/1
tPEbus / tTSV / tNoC (hop) / tSERDES (hop) (ns)	1/1/1/0.08
RD,WR / PRE,ACT / AddrRF / DataRF (J/access)	0.52n/0.22n/0.43p/2.66p
SIMD Unit / Int ALU (J/access)	87.37p/11.05p
PEbus / TSV / SERDES (J/bit)	0.017p/4.64p/4.50p
DRAM_rowbuffer_policy / DRAM_schedule	open_page / FR-FCFS

ramulator [44] by integrating customized compute-logic and buffers with DRAM banks. iPIM is designed to run at a clock frequency of 1GHz under the 22nm technology node. We use cacti-3DD [93] to evaluate the inter-PE interconnects, TSV, and the 3D DRAM bank access latency and energy. The energy, performance, and area of the address/data register file and process group/vault scratchpad memory are also simulated by cacti-3DD. The base die and the SERDES energy are set based on previous near data processing work [94]. The hardware components of SIMD units and integer ALUs are synthesized by design compiler [113] to derive performance, power, and area results. For all the evaluated components on the DRAM die, we conservatively assume $\times 2$ area overhead considering reduced metal layers in the DRAM process [40]. For the control core on the base logic die, we adopt an in-order ARM cortex-A5 core [114] to evaluate its area and power. For the GPU evaluation, the baseline image processing workloads are written in Halide with manually-tuned schedules. The GPU performance and power are measured from *nvprof* and *nvidia-smi*, respectively.

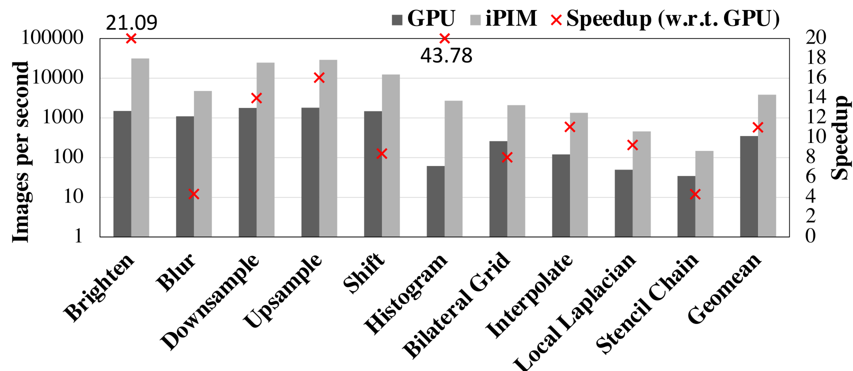


Figure 5.4: Throughput and speedup comparison between iPIM and GPU.

5.4.2 Performance, Energy-efficiency, Area, and Thermal Issues

Performance. iPIM achieves 11.02× average speedup over the GPU as shown in Fig.5.4. From the hardware's point of view, this speedup is mainly attributed to iPIM's ample memory bandwidth as a result of near-bank architecture (more comparisons in Sec.5.4.3). From the software's point of view, this high speedup is achieved through good compiler optimizations.

Next, we explain the variations in the speedup for different benchmarks. First, the Brighten benchmark consists of elementwise operations which are completely bound by memory bandwidth, so iPIM's enormous bank-level bandwidth can provide very good speedup (21.09×). Second, the Histogram benchmark involves data-dependent computation resulting in inferior performance using Halide's default schedule on GPU. The schedule on iPIM converts it into a reduction of parallel reduced partial histogram results, thus it achieves significant performance improvement (43.78×). Third, Blur and Stencil Chain benchmarks only have moderate speedup (4.32× and 4.30×, respectively) on iPIM. Later analysis (Sec.5.4.4) shows that these two benchmarks have higher computation intensity than other benchmarks, and involve a lot of index calculations which

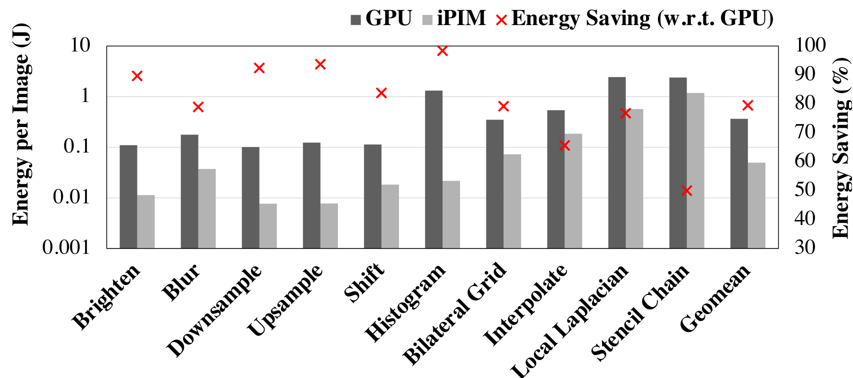


Figure 5.5: Energy comparison between iPIM and GPU.

are bound by address register file. As a conclusion, the results indicate that iPIM can effectively accelerate a wide range of image processing applications.

Energy-efficiency. iPIM achieves 79.49% average energy saving over the GPU (Fig.5.5). The energy saving mainly comes from the reduction of expensive data movement compared with GPU, since iPIM's compute-logic can use the local bank without off-chip data access. Sec.5.4.3 provides a more detailed energy breakdown to show the small overhead of data movement in iPIM. Also, we observe that for each benchmark the energy saving in Fig.5.5 is approximately proportional to the speedup in Fig.5.4. This is because iPIM's increased bank-level bandwidth is a result of near-bank data access, which also contributes to the reduction of data movement energy.

Next, we explain the difference in energy saving between single-stage benchmarks and multi-stage benchmarks (89.26% and 66.81%, respectively). iPIM employs *compute_root* schedule, where intermediate data between pipelines are written back to banks without fusing. In comparison, since Halide employs pipeline fusion for multi-stage benchmarks on GPU, the expensive off-chip memory access can be reduced due to increased on-chip data reuse. As a result, iPIM has a slight drop in energy saving for multi-stage benchmarks.

Area. iPIM's decoupled control-execution architecture is area-efficient because it

Name	Number	Area (mm^2)	Overhead (%)
SIMD Unit	64	2.26	2.36
Int ALU	64	0.32	0.33
Address Register File	64	0.20	0.21
Data Register File	64	1.79	1.86
Memory Controller	16	1.84	1.92
PGSM	16	3.87	4.03
Total	-	10.28	10.71

Table 5.4: Area evaluation of iPIM components on the DRAM die considering DRAM process overhead.

only adds small area overhead (execution part) per DRAM die and the control core can be well fitted on the base logic die. First, we evaluate the area of execution components in the PIM layers considering DRAM process overhead (Fig.5.2(c)), and normalize the total added area to a DRAM die ($96mm^2$ [12]). We show that the added area per DRAM die is small (10.71%) to support programmability according to Table.5.4. Second, we evaluate the area of iPIM's control core on the base logic die (Fig.5.2(b)). The core consumes $0.92mm^2$ total silicon footprint (including the VSM which takes $0.23mm^2$), and it can be well fitted into the extra area of each vault ($3.5mm^2$ [38]) on the base logic die. On the contrary, if this control core is naïvely integrated with each bank, the total area overhead per DRAM die will increase to 122.36%, which is 10.42× larger than that of our decoupled control-execution design.

Thermal Issues. iPIM's peak power is $63W$ per cube considering both DRAM dies and the base logic die, and the peak power density is $593mW/mm^2$. The normal operating temperature for 8Hi HBM2 DRAM dies is $105°C$ [12], and we conservatively assume the DRAM dies in our case operates under $85°C$. A prior study on 3D PIM thermal analysis [115] shows that active cooling solutions can effectively satisfy this thermal constraint ($85°C$). Both commodity-server active cooling solution [116] (peak power density allowed: $706mW/mm^2$) and high-end-server active cooling solution [117] (peak

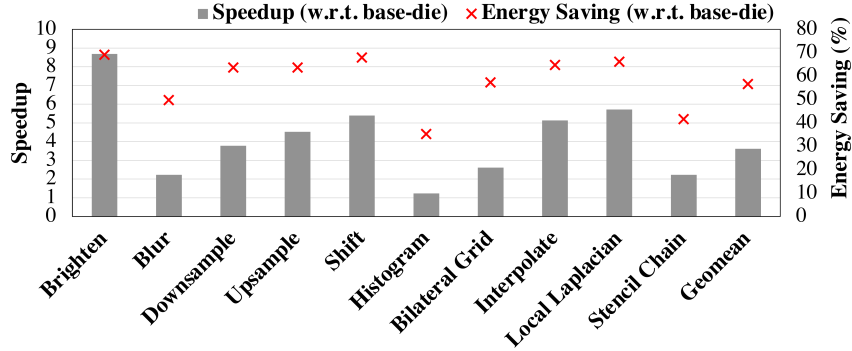


Figure 5.6: Comparison of near-bank and process-on-base-die solutions.

power density allowed: $1214mW/mm^2$)) can be used. Also, compared with previous work [115] where PIM logics are concentrated on the base logic die far from the top heat sink, iPIM distributes the PIM logics evenly to each DRAM dies, so the heat dissipation will be much better [118]. In addition, we note that the majority of the peak power (78.5%) is induced by simultaneously activating/precharging DRAM banks. Since iPIM compiler optimizes row buffer locality for image processing workloads, for memory-intensive workloads with ideal row buffer locality, the frequency of this activity is relatively low.

5.4.3 Architecture Analysis

Comparison of iPIM and process-on-base-die solution

We compare iPIM with the process-on-base-die (PonB) solution and observe that iPIM on average achieves 3.61× speedup and 56.71% energy saving as shown in Fig.5.6. We further explain the PonB configuration and the advantages of iPIM over the PonB solution. The only difference of PonB with iPIM is that all near-bank components are moved to the base logic die, and these components access their DRAM banks through TSVs. We evaluate PonB using the same benchmarks and simulator while serializing the data traffic on the shared TSVs between the base logic die and the DRAM dies.

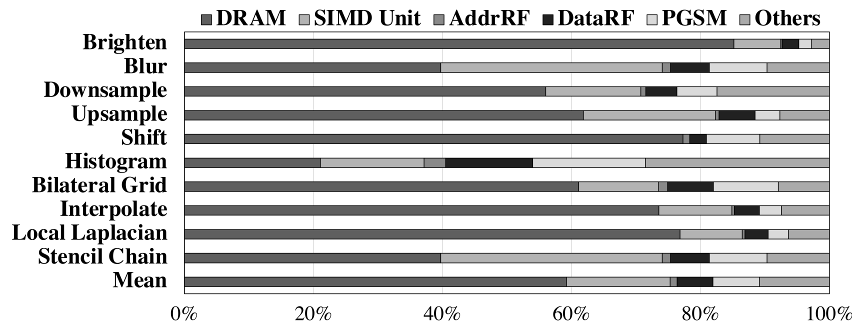


Figure 5.7: Energy breakdown of iPIM programs.

The inferior performance of the PonB solution is because all memory accesses need to go through TSVs with limited bandwidth, which is only 10% of iPIM's peak memory bandwidth. The energy overhead of the PonB solution is induced by expensive in-cube data movement energy, which is 2.48× of iPIM's local bank access energy. We argue that it is impractical for the PonB solution to have the same memory bandwidth as iPIM by increasing the number of TSVs, since this will increase the TSV overhead by 10×, which translated to 187% area overhead per DRAM die.

Energy Breakdown

We provide a detailed energy breakdown of iPIM programs shown in Fig.5.7. The $DRAM$ in this figure contains the background energy, activation/precharge (RAS) energy, read/write (CAS) energy, and refresh energy. The $SIMDunit$ contains all floating/integer operation energy of the SIMD unit. The $AddrRF/DataRF/PGSM$ contains the read/write energy and leakage energy. The $Others$ contains data movement energy and control core's energy on the base logic die. The breakdown shows that iPIM's decoupled execution-control architecture spends most of the energy on PIM dies (89.17%), and only a small part on data movements and the control core (10.83%). This can be further justified by the instruction breakdown analysis in Sec.5.4.4. The low energy consumption of inter-vault and intra-vault data movement is contributed from (1) iPIM's near-bank

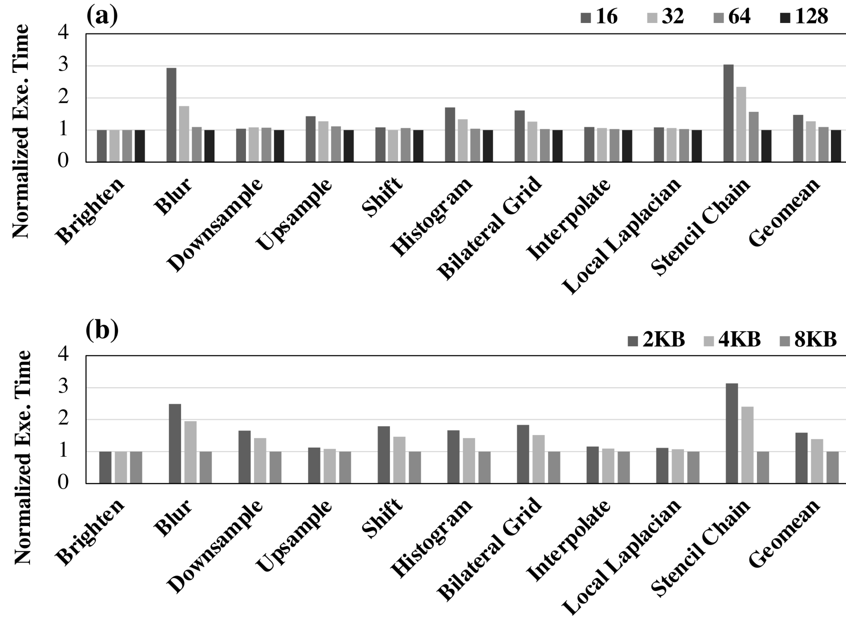


Figure 5.8: Sensitivity of (a) the number of registers; (b) the scratchpad size.

architecture and (2) localized data movement benefited from the memory hierarchy and compiler optimizations.

Sensitivity Analysis

We conduct sensitivity studies on how the number of registers per PE (RF) and Process Group Scratchpad size (PGSM) will impact the execution time. First, to study the RF sensitivity (Fig.5.8(a)), we vary the RF value from 16 to 128 and normalize the execution time to the case when RF=128. We observe that RF=16, RF=32, and RF=64 have 46.8%, 26.8%, and 9.5% performance drop compared to RF=128, respectively. The performance drop is attributed to the decreased number of registers that results in (1) more registers spilling to the local DRAM bank, and (2) increased register data dependency. Second, to study the PGSM sensitivity (Fig.5.8(b)), we change the PGSM from $2KB$ to $8KB$ and normalize the execution time to the case when PGSM=$8KB$. We observe that PGSM=$2KB$ and RF=$4KB$ have 58.9% and 39.0% performance drop com-

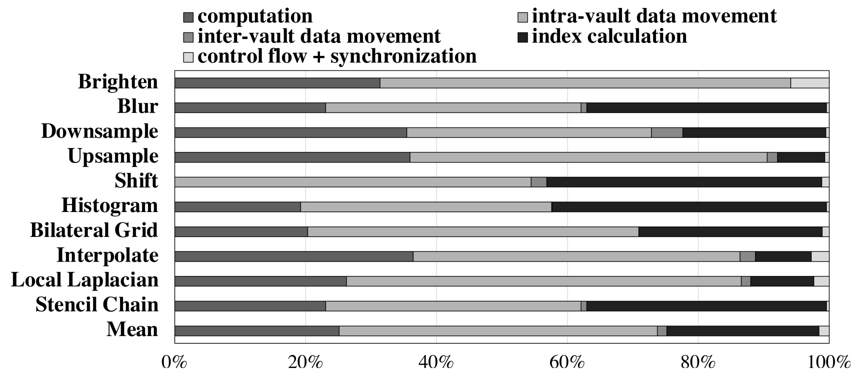


Figure 5.9: Instruction breakdown of iPIM programs.

pared to PGSM=8KB, respectively. This is because reduced scratchpad size will increase the number of accesses to long latency DRAM. For iPIM design, we choose RF=64 and PGSM=8KB as a tradeoff between performance and area overhead (Table.5.4).

5.4.4 Instruction Breakdown

From the instruction breakdown of iPIM programs shown in Fig.5.9, we can observe that each benchmark is a combination of different instructions with varied ratios. From the programmability perspective, this indicates that SIMB ISA efficiently maps heterogeneous pipeline stages which exhibit diverse computation, data movement, and control flow patterns. Therefore, SIMB ISA can provide flexible support for a wide range of image processing applications.

We also find that index calculation instructions on average take 23.25% of total instruction count. Since the image uses 2D memory abstraction and physical memory assumes linear address space, frequent index calculations are required to map the 2D image reference locations to the corresponding memory addresses. This index calculation overhead takes more than 28% of total instruction count for Blur, Shift, Histogram, Bilateral Grid, and Stencil Chain benchmarks. This provides a direct explanation about iPIM's moderate speedup on these benchmarks as shown in Fig.5.4 except for Histogram.

Another important observation is that the inter-vault data movement instructions are a very small part (1.44%) of the total instruction count. This confirms image processing kernels have good data parallelism and can be efficiently mapped onto near-bank architecture with little global data movement.

5.5 Summary

In this chapter, we propose iPIM, a programmable in-memory image processing accelerator using near-bank architecture. iPIM uses a decoupled control-execution architecture to support lightweight programmability. It also contains a novel SIMB (Single-Instruction-Multiple-Bank) ISA to enable various computation and memory patterns for heterogeneous image processing pipelines. In addition, we develop an end-to-end compilation flow extended from Halide with new schedules for iPIM. The compiler backend further contains optimizations for iPIM including register allocation, instruction reordering, and memory order enforcement. Evaluations show that iPIM supports programmability with small area overhead, and provides significant speedup and energy saving compared with GPU. Further analysis demonstrates the benefits of iPIM compared with the previous process-on-base-logic architecture design and the effectiveness of iPIM's compiler optimizations.

Chapter 6

MPU: Towards Bandwidth-abundant SIMT Processor via Near-bank Computing

This chapter is dedicated to solve the memory-bandwidth bottleneck in state-of-the-art single-instruction-multiple-thread (SIMT) accelerator (i.e., GPU) by using the 3D-stacking near-bank design [40, 39, 41, 18]. This design moves simple arithmetic units closer to the DRAM banks to harvest the abundant bank-internal bandwidth (around $10\times$ w.r.t. process-on-logic-die solution [18]). These near-bank accelerators have demonstrated significant speedups (around $2\times - 14\times$ w.r.t. GPU) thus they are promising to tackle the memory bandwidth issue in state-of-the-art GPU accelerators. Despite its potential to provide plentiful memory bandwidth, there are still several challenges for near-bank computing in accelerating general purpose data-intensive workloads. First, the pipeline of SIMT processors contains complex logic components (e.g., load-store-unit [119]) and large register files [120]. Different from the prior near-bank accelerators customized for applications, the SIMT pipeline is needed for general purpose

data-intensive workloads. Naively placing the whole pipeline with complex logic components and complicated data paths in the DRAM die introduces an intolerable area overhead [33, 98, 99]. Second, the efficient support of the SIMT programming model on near-bank computing is needed, especially the inter-thread communication and the dynamic scheduling of warps [121]. As the shared memory is frequently used for inter-thread communication in a thread block, directly placing it on the base logic die will incur enormous Through-Silicon-Via (TSV) traffic. The dynamic scheduling of warps could disrupt the row-buffer locality of DRAM banks, seriously downgrading bandwidth utilization. Third, it requires both the end-to-end support of a parallel programming language to ease the programmers' burden and backend compiler optimizations for near-bank computing to exploit hardware potentials.

In this chapter, we present MPU (Memory-centric Processing Unit), the first SIMT processor based on 3D-stacking near-bank computing architecture, and an end-to-end compiler flow supporting CUDA programs [122] with optimizations tailored to MPU. First, we design a hybrid SIMT pipeline for MPU where only a small number of registers and other lightweight components are added on the DRAM dies. At runtime, instructions are fetched, decoded, and issued on the base logic die while they can be offloaded to near-bank units (NBU) according to either compiler hints or hardware policies. To facilitate this hybrid execution of instructions, we propose an instruction offload engine to make instruction movement decisions, a register track table and a register move engine to flexibly transfer registers, and a load-store unit extension to handle near-bank load/store requests. Second, we propose two architectural optimizations for the SIMT model. For the shared memory, we move it to the DRAM die and restructure the core organization by placing all NBUs associated with the same core on the same DRAM die. For the dynamic scheduling of thread warps, we enable multiple activated row-buffers per DRAM bank to reduce the ping-pong effect thus improving the bandwidth. Third, we propose

an end-to-end compilation flow to support CUDA programs. To optimize instruction offloading location on MPU's hybrid pipeline, we further propose a novel instruction and register location annotation algorithm through the static analysis of instructions, which effectively reduces the data movement among the shared TSVs.

The contributions of this chapter are summarized as follows:

- To the best of our knowledge, we design the first near-bank SIMT processor using a hybrid pipeline with an instruction offloading mechanism. By integrating lightweight hardware components on the DRAM die, MPU achieves a small area overhead for general purpose processing.
- We propose two architectural optimizations for the SIMT model, including the near-bank shared memory to reduce data movement and multiple activated row-buffers to alleviate ping-pong effects in the dynamic warp scheduling.
- We develop an end-to-end compilation flow supporting CUDA programs on MPU and a novel backend optimization annotating the locations of registers and instructions.
- Evaluation results of representative data-intensive workloads show that MPU with all optimizations achieves 3.46× speedup and 2.57× energy reduction on average over an NVIDIA Tesla V100 GPU.

6.1 Motivation

Despite the success of the graphics processing unit (GPU) in accelerating data-intensive parallel programs, we observe from performance characterizations that the memory bandwidth is a serious performance challenge for these workloads on the state-of-the-art GPU. Specifically, we evaluate a set of representative data-intensive workloads

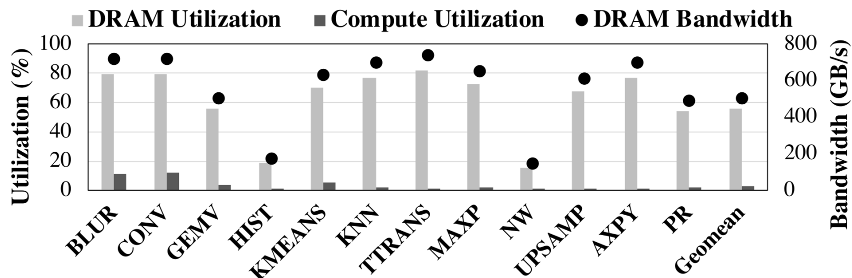


Figure 6.1: Profiling results for data-intensive workloads on NVIDIA Tesla V100 GPU.

from various application domains including deep learning, bioinformatics, linear algebra, and image processing applications as detailed in Table 6.4.1. The measured memory bandwidth, bandwidth utilization, and compute utilization of an NVIDIA Tesla V100 GPU [123] are shown in Fig.6.1. On average, these benchmarks achieve 55.90% DRAM bandwidth utilization (503.10 GB/s) and 2.57% compute utilization. The saturation of DRAM bandwidth and the low utilization of the compute resources exhibit a memory-bandwidth bound behavior. This performance characteristic results from the low arithmetic density and the regular memory access patterns in most of these workloads. Also, we note that the workloads $HIST$ and NW show relatively low bandwidth utilization as a result of the long memory access latency on GPU.

For workloads suffering from either the limited DRAM bandwidth or the long DRAM access latency on GPU, near-bank computing is a promising architecture to alleviate these performance bottlenecks because of both abundant bank-level memory bandwidth and reduced memory access latency. However, prior near-bank computing accelerators [40, 39, 41, 18] are domain-customized, since they have simple data paths, application-specific mapping strategies, and inefficient general purpose programming language support. The lack of programmability for these accelerators confines them to a niche application market, adding non-recurring engineering costs in manufacturing. Moreover, parallel data-intensive workloads usually come from various application domains, making none of these

near-bank accelerators feasible to support all of these parallel programs.

In summary, the memory bandwidth bottleneck on the state-of-the-art GPU urges the need for a higher memory bandwidth for data-intensive parallel programs, and the huge overheads of placing an SIMT processor near banks introduce unique technical challenges. Both of these factors motivate us to design MPU, the first general purpose SIMT processor based on 3D-stacking near-bank computing to exploit bank-level bandwidth and alleviate the GPU bandwidth bottleneck.

6.2 Architecture Design

6.2.1 Microarchitecture Overview

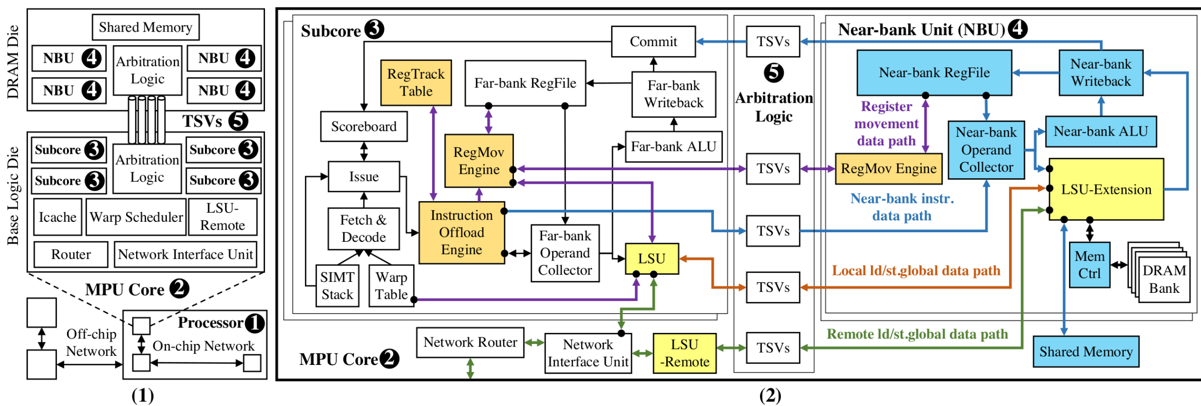


Figure 6.2: (1) MPU architecture overview. (2) Detailed microarchitecture of a MPU core (left: Subcores, middle: TSVs, right: NBUs). Added near-bank components are colored in blue. Newly supported components for instruction offloading and register movement (Sec.6.2.2) are colored in orange. Hybrid load-store unit (LSU) components (Sec.6.2.2) are colored in yellow.

From the high-level, MPU adopts a scalable design with many processors (Fig.6.2❶) interconnected by an off-chip network (similar to SERDES links in the HMC [124]) as shown in the bottom of Fig.6.2 (1). Each processor is a 3D-stacking cube of a base logic die stacked with multiple DRAM dies, connected by vertically shared buses called

the through silicon vias (TSVs) [125] (Fig.6.2❺). The base logic die is horizontally partitioned into an array of SIMT cores (Fig.6.2❷) interconnected by an on-chip 2D-mesh network [106].

To harvest the near-bank bandwidth with small overheads on DRAM dies, MPU's SIMT core adopts a hybrid pipeline design. In the MPU core (Fig.6.2❷), complex logics are placed on the base logic die, and some lightweight components in the execution stage are replicated on near-bank locations. On the base logic die, a core consists of four subcores (Fig.6.2❸), an instruction cache, a warp scheduler, and components for handling inter-core traffic (network interface unit, router, and LSU-Remote). The TSVs (Fig.6.2❺) are evenly divided among the cores (64*b* data buses per core), via which the subcores can communicate with near-bank components on DRAM dies. All the core's near-bank components are located within the same DRAM die, containing four near-bank units (NBUs) (Fig.6.2❹) and the shared memory. To enable efficient processing in this hybrid architecture (Sec.6.2.2), we propose a novel instruction offloading mechanism and a hybrid load-store unit design.

In addition, MPU considers two architectural optimizations for the SIMT programming model in Sec.6.2.3. First, we find that naively implementing shared memory on the base logic die results in poor performance, so we restructure the core's 3D organization and develop a near-bank shared memory design. Second, we observe that the dynamic execution of SIMT warps may disrupt the row-buffer locality of DRAM banks, so we adopt a technique to enable multiple activated row-buffers inside the same DRAM bank.

6.2.2 Hybrid Pipeline

As illustrated in Fig.6.2 (2), an MPU core (Fig.6.2 (2)❷) adopts a hybrid pipeline design that is split between the base logic die (subcore) (Fig.6.2 (2)❸) and the DRAM die

(near-bank unit, NBU) (Fig.6.2 (2)❹). The frontend components of the SIMT pipeline mostly comprise of control flow and data dependency logic, so they are mainly contained in the subcores, including instruction fetch (I-cache, SIMT stack, warp table), decode, and issue (scoreboard) stages. For the backend pipeline, MPU duplicates some simple parts from the subcore to the NBU, including the register file, operand collectors, and ALUs. Other complex units such as LSU and network interface units are left on the base logic die. Note that the memory controller and the shared memory are entirely moved from the base logic die to the DRAM die, since near-bank memory controller will reduce TSV traffic for DRAM commands [40, 18], and shared memory can reduce TSV traffic for register movement (Sec.6.2.3).

In addition to the original pipeline components, to assist flexible instruction offloading, each subcore also adds an instruction offload engine, a register move engine, and an associated register track table (Sec.6.2.2). Besides, the load-store unit (LSU) is modified and augmented to support remote data traffic and near-bank instruction offloading (sec.6.2.2).

Instruction Offloading Mechanism

The instruction offload engine after the issue stage will decide whether to offload the instruction for near-bank execution (*Near-bank instr. data path* in Fig.6.2 (2)). The offloaded instruction will first travel through the TSVs, then access the near-bank operand collector to collect operand data from the near-bank register file. Then, the arithmetic and logic computation instructions will be sent to the near-bank ALUs, and the *ld/st* instructions will be provided to the LSU-Extension for further processing, where the shared memory or the DRAM controller is involved. After the execution finishes, the resulting registers will be written back into the near-bank register file, and the instruction is returned to the subcore for the final commit, where the scoreboard clears its data

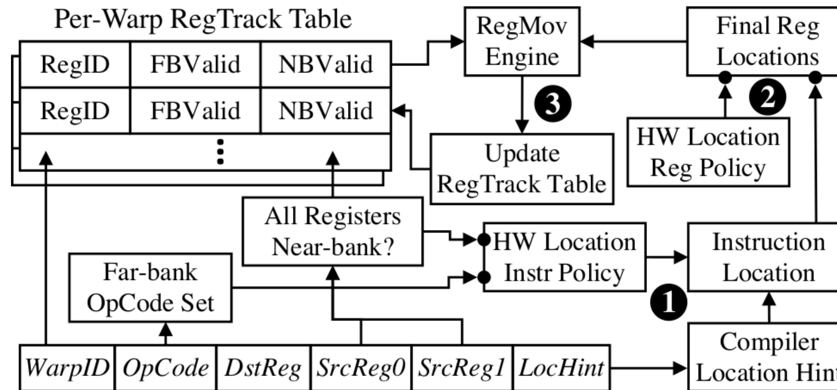


Figure 6.3: Instruction Offloading Mechanism.

dependency.

The first step (Fig.6.3❶) is to identify the target instruction's location according to three policies with decreasing priority. The first policy will set the instruction with the far-bank location if the corresponding operation type ($OpCode$) falls in the far-bank operation set. For example, since address range checking and memory coalescing can only be performed by the LSU in the subcore, currently *ld/st.global* instructions are classified as far-bank locations. If the first hardware policy cannot determine the location, then the compiler hint associated with the instruction will determine whether this instruction will be offloaded to NBU or not. If the instruction has no compiler hints, then the hardware will check the register track table. The instruction will be offloaded to NBU if all source registers have valid near-bank copies. Otherwise, the instruction will be passed to the far-bank. This default policy takes the far-bank subcore as a fall-back location for all instructions as it has the full pipeline support.

The second step (Fig.6.3❷) is to determine the locations for the source registers using the hardware policy or the instruction location derived in the first step. For *ld/st.global*, the hardware policy will set the address register location to be far-bank, since it is required by the LSU. The data register location is set to near-bank. For *ld/st.shared*, both the address register and the data register are set to near-bank location. If the

hardware policy is not given, all the source and destination registers' locations will follow the instruction's location.

The third step (Fig.6.3❸) will move registers to their corresponding locations if they are not currently available in the register track table (*Register Movement data path* in Fig.6.2 (2)). For example, the instruction offload engine may require register $\%r1$ to be valid in the far-bank register file, while the register track table indicates $\%r1$ only exists in the near-bank register file (e.g., *FBValid* is False but *NBValid* is True). Then, the far-bank register move engine initiates a request to the near-bank register move engine, which then reads $\%r1$ from the near-bank register file and returns it to the far-bank register move engine. The far-bank register move engine will write $\%r1$ into the far-bank register file, and the instruction offloading engine will start instruction offloading once all registers are in the target locations. Note that we optimize register locations in Sec.6.3.2, so a hit in the register track table will not cause register movement. In the end, the register track table is updated to reflect the most current register location information.

Hybrid Load-store Unit (LSU)

The original LSU in each subcore is augmented with the LSU-Remote in each core to handle remote traffic (*Remote ld/st.global data path* in Fig.6.2 (2)), and the LSU-Extension in each NBU to handle both near-bank instruction offloading and local DRAM transactions (*Local ld/st.global data path* in Fig.6.2 (2)). We will introduce them and use *ld.global* instruction as an example in the following paragraphs.

LSU: As shown in Fig.6.4 (1), after receiving a *ld.global* instruction, the LSU first performs address range checking (Fig.6.4❶) to split the instruction to remote access and local access. If there is remote access, it is encoded and sent to the network interface unit (Fig.6.4❷) to request remote data. For the local access, the following steps are

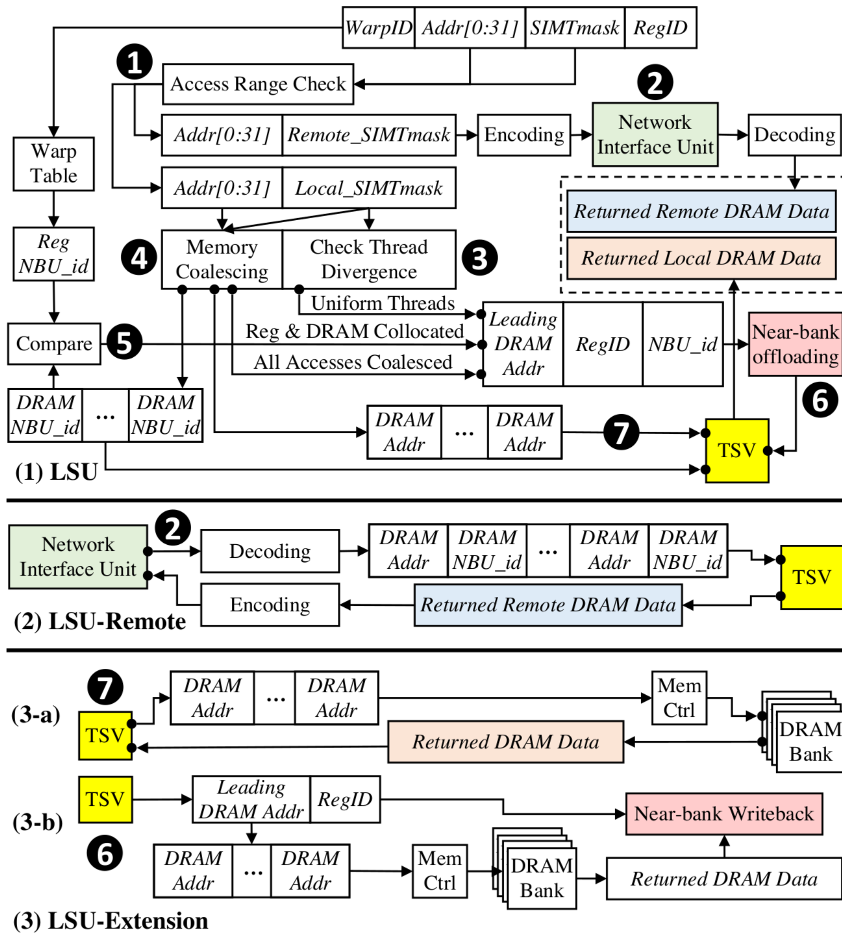


Figure 6.4: Three load-store unit (LSU) components' microarchitecture using *ld.global* data path as an example. (1) LSU in each subcore, (2) LSU-Remote in each core, (3) LSU-Extension in each near-bank unit

performed concurrently. First, the LSU will check if all the SIMT mask fields are valid or not to determine thread divergence (Fig.6.4❸). Second, the LSU will perform memory coalescing (Fig.6.4❹) on the local memory addresses. It will also judge if all the memory addresses are perfectly coalesced, meaning that the load request will access a continuous DRAM address space. Third, the LSU will compare the NBU_id field of the generated DRAM addresses with the NBU_id associated with the register in the given warp (Fig.6.4❺). The *ld.global* instruction is decided for near-bank offloading (Fig.6.4❻) only if all threads are valid, register and DRAM addresses have the same NBU_id, and

all DRAM addresses are coalesced. Note that since all of the above assumptions are satisfied, we only need to transfer the leading DRAM address, register ID, and the NBU_id. The SIMB mask will be ignored and restored in the LSU-Extension side. If the above conditions cannot be met, the LSU will issue DRAM transactions to the LSU-Extension (Fig.6.4❼) and gather returned local DRAM data. Combined with the returned remote DRAM data, the final DRAM data will be used to compose a register write request and transferred for near-bank writeback. The reason to load the DRAM data first to the near-bank register file is that it can benefit near-bank execution due to the reduction of TSV traffic. For far-bank execution, the register data will eventually be brought down to the far-bank register file, so there is no increase in the TSV traffic.

LSU-Remote: As shown in Fig.6.4 (2), LSU-Remote receives remote *ld.global* request and decodes them into a series of DRAM addresses and DRAM NBU_id. It then sends these DRAM transactions to the LSU-Extension through the TSVs. After receiving the returned DRAM data, it encodes it together with the source core location and request ID and composes a response packet, finally sending it back to the original core's LSU who requests this DRAM data.

LSU-Extension: As shown in Fig.6.4 (3), LSU-Extension has two data paths. In Fig.6.4 (3-a), it handles DRAM transaction requests from the TSVs by sending the DRAM addresses to the memory controller, and sends back the returned DRAM data through TSV either to the LSU-Remote or the LSU. In Fig.6.4 (3-b), it handles offloaded local *ld.global* instruction by first restoring the full address list from the leading DRAM address. Then, it sends the DRAM addresses to the memory controller, gathers returned DRAM data, and stores the data into the near-bank register file according to the register ID.

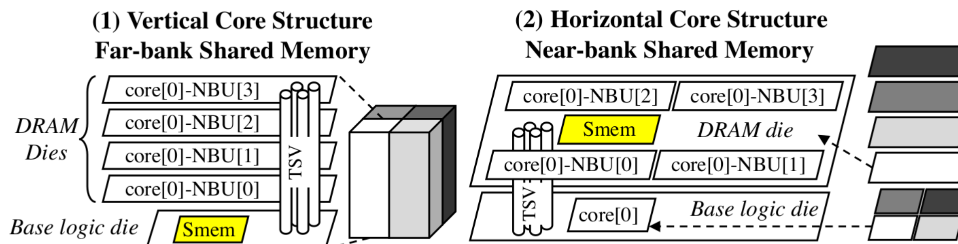


Figure 6.5: (1) Vertical and (2) horizontal core structure.

6.2.3 Optimizations for the SIMT model

Near-bank Shared Memory Design: The shared memory is extensively used for inter-thread communication in the same thread block for a great number of important GPU benchmarks [121]. If the default shared memory location is set in the far-bank subcore on the base logic die, a lot of register data movement traffic will be created and the TSVs will be congested, causing significant performance loss. Thus, it will be desirable to enable a near-bank shared memory design and set the default register location for *ld/st.shared* to the near-bank register file. However, this is impossible in the vertical core structure (Fig.6.5 (1)) in the default HMC-style setting [124], where all the NBUs associated with a core are distributed among multiple 3D stacks. Under such an assumption, moving the shared memory (*Smem*) to each NBUs means that the shared memory is split into each 3D layer and inter-thread shared memory accesses need to go through the bandwidth-bound TSVs. To enable the near-bank shared memory, we restructure the core's 3D-organization as shown in the horizontal core design in Fig.6.5 (2). In our solution, we put all NBUs of the same core into the same DRAM die, so that all NBUs can access the near-bank shared memory without TSV's constraints. In Sec.6.4.3, we confirm the benefits of this optimization on benchmarks that extensively use shared memory.

Multiple Activated Row-Buffers Design: The dynamic execution of warps will create a ping-pong effect on DRAM's row-buffer. Ideally, warps from the same thread

block executing the same memory access instruction will have continuous DRAM addresses and result in a high row-buffer hit rate. However, the hardware dynamically issues available instructions from each warp, resulting in the ping-pong effect of different warps accessing a few row-buffers irregularly. Since MPU has no hardware cache, this ping-pong effect will cause frequent DRAM precharge and activations, significantly downgrade its performance.

To solve this issue in a software transparent way, we observe that for a lot of benchmarks we evaluate, only a small set of row-buffers are active in a short period. If we can enable multiple row-buffers to be simultaneously activated, then this ping-pong effect will be greatly alleviated. Based on the design of MASA (Multitude of Activated Subarrays) [87] which enables multiple subarrays' row-buffers to be activated in parallel, we change our address mapping so that continuous DRAM row addresses will be mapped to interleaved subarrays' physical row. Extra row address latches and access transistors are added to enable different warps to access different row buffers in independent subarrays. Through evaluations in Sec.6.4.3, we confirm that this design can decrease the row-buffer miss rate and increase performance for multiple benchmarks.

6.3 Software Support

Sec. 6.3.1 introduces the role of MPU in a heterogeneous platform and its programming interface. Sec. 6.3.2 discusses the compiler support of transformations from high-level CUDA kernels to optimized programs running on the MPU.

6.3.1 Programming Interface

MPU acts as a standalone accelerator in a heterogeneous platform with a similar usage

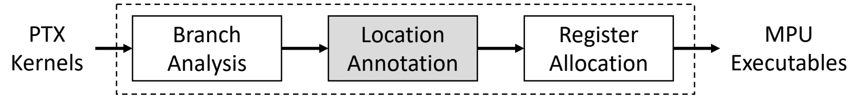


Figure 6.6: The backend of MPU compiler generating MPU executable kernels from PTX kernels.

of the GPU. In terms of the memory system abstraction, MPU has its own memory space independent from the host. In order to use MPU as an accelerator, the host is responsible to allocate the device memory on MPU, transfer input data to MPU, launch computation kernel to MPU, and transfer computation results from MPU. During the kernel launch, MPU runtime is responsible to dispatch the workload of thread blocks to MPU cores according to the thread block configurations. To ease the burden of implementing kernels on MPU, MPU supports CUDA programming language as a realization of the SIMT programming model. As a result, we can leverage a GPU compiler as our front-end to parse the source code and generate intermediate instructions. Then, our compiler backend as detailed in Sec. 6.3.2 is responsible for backend optimizations tailored for MPU architecture.

6.3.2 MPU Compiler

To enable the SIMT programming model, MPU supports an end-to-end compilation flow from CUDA programs to MPU executable programs. This compilation flow contains a novel static analysis stage to optimize the location assignment of instructions by reducing data movement between MPU cores and near-bank units.

The end-to-end compilation flow of MPU includes frontend stages and backend stages. In frontend stages, the MPU compiler reuses *nvcc* [126] to compile CUDA programs [121] to generate kernels in Parallel Thread Extension (PTX) [127] ISA which is a kind of intermediate representation of CUDA kernels. Then, the backend generates the MPU

Workload	App Domain	Reference	Description
BLUR	Image Processing	Halide [100]	3x3 blur.
CONV	Machine Learning	TensorFlow [81]	3x3 conv.
GEMV	Linear Algebra	cuBLAS [129]	Matrix-vector multiply.
HIST	Image Processing	CUB [130]	Histogram.
KMEANS	Machine Learning	Rodinia [131]	K-means clustering.
KNN	Machine Learning	Rodinia [131]	K-nearest-neighbour.
TTRANS	Linear Algebra	cuBLAS [129]	Tensor transposition.
MAXP	Machine Learning	TensorFlow [81]	Max-pooling.
NW	Bioinformatics	Rodinia [131]	Sequence alignment.
UPSAMP	Image Processing	Halide [100]	Image upsample.
AXPY	Linear Algebra	cuBLAS [129]	Vector add.
PR	Linear Algebra	CUB [130]	Parallel reduction.

Table 6.1: The workloads of the benchmark suite.

hardware executables from the PTX kernels, which includes three main stages.

First, the branch analysis stage infers the re-convergence point of each jump instruction so that the hardware can maintain a SIMT stack to handle thread divergence efficiently during the execution [128]. Second, The register allocation stage analyzes the liveness of each virtual register in the program to build a register interference graph. Third, The location annotation is a novel backend stage to optimize the performance by annotating the location of instructions as either the near-bank NBU or the far-bank subcore on the base logic die. As shown in Fig.6.3, when executing the annotated kernels on MPU, the locations annotated on instructions will be used to finalize the runtime instruction offloading decision as explained in Sec.6.2.2.

6.4 Experiment

6.4.1 Experimental Setup

Benchmark. To evaluate the effectiveness of the MPU design in supporting data-intensive parallel programs, we select a set of representative CUDA workloads as shown

Parameter Names	Configuration
Proc/(3D,Core)/(Subcore,NBU/Bank/RowBuf)	8/(4,16)/(4,4/4/4)
SIMT/BankIO/TSV/(on)offchip_bus (Bit)	32/256b/1024/(256)128
Bank/Icache/(Far)Near-bank RF/Smem (Byte)	16M/128K/(32K)16K/64K
tRCD/tCCD/tRTP/tRP/tRAS/tRFC/tREFI [44]	14/2/4/14/33/350/3900
fCore / fTSV / fRouter / f(on)offchip_bus (GHz)	1/2/2/(2)2
RD,WR/PRE,ACT/REF/RF/SMEM [93] (J/access)	0.15n/0.27n/1.13n/40.0p/22.2p
Operand_collector / LSU-Extension (J/access)	41.49p/39.67p
TSV [132] / (on)off-chip bus [93, 94] (J/bit)	4.53p/(0.72p)4.50p
DRAM_rowbuffer_policy / DRAM_schedule	open_page / FR-FCFS

Table 6.2: MPU hardware configuration parameters.

in Table.6.4.1. In particular, these workloads are from various important application domains including image processing, machine learning, linear algebra, and bioinformatics. Because our MPU compiler needs either CUDA source code or PTX kernels to generate MPU executable programs, we have CUDA implementations of these workloads from either well-known GPU benchmarks, such as Rodinia [131], or writing CUDA programs in the same functionality while achieving performance comparable to state-of-the-art libraries, such as cuBLAS [129].

Hardware Configuration. Using the 3D-stacking memory configuration similar to the previous near-bank accelerators [39, 40, 41, 18], MPU needs no changes to DRAM's core circuit except the multiple activated row-buffers enhancement [87]. The detailed hardware configuration, latency values, energy consumption, and DRAM settings are presented in Table.6.4.1. MPU contains 8 processors (total $\sim 926mm^2$) to compare with a Tesla V100 GPU card [97] with 4 HBM stacks (total $\sim 1199mm^2$), where one HBM stack consumes $\sim 96mm^2$ footprint [12].

Simulation Methodology. We develop an event-driven simulator using SimPy [133], which adapts the simulation framework from GPGPU-Sim [134] for SIMT core model, Ramulator [44] for DRAM model, and Booksim [106] for interconnect network model. MPU is designed to run at a clock frequency of 1GHz under the 20nm technology node.

For energy and latency modeling, we first use the design compiler to get the power values for the SIMT core pipeline based on Harmonica project [19]. Then, we use cacti [93] to evaluate the register file, shared memory, and the DRAM bank. Since the major components in the operand collector and the LSU-Extension are SRAM buffers, we also use cacti to evaluate their latency and energy values. We model the router latency and energy consumption using BookSim2's model [106], and the TSV and on/offchip buses adopt parameters from previous studies [132, 93, 94]. For the ALU, we use the measured results from PTX instructions [135, 136]. For area evaluation, we use design compiler [113] to analyse pre-layout area of the vector ALU and the SIMT core pipeline [19]. We use AxRAM's area result [29] for the in-dram memory controller and scale it to 20nm. The area for the shared memory, register file, operand collector, and LSU-Extension are derived from cacti [93]. For all the above-evaluated components on the DRAM die, we conservatively assume $\times 2$ area overhead considering the reduced number of metal layers in the DRAM process [40]. For multi-row-buffer support, we include the overhead of 128 extra row address latches [87] per memory controller to enable simultaneous activations of 4 subarray row buffers. For the GPU evaluation, the GPU performance and power results are collected with the help of *nvprof* and *nvidia-smi*, respectively.

6.4.2 Performance, Area, Energy, and Thermal Analysis

Performance. MPU achieves 3.45× speedup on average over the GPU as shown in Fig.6.7 (1). This speedup is contributed by the improved memory bandwidth from the hybrid-pipeline near-bank architecture, the architecture optimizations for the SIMT programming model (Sec.6.4.3).

To further explain different speedup numbers across workloads, we plot the memory intensity (Byte/Instruction) and the speedup of these workloads in Fig.6.7 (2). First, we

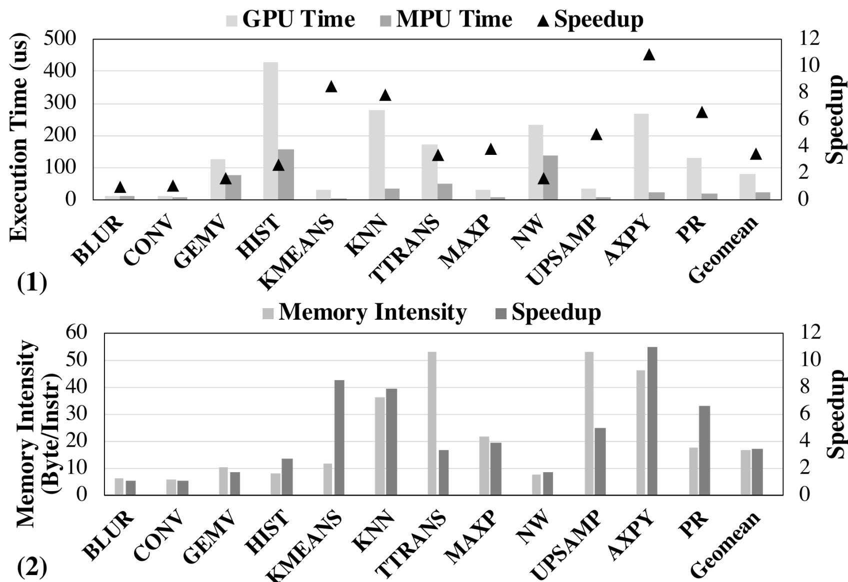


Figure 6.7: (1) Execution time and speedup comparison with the GPU. (2) Workloads memory intensity and speedup.

observe that the speedup number has a strong correlation with the memory intensity because the memory intensity represents the demand of workloads for memory bandwidth. As MPU provides more memory bandwidth than the GPU (4.13× in measurement), for benchmarks with simple memory access and compute patterns (e.g., AXPY), the speedup is proportional to the memory intensity. Second, we find that some benchmarks show higher (KMEANS) and lower (TTRANS, UPSAMP) speedup numbers than their memory intensity. The reason is that memory dependency cannot reflect memory latency characteristics and complex program behaviors. For KMEANS, MPU provides additional latency reduction compared to the GPU, as the compute instructions are mostly data-dependency free so the performance is less sensitive to the number of instructions. For TTRANS and UPSAMP, complicated control flow and data-dependency hinder the memory parallelism, so the abundant memory bandwidth in MPU is not fully utilized.

Area. MPU's hybrid pipeline architecture is area-efficient because only a small part of the pipeline backend components are added in the DRAM die, saving the area for other

Name	Number	Area Per Die (mm^2)	Overhead (%)
Shared Memory	4	0.84	0.88
Register File	16	9.71	10.12
Memory Controller	16	0.63	0.66
Operand Collector	64	2.43	2.53
Vector ALU	16	3.74	3.90
LSU-extension	16	2.43	2.53
Multi-row-buffer Support	64	0.01	0.01
Total	-	19.80	20.62

Table 6.3: Area evaluation of MPU components on the DRAM die considering DRAM process overheads.

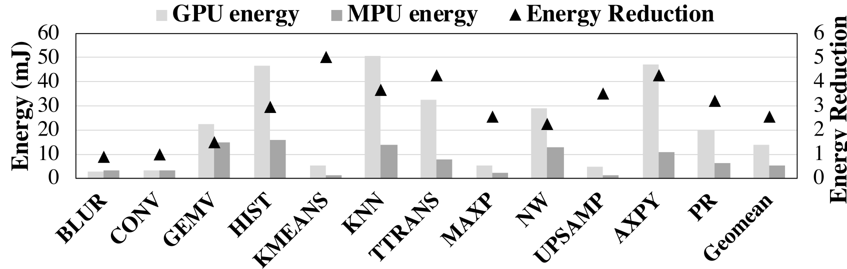


Figure 6.8: Energy and energy reduction comparison with the GPU.

pipeline units. In Table.6.4.2, we evaluate the area of added components and normalize the total overhead to a DRAM die ($96mm^2$ [12]). Thanks for our compiler optimizations which significantly reduce the near-bank register usage, we shrink the near-bank register file to half the size of the far-bank register file. This brings the total area overhead from 30.74% to 20.62%. We argue this overhead is small for a general purpose SIMT processor, comparing to 10.71% area overhead in previous work which only supports domain-acceleration [18]. According to the synthesis result of Harmonica [19] scaled to $20nm$, the $3.4mm^2$ area of the SIMT core with an instruction cache, operand collectors, and a load-store unit can perfectly fit into the available area ($3.5mm^2$ [38]) on the base logic die. On the contrary, if the whole core is placed in the DRAM die, the total area overhead will increase significantly (2× compared with the hybrid pipeline of MPU).

Energy. MPU achieves 2.57× energy reduction on average over the GPU (Fig.6.8).

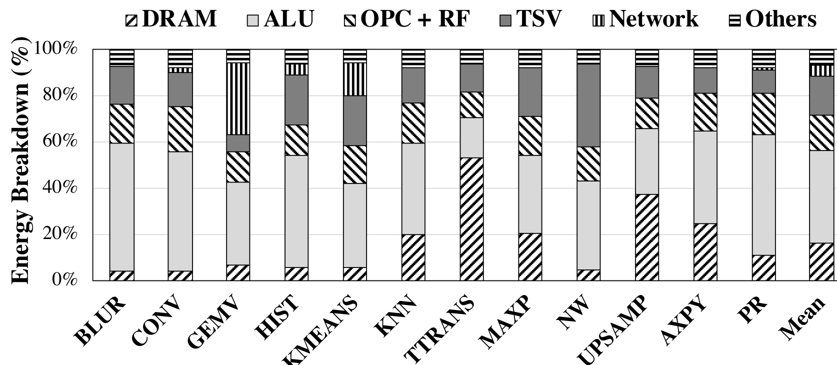


Figure 6.9: MPU energy breakdown.

The energy reduction mainly comes from the reduction of expensive data movement compared with the GPU, since MPU has a much shorter and simpler data path to access a core's local DRAM banks. Compared with the complex data path components in the GPU, where the data needs to travel through the TSVs inside the HBM, off-chip links, L2 cache, crossbar network, and then L1 cache to the local register file, the MPU directly offloads the instruction to the DRAM dies to transfer data between the near-bank register file and the DRAM banks. Also, we observe that for each benchmark the energy reduction in Fig.6.8 is approximately proportional to the speedup in Fig.6.7 (1). This is because MPU's increased bank-level bandwidth is a result of near-bank data access, which also contributes to the reduction of data movement energy.

In order to further analyze the energy consumption, we provide a detailed energy breakdown in Fig.6.9. We discover that most of the energy in MPU (92.94%) is spent on computation (ALU consumes 39.82%), data access (31.90%), and data movement (21.22%). The data access energy contains local register file access (operand collectors (OPC) and register file (RF) consume 15.47%) and DRAM accesses (16.42%). For data movement, the energy spent on remote data movement (Network consumes 4.43%) is significantly smaller than the local data movement (TSV consumes 16.79%). This well explains the data movement saving advantages of MPU compared to GPU to achieve

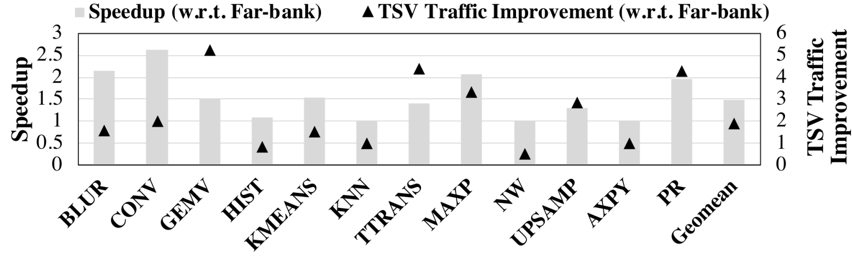


Figure 6.10: Comparison of near-bank / far-bank smem.

great energy reduction.

Thermal Analysis. MPU's peak power is $83W$ per processor considering both DRAM dies and the base logic die, and the peak power density is $552mW/mm^2$. The normal operating temperature for HBM2 DRAM dies is $105°C$ [12], and we conservatively assume the DRAM dies in our case operates under $85°C$. A prior study on 3D PIM thermal analysis [115] shows that active cooling solutions can effectively satisfy this thermal constraint ($85°C$). Both commodity-server active cooling solution [116] (peak power density allowed: $706mW/mm^2$) and high-end-server active cooling solution [117] (peak power density allowed: $1214mW/mm^2$)) can be used.

6.4.3 Architecture Analysis

Shared memory optimization. To understand the benefit of our near-bank shared memory, Fig.6.10 shows performance results compared with placing the shared memory on the base logic die, denoted as far-bank shared memory. In the same figure, we also plot TSV traffic improvement of near-bank shared memory design w.r.t. far-bank shared memory design. On average, near-bank shared memory design achieves 1.48× speedup and 1.89× TSV traffic improvement compared with far-bank shared memory design. The performance benefits of near-bank shared memory come from the extensive use of shared memory. If the shared memory location is far-bank, the contents of near-bank registers need to be brought down to the base logic die for the inter-thread communication through

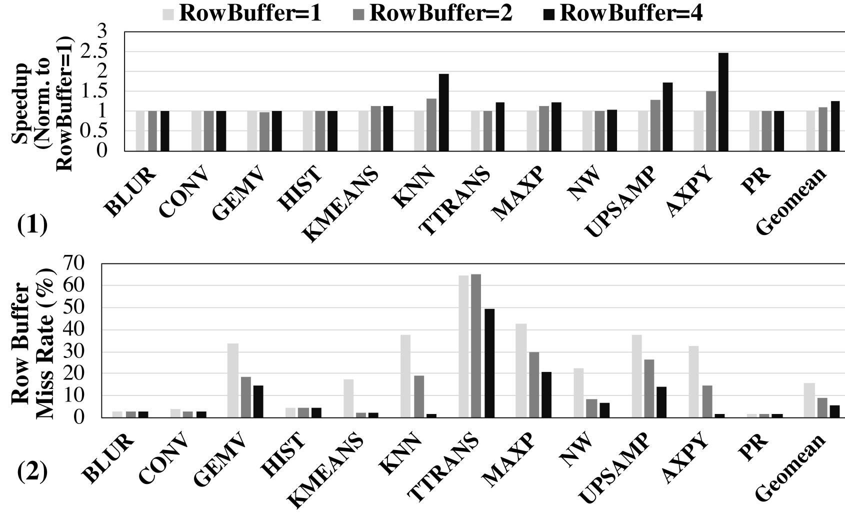


Figure 6.11: Comparison of the number of activated row-buffers on (1) performance and (2) row-buffer miss rate

shared memory. This will create a lot of register movement traffic and congest the TSVs. For the near-bank shared memory design, the default locations of value registers for *ld/st.global* and *ld/st.shared* are all near-bank. Thus less register movement will be involved, easing the bandwidth pressure on the TSVs. However, since the number of instructions offloaded to NBUs also rises, this may increase TSV traffic, as we observe that for some workloads with speedup larger than 1, the TSV traffic improvement may be slightly less than 1 (HIST, NW). For workloads that do not use shared memory, both the performance and TSV traffic are identical to the location of shared memory.

Multiple activated row-buffers analysis. To understand the benefits of multiple activated row-buffers, we compare the performance of all workloads running on MPU with different numbers of activated row-buffers. Fig.6.11 shows such performance comparisons where the speedup is normalized to a single row-buffer. As shown in the Fig.6.11 (1), the speedup numbers are 1.10× and 1.25× when we increase the number of activated row-buffers to 2 and 4, respectively. The row-buffer miss rate in Fig.6.11 (2) indicates that as we increase activated row-buffer numbers to 2 and 4, the miss rate reduces from 15.60%

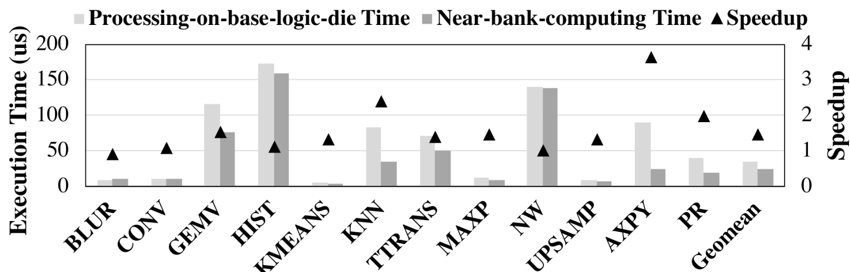


Figure 6.12: Execution time and speedup comparison with the processing-on-base–logic-die solution.

to 9.20% and 5.45%, respectively. Because more activated row-buffers can effectively reduce the row-buffer ping-ping effect in the dynamic scheduling of warps, increasing the number of activated row-buffers effectively reduces average DRAM access latency to improve end-to-end time. Especially, we observe that KNN, UPSAMP, and AXPY significantly benefit from the increased number of activated row-buffers due to severe ping-pong effects on a single row-buffer.

Comparison with processing-on-base-logic-die (PonB) solution. We compare MPU with the state-of-the-art general purpose near-data SIMT processors by placing all compute logic on the base logic die, denoted as PonB. The end-to-end execution time shown in Fig.6.12 demonstrates that on average MPU achieves 1.46× speedup up compared with the PonB solution. This performance improvement is contributed by a significant amount of instructions offloaded for near-bank computations. This reduces data movements on the TSVs which have a much lower bandwidth than bank-level memory bandwidth.

6.5 Summary

In this chapter, we propose MPU (Memory-centric Processing Unit), a SIMT processor based on 3D-stacking near-bank computing architecture. First, we develop a hybrid

pipeline where only a small number of hardware components are added on the DRAM dies and the instructions can be offloaded for near-bank computing. Second, we explore two architectural optimizations for the SIMT programming model, introducing a near-bank shared memory design to reduce data movements, and multiple activated row-buffers designs to increase bandwidth utilization. Third, we present an end-to-end compilation flow for MPU based on CUDA with a backend optimization to annotate the location of instructions as either near-bank or base logic die through the static analysis of programs. The end-to-end evaluation results of MPU on a set of representative benchmarks demonstrate 3.46× speedup and 2.57× energy reduction compared with an NVIDIA Tesla V100 GPU. We further conduct studies to show the performance improvement of MPU over prior 3D-stacking processors and identify the benefits of MPU's software and hardware optimizations.

Chapter 7

NMTSim: Transaction-Command based Simulator for New Memory Technology Devices

This chapter aims to establish a simulation infrastructure for architectural level explorations in emerging non-volatile memory technologies, by using a novel transaction-command based protocol called NVDIMM-P [24]. Unlike DDR timing which assumes synchronous media responses, NVDIMM-P allows asynchronous media activities through a transaction handshaking mechanism. For example, after receiving an $XREAD$ (transaction READ) command, the media controller can respond to the host (RD_RDY) after a non-deterministic time. However, existing memory simulators are either unable to support the novel transaction features in NVDIMM-P, or confined to a limited media scope. Previous DRAM simulators, including DRAMSim2 [42], NVMain2 [43], and Ramulator [44], only employ deterministic DDR timing protocols. Significant modification efforts are required to add handshaking and transaction handling logic in the complex scheduling unit of memory controller. Also, the passive memory module needs to add

extensive new functionalities to become a media controller that can independently process host-issued commands. Previous NVDIMM simulators (e.g., FlashDIMMSim [28]) are customized for traditional flash media with block-granularity. While emerging NVMs have demonstrated better performance and byte-addressability, it is more promising to explore them as memory media for NVDIMM.

In this chapter, we introduce NMTSim, a transaction-based (NVDIMM-P compatible) and cycle accurate memory simulator for new memory technology devices. To support transaction semantics, NMTSim includes a new memory controller with queuing structures, transaction handling logic and a command issuing unit. For the media controller, NMTSim uses a modified DRAMSim2 [42] simulator as the DRAM/NVM back-end, adapts DDR4 timing parameters to simulate emerging NVM devices, and adds a command scheduling unit and corresponding transaction functionalities. We validate NMTSim's simulation accuracy using real hardware measurements from Intel Optane memory [23], which uses a proprietary transaction command protocol for 3D Xpoint memory.

NMTSim also incorporates two architectural level optimizations to reduce latency overhead introduced by transaction commands. The first optimization grants $SEND$ command higher priority than $XREAD$ command, and can significantly reduce access latency under high host request bandwidth. The second optimization enables early host notification from media controller, and can save tRL latency for all host request bandwidth. After applying these optimizations, we thoroughly compare the performance of NVDIMM-P and DDR4 using DRAM and NVM with synthetic benchmarks under different read/write ratios.

The main contributions of this chapter are summarized as follows:

1. We propose NMTSim, a transaction-command based and cycle accurate simulator

for new memory technologies. We show the simulation framework of NMTSim and verify it using Intel Optane memory [23].

2. We incorporate a command issue optimization and an early notification functionality for transaction-command in NMTSim, and demonstrate the latency improvements of these two schemes.

3. We evaluate NMTSim using synthetic benchmarks. Evaluation results on both DRAM and NVM devices show slight latency overhead of NVDIMM-P compared with DDR4.

7.1 Simulator Design

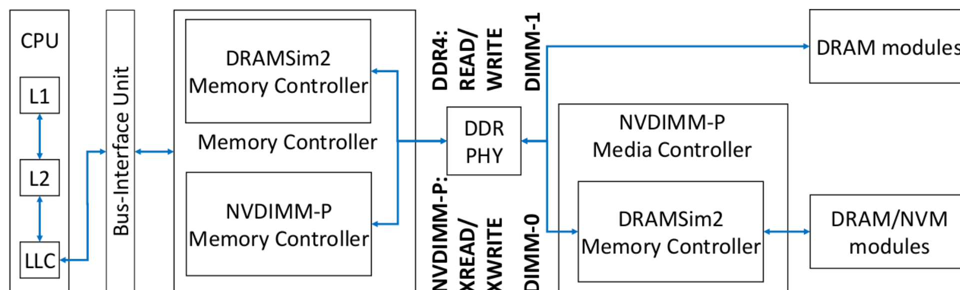


Figure 7.1: **NMTSim's high-level overview.**

We first introduce the high-level overview of NMTSim in Sec. 7.1.1. Then, in Sec. 7.1.2 we explain the detailed simulator components, data paths, and control paths to support transaction commands. Last, Sec. 7.1.3 discusses the new timing parameters to support emerging NVM.

7.1.1 High-level Overview

NMTSim consists of a front-end processor interface, a memory controller, an interface bus, and a media controller with DRAM/NVM modules as shown in Fig. 7.1.

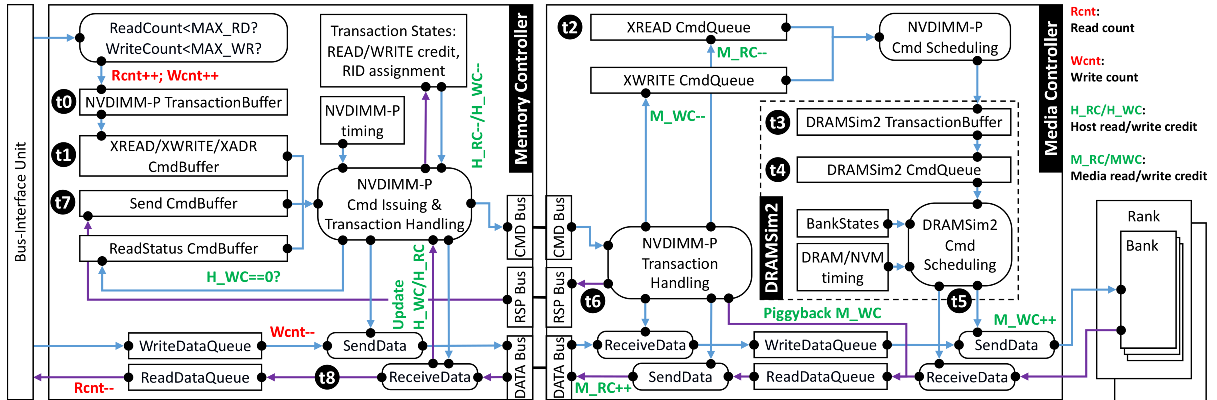


Figure 7.2: **NMTSim's block diagram and data/control paths. The circled numbers represent time stamps in a XREAD command in Fig. 7.3**

In trace-based simulation mode, the front-end processor interface is compatible with DRAMSim2, and can be easily modified to support any new processors. The memory controller supports two memory protocols: DDR4 protocol which is implemented by a modified DRAMSim2 memory controller, and NVDIMM-P protocol which is realized by the proposed NVDIMM-P memory controller. The two protocols coexist on the same memory channel, on which DDR4 uses normal READ/WRITE commands for DIMM-0, and NVDIMM-P uses XREAD/XWRITE commands for DIMM-1. For NVDIMM-P protocol, an additional media controller is included to support transaction semantics. In full-system simulation mode, NMTSim can be integrated into existing architecture simulators (e.g., GEM5) to receive and respond host memory requests.

7.1.2 Support Transaction Commands Simulation

To support transaction commands, a new memory controller with NVDIMM-P queuing structures, command issuing logic, and transaction handling logic are proposed as shown in Fig. 7.2. The NVDIMM-P transaction buffer will store accepted host memory requests, and the command buffer will store translated NVDIMM-P commands. The NVDIMM-P command issuing unit will decide which command can be sent to the media

controller according to transaction states and command priority, and the transaction handling logic will arbitrate command and data traffic to avoid conflicts on the bus according to NVDIMM-P timing. An XREAD (XWRITE) command can only be issued if there is available read (write) credit, and the corresponding credit will decrement by one after a command is sent. The read credit will increment by one if memory controller receives a returned read data packet. Released write credit will be piggybacked through returned read data packet. However, if there is no enough write credit, memory controller can compose a READ_STATUS command and explicitly ask the media controller for more write credit. After receiving a RD_RDY signal from RSP bus, a SEND command will be prepared and issued by the command issuing logic.

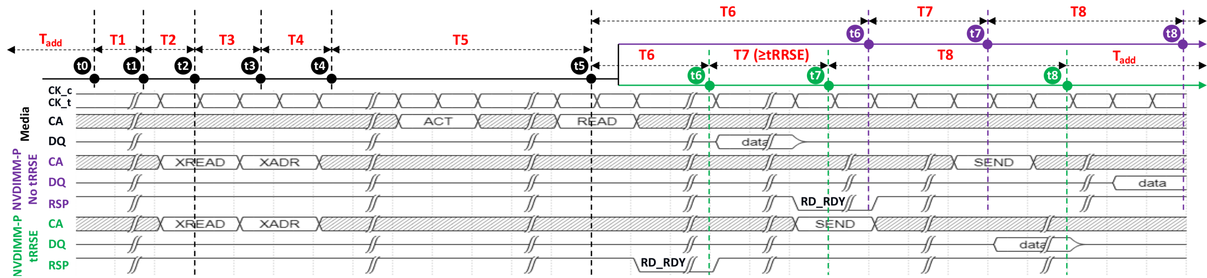


Figure 7.3: **XREAD latency breakdown and illustration of early notification mode. Timing terminologies are detailed in Table. 7.1.2.**

From the media controller side, the received commands will first be interpreted and added to command queues for further scheduling. At the same time, the corresponding credit will decrement by one. Since NVDIMM-P supports out-of-order transactions, the NVDIMM-P command scheduling logic will select a command without data hazard for execution. To optimize read latency, an XREAD command has higher scheduling priorities than an XWRITE command. The write credit will increment after the XWRITE command is consumed by DDRAMSim2. After an XREAD command finishes execution by DRAMSim2, the NVDIMM-P transaction handling logic will inform the memory controller by signaling the RSP bus. Then, the media controller will return read data

Parameter	Definition
$T1$	Memory Controller Transaction Buffer Latency
$T2$	Memory Controller Command Buffer Latency
$T3$	Media Controller Command Queue Latency
$T4$	DRAMSim2 Transaction Buffer Latency
$T5$	DRAMSim2 Command Queue Latency
$T6$	Media Bank Access Latency
$T7$	Memory Controller Response Latency
$T8$	Memory Controller SEND Latency
T_{add}	Cache Access Handling and Bus-interface Unit Latency

Table 7.1: **Latency terminologies for an XREAD command.**

packets and increment read credit after receiving SEND command.

By default, the NVDIMM-P protocol enables 256 maximum read/write transaction commands. However, we find that for memory setting with small bank number, this large command count will cause unnecessary command queuing delay. To reduce this delay, we add maximum read/write count to constrain host issued requests. The maximum read/write count is set to match the total bank number per rank, assuming we use a per-rank queuing structure in DRAMSim2.

7.1.3 Support Emerging NVM Timing Simulation

In order to support emerging NVM timing simulation using DRAM compatible timing parameters, we propose the following changes. First, since NVM is non-volatile in nature, for NVM mode DRAM REFRESH command is disabled and all corresponding timing parameters are set to zero. Second, we change the t_{RCD} to t_{RCD_R} for read and t_{RCD_W} for write considering the asymmetric read/write behavior for NVM. Since NVM write does not require row activation, we set $t_{RCD_W} = 0$ for a full page write. Similarly, we change the t_{RP} to t_{RP_R} for read and t_{RP_W} for write. Since NVM read does not require row precharge, we set $t_{RP_R} = 0$. Third, to account for partial page write overhead, we

Parameter	**NVM**	**DRAM**	**Parameter**	**NVM**	**DRAM**
Channel	1	1	t_{CK}	0.75ns (2666MHz)	
Rank	1	2	t_{RCD_R}	192ns	14.3ns
BankGroup	2	4	t_{RCD_W}	0ns	14.3ns
Bank	4	4	t_{RP_R}	0ns	13.5ns
Row	2^{26}	2^{16}	t_{RP_W}	489ns	13.5ns
Column	2^5	2^{10}	t_{RMW}	t_{RCD_R}+t_{RP_W}	-
DeviceWidth	8	8	t_{REF}	-	7800ns
Capacity	128GB	16GB	t_{RFC}	-	260.3ns
CmdQueueStruct	Per Rank		CmdQueueDepth	16	32
RowBufferPolicy	Close-Page		Scheduling	Rank-Bank Round Robin	

Table 7.2: **NMTSim Architecture and Timing Configuration.**

add a new timing parameter t_{RMW} (Read-Modify-Write). If write request number to the same row is smaller than the page size, then an additional t_{RMW} latency overhead is induced to read the unmodified portion of the page.

The t_{RCD_R} and t_{RP_W} parameters should be derived based on actual NVM measurement results. First, a sequential benchmark should be used to stress the actual NVM hardware, and the maximum achievable read and write bandwidth can be discovered. Then, using the same benchmark and assuming the same configuration as the baseline NVM hardware, we can sweep t_{RCD_R}/t_{RP_W} parameter space to acquire the maximum achievable read/write bandwidth curve. In the end, we can select the t_{RCD_R}/t_{RP_W} value if the corresponding read/write bandwidth matches hardware measurements.

7.2 Optimization

Before introducing the optimizations, we illustrate the latency breakdown for an XREAD command in Fig. 7.3 and related timing terminologies in Table. 7.1.2. Since the load-to-use latency measures the request-to-service interval from a host CPU, we also include a constant additional latency [137] to count for host cache access handling and

bus-interface unit delay, which are extracted from the CPU baseline in Sec. 7.3.1. Specifically, the T_{add} represents the latency between the memory controller and the host load-store interface. All analysis in this section uses synthetic random benchmarks ($64Byte$ access granularity) based on the hardware configuration of Table. 7.2.

7.2.1 Command Issue Optimization

The memory controller command issuing logic needs to decide the priority between SEND and XREAD for optimal performance when the command bus is congested. The first policy ($p1$) grants XREAD with higher priority to reduce memory controller side queuing latency and delays SEND. In contrast, the second policy ($p2$) gives SEND the higher priority which shortens the latency of already-issued XREAD but blocks new XREAD. We plot the latency-bandwidth graph of the two policies for DRAM and NVM in Fig. 7.4 (a-1)/(b-1), respectively. We can observe that for DRAM, $p2$ can significantly reduce access latency compared with $p1$ for high memory bandwidth cases. Further latency breakdown in Fig. 7.4 (a-2) finds that the latency increase in $p1$ is mainly incurred by $T7$, which is the memory controller response queuing latency for SEND. This validates the benefits of $p2$. However, we also observe that $p2$ has larger $T5$ than $p1$, which corresponds to DRAMSim2 command queuing latency. Since we add maximum read count constraints in the simulator, earlier completion of XREAD ($p2$) will encourage the memory controller to accept and send more XREAD to media controller, which in turn increases the queuing delay ($T5$). For NVM, these two policies show little difference in terms of latency. This is because NVM has much lower sustained bandwidth compared with DRAM, so the command bus will not be congested to issue SEND and XREAD commands, as shown in Fig. 7.4 (b-2).

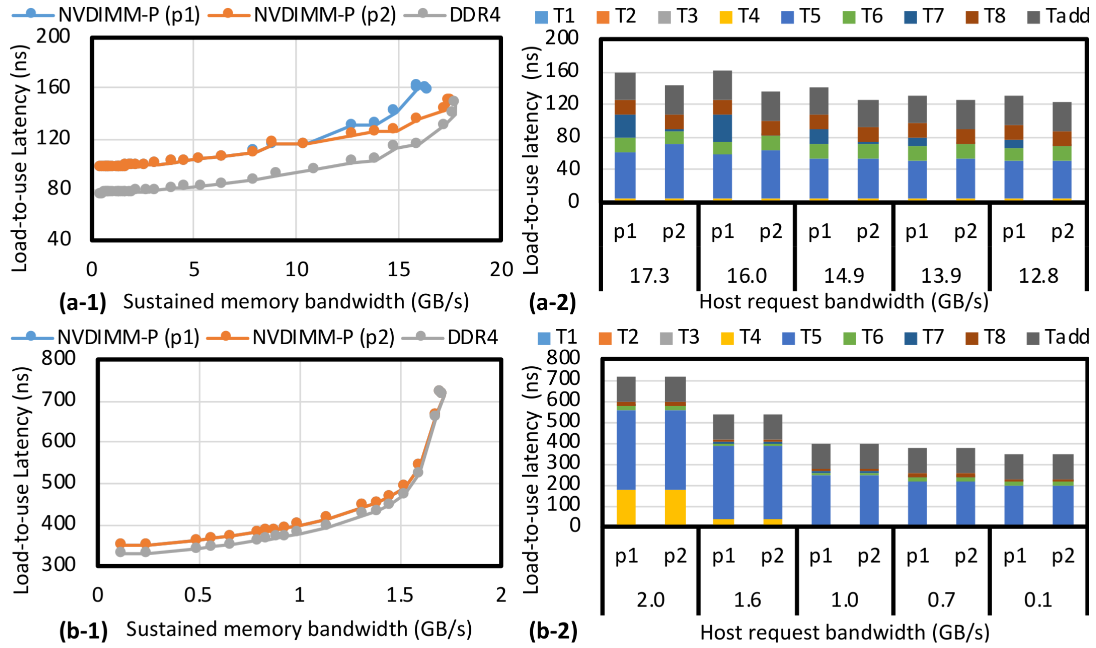


Figure 7.4: **The comparison of two command issuing policies ($p1$,$p2$). (a) DRAM media. (b) NVM media. Left: bandwidth-latency graph. Right: latency breakdown.**

7.2.2 Early Notification

NVDIMM-P supports early host notification to reduce response latency. Fig. 7.3 illustrates the timing differences of disabling (No t_{RRSE}) and enabling (t_{RRSE}) early notification. When early notification is not supported, media controller needs to wait until the media returns read data to inform host (RD_RDY). When early notification is enabled, media controller can inform the host (RD_RDY) in advance, and memory controller needs to wait for a certain time interval (t_{RRSE}) after receiving RD_RDY to issue $SEND$. The choice of early notification timing depends on implementation of memory controller and the design of media controller. Here, for optimistic estimation, we assume host response immediately after receiving early notification and media controller signals RD_RDY at the same time it issues $READ$ to the media. After including the optimization in Sec. 7.2.1, we plot the bandwidth-latency graph of disabling and enabling

early notification for DRAM and NVM in Fig. 7.5 (a-1)/(b-1). We find that for low to medium memory bandwidth, enabling early notification can greatly reduce latency ($\sim 15ns$) for both NVM and DRAM. However, for high memory bandwidth, the latency gap closes between $e1$ and $e2$. We further investigate the latency breakdown in Fig. 7.5 (a-2)/(b-2). First, we observe the latency reduction of $e2$ is mainly contributed by reducing $T6$, since the media bank access latency is overlapped by t_{SEND} in early notification case. Second, we observe that as request bandwidth increases, $T5$ increases rapidly for $e2$ until the latency saturates. Under the constraints of maximum read count, $e1$ require less XREAD to saturates the latency, so the extra XREAD for $e2$ will stack in DRAMSim2's command queue and add to the queuing latency.

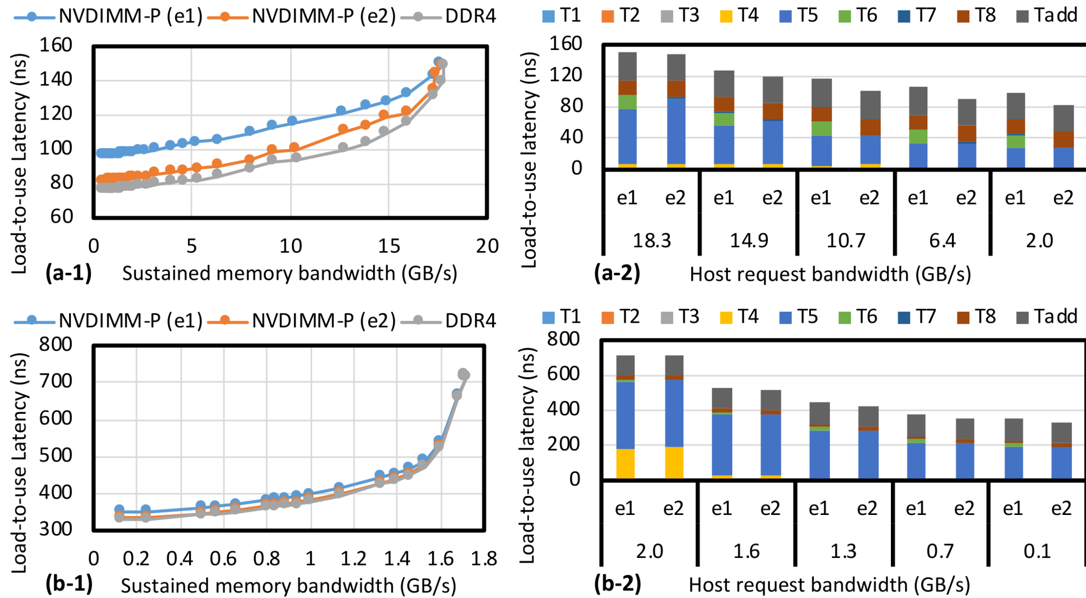


Figure 7.5: **The comparison between disabling($e1$)/enabling($e2$) early notification. (a) DRAM media. (b) NVM media. Left: bandwidth-latency graph. Right: latency breakdown.**

7.3 Experiment

7.3.1 Validation

For the baseline hardware configuration [23], we use an Intel Xeon Platinum 8280L processor with a single memory channel, which contains a 128GB DCPMM Intel Optane memory (App Direct mode) and a 16GB RDIMM Samsung (M393A2K43BB1) DRAM. The baseline uses DDR-T, which is a proprietary transaction protocol of Intel. In comparison, NMTSim assumes NVDIMM-P transaction protocol for both DRAM and NVM media and uses the configuration as detailed in Table. 7.2. The optimizations from Sec. 7.2.1 and Sec. 7.2.2 are also incorporated in NMTSim. We extract the additional latency (T_{add}) of $\sim 35ns$ for RDIMM+DRAM device and $120ns$ for DCPMM+Optane device. For NVM simulation, to match the maximum read bandwidth ($8.3GB/s$) and maximum write bandwidth ($2.2GB/s$), t_{RP_W} is set to 192ns and t_{RP_W} is set to 489ns. We plot the bandwidth-latency graph and use all_read and 2reads_1write random access benchmarks (64Byte granularity) to validate the results of NMTSim with baseline hardware measurements in Fig. 7.6. The validation results show that on average NMTSim has 2.8% and 3.4% latency error compared with the baseline.

7.3.2 Evaluation on Synthetic Benchmarks

After applying the optimizations from Sec. 7.2, we use random access benchmarks (64Byte granularity) with various read/write ratios ($R = 0.8/0.5/0.1$) to characterize the performance of NMTSim using combinations of NVDIMM-P and DDR4 protocols with different media types, as shown in Fig. 7.7. For both DRAM and NVM media, NVDIMM-P adds additional transaction latency with respect to DDR4. For DRAM, the additional latency varies little as memory bandwidth changes. However, for NVM, the

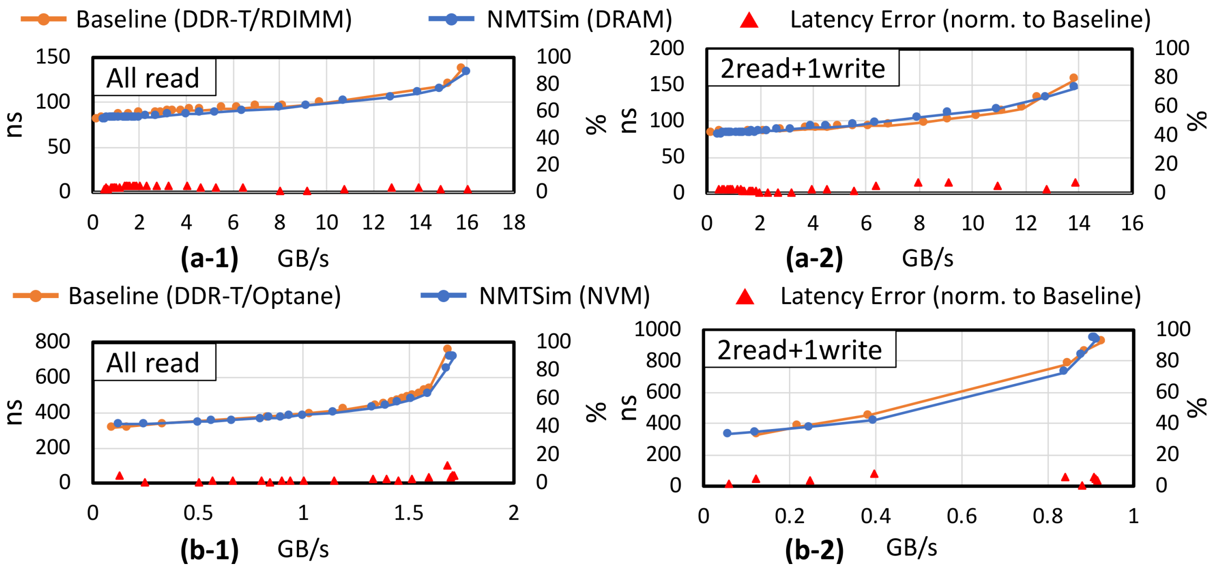


Figure 7.6: **Validation of NMTSim with Intel Optane.**

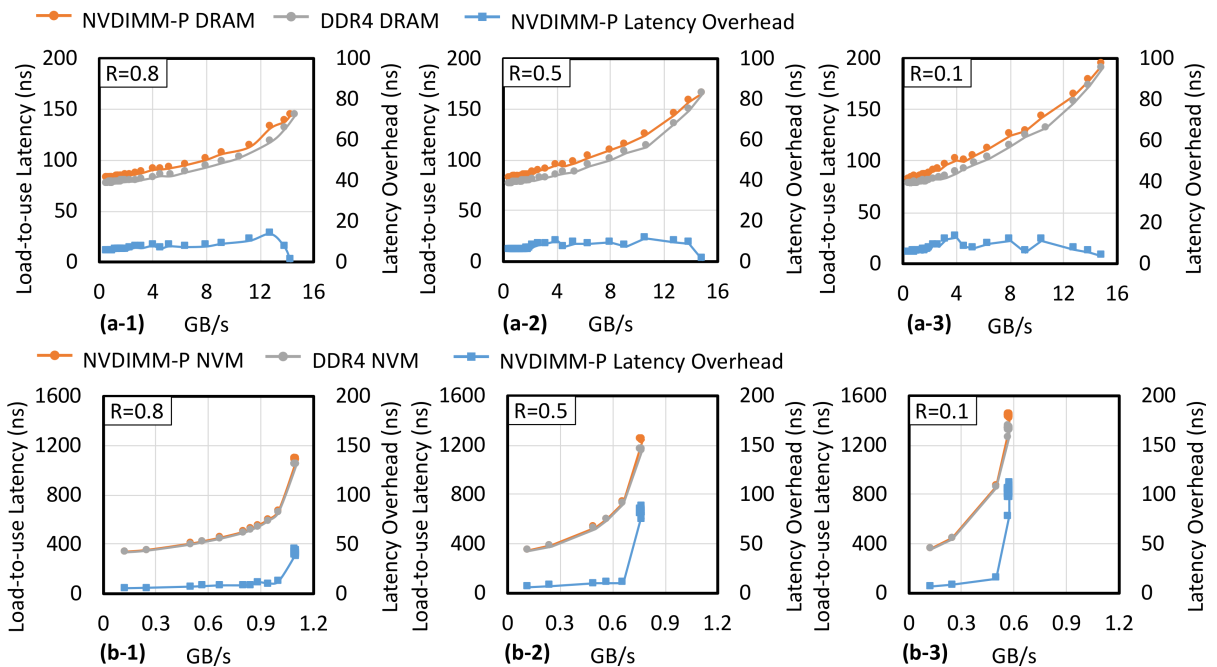


Figure 7.7: **Characterize NMTSim using different Read/Write ratio (R). (a) DRAM media. (b) NVM media.**

additional latency tends to increase as read/write ratio is low. This is because NVM's write performance is much worse than its read performance. As read/write ratio goes low, NVM writes will interfere read request more significantly compared with DRAM. In addition, we observe that NVM's bandwidth-latency curve bends more obvious than DRAM's as read/write ratio becomes lower, which is also explained by NVM's asymmetric read/write performance.

7.3.3 Evaluation on SPEC 2017

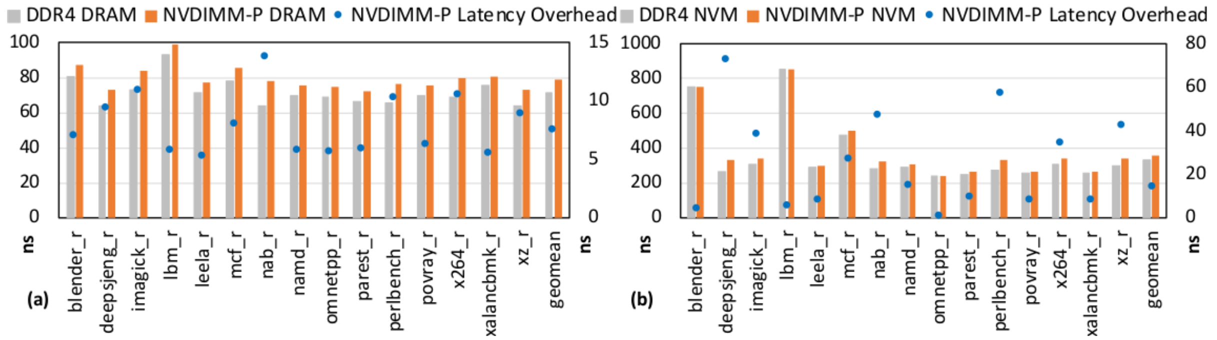


Figure 7.8: **Latency comparison of NVDIMM-P and DDR4 protocol on SPEC 2017 benchmark suite. (a) DRAM media. (b) NVM media.**

We also evaluate NMTSim using selected SPEC 2017 benchmarks as shown in Fig. 7.8. We use the GEM5 simulator (X86 architecture) to generate real-time memory traces and feed them to NMTSim. For each benchmark, we show its latency number using different combinations of memory protocol and media types. The evaluation results show that on average NVDIMM-P adds 7.6ns and 14.76ns latency overheads for DRAM and NVM media, respectively.

7.4 Summary

In this chapter, we present NMTSim, a transaction-command based and cycle accurate simulator for new memory technology. NMTSim introduces a new memory controller with transaction handling and command issuing logic. To enable simulation for emerging NVM using DDR4 standard, we propose some new NVM timing parameters and incorporated them into DRAMSim2 [42]. Furthermore, DRAMSim2 is augmented with transaction handling and command scheduling logic to be the backend for the media controller. In addition, NMTSim incorporates an optimized transaction command issuing policy and an early notification mode to optimize access latency. We verify NMTSim using Intel Optane memory [23], and characterize its performance using synthetic benchmarks with different read/write ratio.

Chapter 8

Summary

The memory bandwidth scaling and the memory capacity scaling are becoming serious challenges to continuously improve performance of the data processing systems. Memory-centric architecture is a promising approach to tackle these problems, and this dissertation studies analog process-in-memory architecture, digital process-near-memory architecture, and enhanced memory architecture related to this direction.

First, we explore optimizing the peripheral logic overheads in analog process-in-memory architecture. We propose a highly optimized memristor crossbar architecture to implement quantized DNN which supports flexible configurations. The architecture supports many non-linear functions at the tile level with a crossbar-based universal approximation engine. Design space explorations are carried out to study the impact of bit precision and hidden layers on the approximation accuracy. We propose a hybrid-precision ADC design to reduce the power and area overheads of ADCs. The memristor crossbars in the proposed architecture are divided into high-precision and reduced-precision configurations with trade-offs between accuracy and energy efficiency. DNN applications that tolerate noise in low-order bits could be computed with less precise memristor crossbars for energy savings. For models requiring high precision, we provide an encoding scheme

to configure the reduced-precision crossbars for high-precision computation, offering great flexibility for application mapping.

Second, we study how to efficiently support memory-intensive deep learning training tasks by integrating lightweight units in near-bank architecture. We propose a near-bank architecture, DLUX, for DNN training acceleration with both high performance and low hardware overhead. We demonstrate the DLUX design, with the highlight of the in-DRAM hierarchical LUT for high-performance FP computing, efficient communication using shared data bus and lightweight support for data transformation. We present the DLUX software design. The intra-layer mapping/scheduling improves utilization, concurrency while minimizing data movement, and the inter-layer data transformation ensures layout consistency in dataflow processing. We evaluate DLUX and compare with the Tesla V100 GPU. The results shows DLUX provides on average 6.3× end-to-end speedup and 42× energy-efficiency improvement on representative data center training workloads.

Then, we explore how to support domain programmable near-bank accelerators. We design a standalone programmable accelerator, iPIM, using 3D-stacking near-bank architecture for image processing applications. By using a decoupled control-execution architecture, iPIM supports programmability with small area overhead per DRAM die ($\sim$ 10.71%). We propose SIMB (Single-Instruction-Multiple-Bank) ISA which enables flexible computation, data access, and communication patterns to support various pipeline stages in image processing applications. We develop an end-to-end compilation flow based on Halide with novel iPIM schedules and various iPIM backend optimizations including register allocation, instruction reordering, and memory-order enforcement. Evaluation results of representative image processing benchmarks, including single stage and heterogeneous multi-stage pipelines, show that iPIM design together with backend optimizations can achieve 11.02× speedup and 79.49% energy saving on average over an NVIDIA

Tesla V100 GPU. The backend optimizations improve 3.19× performance compared with the naïve baseline.

Next, we further extend the application scenario to support general purpose parallel computing programs on near-bank architecture. We design a near-bank SIMT processor using a hybrid pipeline with an instruction offloading mechanism. By integrating lightweight hardware components on the DRAM die, MPU achieves a small area overhead (20.62%) for general purpose processing. We propose two architectural optimizations for the SIMT model, including the near-bank shared memory to reduced data movement and multiple activated row-buffers to alleviate ping-pong effects in the dynamic warp scheduling. We develop an end-to-end compilation flow supporting CUDA programs on MPU and a novel backend optimization annotating the locations of registers and instructions. Evaluation results of representative data-intensive workloads show that MPU with all optimizations achieves 3.46× speedup and 2.57× energy reduction on average over an NVIDIA Tesla V100 GPU.

In the end, we study how to support various emerging memory technologies by proposing a novel simulation framework. We propose NMTSim, a transaction-command based and cycle accurate simulator for new memory technologies. We show the simulation framework of NMTSim and verify it using Intel Optane memory. We incorporate a command issue optimization and an early notification functionality for transaction-command in NMTSim, and demonstrate the latency improvements of these two schemes. We evaluate NMTSim using synthetic benchmarks. Evaluation results on both DRAM and NVM devices show slight latency overhead of NVDIMM-P compared with DDR4.

We hope the work in this thesis would be useful and inspirational for memory-related system designs and memory-centric accelerator designs.

Bibliography

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, *Edge computing: Vision and challenges*, *IEEE internet of things journal* **3** (2016), no. 5 637–646.

[2] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, *Nature* **521** (2015), no. 7553 436–444.

[3] M. M. Waldrop, *The chips are down for moore's law*, *Nature News* **530** (2016), no. 7589 144.

[4] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, *et. al.*, *Scaling the power wall: a path to exascale*, in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841, IEEE, 2014.

[5] K. Rupp, *40 years of microprocessor trend data*, in *GitHub*, 2018.

[6] I. Insight, *Transistor count trends continue to track with moore's law*, *The McClean Report* (2020).

[7] M. Nemirovsky and D. M. Tullsen, *Multithreading architecture*, *Synthesis Lectures on Computer Architecture* **8** (2013), no. 1 1–109.

[8] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, *Scaling the bandwidth wall: challenges in and avenues for cmp scaling*, *ACM SIGARCH Computer Architecture News* **37** (2009), no. 3 371–382.

[9] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, *Disaggregated memory for expansion and sharing in blade servers*, *ACM SIGARCH computer architecture news* **37** (2009), no. 3 267–278.

[10] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, *et. al.*, *Cascade lake: Next generation intel xeon scalable processor*, *IEEE Micro* **39** (2019), no. 2 29–36.

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

[12] K. Sohn, W. Yun, R. Oh, C. Oh, S. Seo, M. Park, D. Shin, W. Jung, S. Shin, J. Ryu, H. Yu, J. Jung, K. Nam, S. Choi, J. Lee, U. Kang, Y. Sohn, J. Choi, C. Kim, S. Jang, and G. Jin, *A 1.2 v 20 nm 307 gb/s hbm dram with at-speed wafer-level io test scheme and adaptive refresh considering temperature distribution*, *IEEE Journal of Solid-State Circuits* **52** (2017), no. 1 250–260.

[13] M. Horowitz, *1.1 computing's energy problem (and what we can do about it)*, in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[14] P. Gu, X. Xie, S. Li, D. Niu, H. Zheng, K. T. Malladi, and Y. Xie, *Dlux: a lut-based near-bank accelerator for data center deep learning training workloads*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[15] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, *Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 587–599, 2019.

[16] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, *Near-data processing: Insights from a micro-46 workshop*, *IEEE Micro* **34** (2014), no. 4 36–42.

[17] L. Xia, P. Gu, B. Li, T. Tang, X. Yin, W. Huangfu, S. Yu, Y. Cao, Y. Wang, and H. Yang, *Technological exploration of rram crossbar array for matrix-vector multiplication*, *Journal of Computer Science and Technology* **31** (2016), no. 1 3–19.

[18] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, *ipim: Programmable in-memory image processing accelerator using near-bank architecture*, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 804–817, IEEE, 2020.

[19] C. D. Kersey, H. Kim, and S. Yalamanchili, *Lightweight simt core designs for intelligent 3d stacked dram*, in *Proceedings of the International Symposium on Memory Systems*, pp. 49–59, 2017.

[20] J. S. S. T. Association *et. al.*, *Low power double data rate (lpddr) sdram standard, JESD209B, JEDEC Standard* (2010).

[21] J. Standard, *Double data rate (ddr) sdram specification*, *JEDEC Solid State Technology Assoc* (2005).

[22] J. Standard, *Graphics double data rate (gddr5x) sgram standard*, *JESD212C, February* (2016).

[23] B. Tristian, L. Travis, and C. Jamie, *Analyzing the performance of intel optane dc persistent memory in app direct mode in lenovo thinksystem servers*, *Lenovo Press* (2019).

[24] B. Gervasi, D. Designs, and J. Hinkle, *Overcoming system memory challenges with persistent memory and nvdimm-p*, in *JEDEC Server Forum*, vol. 2017, 2017.

[25] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, *Phase change memory*, *Proceedings of the IEEE* **98** (2010), no. 12 2201–2227.

[26] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, *Evaluating stt-ram as an energy-efficient main memory alternative*, in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, IEEE, 2013.

[27] H.-S. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, *Metal–oxide rram*, *Proceedings of the IEEE* **100** (2012), no. 6 1951–1970.

[28] P. Enns, *FlashDIMMSim: A reasonably accurate flash DIMM simulator*.

[29] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, *Rram-based analog approximate computing*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34** (2015), no. 12 1905–1917.

[30] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, *Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars*, in *Proc. ISCA*, 2016.

[31] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, *Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory*, in *Proceedings of ISCA*, vol. 43, 2016.

[32] B. L. Y. W. H. Y. Tianqi Tang, Lixue Xia, *Binary convolutional neural network on rram*, in *Asia and South Pacific Design Automation Conference*, 2017.

[33] P. M. Kogge, *Execube-a new architecture for scaleable mpps*, in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1, pp. 77–84, IEEE, 1994.

[34] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, *Computational ram: Implementing processors in memory*, *IEEE Design & Test of Computers* **16** (1999), no. 1 32–41.

[35] M. Gokhale, B. Holmes, and K. Iobst, *Processing in memory: The terasys massively parallel pim array*, *Computer* **28** (1995), no. 4 23–31.

[36] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, *ACM SIGARCH Computer Architecture News* **43** (2016), no. 3 105–117.

[37] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, *Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 380–392, IEEE, 2016.

[38] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, *Tetris: Scalable and efficient neural network acceleration with 3d memory*, in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 751–764, ACM, 2017.

[39] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, *Mcdram: Low latency and energy-efficient matrix computations in dram*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2018), no. 11 2613–2622.

[40] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, *In-dram near-data approximate acceleration for gpus*, in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, p. 34, ACM, 2018.

[41] S. Aga, N. Jayasena, and M. Ignatowski, *Co-ml: a case for collaborative ml acceleration using near-data processing*, in *Proceedings of the International Symposium on Memory Systems*, pp. 506–517, ACM, 2019.

[42] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, *Dramsim2: A cycle accurate memory system simulator*, *IEEE computer architecture letters* **10** (2011), no. 1 16–19.

[43] M. Poremba, T. Zhang, and Y. Xie, *Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems*, *IEEE Computer Architecture Letters* **14** (2015), no. 2 140–143.

[44] Y. Kim, W. Yang, and O. Mutlu, *Ramulator: A fast and extensible dram simulator*, *IEEE Computer architecture letters* **15** (2015), no. 1 45–49.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et. al.*, *Imagenet large scale visual recognition challenge*, *International Journal of Computer Vision* **115** (2015), no. 3 211–252.

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

[48] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, .

[49] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, *Yodann: An ultra-low power convolutional neural network accelerator based on binary weights*, in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 236–241, IEEE, 2016.

[50] M. Courbariaux, Y. Bengio, and J.-P. David, *Binaryconnect: Training deep neural networks with binary weights during propagations*, in *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

[51] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, *Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients*, *arXiv preprint arXiv:1606.06160* (2016).

[52] P.-Y. C. H. W. B. G. D. W. W. W. S. Yu, Z. Li and H. Qian, *Binary neural network with 16 mb rram macro chip for classification and online training*, 2016.

[53] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, *arXiv preprint arXiv:1502.03167* (2015).

[54] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, *Neural networks* **2** (1989), no. 5 359–366.

[55] J. Li, C.-I. Wu, S. C. Lewis, J. Morrish, T.-Y. Wang, R. Jordan, T. Maffitt, M. Breitwisch, A. Schrott, R. Cheek, *et. al.*, *A novel reconfigurable sensing scheme for variable level storage in phase change memory*, in *2011 3rd IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE, 2011.

[56] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, *Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0*, in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 3–14, IEEE, 2007.

[57] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, *Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication*, in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[58] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et. al.*, *Dadiannao: A machine-learning supercomputer*, in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE, 2014.

[59] G. Khodabandehloo, M. Mirhassani, and M. Ahmadi, *Analog implementation of a novel resistive-type sigmoidal neuron*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **20** (2011), no. 4 750–754.

[60] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, *et. al.*, *Freepdk: An open-source variation-aware design kit*, in *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*, pp. 173–174, IEEE, 2007.

[61] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, *A 3.1 mw 8b 1.2 gs/s single-channel asynchronous sar adc with alternate comparators for enhanced speed in 32 nm digital soi cmos*, *IEEE Journal of Solid-State Circuits* **48** (2013), no. 12 3049–3058.

[62] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn, *Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation adcs*, *IEEE Transactions on Circuits and Systems I: Regular Papers* **58** (2011), no. 8 1736–1748.

[63] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, *Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **31** (2012), no. 7 994–1007.

[64] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[65] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[66] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, *Deepface: Closing the gap to human-level performance in face verification*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708, 2014.

[67] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et. al.*, *Going deeper with embedded fpga platform for convolutional neural network*, in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, ACM, 2016.

[68] R. Berdan, T. Prodromakis, and C. Toumazou, *High precision analogue memristor state tuning*, *Electronics letters* **48** (2012), no. 18 1105–1107.

[69] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, *Processing-in-memory for energy-efficient neural network training: A heterogeneous approach*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 655–668, IEEE, 2018.

[70] D. Kim, T. Na, S. Yalamanchili, and S. Mukhopadhyay, *Deeptrain: A programmable embedded platform for training deep neural networks*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (Nov, 2018) 2360–2370.

[71] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, *Neurostream: Scalable and energy efficient deep learning with smart memory cubes*, *IEEE Transactions on Parallel and Distributed Systems* **29** (2017), no. 2 420–434.

[72] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, *A scalable near-memory architecture for training deep neural networks on large in-memory datasets*, *arXiv preprint arXiv:1803.04783* (2018).

[73] L. Xu, D. P. Zhang, and N. Jayasena, *Scaling deep learning on multiple in-memory processors*, in *Proceedings of the 3rd Workshop on Near-Data Processing*, 2015.

[74] Y.-B. Kim and T. W. Chen, *Assessing merged dram/logic technology*, *Integration* **27** (1999), no. 2 179–194.

[75] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, *Neural collaborative filtering*, in *Proceedings of the 26th international conference on world wide web*, pp. 173–182, 2017.

[76] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et. al.*, *Deep speech: Scaling up end-to-end speech recognition*, *arXiv preprint arXiv:1412.5567* (2014).

[77] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[78] A. M. Rush, S. Harvard, S. Chopra, and J. Weston, *A neural attention model for sentence summarization*, in *ACLWeb. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2017.

[79] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, *Skip-thought vectors*, in *Advances in neural information processing systems*, pp. 3294–3302, 2015.

[80] G. Toderici, D. Vincent, N. Johnston, S. Jin Hwang, D. Minnen, J. Shor, and M. Covell, *Full resolution image compression with recurrent neural networks*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5306–5314, 2017.

[81] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et. al.*, *Tensorflow: A system for large-scale machine learning*, in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[82] J. T. Pawlowski, *Hybrid memory cube (hmc)*, in *2011 IEEE Hot Chips 23 Symposium (HCS)*, pp. 1–24, IEEE, 2011.

[83] "Berkeley hardware floating-point units." `https://github.com/ucb-bar/berkeley-hardfloat`.

[84] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys*. Springer Science & Business Media, 2012.

[85] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, *Mcdram: Low latency and energy-efficient matrix computations in dram*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2018), no. 11 2613–2622.

[86] F. De Dinechin and B. Pasca, *Designing custom arithmetic data paths with flopoco*, *IEEE Design & Test of Computers* **28** (2011), no. 4 18–27.

[87] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, *A case for exploiting subarray-level parallelism (SALP) in DRAM*, *ACM SIGARCH Computer Architecture News* **40** (2012), no. 3 368–379.

[88] H. Consortium *et. al.*, *Hybrid memory cube specification 2.1*, *Retrieved from hybridmemorycube. org* (2013).

[89] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cudnn: Efficient primitives for deep learning*, *arXiv preprint arXiv:1410.0759* (2014).

[90] "MLPerf." `https://github.com/mlperf/training`.

[91] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, *Fathom: Reference workloads for modern deep learning methods*, in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, IEEE, 2016.

[92] X. Xie, X. Hu, P. Gu, S. Li, Y. Ji, and Y. Xie, *NNBench-x: Benchmarking and understanding neural network workloads for accelerator designs*, *IEEE Computer Architecture Letters* **18** (2019), no. 1 38–42.

[93] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, *Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory*, in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 33–38, EDA Consortium, 2012.

[94] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, *Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads*, in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 190–200, IEEE, 2014.

[95] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, *Scaledeep: A scalable compute architecture for learning and evaluating deep networks*, in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, vol. 45, pp. 13–26, ACM, 2017.

[96] S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, *Communications of the ACM* **52** (2009), no. 4 65–76.

[97] *NVIDIA Tesla V100 GPU Architecture*, 2018. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[98] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, *A case for intelligent ram*, *IEEE micro* **17** (1997), no. 2 34–44.

[99] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, *et. al.*, *The architecture of the diva processing-in-memory chip*, in *Proceedings of the 16th international conference on Supercomputing*, pp. 14–25, 2002.

[100] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, *Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines*, in *Acm Sigplan Notices*, vol. 48, pp. 519–530, ACM, 2013.

[101] J. Fung and S. Mann, *Using graphics devices in reverse: Gpu-based image processing and computer vision*, in *2008 IEEE international conference on multimedia and expo*, pp. 9–12, IEEE, 2008.

[102] M. D. M. F. J. L. Shuhang Gu, Andreas Lugmayr and R. Timofte, *Div8k: Diverse 8k resolution image dataset*, in *International Conference on Computer Vision (ICCV) Workshops*, October, 2019.

[103] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, *arXiv preprint arXiv:1804.06826* (2018).

[104] J. Clemons, C.-C. Cheng, I. Frosio, D. Johnson, and S. W. Keckler, *A patch memory system for image processing and computer vision*, in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 51, IEEE Press, 2016.

[105] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, *Memory access scheduling*, in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 128–138, ACM, 2000.

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

[107] J. Chen, S. Paris, and F. Durand, *Real-time edge-aware image processing with the bilateral grid*, in *ACM Transactions on Graphics (TOG)*, vol. 26, p. 103, ACM, 2007.

[108] S. Paris, S. W. Hasinoff, and J. Kautz, *Local laplacian filters: Edge-aware image processing with a laplacian pyramid.*, *ACM Trans. Graph.* **30** (2011), no. 4 68.

[109] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[110] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, *Darkroom: compiling high-level image processing code into hardware pipelines.*, *ACM Trans. Graph.* **33** (2014), no. 4 144–1.

[111] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, *A dsl compiler for accelerating image processing pipelines on fpgas*, in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 327–338, IEEE, 2016.

[112] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, *et. al.*, *Spatial: A language and compiler for application accelerators*, in *ACM Sigplan Notices*, vol. 53, pp. 296–311, ACM, 2018.

[113] G. Dupenloup, *Automatic synthesis script generation for synopsys design compiler*, Dec. 28, 2004. US Patent 6,836,877.

[114] *ARM Cortex-A5 processor*, 2009. `https://https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a5`.

[115] Y. Zhu, B. Wang, D. Li, and J. Zhao, *Integrated thermal analysis for processing in die-stacking memory*, in *Proceedings of the Second International Symposium on Memory Systems*, pp. 402–414, 2016.

[116] D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsari, *et. al.*, *Thermal characterization of cloud workloads on a power-efficient server-on-chip*, in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 175–182, IEEE, 2012.

[117] Y. Eckert, N. Jayasena, and G. H. Loh, *Thermal feasibility of die-stacked processing in memory*, .

[118] A. Agrawal, J. Torrellas, and S. Idgunji, *Xylem: Enhancing vertical thermal conduction in 3d processor-memory stacks*, in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 546–559, IEEE, 2017.

[119] L. Nyland, J. R. Nickolls, G. Hirota, and T. Mandal, *Systems and methods for coalescing memory accesses of parallel threads*, Dec. 27, 2011. US Patent 8,086,806.

[120] J. E. Lindholm, M. Y. Siu, S. S. Moy, S. Liu, and J. R. Nickolls, *Simulating multiported memories using lower port count memories*, Mar. 4, 2008. US Patent 7,339,592.

[121] D. Kirk *et. al.*, *Nvidia cuda software and gpu parallel computing architecture*, in *ISMM*, vol. 7, pp. 103–104, 2007.

[122] C. Nvidia, *Compute unified device architecture programming guide*, .

[123] M. Martineau, P. Atkinson, and S. McIntosh-Smith, *Benchmarking the nvidia v100 gpu and tensor cores*, in *European Conference on Parallel Processing*, pp. 444–455, Springer, 2018.

[124] J. D. Leidel and Y. Chen, *Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations*, in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 621–630, IEEE, 2016.

[125] Y. Xie and J. Zhao, *Die-stacking architecture*, *Synthesis Lectures on Computer Architecture* **10** (2015), no. 2 1–127.

[126] C. NVIDIA, *Compiler driver nvcc, Options for Steering GPU Code Generation URL* (2013).

[127] NVIDIA, *Parallel Thread Extension ISA*, 2020.
`https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[128] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, *Dynamic warp formation and scheduling for efficient gpu control flow*, in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 407–420, IEEE, 2007.

[129] NVIDIA, *cuBLAS Library*, 2020.
`https://docs.nvidia.com/cuda/cublas/index.html`.

[130] NVIDIA, *CUB Library*, 2020. `https://github.com/NVlabs/cub`.

[131] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, *Rodinia: A benchmark suite for heterogeneous computing*, in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

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

[133] N. Matloff, *Introduction to discrete-event simulation and the simpy language*, *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* **2** (2008), no. 2009 1–33.

[134] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, *Analyzing cuda workloads using a detailed gpu simulator*, in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, IEEE, 2009.

[135] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, *Low overhead instruction latency characterization for nvidia gpgpus*, in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2019.

[136] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chennupati, S. Eidenbenz, and N. Santhi, *Verified instruction-level energy consumption measurement for nvidia gpus*, in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pp. 60–70, 2020.

[137] R. S. Verdejo, K. Asifuzzaman, M. Radulovic, P. Radojković, E. Ayguadé, and B. Jacob, *Main memory latency simulation: the missing link*, in *Proceedings of the International Symposium on Memory Systems*, pp. 107–116, 2018.