University of California
Santa Barbara

Efficient In-DRAM Near-Bank Processing for Emerging Parallel Computing Workloads

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Xinfeng Xie

Committee in charge:

Professor Yuan Xie, Chair
Professor Dmitri Strukov
Professor Tim Sherwood
Professor Yufei Ding

March 2022

The Dissertation of Xinfeng Xie is approved.

Professor Dmitri Strukov

Professor Tim Sherwood

Professor Yufei Ding

Professor Yuan Xie, Committee Chair

March 2022

Efficient In-DRAM Near-Bank Processing for Emerging Parallel Computing Workloads

by

Xinfeng Xie

I am dedicating this dissertation to my parents, Jieli Fang and Boxuan Xie, for bringing me to this world and giving me their unconditional support and love throughout my life.

Acknowledgements

I have received help from many people along my Ph.D. program. I would like to express my deepest gratitude to these kind people and their nice help. Without their generous help, it would be impossible for me to complete this dissertation.

First of all, I would like to thank my advisor, Prof. Yuan Xie. As my advisor, Prof. Yuan Xie is very supportive of my research and professional skill development. He leaves me the highest degree of freedom to select research topics and schedule project agendas, and he always encourages me to explore topics I am most interested in. Moreover, when I was frustrated with obstacles and difficulties, he always encourages me to insist on my research direction. In addition to my research projects, he also supports all of my decisions about industrial internships. These internship experiences deeply shape my research tastes and motivate me to focus more on real-world problems.

Second, I would like to thank all members of my Ph.D. committee, Prof. Dimitri Strukov, Prof. Tim Sherwood, and Prof. Yufei Ding, for their professional services. Especially, I would like to thank Prof. Yufei Ding for her suggestions and help to improve my research work in this dissertation.

Third, I would like to thank all of my labmates and collaborators, including but not limited to Dr. Lei Deng, Dr. Fengbin Tu, Dr. Jiayi Huang, Dr. Maohua Zhu, Dr. Abanti Basak, Alvin Glova, Liu Liu, Wenqin Huangfu, Nan Wu, Tianqi Tang, Gushu Li, Ling Liang, Bangyan Wang, Jilan Lin, Zheng Qu, Zhaodong Chen, Guyue Huang, and Anbang Wu. Working in a research lab of so many great researchers with diverse research directions, I learned a lot from them in a wide spectrum of research fields. Among all labmates and collaborators, I would like to especially thank Dr. Xing Hu and Dr. Peng Gu for their help to both my research projects and my personal life. Dr. Xing Hu is the senior researcher who helped me build fundamental skills in the early stage of my Ph.D.

program. She also helped me a lot when I was settling down in Santa Barbara. Dr. Peng Gu is my most important collaborator, who is the co-primary contributor to two projects in this dissertation. He taught me a lot about memory technology and hardware knowledge. Also, his hard-working and enthusiasm for research always motivate me to push myself forward in my research projects.

In addition to the help I received in my academic research projects, I also received a lot of help from industrial mentors in internships during my Ph.D. program. I would like to thank Dr. Andrea Di Blas, Dr. Qiuling Zhu, Dr. Yuchen Hao, Dr. Jianyu Huang, Dr. Peter Ma, and Dr. Yanqi Zhou for their great mentorship during my stays in Google and Facebook. I sincerely appreciate their efforts to help me build professional skills and help me set up a good starting point in my career path.

Last but not least, I am thankful to my friends and my family members for their continuous support and encouragement not only in this Ph.D. program but also in my whole life. Especially, I thank my parents, Jieli Fang and Boxuan Xie, for bringing me to this world and encouraging me to explore the beauty, art, and adventure of this journey.

Curriculum Vitæ

Xinfeng Xie

Education

2022	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara, United States.
2020	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara, United States.
2017	B.S. in Microelectronics Science and Engineering, Peking University, Beijing, China.

Publications

[C1] Pengfei Zuo, Yu Hua, Ling Liang, **Xinfeng Xie**, Xing Hu, Yuan Xie. "SEALing Neural Network Models in Encrypted Deep Learning Accelerators." 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021.

[C2] **Xinfeng Xie**, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, Yuan Xie. "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator." 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021.

[C3] Abanti Basak, Jilan Lin, Ryan Lorica, **Xinfeng Xie**, Zeshan Chishti, Alaa Alameldeen, Yuan Xie. "Saga-bench: Software and hardware characterization of streaming graph analytics workloads." 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2020.

[C4] Peng Gu, **Xinfeng Xie (co-primary author)**, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, Yuan Xie. "iPIM: Programmable in-memory image processing accelerator using near-bank architecture." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[C5] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, **Xinfeng Xie**, Yufei Ding, Chang Liu, Timothy Sherwood, Yuan Xie. "Deepsniffer: A dnn model extraction framework based on learning architectural hints." Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020.

[C6] Kun Wu, Guohao Dai, Xing Hu, Shuangchen Li, **Xinfeng Xie**, Yu Wang, Yuan Xie. "Memory-bound proof-of-work acceleration for blockchain applications." Proceedings of the 56th Annual Design Automation Conference 2019. 2019.

[C7] Yu Ji, Youyang Zhang, **Xinfeng Xie**, Shuangchen Li, Peiqi Wang, Xing Hu, Youhui Zhang, Yuan Xie. "Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture." Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019.

[C8] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, **Xinfeng Xie**, Li Zhao, Xiaowei Jiang, Yuan Xie. "Analysis and optimization of the memory hierarchy for graph processing workloads." 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019.

[C9] Peiqi Wang, **Xinfeng Xie**, Lei Deng, Guoqi Li, Dongsheng Wang, Yuan Xie. "Hitnet: Hybrid ternary recurrent neural network." Proceedings of the 32nd International Conference on Neural Information Processing Systems. 2018.

[C10] Jie Wang, **Xinfeng Xie**, Jason Cong. "Communication optimization on GPU: A case study of sequence alignment algorithms." 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2017.

[J1] **Xinfeng Xie**, Peng Gu, Jiayi Huang, Yufei Ding, Yuan Xie. "MPU-Sim: A Simulator for In-DRAM Near-Bank Processing Architectures." IEEE Computer Architecture Letters, vol. 21, no. 1, pp. 1-4, 1 Jan.-June 2022, doi: 10.1109/LCA.2021.3135557.

[J2] **Xinfeng Xie**, Prakash Prabhu, Ulysse Beaugnon, Mangpo Phitchaya Phothilimthana, Sudip Roy, Azalia Mirhoseini, Eugene Brevdo, James Laudon, Yanqi Zhou. "A Transferable Approach for Partitioning Machine Learning Models on Multi-Chip-Modules." arXiv preprint arXiv:2112.04041 (2021).

[J3] Xiaobing Chen, Yuke Wang, **Xinfeng Xie**, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, Yuan Xie. "Rubik: A Hierarchical Architecture for Efficient Graph Neural Network Training." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2021).

[J4] **Xinfeng Xie**, Peng Gu, Yufei Ding, Dimin Niu, Hongzhong Zheng, Yuan Xie. "MPU: Towards Bandwidth-abundant SIMT Processor via Near-bank Computing." arXiv preprint arXiv:2103.06653 (2021).

[J5] **Xinfeng Xie**, Xing Hu, Peng Gu, Shuangchen Li, Yu Ji, Yuan Xie. "NNBench-X: A Benchmarking Methodology for Neural Network Accelerator Designs." ACM Transactions on Architecture and Code Optimization (TACO) 17.4 (2020): 1-25.

[J6] Peng Gu, **Xinfeng Xie**, Shuangchen Li, Dimin Niu, Hongzhong Zheng, Krishna T Malladi, Yuan 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).

[J7] Peiqi Wang, Dongsheng Wang, Yu Ji, **Xinfeng Xie**, Haoxuan Song, XuXin Liu, Yongqiang Lyu, Yuan Xie. "QGAN: Quantized generative adversarial networks." arXiv preprint arXiv:1901.08263 (2019).

[J8] **Xinfeng Xie**, Xing Hu, Peng Gu, Shuangchen Li, Yu Ji, Yuan Xie. "Nnbench-x: Benchmarking and understanding neural network workloads for accelerator designs." IEEE Computer Architecture Letters 18.1 (2019): 38-42.

[J9] **Xinfeng Xie**, Dayou Du, Qian Li, Yun Liang, Wai Teng Tang, Zhong Liang Ong, Mian Lu, Huynh Phung Huynh, Rick Siow Mong Goh. "Exploiting sparsity to accelerate

fully connected layers of cnn-based applications on mobile socs." ACM Transactions on Embedded Computing Systems (TECS) 17.2 (2017): 1-25.

Abstract

Efficient In-DRAM Near-Bank Processing for Emerging Parallel Computing Workloads

by

Xinfeng Xie

Despite the success of parallel architectures and domain-specific accelerators in boosting the performance of emerging parallel workloads, contemporary computer organizations still face the bottleneck of data movement between processors and the main memory. Processing-in-memory (PIM) architectures, especially those designs integrating compute logics near DRAM memory banks, are promising to address this bottleneck. However, such an in-DRAM near-bank integration faces hardware and software design challenges in performance, area overheads, architecture complexity, and programmability.

To address these challenges, this dissertation focuses on developing efficient hardware and software solutions for in-DRAM near-bank computing. First, this dissertation investigates the memory bandwidth bottleneck of contemporary hardware platforms through in-depth workload characterization, which motivates in-DRAM near-bank processing solutions. Second, this dissertation proposes multiple full-stack in-DRAM near-bank processing solutions targeting different application scopes that vary from application-specific to general-purpose computing. These solutions reveal a wide spectrum of trade-off points among hardware efficiency, architecture flexibility, and software complexity. On top of these solutions, this dissertation introduces an open-source simulation framework that supports the architectural and software optimization studies of in-DRAM near-bank processing. Finally, this dissertation develops novel machine learning-based compiler optimizations for partitioning workloads on a chiplet hardware platform that has a distributed compute-memory abstraction similar to in-DRAM near-bank architectures.

Contents

Chapter 1

Introduction

The slowdown of technology node scaling urges hardware architecture innovations to sustain the performance improvements of modern computing platforms. Over the last several decades, parallel computing architectures, including multi-core CPUs and many-cores GPUs, have achieved great success in accelerating many data-intensive parallel computing workloads. Additionally, domain-specific hardware designs further boost the performance improvements in many important application domains, such as TPU [1] for machine learning applications. Despite these successful architecture innovations, the advances in computation capability significantly outpace the improvements of memory technology. Compared with the growth of computation capability on modern computing platforms, the increase of memory bandwidth is much slower. This comes from the difficulties in increasing the number of off-chip I/O pins and the frequency of memory bus under stringent area, thermal, and signal integrity constraints. As a result, the memory bandwidth wall emerges as a new bottleneck where data movements between processors and memory units can hardly catch up with the computation throughput.

The 3D-stacking near-data-processing (3D-NDP) architecture [2] emerges as a promising approach to alleviate this memory bandwidth bottleneck. Currently, high-end GPUs

are equipped with high-bandwidth memory (HBM) stacks, where off-chip data transfers need to go through the low performance I/Os on the silicon interposer. The principal idea of 3D-NDP is to closely integrate affordable logic components adjacent to the memory stack. A large number of pioneering studies have adopted the processing-on-base-logic-die (PonB) architecture, where general purpose cores (e.g., SIMT cores [3–10]) are placed on the base logic die of the 3D stack to benefit from the intra-stack bandwidth enhancement (around 2× w.r.t. HBM [10]). This solution provides a mediocre bandwidth improvement because intra-stack memory accesses are still bounded by the limited number of through-silicon-vias (TSVs) between memory dies and the base logic die. To overcome this bandwidth bottleneck of TSVs, recent near-bank accelerators [11–13] further move simple arithmetic units closer to the DRAM banks to harvest the abundant bank-internal bandwidth (around 10× w.r.t. process-on-logic-die solution). These near-bank accelerators have demonstrated significant speedups (around $2\times - 14\times$ w.r.t. GPU). Moreover, recent industrial near-bank prototypes from UPMEM [14] and Samsung [15] have demonstrated that it is feasible to place compute-logic on DRAM dies. Thus near-bank processing is promising to tackle the memory bandwidth issue.

Despite the promising future of near-bank processing, there are still hardware and software challenges for efficient near-bank processing that need innovative solutions and comprehensive studies. In terms of hardware designs, we need to not only innovate area-efficient near-bank processing architectures but also investigate the trade-off between hardware programmability and efficiency. Although prior work demonstrates several near-bank processing designs through domain customization [11, 12, 15], the solutions for efficient programmable in-DRAM compute-logics are still missing. Moreover, the fabrication of such compute-memory hybrid chips is expensive so that the trade-offs among application scopes, hardware costs, and performance are important. In terms of software support, the unique hardware features of near-bank processing architectures together with

a wide spectrum of hardware designs open a new space for studying efficient programming language and system supports. Because of domain specialization, prior studies usually only narrow this large space to their applications. In particular, near-bank processing architectures have unique hardware features different from traditional processors that are not taken into consideration in modern compilers and system software. For example, different access patterns to the same memory bank could result in significant bandwidth differences because DRAM row buffer locality affects DRAM timing. These unique hardware features make the optimizations of not only data locality but also memory access order important and challenging. Finally, an open-source simulator for studying the hardware and software challenges of near-bank processing is missing. Even though there are some existing simulators for PIM architectures, they can hardly support the unique architecture features and the best fit programming model of near-bank processing. This missing piece significantly impedes the research and development of efficient in-DRAM near-bank processing solutions.

To address the aforementioned challenges for efficient in-DRAM near-bank processing, we make the following contributions in this dissertation:

Architecture Designs: In this dissertation, we develop three near-bank processing architectures targeting different application scopes. These architectures include application-specific, domain-specific, and general-purpose computing designs. In particular, we develop SpaceA for sparse matrix-vector multiplication, iPIM for image processing applications, and MPU for data-intensive parallel computing workloads Our architectural innovations include lightweight hardware components through customization and decoupled control-execution to minimize in-DRAM overheads. Through our evaluations of performance, power, and area overheads among these three architectures, this dissertation provides a comprehensive view of trade-offs between application scopes and these design metrics. In particular, application-specific designs provide the most significant

performance and energy efficiency improvements with the smallest area overhead while general-purpose design can also achieve considerable performance benefits with affordable overheads.

Software Support: To address the challenges of unique memory abstraction and fully exploit the hardware potentials of near-bank processing platforms, we develop several optimizations in this dissertation. These optimizations include data locality exploration in SpaceA by clustering matrix rows with similar column indices to the same memory bank, memory order enforcement in iPIM, and register location analysis in MPU. In addition to these architecture-dependent optimizations, we also study the workload partitioning problem on multi-chip-modules (MCMs) that have a similar memory space abstraction as near-bank processing architectures. We develop a novel constrained reinforcement learning method to partition machine learning workloads on an MCM-based ML accelerator, and our real hardware evaluation demonstrates its significant improvements over existing search methods in not only the final achieved speedups but also the compilation time.

Simulation Infrastructure: Through our research projects for near-bank processing architectures, we build and release an open-source simulator, MPU-Sim, for studying hardware and software challenges. In particular, MPU-Sim supports unique architecture features that are important to near-bank processing, such as individual memory bank control, shared bus arbitration, and decoupled execution pipelines. Moreover, MPU-Sim supports the single instruction multiple threads (SIMT) programming model that is the best fit to control a massive number of parallel memory banks. These unique hardware and software features are missing from existing PIM simulators. We conduct calibration and case studies to validate MPU-Sim and demonstrate its potential usage for the future research of near-bank processing solutions.

The remainder of this dissertation is organized as follows: Chapter 2 presents a holistic benchmarking methodology for neural network accelerator designs, which moti-

vates near-bank computing architectures to overcome the memory bandwidth bottleneck. Chapter 3, Chapter 4, and Chapter 5 present near-bank processing architectures for application-specific accelerator, domain-specific accelerator, and general-purpose computing respectively. In particular, Chapter 3 presents SpaceA for sparse matrix vector multiplication, Chapter 4 presents iPIM for image processing applications, and Chapter 5 presents MPU for data-intensive parallel computing workloads. We also develop and detail our open-source simulator, MPU-Sim, in Chapter 6 that is an important tool in studying hardware and software challenges for efficient near-bank processing. In addition to simulation tools, we also optimize the workload partitioning problem on real hardware in Chapter 7 that has a similar memory space abstraction as near-bank processing architectures. Finally, Chapter 8 summarizes the research projects and contributions of this dissertation for efficient in-DRAM near-bank computing solutions.

Chapter 2

NNBench-X: A Benchmarking Methodology for Neural Network Accelerator Designs

In this chapter, we introduce NNBench-X which is a benchmarking methodology for neural network accelerator designs. This project conducts workload characterization and hardware evaluation for neural network (NN) applications, which derives insights about both workload characteristics and future hardware design guidelines. One of the most important observations in this project is the memory bandwidth bottleneck in many NN models despite the wide usage of compute-intensive tensor operations. Our hardware evaluation reveals that these applications urge the memory bandwidth of future hardware designs.

Neural network (NN) algorithms have demonstrated better accuracy than traditional machine learning algorithms in a wide range of application domains, such as computer vision (CV) [16–19] and natural language processing (NLP) [20–22]. These breakthroughs indicate a promising future for their real-world deployment. Deploying these applications,

especially for the inference stage, requires high performance under stringent power budgets, which boosts the emergence of accelerator designs for these applications. However, designing such an NN accelerator using application-specific integrated circuits (ASICs) is challenging because NN applications are changing rapidly to support new functionalities and improve accuracies, while ASIC design requires a long design and manufacturing period. The accelerator design could be prone to becoming obsolete if the design fails to capture key characteristics of emerging models. Therefore, a benchmark to capture these workload characteristics is crucial to guiding NN accelerator design.

In this work, we propose an end-to-end benchmarking approach for software-hardware co-design to quantitatively select applications and benchmark software-hardware co-design by decoupling our approach into three stages, workload characterizations, software-level model compression strategies, and hardware-level accelerator evaluations. In the first stage, *application set selection*, we characterize NN applications of interest without considering any software optimization techniques. After gathering their performance features, we select representative applications for the original application set. In the second stage, *benchmark suite generation*, users can refine the selected applications to generate the final benchmark suite according to their model compression strategies. New NN models for each application in the original benchmark suite will be generated according to software-level optimizations, such as quantizing and pruning techniques. In the last stage, *hardware evaluation*, users can provide the performance models of their accelerator designs together with the assumptions of interconnection and host. Accelerators are evaluated with the benchmark suite generated from the second stage. Power, performance, and area results are derived according to input performance models.

To demonstrate the functionality of our benchmark, we conduct a case study on designing NN accelerators for general NN applications. First, we comprehensively analyze 57 models with 224,563 operators from the TensorFlow (TF) Model Zoo [23]. Second, we

generate benchmark suites by using several state-of-the-art software-level optimizations including quantizing and pruning NN models. Finally, we evaluate several representative accelerators including general-purpose processors (CPU and GPU), accelerator architecture (DianNao [24]), near-data-processing architecture (Neurocube [25]), and sparse-aware architecture (Cambricon-X [26]).

Our contributions can be summarized as follows:

- We propose a novel benchmarking method, which selects the benchmark by analyzing a user-input candidate application pool and covers software-hardware co-design configurations with high flexibility. Therefore, our benchmark method is able to provide guidelines for architecture design to trade-off application compatibility, algorithm accuracy, and hardware performance.

- We conduct a case study of generating a general-purpose NN benchmark suite from the TF Model Zoo while applying state-of-the-art NN model compression techniques and evaluate it on representative architectures to demonstrate the functionality of our benchmark method. Our case study reveals that CV and NLP applications show very different performance characteristics and favor different compression techniques and hardware architectures.

2.1 Motivation

2.1.1 System Stack and Neural Network Models

Modern NN development and deployment system stacks are decoupled into several levels. As shown in Figure 2.1, the whole system stack includes application, framework, primitive, and hardware levels. From top to bottom, the application level focuses on developing high accuracy algorithms, and sometimes makes trade-offs between accuracy

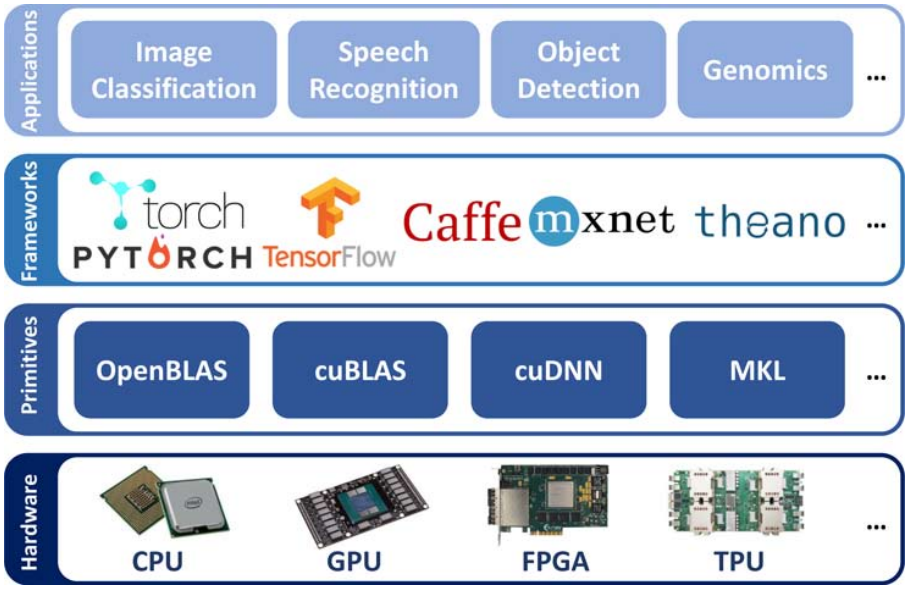


Figure 2.1: System stack for the development and deployment of NN applications including (1) application layer, (2) framework layer, (3) primitive layer, and (4) hardware layer.

and performance when exploring different NN structures. The framework level focuses on transforming high level abstractions into hardware primitives by providing a flexible programming model and efficient runtime environment. Meanwhile, the primitive level provides simple and well-optimized primitives for the hardware. For example, cuDNN [27] provides well-optimized library for executing convolution on GPUs. At the bottom of the whole development and deployment stack, the hardware level provides efficient hardware platforms for executing NN applications.

Across these system stack levels, each NN model is represented by a computation graph, which abstracts tensor operators as vertexes and tensor operands as edges to present an NN model. The topology of computation graphs indicate the data dependency among tensor operators. Figure 2.2 provides an example NN represented by these two abstractions to demonstrate their differences. The computation graph abstraction brings a more flexible representation of NN models and modern frameworks, such as TensorFlow [28] and PyTorch [29], adopt computation graph as the programming model.

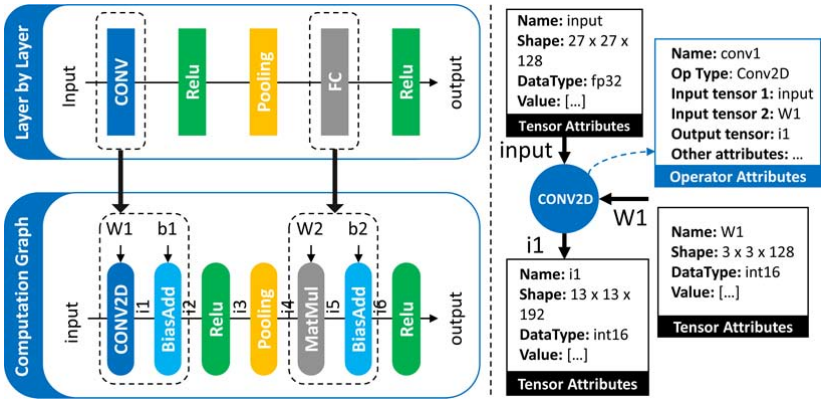


Figure 2.2: An NN example represented by the layer-by-layer abstraction and the computation graph with the detailed components of a Conv2D operator to explain what are included in an operator.

Thus, the computation graph representation is general across different NN frameworks. Moreover, because the computation graph does not have any constraint on the graph topology, it is fully compatible with all widely used NN models including RNN models even though it could introduce loops in the computation graph. All models from TensorFlow Model Zoo [23] are represented by TensorFlow graphs, which is an implementation of the computation graph concept. In the rest of this paper, we adopt this abstraction taking an NN model as a computation graph.

2.1.2 Neural Network Benchmarks

Although many NN benchmark suites have recently been proposed, through analyzing available suites we see that many demands are not met. We first narrow down the analysis of existing suites by categorizing all previous benchmarks in terms of *benchmark-suite* and *benchmark-object*. Then, we highlight the novelty of this work by comparing it to BenchIP [30] and Fathom [31] in four detailed aspects.

All previous NN benchmarks can be categorized according to the *benchmark-suite* and *benchmark-object*. A *benchmark-suite* consists of a set of representative workloads

Table 2.1: Classifying NN benchmarks w.r.t. *benchmark-suite* and *benchmark-object*. (★: benchmark-suite, □: benchmark-object)

	BenchNN [32]	BenchDL [33]	DeepBench [34]		Fathom [31]	BenchIP [30]	**Our Work**
Application	★	★			★	★	★ □
Framework		□					
Primitive			★			★	
Hardware	□		□		□	□	□

Table 2.2: The uniqueness of our benchmarking methodology. (✗ means the corresponding feature is not supported, and ✓ means the corresponding feature is supported)

	Fathom	BenchIP	**Ours**
Analysis based App. Selection	✗	✗	✓
Flexible with Update/Customize	✗	✗	✓
SW/HW Co-design	✗	*fixed*	*general*
Evaluation on Accelerators	✗	*ASIC*	*ASIC/NDP*

to be evaluated on different *benchmark-objects*. We classify the benchmark-suite and benchmark-object into different levels in the system stack, as shown in Table 2.1. Although BenchNN [32] is one of earliest efforts in building an NN benchmark, the benchmark-suite is a bit out of date without updates. Prior study [33] (denoted as BenchDL) proposes a benchmark suite for evaluating different deep learning software tools, i.e., frameworks in our system stack of NN applications. DeepBench [34] is a benchmark suite comparing the performance of different primitives on different platforms. However, benchmarking NN applications from the primitive layer loses the whole picture. Fathom [31] and BenchIP [30] serve a similar purpose as our work. However, they do not take software-hardware co-design as the benchmark object. *Different from all of them, our benchmarking methodology targets at capturing end-to-end application-to-hardware characteristics to guide architecture design for state-of-the-art NN workloads.* Since both Fathom and BenchIP serve a similar purpose of benchmarking NN accelerator designs, we further detail our comparison with Fathom and BenchIP in four aspects, as summarized

in Table 2.2.

Quantitative analysis based benchmark selection: Accelerator designers usually know the application domain they are interested in, which could include a large number of NN applications. Thus, it is important to select representative NN applications to guide hardware architecture design. Fathom and BenchIP pick their applications with some empirical guidelines but not by any quantitative analysis. Even though they show the effectiveness of their selected suits afterward, there is no guarantee that their selections are the **most** representative. *On the contrary, our approach selects benchmarks according to the results of extensive profiling and analyzing.* Our method characterizes NN applications through application features that are key to the performance, from the perspective of architecture designs. At the end of Section 2.3.1, we show how our method captures additional features that other benchmarks fail to cover.

Flexible with updates and customizations: We propose a benchmarking methodology, **not** simply a benchmark suite. By doing this, we are subject to updates due to the rapid developing NN algorithms. Statistics [30] have shown that within one year, the NN models proposed in top tier conferences double. For a fixed benchmark suite, it is difficult to know whether to extend the suite and whether a new accelerator is needed when a new model appears. Although evaluating a new model on existing accelerators can help us understand its characteristics to some extent, the demand for updating the benchmark suite and designing a new accelerator would be challenging without an in-depth workload characterization. In addition, most of the accelerators target a certain application scenario (e.g., autonomous cars), instead of a general NN processor. A single one-for-all benchmark suite does not adequately address these needs. Instead, we generate different suites according to the user-customized candidate application pool.

SW/HW co-design: Recent NN accelerator designs usually include both software optimizations, such as model pruning and quantization, and hardware optimizations.

Our benchmark method is the first for accelerators with a comprehensive awareness of software-hardware co-design. Although BenchIP [30] includes sparse models, such as Sparse VGG, into their application set as representative workloads, these considerations are insufficient due to two reasons. First, pruned models are very similar to their original models in their work. For example, Sparse VGG performs very similar to VGG in terms of extracted performance features, making it redundant. Second, their sparsity benchmark cannot consider all model compression techniques. For example, structural sparsity [35] is not covered.

Diversity of evaluation platforms: Because of the growing heterogeneity of hardware platforms, targeting only ASIC designs is not sufficient. We evaluate our benchmarks not only on CPU/GPU and ASICs but also on other innovative architectures such as NDP architectures. In addition, our evaluation method is not limited to any NN framework. Instead, we use the computation graph as a programming model with a general abstraction for the execution of NN applications across different platforms.

2.2 Benchmarking Methodology

An overview of our benchmarking method is shown in Figure 2.3. Our benchmarking method includes three stages. The first stage is *application set selection*, with an application candidate pool as its user input and original application set as its output [36]. The second stage is *benchmark suite generation*, with the model compression technique as the user input and the previous generated original application set as another input. The last stage is the *hardware evaluation*, which takes the generated benchmark suite and the hardware performance models as its inputs, and then outputs the performance results. The rest of this section will introduce these three stages in detail.

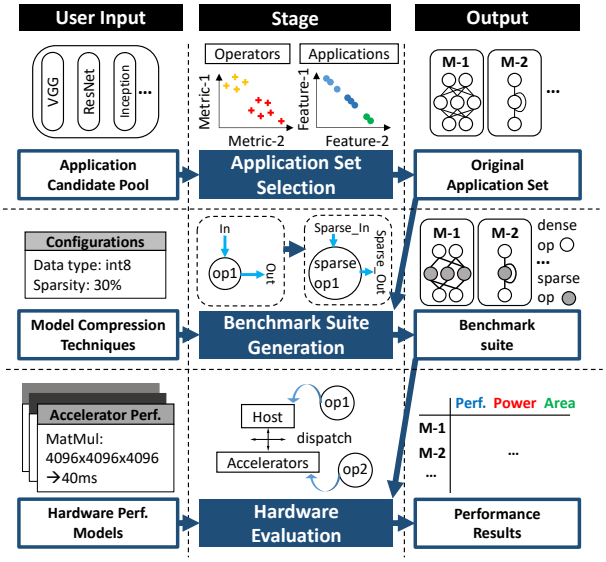


Figure 2.3: Benchmark method overview with three main stages and their corresponding inputs and outputs.

2.2.1 Application Set Selection

In the first stage, application set selection, we select diverse and representative NN applications from the application candidate pool which includes the applications of the user's interests.

The proposed application set selection consists of two phases, operator-level and application-level analysis, as shown in Figure 2.4. Since tensor operators are the primitives of NN applications, operator-level analysis is conducted first, before application-level analysis. In the operator-level analysis, we extract all operators from the application candidate pool, and use two important metrics, locality and parallelism, as the performance feature to represent an operator. Then, all the operators are clustered into several groups according to the extracted operator features. This process of getting operator clusters is detailed as Algorithm 1. After the operator-level clustering, application-level analysis is performed as the second phase. Applications are first profiled on baseline architectures before they are quantified by time breakdown on the different operator clusters. The

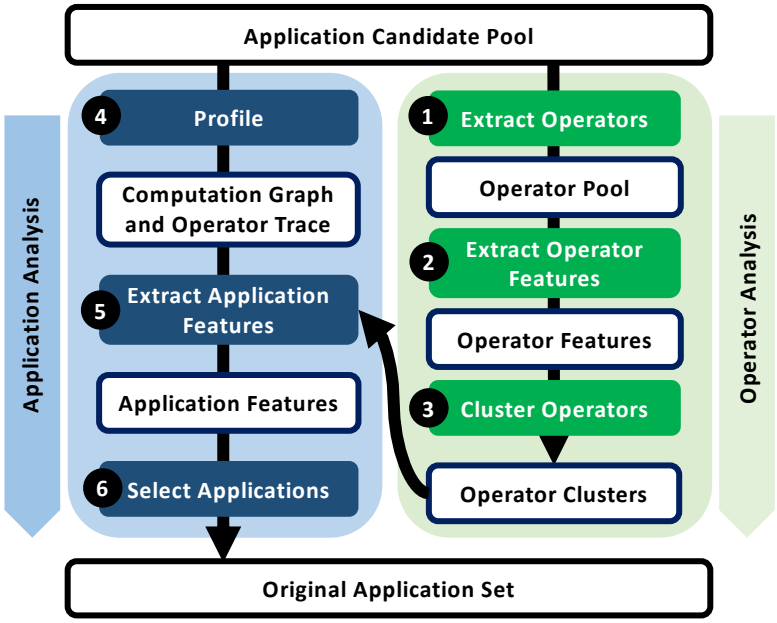


Figure 2.4: Application set selection process with two phases: operator-level analysis phase and application-level analysis phase.

process of getting application features is detailed as Algorithm 2. After obtaining these application features, we conduct a similarity analysis for all applications. Finally, an application set composed of diverse and representative workloads can be selected out of the application candidate pool. Instead of clustering operators according to their functionalities, as in prior work [31], our work is fundamentally different because it clusters tensor operators according to their architectural features, i.e. locality and parallelism. We observe that functionality-based classification is not sufficient and can cause incorrect bottleneck characterization, as validated by the experiments at the end of Section 2.3.1.

Operator-level Analysis

As shown in Figure 2.4, we perform operator-level analysis in the first phase to extract operator features and cluster operators based on these operator features. Our operator-level analysis first extracts all operators from the applications in the application candidate pool. Then, we analyze operator features from the perspective of architecture designs.

Finally, we cluster these operators.

To improve the generality of the generated benchmark suite, we use platform-independent metrics as the operator feature. Specifically, we define two platform-independent metrics, *Locality* and *Parallelism*, for the operator-level analysis to reflect general architecture considerations when designing accelerators for tensor operators. A common practice in accelerator design is to consider customized data-path designs, such as the different dataflow structures in Eyeriss [37], that can leverage both the locality of these operators and can utilize multiple processing elements (PEs) to exploit the available parallelism. Thus, these two platform-independent metrics can be useful to help understand operators from the viewpoint of architectural designs for overall demands. The definition of these two metrics used to represent the architectural feature of an operator is detailed as follows.

Locality. This metric is defined as the amount of data needed by an operator divided by the number of scalar arithmetic computations it needs. The amount of data needed by an operator is equal to the sum of the input tensor size and the output tensor size. Input tensors include all input data needed by this operator, such as model weights. Our locality metric reflects the overall locality of an operator because it indicates the average times of a byte used in the scalar arithmetic computations. Moreover, the average times of a byte used in the computation indicates the locality in an ideal memory system where a cache hit happens if the same location was accessed before. For example, when the locality metric of an operator equals to 0.1, it means that this operator performs an arithmetic scalar computation on 0.1 byte of data on average. In other words, each byte is used for 10 ($= \frac{1}{0.1}$) scalar computations on average. In an ideal memory system, this byte is accessed 10 times (1 access per arithmetic computations), and the miss rate is 10% because only the first access of these ten accesses will result in a cache miss. Another example is that when the locality metric of an operator equals to 12, it means that this

operator performs an arithmetic scalar computation on 12 bytes of data on average. In this case, each data is accessed only once for the computation, and the miss rate in an ideal memory system is 100% because there is no data-reuse. In summary, the cache miss rate of an operator in an ideal memory system is $min\{Locality, 100\%\}$ when the cache line size is 1 byte. Thus lower values of this metric indicate better locality for the operator.

Parallelism. This metric is defined as the ratio of scalar arithmetic operations which can be executed in parallel, assuming sufficient hardware resources. Thus the quantitative value of this metric falls into the range between 0 and 1. Higher values of this metric express greater available parallelism for the operator. This metric reflects the parallelism of computations in terms of data dependency. For example, a tensor *Add* operator which adds two tensors with N elements in an element-wise manner has N scalar-add operations. All of these scalar-add operations can be executed in parallel without any true dependency. Therefore, the parallelism for this tensor *Add* operator is 100%. Take a tensor *Max* operator as another example. The functionality of a tensor *Max* operator is to find the maximum value in the input tensor with N elements. A tree-based reduction can explore the parallelism with $logN$ sequential steps that must be executed in a sequential manner. In each step of this tree-based reduction, all of the N scalar-max operations can be executed in parallel given sufficient hardware resources. As a result, the parallelism for a tensor *Max* operator is $\frac{1}{logN}$.

After obtaining operator features in the aforementioned metrics, we can group operators into several clusters according to these operator features.

Application-level Analysis

As shown in Figure 2.4, we perform application-level analysis in the second phase to extract application features and select applications based on these application fea-

tures. We define the performance feature of an application as the time breakdown on the different operator clusters obtained from the operator-level analysis. We denote the number of operator clusters as n. Specifically, the performance feature is denoted as $\vec{f} = (R_1, R_2, ..., R_n)$ where R_i represents the percentage of the elapsed time spent in the i-th class operators. We profile each application from the application candidate pool on the baseline hardware, usually a CPU or a GPU, to obtain its time spent in each operator cluster. By analyzing applications in terms of time breakdown, benchmark users can have a better understanding of which operator class acts as a bottleneck on the baseline hardware. Because operators are grouped by their architecture features of both locality and parallelism, it provides clearer guidelines to design specialized hardware to accelerate the bottleneck operator cluster.

We rely on the application-level analysis phase to understand the application characteristics on baseline platforms. Thus there are several major design decisions when we are building application features. First, we use profiling information on existing baseline platforms for a more accurate analysis. Although baseline platforms are usually general-purpose processors, such as CPU or GPU, they can be changed to other hardware devices depending on design goals. For example, if NNBench-X is used to develop the second generation of TPU, the first version of TPU could be the baseline device [1]. Second, because this phase in the application set selection stage, this phase needs to be independent from software-hardware co-design solutions to be evaluated by NNBench-X. Specifically, this phase does not take any software-hardware co-design solutions as inputs and extracts application features based on performance models of these co-designs, such as the roofline model [38]. Third, we do not consider inter-operator parallelism as a part of application features because software frameworks usually take operators as the granularity of scheduling. These frameworks will offload operators to hardware instead of the whole computation graph and they are responsible to exploit inter-operator parallelism.

Algorithm 1 Operator-level analysis to get operator clusters.

```
Input: A list of models (M) and the number of operator clusters (N)
Output: Operator cluster centers
Init All_Op_Features = []
for m in M do
  for op in m.operator_list() do
    op_features = ExtractOperatorFeatures(op)
    All_Op_Features.append(op_features)
  end for
end for
cluster_centers = kMeans(All_Op_Feastures, num_clusters=N)
Return cluster_centers
```

Algorithm 2 Application-level analysis to get application features.

```
Input: A list of models (M) and the centers of operator clusters (cluster_centers)
Output: The application features for each model (All_App_Features)
Init All_App_Features = []
for m in M do
  Init app_feature = [0.0] * len(cluster_centers)
  Init total_time = 0.0
  for op in m.operator_list() do
    op_features = ExtractOperatorFeatures(op)
    cluster_id = GetNearestClusterCenterID(op_features, cluster_centers)
    app_feature[cluster_id] += op.elapsed_time
    total_time += op.elapsed_time
  end for
  app_feature = app_feature / total_time
  All_App_Features.append(app_features)
end for
Return All_App_Features
```

However, when designing an accelerator taking the whole computation graph as inputs, this metric can be added into application features as discussed in Section 2.4.

After this two-level analysis, we select representative applications out of the application candidate pool to build the original application set.

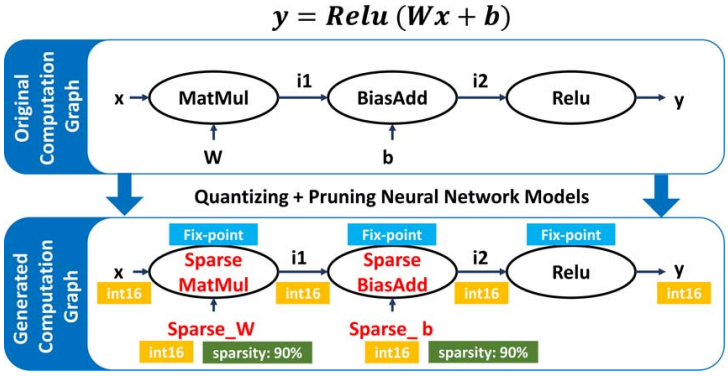


Figure 2.5: An example for benchmark suite generation to generate a new computation graph according to user-provided quantizing (int16) and pruning strategies (sparse weights).

2.2.2 Benchmark Suite Generation

In the second stage, benchmark suite generation, we provide interfaces for users to customize their NN compression techniques to generate the final benchmark suite.

This stage is motivated by the success of model compression techniques, either quantizing or pruning, and the fact that state-of-the-art accelerator designs leverage these techniques for better computation and memory access efficiency by designing specialized hardware, either fixed-point ALU or sparse tensor computation engines. Although we obtain a diverse and representative application set after the first stage, we cannot benchmark different accelerators using only one set of applications because of the diversity of NN model compression techniques.

Each application from the original application set is a computation graph. To customize different NN model compression techniques, we provide interfaces for the users to specify the data type of tensors in this computation graph. For tensors storing the pre-trained weights, users can overwrite these weights by using pruned weights so that these tensors become sparse. Sparsity information can also be included as an additional attribute in the tensors storing weights. The sparsity of the tensors produced by activation functions, such as ReLU, can be computed in runtime. Figure 2.5 illustrates a case for

these interfaces. Suppose we quantize the original application from the single-precision floating-point into 16-bit fixed-point, and prune weights by 90%, the structure of the computation graph remains the same but the operators and tensors are changed accordingly, as shown in Figure 2.5. Users can define and import model compression methods, and change the information of operators and tensors to generate the final benchmark suite according to their software-level studies in the training stage. Compression techniques resulting in intolerable accuracy degradations should not be imported into this stage. At the end of this stage, NNbench-X produces the final test set of applications composed of quantized and pruned NN models for evaluations.

Because our benchmark methodology provides interfaces for the users to specify their own compression methods instead of defining several patterns, NNBench-X is able to support a wide range of compression methods. For example, when NNBench-X is used to evaluate software-hardware co-designs exploiting the structural sparsity [39, 40], NNBench-X passes model weights to compression methods provided by the users to generate weights in structural sparse patterns. In this case, the pruned models with weights in structural sparse patterns will be in the generated benchmark suite at the end of this stage.

2.2.3 Hardware Evaluation

In the final stage, the hardware evaluation, we evaluate the generated benchmark suite on accelerator designs.

Although this stage can be completed by users with detailed simulation results of accelerators, we build a system-level simulator for fast performance estimation in the initial architecture design stages to provide high-level guidelines for accelerator designs. Our system-level simulator evaluates accelerators on the generated benchmark suite by using

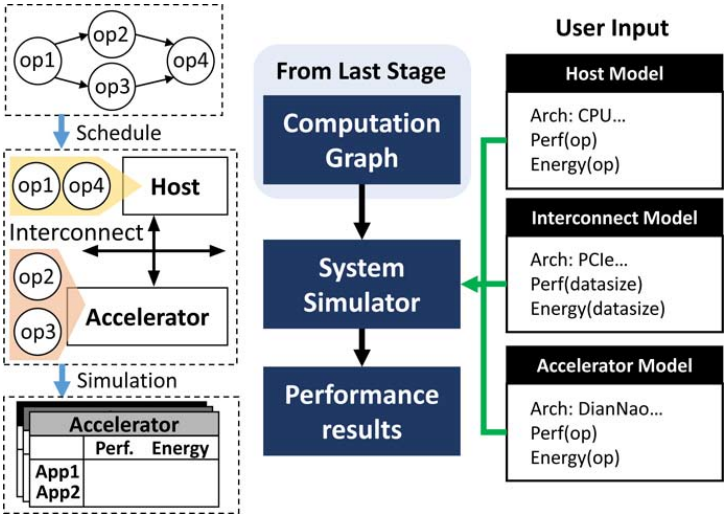


Figure 2.6: The workflow of hardware evaluation with the user-inputs for hardware modeling including models for host, interconnect, and accelerators.

the performance models of the accelerator, the host, and the interconnection between the accelerator and the host. These performance models are provided by users so that they can be as simple as a roofline model or as complicated as a cycle-accurate simulator depending on the demands of hardware evaluation. For example, early design stages could use the roofline model to decide the balance between computation and memory resources while later design stages could need cycle-accurate simulators to model more hardware details. The inputs and outputs of our system-level simulator are shown in Figure 2.6. For each application in the generated benchmark suite, our simulator schedules operators into either the accelerator or the host by a first-come-first-serve scheduling algorithm. When an operator is not supported by the accelerator, it will be launched into the host with subsequent data transfer between the accelerator and the host. The performance results of running supported operators on accelerators and overheads of data transfer between the host and accelerators are provided by input hardware models which are a part of inputs to our system-level simulator. To demonstrate the usage of our system-level simulator, we use a simple but effective analytical model, the roofline model, in

Section 2.3.2 to evaluate various architectures including DianNao [24], Neurocube [25], and Cambricon-X [26].

Our system-level simulator plays a role similar to that of frameworks. Our straightforward scheduling policy may not consistently achieve optimal performance, but integrating accelerators into the whole system with developed primitives is time-consuming and impractical in the initial design space exploration stage for architectures. As the case study shown in Section 2.3.2, the performance speedups of different architectures could vary in orders of magnitudes. Therefore, our coarse but fast estimations can still provide insightful guidelines in architectural designs. Furthermore, the accuracy of estimation in this stage depends on the accuracy of performance models provided by users. Although we use a simple analytical model, roofline model, in Section 2.3.2 as a demo case, users can provide models capturing more hardware details to fit their demands exploiting various hardware designs. For example, when it is decided to use dataflow architectures in NN accelerators and our benchmark methodology is used to evaluate and compare different dataflow designs, the MAESTRO [41] framework can be used to provide the performance results of different architectures for supported operators. Another example is that when the users want to evaluate software-hardware co-designs exploiting structural sparsity, the user-provided performance models of hardware designs need to take the sparsity into account [39, 40]. In both examples, our system-level simulator is responsible to provide operator information, such as input tensor shapes and operator weights, while users need to implement their own performance models as the backend to return the performance results of running the operator on their accelerators. For accelerator designs in Section 2.3.2, we implement a roofline model as the backend for various accelerator designs which returns the performance by using the roofline model according to operator information and hardware specifications. For the performance of operators on real devices, such as CPU and GPU, we implement the backend performance model

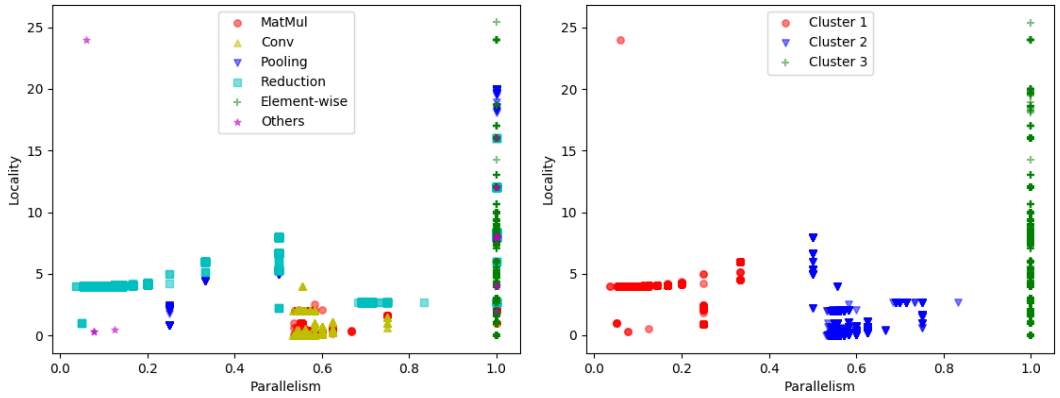
by running the operator on the real device and returning the measured time.

2.3 Workload Characterization

We conduct a case study of benchmarking NN inference accelerators to demonstrate the usage of our benchmark approach. To this end, we set the TensorFlow (TF) Model Zoo [23] (with 57 NN models and 224,563 operators) as the application candidate pool, and our software-hardware co-design evaluation includes several state-of-the-art model compression techniques and hardware designs. The version of the TF Model Zoo we used in this case study contains 57 NN models from 24 different applications. These NN models have very diverse structures including convolutional neural networks (CNNs) and recurrent neural networks (RNNs). From the perspective of learning algorithms, these models are from different learning methods, including supervised learning, unsupervised learning, and reinforcement learning. Thus our application pool has very good coverage on existing NN applications from different application domains, with different model structures, and trained by different learning algorithms. This section follows the three-step process introduced in Section 2.2. First, Section 2.3.1 studies our application set selection process to select representative applications from TF Model Zoo. By comparing to the application set of prior benchmarks, we also demonstrate the advantages by the end of Section 2.3.1. Then, Section 2.3.2 evaluates several software-hardware co-designs on these selected applications. In the process of both application set selection and evaluating software-hardware co-designs, we conclude several observations on application characteristics and architecture design guidelines from these studies.

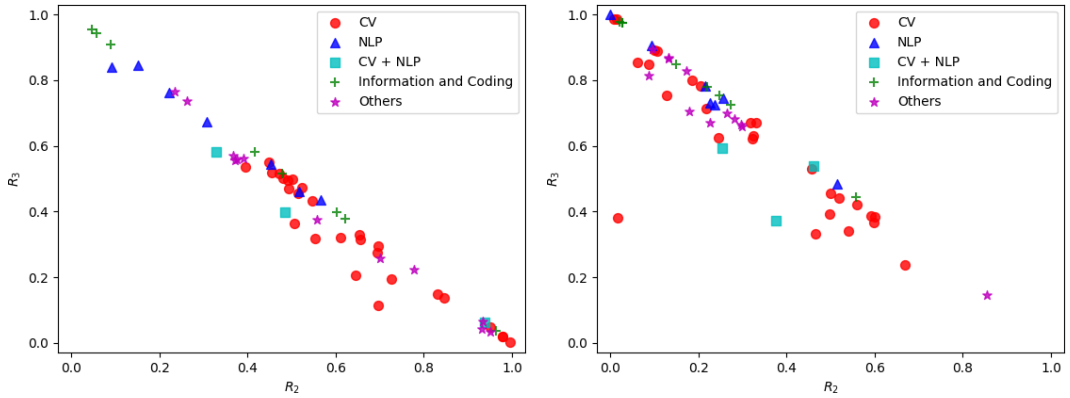
2.3.1 Application Selection from TensorFlow Model Zoo

As the first step of our analysis flow, we apply the operator-level analysis to most of the applications from the TensorFlow Model Zoo [23]. We first perform *extract operators* Figure 2.4-❶) to all 224,563 operators from the application candidate pool. We then *extract operator features* (Figure 2.4-❷) and measure both the locality and the parallelism of the operators as defined in Section 2.2.1. The resulting distribution of operator features is shown in Figure 2.7a. It labels different operator functionalities including matrix multiplication (MatMul), convolution (Conv), pooling, reduction, element-wise, and other irregular operators (Others) where computations and memory accesses are dependent on input tensor values. Based on the performance feature distribution, we conduct *cluster operations* step (Figure 2.4-❸), which groups these operators into three clusters. We apply the k-means algorithm and obtain the cluster results shown as Figure 2.7b. After this, we conduct an application-level analysis. Because most accelerator designs compare their performance to two kinds of general-purpose processors, CPU and GPU, we *profile* (Figure 2.4-❹) all applications from the application candidate pool on Intel Xeon E5-2680 CPU and NVIDIA Titan Xp GPU. To *extract application features* (Figure 2.4-❺), we use the three operator classes from previous operator analysis. The application performance feature in this case study is denoted as $\vec{f} = (R_1, R_2, R_3)$, where R_1, R_2, and R_3 represent the time breakdown of an application into three operator clusters. The performance feature distributions measured on CPU and GPU are shown as Figure 2.8a and Figure 2.8b. Since $R_1 + R_2 + R_3 = 1$, we plot two dimensional scatter figures where x-axis stands for the R_2, y-axis stands for the R_3, and R_1 can be derived by $1 - R_2 - R_3$. Finally, we *select applications* (Figure 2.4-❻). Based on the distribution of the application features on CPU, we select ten diverse and representative applications as the original application set by evenly sampling the application candidate pool. The



(a) Operators from the pool. (b) Operator clusters.

Figure 2.7: The distribution of operator features for all operators from the application candidate pool (TF Model Zoo) and the clustering results by running k-means.



(a) CPU (b) GPU

Figure 2.8: The distribution of application features using CPU and GPU as baseline devices.

distribution of these ten applications is shown in Figure 2.9. Brief descriptions for these ten applications can be found in Table 2.3.

Observations on the operator-level analysis. We classify operators into several categories to obtain observations on their architectural characteristics. The operator categories are designed to reflect operator functionalities or data access patterns. Among these operator categories, matrix multiplication (MatMul), convolution (Conv), and pooling attract intensive attention in many accelerator designs because of their importance in early NN models, such as VGG models [16]. The activation functions are also very

Table 2.3: Brief descriptions for ten applications selected into the original application set.

Application	Description	Application Domain
textsum [21]	Text summarization	Natural Language Processing
skip_thoughts [22]	Sentence-to-vector encoder	Natural Language Processing
pcl_rl [42]	Reinforcement learning	Others
entropy_coder [43]	Image file compression	Information and Coding
mobilenet [19]	Image classification	Computer Vision
inception_resnet_v2 [17, 18]	Image classification	Computer Vision
image_decoder [44]	Image file decompression	Information and Coding
rfcn_resnet101 [45]	Object detection	Computer Vision
faster_rcnn_resnet50 [46]	Object detection	Computer Vision
vgg16 [16]	Image classification	Computer Vision

common in NN models, such as ReLU operation in convolutional neural networks [16–18], and all of them are vector-like element-wise operations. Thus we create a category as Element-wise in Figure 2.7a for all operators performing vector-like operations. We also create a separate category named as reduction for operators with reduction patterns, such as the *Softmax* and *Argmax* operations. Although these five categories cover most of the operators, we put the rest of operators into the last category as others.

We make several observations from the results of operator clustering (Figure 2.7a-2.7b). First, convolution and matrix multiplication operators are similar to each other, and most of them have good locality. Because of existing reduction patterns along some tensor dimensions, such as input channels in convolution operators, these two kinds of operators possess moderate parallelism. Second, all element-wise operators have identical parallelism while the computation intensity on each tensor element can vary significantly. Because of fully parallel scalar operations for all elements in element-wise operators, element-wise operators have the largest degree of parallelism (100%). Third, *operators with the same or similar functions can have very different performance features*, such as reduction and pooling operators. Clustering these operators by functions and designing hardware accordingly would result in bottleneck mis-prediction.

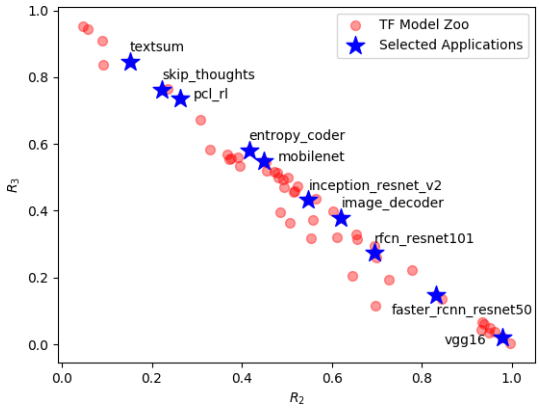


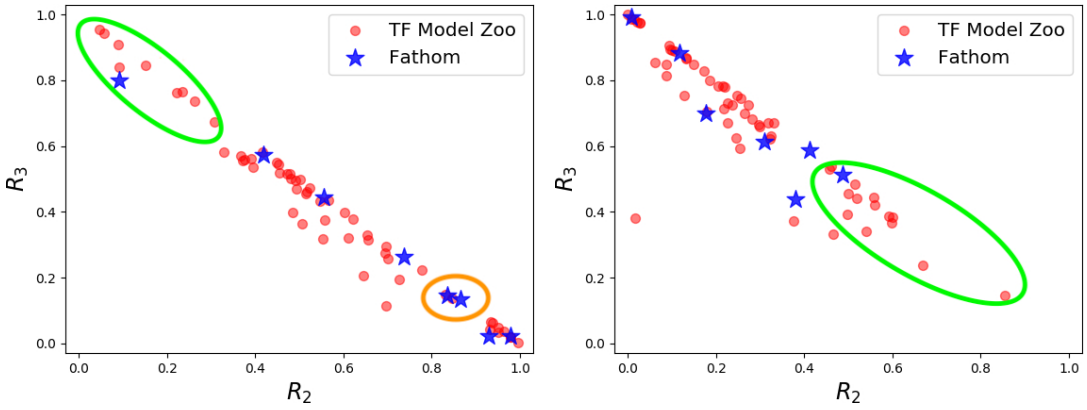
Figure 2.9: The distribution of application features for selected applications out of the application candidate pool (TF Model Zoo).

Architecture implications of operator clusters. The application feature in our work is directly associated with the breakdown of execution time spent on different operator clusters. Since we cluster operators according to their architecture features, i.e. locality and parallelism, operators in the same cluster could favor similar architecture designs. Specifically, operators in the first cluster have limited parallelism and moderate locality, whose execution time contributes to R_1. These operators could benefit from the locality optimizations while they can hardly benefit from more parallel processing elements (PEs). Operators from the second cluster have both moderate parallelism and locality, such as matrix multiplication and convolution, whose execution time contributes to R_2. These operators could benefit from parallel PE design, more computation resources, and optimizations on locality, such as the careful design of data-flow to exploit data reuse. Finally, operators from the third cluster can be fully parallelized whose execution time contributes to R_3. Increasing the number of PEs is helpful to exploit the parallelism while these operators will become bounded by memory bandwidth when the number of PEs is sufficient.

From the perspective of applications, application features indicate the distribution of execution time on these operator clusters. Thus these application features help iden-

tify the application bottleneck from the perspective of operator clusters, which further provides architecture design guidelines. For example, an application with a large R_2 indicates that its bottleneck comes from operators in the second cluster, which could prefer architecture designs with more computation resources or larger on-chip memory. Similarly, an application with a large R_3 could prefer memory-centric architectures for higher effective memory bandwidth because it is bounded by operators in the third cluster.

Observations on the application-level analysis. For the application-level analysis in Figure 2.8a-2.8b, we summarize the following observations. First, *Conv*, *MatMul*, and *Element-wise* operators take up a majority of the application time in most of the applications, since most of the applications distribute near the line $R_2 + R_3 = 1$. Second, in contrast to CPU, GPU is more likely to be bounded by R_1, due to its more powerful computing resource and higher memory bandwidth. In addition, R_3 takes a larger percentage on GPU, indicating there are opportunities for GPU memory system optimization. Third, the consideration of application scenarios reveals additional trends. Both of Figure 2.8a and Figure 2.8b label different application domains including computer vision (CV), natural language processing (NLP), hybrid CV and NLP (CV+NLP), information and coding, and others. We classify applications into application domains according to the task of applications. Applications for traditional CV or NLP tasks are labeled as CV or NLP respectively. The task of some applications is mixed by traditional CV or NLP tasks. For example, image captioning requires image understanding and caption generation where image feature extraction is a CV task while the caption generation involving text summary is an NLP task. The application domain of these mixed tasks is denoted as *CV+NLP*. In addition to these traditional CV or NLP tasks, some tasks focus on the coding of information, such as file compression, decompression, and encryption. The application domain of these tasks related to information and coding is labeled as *Information and Coding* although they could need domain knowledge related to CV or



(a) CPU (b) GPU

Figure 2.10: The application feature distribution of applications from our application candidate pool (TF Model Zoo) compared to the distribution of applications in Fathom.

NLP when handling corresponding information, such as image compression. The domain labeled as *Others* includes the rest of the applications, most of the applications in this category belong to applications using reinforcement learning, such as robotics applications. *Most CV applications are bounded by operations from R_2 (mostly* Conv *and* MatMul*). On the contrary, most NLP applications are bounded by operations from the R_3 (mostly element-wise operators).* This indicates that memory-centric computing architectures can be helpful for these NLP applications.

The advantage of our methodology. We first demonstrate the advantage of the operator-level analysis by showing how misleading bottleneck diagnosis would occur if the aforementioned analysis is neglected. Without operator-level clustering, one has to extract the application feature with function-based operator clustering. For example, as described by Fathom, *Add* operators are clustered as the category *Elementwise Arithmetic*, but *transpose* operators are clustered as another category *Data Movement*. However, when using our operator-level analysis, these two clusters should be in the same category (R_3 in our notation), since they have very similar architecture features in terms of locality and parallelism. There would be an issue in the case where R_3 is

the application's bottleneck, but as part of R_3, neither *Elementwise Arithmetic* nor *Data Movement* individually shows as a bottleneck. *The bottleneck is then misunderstood. The described problem happens for 15 out of 57 models in the TF Model Zoo.* Taking application *video_prediction_stp* [47] for example, according to the performance feature defined in Fathom, it will show *Conv2D* as the bottleneck (taking 38% of total time). However, the elapsed time of operators from the R_3 cluster takes 52% of total time, making R_3-like operators (memory-intensive highly parallel operators) the actual bottleneck, not *Conv2D*. Instead of accelerating *Conv2D*, which would result in more computation resources or larger on-chip memory, our analysis recommends that the architecture should be designed with higher effective memory bandwidth, such as processing-in-memory architectures [25,48–50], for R_3-like operators because they take the majority of the elapsed time.

Second, our benchmark process selects more diverse and representative applications. Compared to Fathom, our method selects applications from a large application candidate pool based on extracted application features. Therefore, our analysis-based selection guarantees the diversity and representativeness of selected applications from the viewpoint of performance features. To understand the representativeness of Fathom applications on the TF Model Zoo, we go through the same application analysis process for applications (8 applications in total) from Fathom. The results measured on the CPU and the GPU are shown as Figure 2.10a and 2.10b. Through comparisons, we can conclude that the application selection in Fathom is fairly good due to its similar distribution as TF Model Zoo. However, compared with Fathom, our benchmark selection in Figure 2.9 is more evenly distributed, making it more representative as a general benchmark. For example, the two selected benchmark applications in the orange circle in Figure 2.10a are too close to each other, making one of them redundant. In addition, some applications are underrepresented, such as applications in green circles in Figure 2.10a and 2.10b. The

Table 2.4: The description of hardware platforms.

Platform	GPU	Nuerocube	DianNao	Cambricon-X
Peak Comp (GOPs)	12,100	132.4	482	528
Peak Mem (GB/s)	547.7	320	250	250
TDP (W)	250	21.5	0.485[1]	0.954[1]
Area (mm^2)	471	68	3.02[1]	6.38[1]
Tech Node (nm)	16	15	65	65

[1]These power and area data are from their original papers without considering the power consumption and area cost of DRAM dies.

applications from Fathom in these green circles are not sufficiently representative of the other applications with similar characteristics.

2.3.2 Hardware Evaluation

We need the benchmark generation step (Section 2.2.2) after application selection, to plug-in the NN compression setup. This step is user-customized. According to our evaluation target, we generate our benchmark suite with three config:nnbench-x:urations: no compression (for GPU), quantized 16-bit fixed-point (for DianNao), and 16-bit fixed-point quantized and 90%/95% pruned (for Cambricon-X).

Finally, we conduct studies on evaluating several state-of-the-art software-hardware solutions in this section. In particular, we evaluate GPU (Titan Xp), Neurocube [25], DianNao [24], and Cambricon-X [26] with different model compression techniques. Among these hardware platforms we evaluated, GPU is a representative many-core processor exploiting the massive parallelism in tensor operators. Neurocube is an NDP design that exploits an internal memory bandwidth of memory cubes to accelerate memory-bound operators while DianNao is a compute-centric accelerator design with on-chip computation and data movements tailored for NN applications. Both of these two platforms are designed for computing fixed-point arithmetic, which needs the help of NN model quantization from the software-level. Cambricon-X has a similar design as DianNao except that

its design is intensively customized to exploit the sparsity of NN models, which needs the help of NN model pruning. For the purpose of architecture comparison, Figure 2.11 shows the architecture of Neurocube, DianNao, and Cambricon-X.

Table 2.4 includes comparisons among these platforms in terms of power, performance, and area. These numbers are collected from official product specifications or their original papers. Due to the lack of detailed power models and area models on these platforms, such as the off-chip DRAM power and area data of DianNao and Cambricon-X, we only estimate the performance in our case studies. We use our system-level simulator to estimate the performance of these platforms compared to the CPU baseline implementation. According to the performance results presented in the original papers, we derive an analytical model based on the roofline model [38] to estimate the performance of each supported tensor operators on accelerators. Results on the GPU are profiled and measured from the execution on a real machine. We assume that these heterogeneous platforms are connected to a host CPU, Intel Xeon E5-2680 CPU, through PCIe and any unsupported operator will be offloaded into the CPU for computation. The time of execution on the host CPU and data transfers triggered by offloading unsupported operators will be counted in the final elapsed time. However, we exclude the time used for transferring input data and model weights into these platforms because transferring different batches of input data can overlap in real-world inference stage, and loading trained weights into these platforms is a one-time overhead.

Our simulation results are shown in Figure 2.12. The original application set is evaluated on the GPU, and results are shown in Figure 2.12a. Figure 2.12b presents the performance results on Neurocube for applications quantized into 16-bit fixed-point data-type. Figure 2.12c presents the performance results for DianNao and Cambricon-X. Applications executed on DianNao are also quantized into 16-bit fixed-point. We evaluate two pruning strategies for applications executed on Cambricon-X which prunes

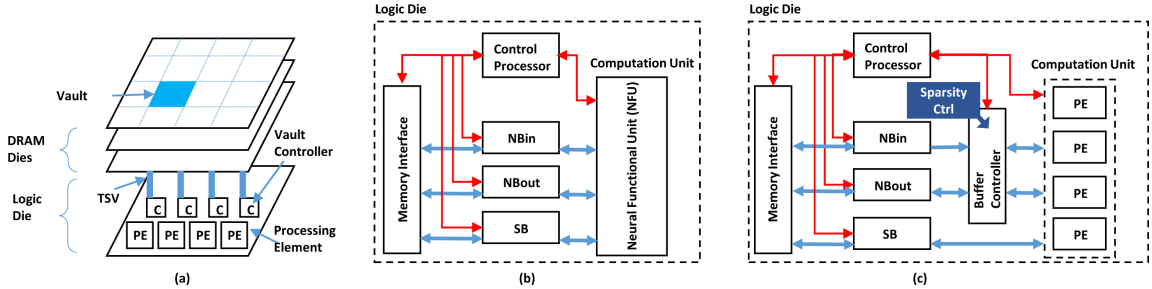


Figure 2.11: The architecture overview of (a) Neurocube, (b) DianNao, and (c) Cambricon-X to distinguish key architecture differences among them: (a) an NDP design, (b) a compute-centric design, and (c) a compute-centric design with the support for sparsity.

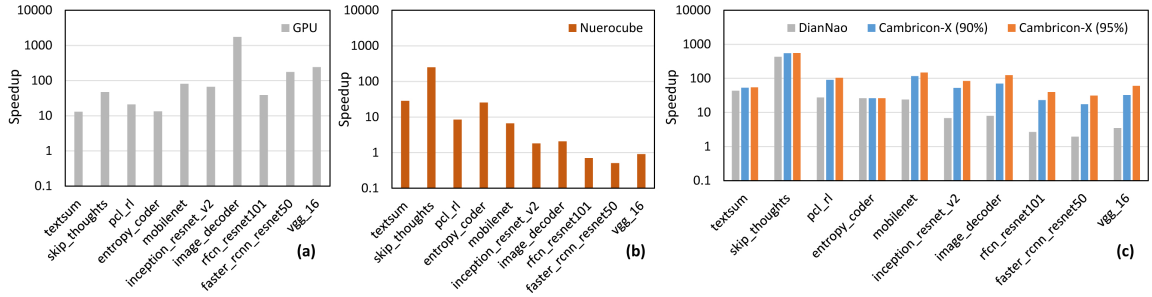


Figure 2.12: The speedups over CPU baseline of applications on (a) GPU without any model compression (b) Neurocube with models quantized into 16-bit fixed-point (c) DianNao with models quantized into 16-bit fixed-point, Cambricon-X (90%) with models further pruned 90% weights, and Cambricon-X (95%) with models further pruned 95% weights.

90% and 95% weights of models, denoted as Cambricon-X (90%) and Cambricon-X (95%), respectively.

Insights from the result. By evaluating three representative accelerator designs with various compression config:nnbench-x:uration, we make the following observations from Figure 2.12. First, GPU can benefit these applications with a higher R_2 ratio in their performance features. These applications are usually computation bound. Since applications on the x-axis are ordered by the increasing order of R_2, applications closer to the right direction along the x-axis spend more time in the second cluster operators, of which most are convolution and matrix multiplication operations. As shown in Figure 2.12a, GPU obtains higher speedups on applications on the right side of the x-axis. Second,

near-data computing architectures favor applications (mostly NLP related) with a higher R_3 ratio. Figure 2.12b shows that Neurocube achieves higher speedups on applications on the left side of the x-axis. Finally, *we found that weight pruning is less attractive for NLP applications than it is for CV applications.* Figure 2.12c shows the comparison of DianNao and Cambricon-X in terms of performance benefits from pruning NN model weights, which reduces the computation and memory workloads of matrix multiplication and convolution operations. Comparing Cambricon-X (90%) to DianNao, Cambricon-X can achieve higher speedups than DianNao, which mainly benefits from the reduction of computation and memory workloads due to pruned models. Such speedups are more significant for computation-bound applications as opposed to memory-bound applications. The results of models with different sparsities, Cambricon-X (90%) and Cambricon-X (95%), indicate that pruning more weights can have slight benefits on memory-bound applications while significant benefits on computation bound applications.

2.4 Discussion

Software-hardware co-design in MLPerf. Neural network applications, especially the inference stage, benefit from the hardware-software co-design methodology. Thus our work urges taking the whole software-hardware co-design solution as a benchmark object instead of benchmarking pure hardware designs by providing a fixed set of applications. The recently released MLPerf inference benchmark [51] includes an Open Division under the same motivation as our study, although they are a preliminary release and the rules of Open Division are immature. Compared to the immature rules in this preliminary release, our methodology provides a concrete interface to take the model compression techniques as the input, and generate the compressed models as the output. Our work takes model compression techniques as the software optimizations in the end-to-end methodology,

and our case studies reveal new insights for the impact of software optimizations on hardware designs. We still need to further refine stages in our methodology to embrace a larger scope of software solutions varying model architectures for the same prediction task, which is an important perspective of our future work.

Extensibility of our benchmark methodology. There are many configuration choices in our case study, which should be configured case by case. For example, we use locality and parallelism as operator features to capture various architecture designs. They are sufficient to indicate the overall architecture demand, such as compute-centric vs. memory-centric designs, because these two metrics are major considerations among different architecture designs to capture memory access patterns and computation intensity. However, these two metrics are not able to capture finer-grain locality and parallelism characteristics. When finer-grain operator characteristics are needed, the operator-level analysis phase needs to be adapted to new features, such as adding the reuse distance [52] to reflect the average distance between data reuses. Another example is adding new application features. We consider time breakdowns in application features because we think inter-operator parallelism is usually implemented in software frameworks, such as TensorFlow [28], for a higher flexibility of scheduling. However, when designing an accelerator taking the whole computation graphs as inputs and exploiting inter-operator parallelism at the hardware-level, the characteristics of computation graphs, such as the average of node degrees, can be added to the application features. In summary, configurations in our benchmark methodology are not fixed and some of them are tailored to our case study. We expect this benchmark methodology to be used by varying configurations case by case. Despite the change of configurations, such as adding reuse distance into the operator features, the key principles of our benchmark methodology, selecting applications quantitatively and benchmarking software-hardware co-designs, remain the same.

NNBench-X for new NN workloads. Because of the promising results from NN

techniques, there are new algorithms developed for challenges in various applications. In these fast-growing algorithm studies, our benchmark methodology is feasible to characterize new NN workloads to provide insights for accelerator designs. For example, Bayesian neural networks (BNNs) [53,54] attract attention due to its ability to deal with uncertainty during the estimation. In our benchmark methodology, we decompose BNN models into operators and go through the application set selection flow to understand their characteristics. The only class of operators in BNN models different from existing NN models in the TF Model Zoo is sampling to generate the learned distribution of weights. Because each element in weights is sampled independently from its distribution in BNNs, these sampling operators have 100% parallelism. Thus these operators belong to the third cluster in Figure 2.7b and their execution time contributes to R_3. BNN models with these sampling operators will have a larger R_3 than NN models with the same NN architecture. As a result, BNN models could have a larger demand on memory bandwidth, which is the same as most of the workloads bounded by R_3. If the performance of BNN models is significantly bounded by these sampling operators, efficient sampling implementations in the accelerator should also be considered [55]. These architectural implications will be helpful when designing new accelerators for BNN models.

2.5 Conclusion

In this work, we propose a novel end-to-end benchmarking method for NN accelerator designs. To select the most representative NN applications and evaluate software-hardware co-designs, our benchmark method is composed of three stages: application set selection, benchmark suite generation, and hardware evaluation. The application set selection stage selects representative NN applications according to quantitative metrics to ensure the diversity of the benchmark suite. The benchmark suite generation and

hardware evaluation stages refine the selected applications according to user-provided model compression techniques and evaluate the compressed models on accelerator designs. We conduct a study of benchmarking several state-of-the-art NN accelerator designs to demonstrate the usage of our benchmark method. We analyze applications from the TensorFlow Model Zoo and observe that applications from the same application domains have similar bottlenecks. Moreover, we evaluate several state-of-the-art software-hardware co-design solutions, including hardware designs for quantized and pruned NN models. From our case studies, we observe that computation-centric and memory-centric architectures can have different benefits for different application domains. Also, we find that pruning NN models provides little benefit to memory-bound applications. Through our case studies and observations, we are convinced that our benchmark method is practical and feasible to provide insightful guidance to NN accelerator designs.

Chapter 3

SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator

This chapter presents the first of three full-stack near-bank processing solutions in this dissertation. In particular, this near-bank processing solution is an application-specific accelerator for sparse matrix vector multiplication (SpMV). We first introduce the background of SpMV and the performance characterization study of SpMV on an NVIDIA GPU that is one of the state-of-the-art hardware platforms. Our profiling results reveal a high DRAM utilization, which indicates that SpMV has been well-optimized on GPU and the memory bandwidth becomes the bottleneck.

[1] Although near-bank processing architectures provide higher effective bandwidth compared to the traditional memory interface between processors and memory, there are

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

several challenges in accelerator designs. First, the memory latency to access data in other banks is much higher than in the local bank. The processing element (PE) design should hide such a high latency for fully utilizing the bank level bandwidth. Second, since interconnect bandwidth is much smaller than that of the memory bank, memory access should be kept as local as possible to ease the burden on the interconnect. Third, PEs near the memory banks have a strict area budget, which requires the compute logic to remain fairly simple, but effective. In addition, challenges of workload balancing and locality exploitation of the input vector also exist when distributing non-zero elements across PEs.

Our accelerator, SpaceA, is designed to overcome these challenges. To overcome the first challenge, each PE near the memory bank possesses a queue to hold the non-zero elements for processing and memory requests to input vector according to the column index of non-zero elements in this queue. Memory requests are non-blocking to hide the memory access latency to other banks by exploiting memory-level parallelism (MLP). To address the second challenge, content addressable memory (CAM) is integrated at the bank level to cache elements from the input vector so that the amount of memory access to other banks is reduced by exploiting the locality. This helps alleviate the bandwidth pressure on the TSVs. The third challenge related to the strict area budget is tackled by the fact that our PE design only includes a queue and a floating-point unit (FPU). Therefore, our PE occupies a very small area overhead, which makes it practical to be integrated near the memory banks. In addition to these design options to overcome hardware challenges, we develop a mapping scheme for SpaceA to distribute the non-zero elements of the sparse matrix across different memory banks to achieve workload balance among PEs and to exploit the locality of data from the input vector.

In summary, the contributions of this project are as follows:

- We design an accelerator, named SpaceA, to leverage outstanding memory requests to hide the memory access latency to non-local banks. To reduce the memory traffic to non-local banks, we integrate CAM buffers in SpaceA to exploit the locality of input vectors.
- We develop a mapping scheme for SpaceA to distribute the non-zero elements across different banks to achieve workload balance among PEs and to exploit the data locality of the input vector.
- Our evaluation of SpaceA with the proposed mapping scheme on matrices [56] from real-world applications reveals 13.5x speedup and 87.49% energy saving on average over the GPU baseline with only 4.86% area overhead. Additionally, our case study on graph applications demonstrates a better performance than state-of-the-art graph accelerators, Tesseract [57] and GraphP [58], because of the higher effective bandwidth provided by near-bank integration instead of placing compute-logic on the base die.

3.1 Motivation

3.1.1 SpMV Workloads

SpMV is a widely used operation in many applications that use algorithms based on a large amount iterations of matrix-vector multiplication in which the coefficient matrix is sparse. We denote an SpMV operation as $Y = Y + AX$ where X is the input vector, A is the input matrix, and Y is the output vector. We denote the dimensions of the input matrix as m and n, which indicate that the matrix has m rows and n columns. Additionally, we denote nnz as the number of non-zero elements. Each component of the output vector can be computed as $Y_i = Y_i + \sum_{j=0}^{n} A_{ij}X_j$ where Y_i is the i-th component

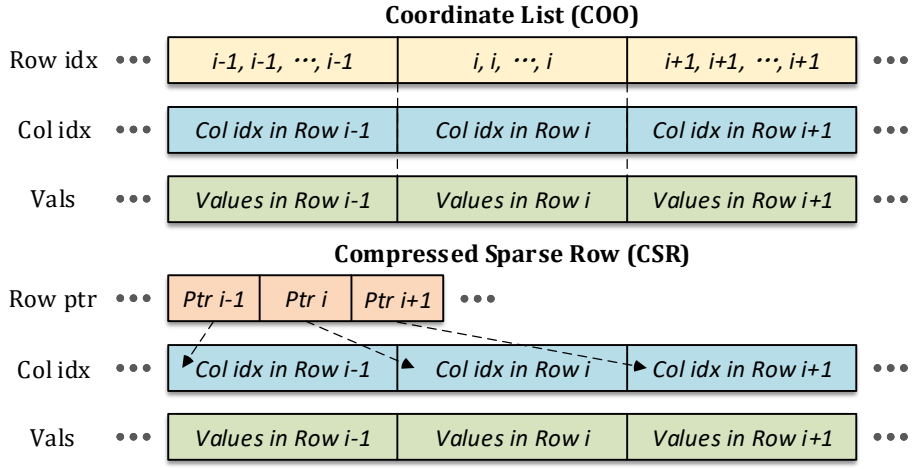


Figure 3.1: The compressed sparse row (CSR) format of a sparse matrix.

of vector Y, X_j is the j-th component of vector X, and A_{ij} is the element located in i-th row and j-th column of matrix A. For a sparse matrix, the computation of $A_{ij}X_j$ can be skipped for locations where $A_{ij} = 0$.

For highly sparse matrices, compressed storage formats such as Coordinate List (COO) and Compressed Sparse Row (CSR) store the non-zero elements efficiently and remove ineffective computation for the zero elements. The COO format is composed of three lists of length nnz. These three lists store the row index, the column index, and the value, respectively, for each non-zero element. The CSR format, on the other hand, consists of three arrays: 1) row ptr, 2) col idx, and 3) vals. Each entry in the row ptr array points to an entry in the col ids array which represents the beginning of the list of column ids containing non-zero elements in that row. The row ptr entry simultaneously points to the entry in the vals array which records the value of the non-zero elements. Figure 3.1 demonstrates how non-zero elements of i-th row are stored.

Compared to COO, CSR is more compact since it saves the memory space of row index array from the length of nnz to the length of $m + 1$. Therefore, CSR is the most widely used sparse matrix format and $csrmv()$ [59] is supported in almost all libraries

on multi-core and many-core architectures to compute SpMV. SpaceA is designed to perform SpMV based on the CSR format.

3.1.2 SpMV on GPU

The poor reuse opportunity in the sparse matrix and irregular memory access patterns make SpMV memory-bound on multi-core and many-core processors. Compared to the CPU, GPU provides higher memory bandwidth through the GDDR memory bus and exploits memory-level parallelism to hide long memory access latency. To understand the state-of-the-art implementation of SpMV on GPU, we profiled SpMV computation with a collection of real-world matrices from the University of Florida sparse matrix collection [56]. The names, application domains, and characteristics of these matrices are elaborated upon Table 3.1. For the implementation of SpMV on GPU, we use the library routine $csrmv()$ from the vendor-provided library cuSPARSE [59], which is a library optimized for sparse linear algebra operations on NVIDIA GPU. We measured the performance and profiled the DRAM metrics of SpMV on NVIDIA GPU, Titan Xp. DRAM read throughput is collected by $nvprof$. In addition, we measured effective read throughput which is computed as nnz times the size of a non-zero element over the measured execution time. Moreover, we compute the achieved GFLOPs of SpMV as nnz over the execution time, and the ALU utilization as the achieved GFLOPs over the maximum GFLOPs provided by GPU. Compared to the maximum DRAM bandwidth of Titan Xp, which is 547.8 GB/s, Figure 3.2 shows that the current average bandwidth utilization (as represented by the mean orange bar) of SpMV on GPU is 27.08% and 43.39% when excluding matrices 12, 13, and 14[2]. In addition, Figure 3.2 shows that the ALU utilization is only 2.68%. Figure 3.2 provides two important starting points

[2]Exceptions represent social networks and web graphs which show relatively poorer utilization of the DRAM bandwidth, in agreement with prior studies [60–62].

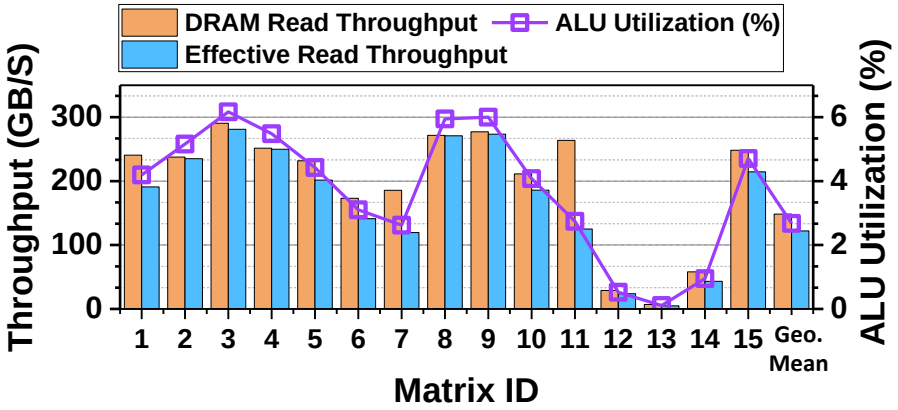


Figure 3.2: Profiling results of SpMV on GPU (The details of each matrix are listed in Table 3.1).

for our work. First, the small ALU utilization compared to the much larger DRAM bandwidth utilization demonstrates the memory-bound behavior of SpMV, motivating our PIM-based architecture. Second, the effective bandwidth utilization (represented by the blue bar) is close to the actual bandwidth utilization (represented by the orange bar), which indicates that actual hardware innovation (rather than algorithmic innovation to eliminate redundant DRAM accesses) is required for higher performance SpMV.

3.2 SpaceA Architecture

3.2.1 Overview

The architecture design of SpaceA is demonstrated in Figure 3.3. As shown in Figure 3.3(a) and 3.3(b), SpaceA is composed of several 3D stacked memory cubes connected in a memory network. To exploit bank-level memory bandwidth, SpaceA integrates a PE near every memory bank. The input/output vectors are evenly partitioned and stored in memory banks on the DRAM layer just above the base logic die, whereas the sparse matrix is statically distributed by the mapping algorithm (Section 3.3) on all the other DRAM layers. The separation of the storage allows each PE to process the sparse matrix

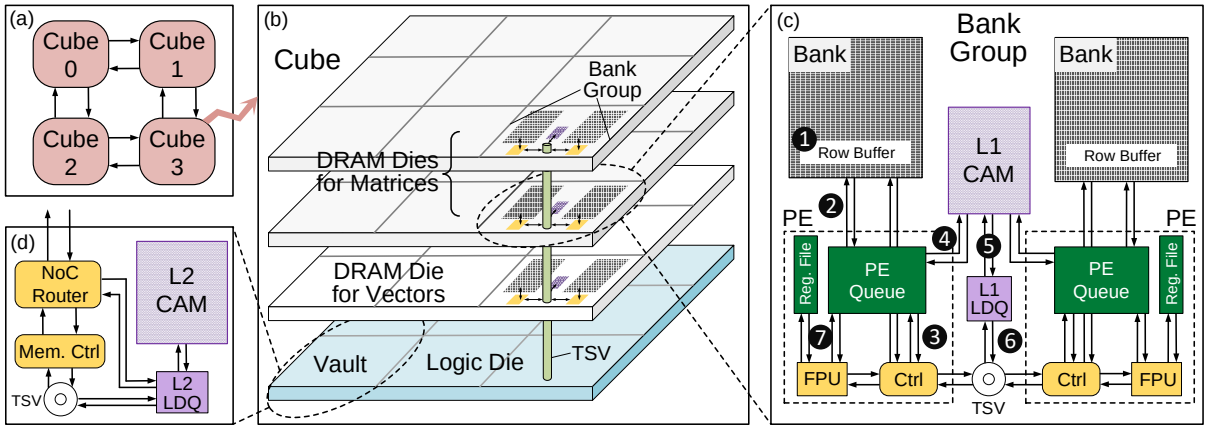


Figure 3.3: SapceA architecture design: (a) memory cubes connected through memory network, (b) the overview of a cube, (c) components in a bank group, and (d) components in a vault controller.

in a streaming manner to maximize the read bandwidth. In addition, on the DRAM die for vectors, the elements with the same index from input and output vectors are stored in the same memory bank. This is because, in an iteration of SpMV, the output of i-th iteration is the input of $i+1$-th iteration. Therefore, this storage scheme can eliminate data movement between iterations for input and output vectors.

3.2.2 PE Design

In SpaceA design, there is a PE dedicated for each memory bank. Since matrix and vector data are separated into memory banks on different dies, PEs attached to these memory banks have different functionalities. The PE of memory banks storing the sparse matrix computes partial dot-product results, while the PE of memory banks storing vectors accumulates the partial results which is finally stored into the output vector. We denote the first type of PE as **Product-PE** and the second type of PE as **Accumulation-PE**. Although these two types of PEs have different functionalities, they can be realized by the same set of hardware components. The hardware components of a

bank group are shown in Figure 3.3(c). Following is the description of how these components are designed for the Product-PE and how they are used for the Accumulation-PE.

Product-PE: Product-PE is responsible for processing non-zero elements of the sparse matrix in its local memory bank. After t_{RAS} cycles, a DRAM row will be loaded into the row buffer of the memory bank (Figure 3.3(c)-❶). Using the similar idea of CSR matrix format, when distributing non-zero elements into memory banks, the mapping algorithm aligns the number of non-zero elements of a row into the size of a DRAM row. This alignment causes non-zero elements of the same DRAM row to end in the same row index in the original sparse matrix. As a result, when storing non-zero elements in a DRAM row, the leading 4 bytes are used to indicate the row index of non-zero elements in this DRAM row, and the rest of the space is used to store pairs of column index and value.

As shown in Figure 3.3(c)-❷, non-zero elements from a DRAM row buffer are pushed into a PE queue if the PE queue is not full. The PE queue is physically realized with scratchpad memory while the control logic in Product-PE accesses elements inside it as a logical cyclic queue. For each non-zero element in the PE queue, it needs to compute the partial result $A_{ij}X_j$ where the row index i, column index j, and value A_{ij} are already known. The control unit scans the PE queue in a cyclic manner when it is not empty, and then processes a non-zero element every L_p cycle (Figure 3.3(c)-❸). For each unprocessed non-zero element, it will check whether X_j is ready in the register file.

Case I: X_j is not ready: When X_j is not ready, the Product-PE needs to read it from other memory banks because vectors are stored separately from the matrix. The access latency to a remote bank is significantly larger than the access to its local memory. To exploit the locality of the input vector, SpaceA integrates an L1 CAM for PEs in the same bank group. This L1 CAM provides a key-value store so that it can help to search the value X_j according to the column index j (Figure 3.3(c)-❹). When the access to

L1 CAM misses, it will return a miss signal which indicates that the remote access is unavoidable. To hide the latency of remote access, the control unit will continue to process the next element in the PE queue instead of waiting for the value X_j. Since this is a logical cyclic queue, the control unit will access this non-zero element again after scanning the rest of non-zero elements in the PE queue. Meanwhile, L1 CAM will send the requested column index j to the load queue (LDQ) (Figure 3.3-❺) to remove the duplication of data requests. If this column index has not been requested yet, it will send out the request through TSV (Figure 3.3(c)-❻). When the requested value X_j comes back, it will be written into both L1 CAM and register file. The corresponding load request j will be removed from the load queue. Since the control unit repeatedly iterates through all elements in the PE queue, X_j will become ready when the control unit accesses it again after the requested value comes back.

Case II: X_j is ready: When X_j is ready in the register file, it will send A_{ij}, X_j, and the current partial result Y_i to Floating-point Unit (FPU) for computing $Y_i = Y_i + A_{ij}X_j$ (Figure 3.3(c)-❼). After accumulating the partial result into Y_i, the non-zero element is labelled as processed. When all of non-zero elements from the same DRAM row in the front of the PE queue are processed, they are popped out of the queue and the control unit moves the front pointer of the queue. The granularity for popping non-zero elements is the same size of a DRAM row buffer so that the whole row buffer of new data can be pushed into the PE queue, and checked as to whether the row index of this new row is different from the existing row index. The partial result Y_i is flushed out through TSV when the new row index is different from the existing row index.

Accumulation-PE: Bank groups with Accumulation PE serve two purposes. First, since the memory banks of this bank group store some elements of the input vector, it will respond the value X_j according to the requested column index j. For this purpose, the request first goes to L1 CAM, and then goes to the memory bank if X_j is not in the

L1 CAM (CAM miss). This part only needs the help of the memory bank, L1 CAM, and control unit. Second, since the memory banks of this bank group store some elements of the output vector, they need to accumulate partial results Y_i. To achieve this purpose, the SRAM of the PE queue is used to realize an update buffer where the elements of the output buffer are stored. When Y_i comes, it will first look up the output buffer by the row index stored in the register file. If corresponding output elements are not in the update buffer, it will be loaded into the update buffer from the memory bank. Then existing Y_i and new partial result Y_i will be accumulated with the help of the FPU. When the update buffer is full, it will write the logical first row back to the memory bank, and load a new row containing Y_i from the memory bank.

3.2.3 Vault Controller

The components of a vault controller on the base die are shown in Figure 3.3(d). In addition to the existing NoC router for inter-vault communication and the memory controller to read and write memory banks attached to the same TSVs, SpaceA integrates a L2 CAM and a corresponding load queue to exploit the locality of the input vector in the communication path between bank groups. There are three types of packets a vault controller could potentially process.

Type I: X_j request. When the vault controller receives the request for the value of X_j, it will first look up the L2 CAM according to the column index j. If X_j exists in L2 CAM, the vault controller will generate a response packet with the value of X_j, and send it back to the source of the request packet, either by NoC router to other vaults or TSV to bank groups attached to the same TSV. If X_j does not exist in L2 CAM, it will look up the load queue (LDQ) to remove duplicated requests for X_j. The vault controller will forward this request to the bank group storing X_j according to the column index j by

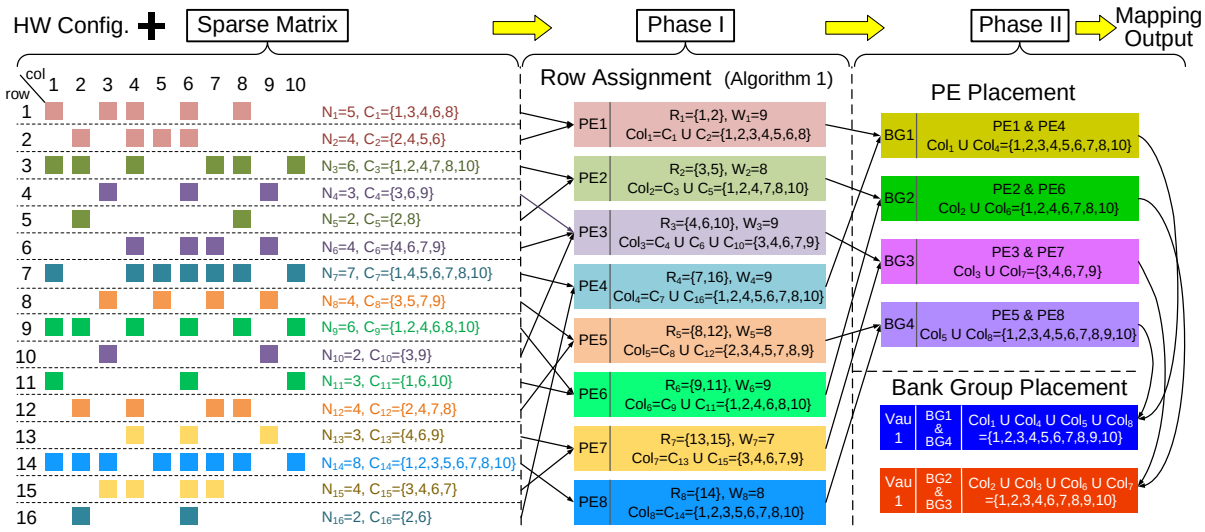


Figure 3.4: The flow of our mapping algorithm which is composed of two phases: row assignment to logical PEs (Phase I) and PE placement to bank groups and vaults (Phase II).

either the NoC router or the TSV.

Type II: X_j response. When the vault controller receives the response for the value of X_j, the vault controller will have the same logic as the Product-PE hearing back the value of X_j. Besides forwarding this packet to its destination, the vault controller will write the value X_j into its L2 CAM and remove the corresponding entry in the load queue.

Type III: Y_i partial result: The vault controller for partial result Y_i will forward it to the corresponding vault storing Y_i according to the row index i. If Y_i is stored in the same vault, it will forward it to the bank groups on the bottom of DRAM dies by TSVs so that the partial result of Y_i can be accumulated with the help of the Accumulation-PE.

3.3 Mapping Method

Algorithm 3 Row assignment to logical PEs.

Init $\overline{nnz} = \frac{nnz}{\#PEs}$
Init K_p = a large constant value
for $pid = 0$ to $\#PEs$ **do**
  Init the set of assigned rows, $R_{pid} = \emptyset$
  Init the set of unique column indexes, $COL_{pid} = \emptyset$
  Init the number of assigned non-zero elements, $W_{pid} = 0$
end for
for $i = 0$ to m **do**
  N_i: the number of non-zero elements in i-th row
  C_i: the set of column indexes in i-th row
  for $pid = 0$ to $\#PEs$ **do**
    if $W_{pid} + N_i > \overline{nnz}$ **then**
      $Score_{pid} = -(W_{pid} + N_i - \overline{nnz}) \times K_p$
    else
      $Overlap = |C_i \bigcap COL_{pid}|$
      $Score_{pid} = max\{\frac{Overlap}{N_i}, \frac{1}{W_{pid}+N_i}\}$
    end if
  end for
  $maxID$ = the pid with highest $Score_{pid}$
  $R_{maxID} = R_{maxID} \bigcup \{i\}$
  $COL_{maxID} = COL_{maxID} \bigcup C_i$
  $W_{maxID} = W_{maxID} + N_i$
end for

3.3.1 Overview

The proposed mapping method distributes the non-zero elements of a sparse matrix into the memory banks of SpaceA for Product-PE to process. There are two overall metrics to efficiently use SpaceA hardware: 1) workload balance and 2) locality. First, since all PEs process non-zero elements in parallel, the performance is bounded by the slowest PE, which requires workload balance among PEs. Second, since SpaceA integrates L1 CAM at the bank group level and L2 CAM at the base die of each vault to mitigate the latency of accessing the data of input vector, non-zero elements assignment should consider the column index locality of non-zero elements to leverage L1 and L2 CAM for keeping access local.

To achieve workload balance and leverage the locality of the input vector, we design the overall mapping pipeline shown in Figure 3.4, which takes the hardware configuration of SpaceA and the sparse matrix as the input. As shown in Figure 3.4, the first phase assigns different rows to PEs, which are logical PEs without any physical location information. In this phase, we exploit the intra-PE locality by assigning the rows of non-zero elements with similar column index pattern to the same PE. Meanwhile, this phase also balances the number of non-zero elements assigned to PEs. The algorithm used in this phase is further introduced in Section 3.3.2. In the second phase, we place all logical PEs into the physical location in SpaceA. This phase clusters PE workloads with similar sets of column index from the non-zero elements, and minimizes the maximal number of unique column indexes across bank groups and vaults to achieve workload balance. The formulated optimization problem is detailed in Section 3.3.3.

3.3.2 Logical PE Workload

The first phase assigns multiple rows of the sparse matrix into PEs, each of which is considered equivalent. In this phase, we balance the workload among PEs and maximize the intra-PE locality. Algorithm 3 shows the scheme, which iterates through all rows and determines which PE is the best to be assigned for a specific row according to the current assignment of previous rows. The metric used to determine the best PE for processing this row is designed according to the following two principles. First, if the current row i assigned to the current PE *pid* makes the current PE process the number of elements larger than $\overline{nnz}$, we add a penalty for the number of elements exceeding this budget. The budget $\overline{nnz}$ is computed as nnz over the total number of logical PEs. When each PE processes $\overline{nnz}$ elements, the workloads of PEs are perfectly balanced. Second, the column index overlap between non-zero elements of the current row i and the non-zero

elements of existing rows assigned to the current PE *pid* is computed. In case of an overlap, the overlap ratio is taken as the score, which is the number of overlap non-zero elements over the number of non-zero elements of row i. When there is no overlap, the factor one over the number of non-zero elements assigned to the current PE is taken as the score. This score rating metric means we optimize locality first and the workload balance when the number of non-zero elements does not exceed the given budget $\overline{nnz}$. After computing the score of each PE, the row i is assigned to the PE with the highest score.

Although mapping the sparse matrix optimally to logical PEs is an NP-hard problem, our mapping heuristic is feasible in terms of time complexity. We denote P as the number of PEs, W_{pid} as the number of non-zero elements assigned to PE *pid*, and N_i as the number of non-zero elements in the row i. Each row needs the time complexity of $O(N_i \sum_{pid=1}^{P} logW_{pid})$. Since W_{pid} is always smaller than nnz (the total number of non-zero elements), the upper bound of the time complexity of a row assignment can be simplified as $O(N_i Plognnz)$. Summing up the time complexity of assigning all rows, since $\sum_{i=1}^{m} N_i = nnz$, the time complexity for finishing all row assignments has an upper bound $O(P \times nnzlognnz)$. This time complexity is scalable in terms of the number of PEs and the number of non-zero elements. Therefore, the algorithm of this phase is practical enough, and its effectiveness is further demonstrated in Section 3.4.3.

3.3.3 Logical PE Placement

In this phase, each logical PE is placed into the position of a physical PE. We decouple this phase into two stages. First, logical PEs are clustered into bank groups. Second, bank groups are clustered into vaults. To achieve locality and workload balance, this phase minimizes the maximal number of unique column indexes across bank groups and

vaults when clustering banks and bank groups. Both stages, clustering logical PEs and bank groups, represent similar problems in terms of problem structure and optimization target. Therefore, we abstract the problem of both stages as follows. Given p sets $S_1, S_2, ..., S_p$, we divide them evenly into q groups, and each group has the same number of sets k where $p = kq$. Therefore, we denote C_{gw} as the w-th set assigned to the group g. The value of C_{gw} should be one of the values between 1 and p. To optimize locality, we want sets assigned to the same group to have a larger overlap. Locality indicates the preference to assign sets with a larger number of overlap while workload balance implies that the maximal number of unique elements should be minimized. The problem is formulated as Formula 3.1 where $F(C)$ stands for the maximum number of unique elements across all groups under the assignment C.

$$\begin{aligned} \underset{C}{\text{minimize}} \quad & F(C) \\ \text{subject to} \quad & F(C) = \max_{1 \le g \le q} \{| \bigcup_{w=1}^{k} S_{C_{gw}} |\}, \\ & C_{gw} \in \{1, 2, ..., p\},\ \forall 1 \le g \le q,\ 1 \le w \le k \\ & C_{g_1 w_1} \ne C_{g_2 w_2},\ \forall (g_1, w_1) \ne (g_2, w_2) \end{aligned} \tag{3.1}$$

In the first stage, p equals the number of logical PEs and q equals the number of bank groups while in the second stage, p equals the number of bank groups and q equals the number of vaults. The formulated problem is also an NP-hard problem, thus we use a heuristic algorithm similar to Algorithm 3 to solve it. The effectiveness of the mapping algorithm is quantitatively shown in Section 3.4.3.

3.4 Evaluation

In this section, we first introduce the experimental setup in Section 3.4.1. Next, we detail the overall performance, power, and area results of our design compared to state-of-

Table 3.1: The information of sparse matrices used to evaluate SpMV on GPU and SpaceA. The number of non-zero elements (nnz), average number of non-zero elements per row (μ), and the standard deviation of the number of non-zero elements in each row (σ) are shown to reflect the pattern of non-zero elements distribution.

ID	Matrix	Domain	Dimensions	nnz	μ	σ
1	bcsstk32	Structural Problem	44609 x 44609	2014701	45.16	15.48
2	cant	2D/3D Problem	62451 x 62451	4007383	64.17	14.06
3	consph	2D/3D Problem	83334 x 83334	6010480	72.13	19.08
4	crankseg_2	Structural Problem	63838 x 63838	14148858	221.64	95.88
5	ct20stif	Structural Problem	52329 x 52329	2600295	51.57	16.98
6	lhr71	Chemical Process Simulation Problem	70304 x 70304	1494006	21.74	26.32
7	ohne2	Semiconductor Device Problem	181343 x 181343	6869939	61.01	21.09
8	pdb1HYS	Weighted Undirected Graph	36417 x 36417	4344765	119.31	31.86
9	pwtk	Structural Problem	217918 x 217918	11524432	53.39	4.74
10	rma10	Computational Fluid Dynamics Problem	46835 x 46835	2329092	50.69	27.78
11	shipsec1	Structural Problem	140874 x 140874	3568176	55.46	11.07
12	soc-sign-epinions	Directed Weighted Graph	131828 x 131828	841372	6.38	32.95
13	Stanford	Directed Graph	281903 x 281903	2312497	8.20	166.33
14	webbase-1M	Weighted Directed Graph	1000005 x 1000005	3105536	3.11	25.35
15	xenon2	Materials Problem	157464 x 157464	3866688	24.56	4.07

the-art SpMV implementations on GPU in Section 3.4.2. Section 3.4.3 demonstrates the advantages of our proposed mapping methods. Section 3.4.4 shows the sensitivity studies of SpaceA performance to hardware configurations. Section 3.4.5 studies the scalability of SpaceA design. Finally, we conduct a case study of using SpaceA to accelerate graph analytics to show its potential for benefiting applications built on SpMV.

3.4.1 Evaluation Methodology

Workload. We evaluate SpaceA by executing SpMV using fifteen real-world matrices from various application domains including scientific computing and graph analytics. These matrices come from the University of Florida collection [56], and they are used in prior studies for accelerating SpMV on GPU [63] and Intel Xeon Phi processors [64]. In terms of the distribution of non-zeros, these matrices cover both structural patterns (i.e. a smaller standard deviation of the number of non-zeros in each row) and non-structural patterns (i.e. a larger standard deviation of the number of non-zeros in each row). The details of these matrices are listed in Table 3.1.

Hardware Configuration. We adopt an HMC-like [65] design to realize the architecture design of SpaceA. The rest of the evaluation results assume an HMC-like architecture, a detailed further discussion between HMC and HBM technology can be found in Section 3.6. We use an HMC configuration specified in the prior HMC characterization study [66]. Specifically, a memory cube has 16 vaults that use 1024 TSVs running at the bit rate of 2 Gbps to communicate with 8 stacked DRAM die layers. Each bank group has 2 banks; each bank has a capacity of 128 Mb with a 2 Kb row buffer. Therefore, there are 256 memory banks in a memory cube with a total of 4 GB capacity, and a memory cube has a footprint of $48mm^2$. We use NVIDIA Titan Xp as a representative of GPU architecture for comparison which has processors with a die size 471 mm^2, an area equivalent to that of 10 cubes. We assume that the area of GPU DRAM dies is comparable to processors in Titan Xp, thus the default configuration of SpaceA uses 16 cubes, occupying 768 mm^2 – a similar area footprint as Titan Xp. Inside each PE, there is a 16 Kb scratchpad memory for the PE queue, which enables the PE to process non-zero elements from 8 DRAM rows concurrently. Register file has the same size as the number of non-zero elements stored in a PE queue. To support double-precision SpMV in scientific computing, each PE includes a floating-point unit (FPU). PEs from the same bank group share an L1 CAM with 32 sets and 4 ways per set. Each way in L1 CAM has 32 bytes, which is equivalent to the size of 4 input vector elements. The configuration of the number of ways per set and the size of each way in L2 CAM is the same as L1 CAM for simplicity, whereas L2 CAM has a larger number of sets, which is 2048 by default. The size of L1 and L2 CAM are 4 KB and 256 KB respectively. The load queues for L1 and L2 CAM are used to remove duplicate requests, and they are realized with fully associated CAM which have the sizes of 512 and 8192 elements respectively. The default configuration of L1 and L2 CAM is an intuitive design point; a detailed sensitivity for L1 and L2 CAM will be demonstrated in Section 3.4.4 to further justify our design point.

Simulation Method. We develop an event-based in-house simulator for the performance and power simulation. The performance simulation is based on triggering events according to the behavior of each hardware component described in Section 3.2. The triggered events are simulated to happen after a deterministic latency of the event triggering it which is based on the latency model of each hardware component. The events in the performance simulator cover FPU computation, the read/write to DRAM banks, on-chip SRAM (register file, PE queue, L1 CAM, L1 load queue, L2 CAM, and L2 load queue), TSV, and NoC packet transfer. Additionally, our simulator maintains a data structure tracking values stored in DRAM banks and on-chip SRAM when simulating each event, and some events will modify this data structure. At the end of the simulation, the correctness of the event triggering mechanism is validated by the values of the output vector.

After validating the event triggering mechanism, the fidelity of our performance simulation relies on the latency of each event. Therefore, we use an existing well-validated simulator CACIT-3DD [67] and tape-out FPU design [68] to provide the latency model of each hardware component. Specifically, CACTI-3DD provides the access latency for DRAM banks, on-chip SRAM, and data transfer via TSV. Prior work [68] provides the latency of the FPU design.

Our event-based simulator logs a detailed event trace including read/write transactions to DRAM banks and on-chip SRAM, TSV data transfer, and FPU computation. Meanwhile, CACTI-3DD provides the energy consumption for each read/write transaction, TSV data transfer, and the static power of these components. FPU design [68] provides both dynamic and static power. Finally, we estimate the total energy consumption by accumulating the energy needed for each activity and the energy spent in the static power.

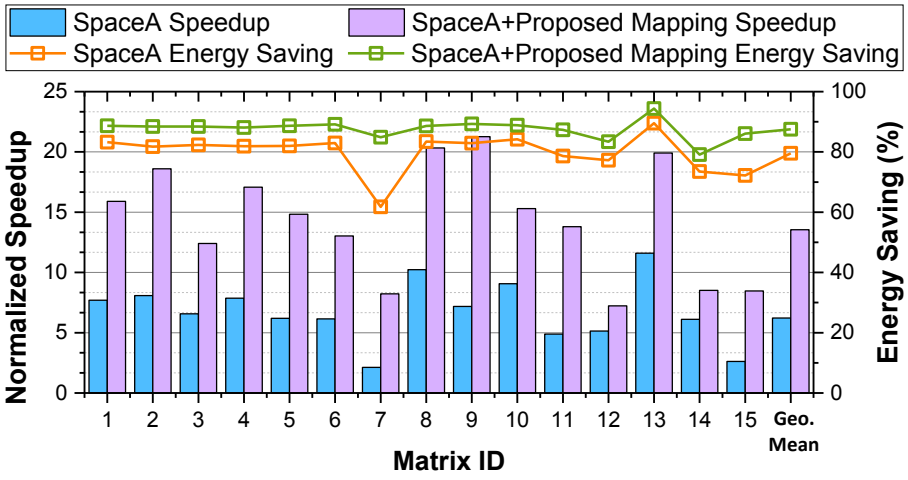


Figure 3.5: Overall speedup and energy savings w.r.t GPU.

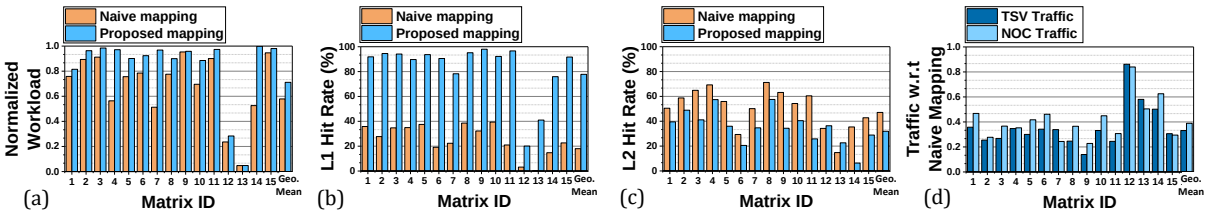


Figure 3.6: Performance metric comparisons between the naive mapping and our proposed mapping: (a) normalized workload, (b) L1 CAM hit rate, (c) L2 CAM hit rate, and (d) traffic between bank groups and vaults (TSV and NoC) with respect to that of the naive mapping.

3.4.2 Overall Performance, Power, and Area

Figure 3.5 shows the performance and energy efficiency for both our architecture design and the proposed mapping algorithm. As shown in Figure 3.5, the architecture design of SpaceA obtains 6.22x speedup and reduces the energy consumption by 4.89x (79.55% energy saving) on average compared to the GPU baseline. The results of SpaceA shown in Figure 3.5 uses a naive mapping which randomly assigns rows from the sparse matrix to PEs, so the performance and energy efficiency benefits mainly come from the advance of the architecture design. Figure 3.5 also demonstrates the overall performance energy efficiency with the proposed mapping method. SpaceA with the proposed mapping

Table 3.2: The area and power density of components in a bank group.

Component	Area	Power Density
PE Queue (x2)	0.0048 mm^2	43.75 mW/mm^2
Register File (x2)	0.0058 mm^2	49.66 mW/mm^2
PE Logic (x2)	0.0994 mm^2	28.21 mW/mm^2
L1 CAM (4 KB)	0.0286 mm^2	66.56 mW/mm^2
L1 Load Queue	0.0072 mm^2	56.29 mW/mm^2
Total / Peak	0.1458 mm^2	66.56 mW/mm^2

achieves 13.54x speedup and reduces 7.99x energy consumption (87.49% energy saving) on average compared to the GPU baseline. The comparison between the results of SpaceA using two mapping methods reveals that our proposed mapping method contributes 2.18x speedup and saves 1.63x energy consumption over the naive mapping method.

We estimate the area of the hardware components needed by SpaceA in addition to the existing HMC memory with CACTI-3DD [67] and an existing FPU design [68]. These hardware components are assumed to be fabricated in the 22 nm technology. According to prior studies [11], the area of compute-logic fabricated in the DRAM process could be up to 2x larger than the one fabricated in the CMOS process due to the less number of metal layers. Thus we multiply all area results from CACTI-3DD and existing FPU design by 2x to estimate the area of these components in the DRAM process. The area of hardware components in a bank group is shown in Table 3.2. As shown in Table 3.2, SpaceA only has an area overhead of 0.1458 mm^2 on the bank group level, which is only 4.86% of the area of a bank group and 5.96% of the area of memory banks. Thus the design of SpaceA has very little area overheads when integrating PE with memory banks. We estimate the area of L2 CAM and L2 load queue which reside on the base die in a similar way. In the default configuration, the area of an L2 CAM is 0.1898 mm^2 and the area of an L2 load queue is 0.0760 mm^2. The area of these two components is 0.2658 mm^2 in total, which is 8.86% area of a vault. The base die in the vanilla HMC memory has 10% to 30% area budget where other prior work integrates compute-logic [57, 69].

As long as the area of components on the base die does not exceed this budget, these components do not introduce any area overhead. In our work, we conservatively assume that the area budget on the base die is only 10%, thus the area of our L2 CAM and load queue is still within such a conservative area budget.

Recent research studies [11] and industrial prototypes [14] have demonstrated the feasibility of fabricating compute-logic in the DRAM process. However, the thermal issue is still a well-known challenge for PIM architecture based on 3D memory [70, 71]. We demonstrate the power density of components on DRAM dies in Table 3.2. As shown in Table 3.2, the peak power density per footprint is 532.48 mW/mm^2 (66.56 mW/mm^2 x 8 layers), which is under the constraint of power density from both commodity server active cooling [72] (706 mW/mm^2) and high-end server active cooling [71] (1214 mW/mm^2).

3.4.3 Mapping

In this section, we discuss performance metrics and power breakdown in detail in order to gain a better understanding of the source of these performance and energy efficiency benefits from our proposed mapping method.

Workload Balance. Since the performance of SpMV in SpaceA is bounded by the slowest PE, one goal of the proposed mapping method is to balance workloads among PEs. To quantify the workload balance, we do the following. First, we define the amount of work done by a PE to be the number of non-zero elements processed by it. Next, we define *normalized workload*, which indicates the ratio of the average amount of work done across all the PEs and the maximum amount of work done by any single PE. We use the normalized workload to represent the quantitative metric for workload balance (higher the better). The choice of the denominator in this ratio calculation is explained by the fact that workload balance is bottlenecked by the slowest PE, i.e., the PE which does the

largest amount of work. In the ideal case where non-zero elements are evenly distributed among PEs, the normalized workload should equal one due to the equivalence between the average and the maximum PE workloads. The difference of normalized workload between the naive mapping and the proposed mapping is shown in Figure 3.6(a). Figure 3.6(a) shows that the normalized workload of the naive mapping is only 81% of that of the proposed mapping on average, which indicates that the maximum PE workload in the proposed mapping is only 81% of that in the naive mapping. The smaller maximum PE workload demonstrates a better workload balance in the proposed mapping.

Locality Improvement. In the flow of our proposed mapping method, we consider locality optimization. To demonstrate the locality improvement in the proposed mapping, we profile the hit rate of both L1 CAM and L2 CAM, the traffic on TSV for intra-vault communication, and the traffic on NoC for inter-vault communication. Since intra-vault communication through TSVs has a uniform latency while inter-vault communication through NoC has non-deterministic latency, we define the traffic of TSV as the amount of data transferred through TSVs and the traffic of NoC as the size of a packet multiplied by the distance between the source and the destination of the packet. Figure 3.6(b)-(d) demonstrate these profiling results. Overall, Figure 3.6(d) shows that the traffic on TSV and NoC is only 33.11% and 38.89% with respect to that of the naive mapping, which indicates a significant amount of communication savings resulted from the improvement of the locality. In details, Figure 3.6(b) shows that the proposed mapping improves the average L1 CAM hit rate of all L1 CAMs significantly from 18% and 78% on average while Figure 3.6(c) shows that the L2 CAM hit rate decreases in the proposed mapping from 47.09% to 31.93%. The main reason for the decreasing L2 CAM hit rate comes from the reduction of requests to L2 CAM with the same amount of cold miss. As a result, the saving of NoC traffic is less than the saving of TSV traffic.

Energy Breakdown. To understand the energy efficiency between the naive map-

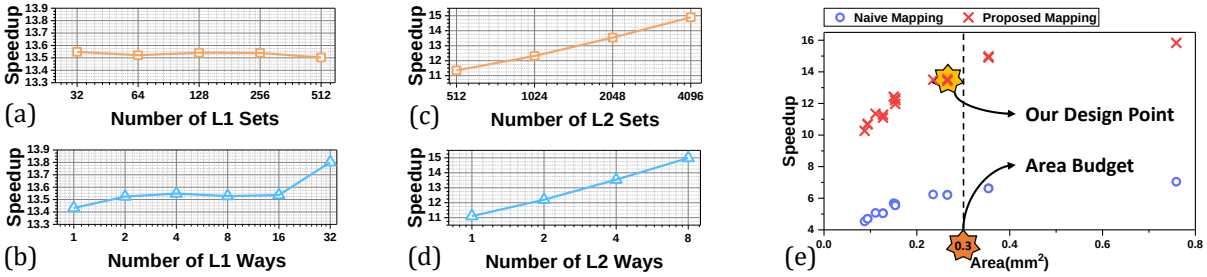


Figure 3.7: The sensitivity of performance to (a) the number of L1 sets, (b) the number of L1 ways, (c) the number of L2 sets, and (d) the number of L2 ways. (e) The trade-off between performance and area in L2 CAM design.

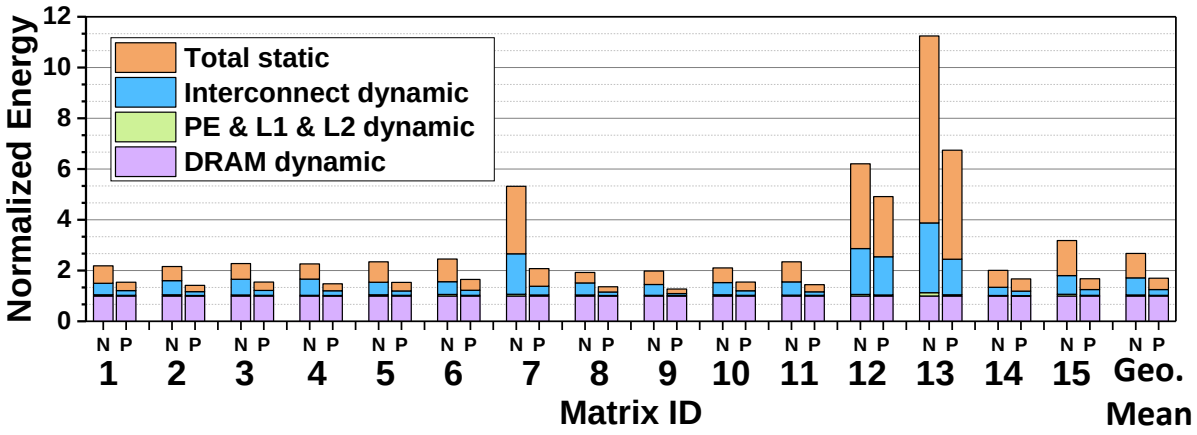


Figure 3.8: The energy consumption breakdown of SpaceA for the naive mapping (denoted as N) and our proposed mapping (denoted as P).

ping and the proposed mapping method, we demonstrate the energy consumption breakdown for these two mapping methods. We normalize the energy consumption of different parts into the energy consumption of DRAM dynamic power mapped by the naive mapping. We divide the overall energy consumption into four parts. The first part is the DRAM dynamic power. The second part is the dynamic power of PE, L1 CAM with its load queue, and L2 CAM with its load queue. The third part is the dynamic power of interconnect, which includes TSV and NoC. The last part is the static power of the whole chip. The energy breakdown of these four parts for the naive mapping and the proposed mapping is shown in Figure 3.8. We have several observations from Figure 3.8. First,

the dynamic power of hardware components added by SpaceA design is negligible (PE & L1 & L2 dynamic). Second, 65.55% on average of the dynamic power of interconnect is saved by the proposed mapping, which is the result of a reduced traffic amount on TSV and NoC shown in Figure 3.6(d). Finally, the proposed mapping method saves 54.05% energy consumption of the static power part: the result of improved performance. The static power dominates the overall energy consumption in matrix 7, 12, and 13. The energy consumption of static power is saved in the proposed mapping method of the matrix 7 due to a 3.87x speedup over the naive mapping. Matrix 12 and 13 have a relatively poor access pattern, thus pushing heavy traffic in the interconnect and resulting in a long execution time while DRAM banks and PEs are idle in most of the cycles.

3.4.4 Sensitivity Study

We conduct sensitivity studies for L1 CAM, L2 CAM, and TSV transfer latency to justify the selected design points in SpaceA architecture.

L1 and L2 CAM Sensitivity Study. We study the performance sensitivity of L1 and L2 CAM by varying either the number of sets or the number of ways. The average speedups compared to GPU for different numbers of sets and different numbers of ways in L1 and L2 CAM are shown in Figure 3.7(a)-(d). As shown in Figure 3.7(a) and (b), the performance of SpaceA is not sensitive to the size of L1 CAM. Although varying the number of ways could help the average speedup from 13.43 to 13.80, the benefit from such a large number of ways is relatively insignificant. Therefore, we keep the number of sets as small and the number of ways as large, resulting in the design with an L1 CAM composed of 32 sets and 4 ways per set. As shown in Figure 3.7(c) and (d), the performance is moderately sensitive to L2 CAM settings. Although the changes of speedup are not significant, these speedup changes are still noticeable, from

11x to 15x, among different CAM settings. Since L2 CAM can be as large as within the area budget, we study the trade-off between the performance and L2 CAM area as shown in Figure 3.7(e). Figure 3.7(e) shows that a larger L2 CAM usually result in a better speedup. Thus we select the largest one under our area budget, 10% area of a vault. Figure 3.7(e) also shows that our proposed mapping algorithm can leverage a smaller L2 CAM while achieving a better performance compared to the naive mapping. The naive mapping with an L2 CAM as large as 0.76 mm^2 achieves only a 68.61% speedup of the proposed mapping with an L2 CAM as small as 0.09 mm^2. The results further demonstrate the advantage of our proposed mapping method in terms of efficient hardware resource usage.

TSV Sensitivity Study. Most of PIM architecture design based on 3D memory technology leverages the low latency of TSV data transfer. We conduct a sensitivity study for TSV latency by varying the latency setting in our performance simulator. Figure 3.9 shows the performance slowdown of different TSV data transfer latency. Figure 3.9 shows that there is little difference between the latency of 1 cycle or 2 cycles for most of matrices. For the scenario where TSV transfer is 4 cycles, some matrices are not affected significantly (within 10% performance slowdown) while some matrices exhibit significant performance slowdown up to 2x. Thus the average slowdown of the performance is 1.3x, a factor which can hardly be ignored. When the TSV transfer latency is increased to 16 cycles, the performance incurs a 2x slowdown on average. In summary, our design is not sensitive to the TSV latency when it is low enough while the performance of design will start to degrade when the TSV latency is large enough, which justifies the reason for a design based on 3D memory technology bringing the low latency of TSV transfers.

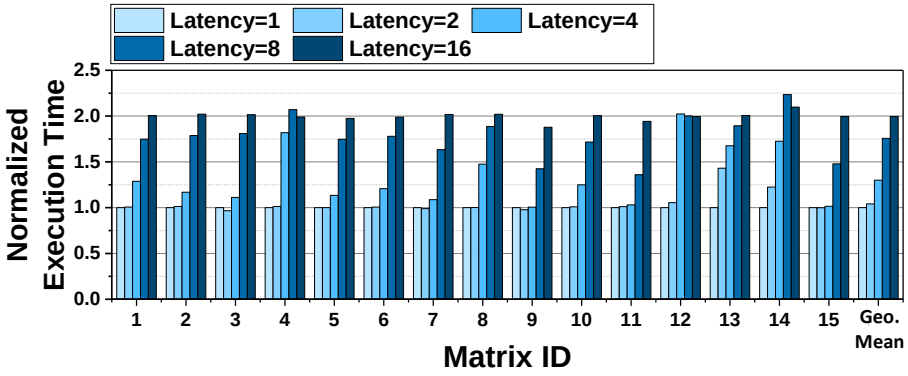


Figure 3.9: The sensitivity of performance to TSV transfer latency.

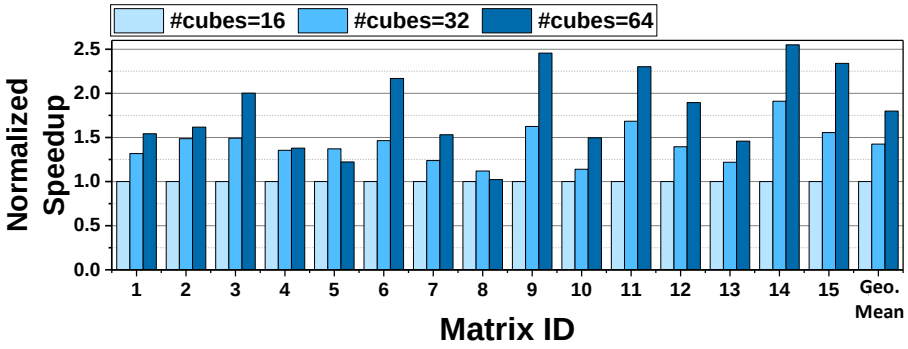


Figure 3.10: The scalability of SpaceA with the increase of the number of cubes.

3.4.5 Scalability

We show the scalability of SpaceA by increasing the number of cubes in Figure 3.10. Figure 3.10 shows that SpaceA with 32 cubes achieves 1.42x speedup and SpaceA with 64 cubes achieves 1.8x speedup on average compared to the default configuration. These results reveal moderate scalability where overheads come from a more expensive inter-vault communication with an increase of cube amount. Although the scalability of SpaceA is moderate, the memory capacity of baseline design (64 GB) is able to accommodate most of the matrices from the University of Florida collection (max size about 50 GB) [56]. When the number of cubes increases, the latency of memory access to other cubes becomes larger, which makes the size of the current PE queue not large enough to hide the latency of remote memory access. Using a larger PE queue and L1 load queue to exploit

Table 3.3: The speedup comparison among Tesseract, GraphP, and SpaceA for PageRank (PR) and Single-Source Shortest Path (SSSP) algorithms on Wiki (WK) and LiveJournal (LJ) datasets over CPU baseline.

	Tesseract	GraphP	SpaceA
PR + WK	18.19	22.58	29.73
SSSP + WK	43.70	52.17	103.57
PR + LJ	21.09	34.08	58.34
SSSP + LJ	40.10	42.83	51.47

larger memory-level parallelism (MLP) will introduce a larger area overhead in the bank group level.

3.4.6 Case Study: Graph Analytics

Since SpMV is a building primitive in many application domains, such as scientific computing and graph analytics, SpaceA can be used to accelerate these applications. In order to study the performance benefits of SpaceA for these applications, we conduct a case study of running graph workloads on SpaceA, and compare the performance with state-of-the-art graph accelerators, Tesseract [57] and GraphP [58]. For comparing both Tesseract and GraphP, we use algorithms and input graphs evaluated in both of them. As a result, we use PageRank (PR) and Single-Source Shortest Path (SSSP) algorithms and Wiki (WK) and LiveJournal (LJ) input graphs [73] in this case study. Then, we run the implementation of these two algorithms from the GAP benchmark [74] on NVIDIA DGX-1 server (Intel Xeon CPU E5-2698 x2) as the baseline. To obtain the performance on SpaceA, we rewrite SSSP and PR algorithms into iterations of SpMV [75], and run them on SpaceA under the same number of cubes, vaults, and memory banks as the Tesseract configuration. We assume Tesseract and GraphP can obtain the same speedup as claimed in their paper, and the speedup of Tesseract, GraphP, and SpaceA over CPU is summarized as Table 3.3. This assumption overestimates the performance of Tesseract

and GraphP because our CPU baseline is more performant than theirs. Specifically, the CPU in our baseline has more cores (40 vs. 32), the same L1 and L2 cache per core while larger L3 cache in total (100 MB vs. 32 MB), and higher memory bandwidth (153.6 GB/s vs. 102.4 GB/s). Moreover, we use a well-optimized GAP benchmark as the CPU baseline instead of in-house C++ implementations used in Tesseract [57]. The results in Table 3.3 show that SpaceA obtains better performance than Tesseract and GraphP despite the overestimation of their speedups. The performance improvement of SpaceA mainly comes from the higher bandwidth provided by the near-bank integration instead of placing compute-logic on the base die. In summary, SpaceA can significantly accelerate graph analytics and it has the potentials to benefit other workloads built on SpMV computation.

3.5 Related Work

SpMV workloads. The study on efficient SpMV implementation starts from the CPU platform where the exploration of the locality of SpMV computations to efficiently use the memory bandwidth plays a major role [76–79]. GPU provides massive memory-level parallelism and high memory bandwidth, which makes it a promising solution when it comes to accelerating SpMV workloads [80]. Although existing studies develop efficient implementations for the widely used compressed sparse row (CSR) format on GPUs [81, 82], new matrix compression formats, such as AMB [83], BRO [63], Cocktail [84], BCSC [85], and BCCOO [86], are proposed to address the challenges of irregular memory access and workload imbalance across different processing units in more efficient and scalable manners. The road-map for SpMV on other many-core architectures, such as Intel Xeon Phi and Intel Knight Landing, is similar to GPGPU where customized matrix compression formats [64, 87, 88] are designed together with the paral-

lel algorithms SpMV to partition workloads across cores. Although these studies exploit existing memory bandwidth very well, they can not overcome the problem of limited memory bandwidth. Unlike these prior works optimizing SpMV on multi-core (CPU) or many-core processors (GPU), we exploit PIM-based architecture for superior bandwidth to overcome the bandwidth problem in multi-core and many-core processors.

PIM and NDP accelerators. There are several studies for PIM and NDP architectures in recent years for general purpose programs on different memory technologies, such as non-volatile memory (NVM) [89,90] and DRAM [3,11,12,91,92]. These architectures are usually equipped with compute logic designed for basic arithmetic primitives to support general purpose programs. Meanwhile, PIM architectures are also very promising in accelerator designs, which are customized for specific application domains, such as neural networks, block-chain [93], and image processing [50] workloads. In particular, many neural network workloads are memory intensive [36], thus prior studies exploit PIM architectures for different application phases: both training [94–99] and inference phases [25, 48, 49, 69, 100–102]. Among these application domains prior work studied for exploiting PIM architectures, graph analytics is the closest to SpMV workloads. In the vertex-centric programming model, a graph algorithm is equivalent to multiple iterations of SpMV when edges are stored in an adjacency matrix. Prior work studied graph workloads in PIM architectures for various memory technologies [57, 103–105] and efficient graph data partition methods [58]. Our case study in Section 3.4.6 shows that SpaceA achieves higher performance than prior designs placing compute-logic on the base die because of higher effective bandwidth exploited at the memory bank level. Prior studies have also exploited similar sparse linear algebra primitives, such as sparse matrix-matrix multiplication (SpGEMM) [106]. However, SpGEMM is very different from SpMV because of its poor data reuse opportunity. Other research discussing in-memory computing for the scientific workloads [107] has also been conducted. However, these studies do not

use compact sparse formats leading to both storage and performance overheads. Overall, different from all of these PIM accelerators, SpaceA is the first to design lightweight compute-logic near DRAM banks for irregular workloads whose memory access pattern is highly irregular thus introducing challenges for increasing the utilization of bank-level memory bandwidth.

Sparse linear algebra primitive accelerators. There are prior studies designing accelerators for SpMV [108] or other sparse linear algebra primitives [109,110]. In particular, because of the model compression techniques for neural network applications, such as weight pruning [111] and weight quantization [112], a lot of dense linear algebra primitives are transformed to sparse ones. As a result, these sparse neural network training and inference workloads attract intensive attention to accelerator designs for sparse linear algebra primitives [113–116]. Although these studies optimize SpMV for better locality or workload balance for on-chip computation, these compute-centric hardware designs have limited memory bandwidth. SpaceA exploits a PIM-based architecture superior bandwidth to overcome the bandwidth problem in CPU, GPU, and compute-centric accelerators.

3.6 Discussion

System and programming interface: Since SpaceA is designed as a standalone accelerator attached to the PCIe bus, it copies the sparse matrix and input vector from the CPU, offloads the computation of SpMV, and finally copies the output vector back to the CPU. The software support of SpaceA needs to provide APIs for memory allocation, data transfer, and SpMV computation invocation so that CPU programs can offload SpMV computation to SpaceA. Because the data format is different between sparse matrices and vectors, these two data structures need different driver APIs to support data allocation

and transfer. Additionally, the sparse matrix needs to be pre-processed on the CPU for assigning different rows across PEs before it is transferred to SpaceA. This execution model has been proven practical by prior studies offloading SpMV into GPU [63,83,84,86].

HMC vs. HBM: Although our architecture design of SpaceA is demonstrated and evaluated based on HMC-like configuration, SpaceA can also be realized by HBM [117] achieving similar performance and power under an equivalent configuration. The effectiveness of SpaceA architecture design mainly relies on two perspectives, near-bank logic integration and low latency communications for banks within the same channel. Although memory banks are grouped into the same channel horizontally in HBM while vertically in HMC, both of these two architectures have low latency TSV for communications among banks in the same channel. Therefore, the proposed approach would be applicable to HBM with a similar conclusion on performance and energy improvement.

3.7 Conclusion

In this project, we design an accelerator, SpaceA, based on PIM architecture by integrating compute-logic at the memory bank level to provide orders of magnitude higher effective bandwidth than GPU for SpMV computation. To exploit such a high bandwidth, our PE design is composed of a queue that holds memory requests to hide the latency of memory access to data in other memory banks. To exploit locality and to reduce traffic among memory banks, we integrate CAM buffers in SpaceA to cache data from the input vector. In addition to the architecture design, we develop a mapping scheme for SpaceA to balance workload and exploit locality among PEs. Our evaluation of 15 real-world matrices shows that SpaceA is highly competitive in terms of performance and energy-efficiency compared to the state-of-the-art GPU baseline.

Chapter 4

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

This chapter focuses on developing a domain-specific accelerator based on near-bank processing architectures for image processing workloads. Despite the success of SpaceA in accelerating SpMV operations through application-specific customization, we aim to study the potentials, benefits, and trade-offs of domain-specific near-bank processing architectures in this project. We select image processing as our target application domain for two reasons. First, image processing is becoming an increasingly important domain on workstations [118] and the data-center [119] platforms for various applications, such as machine learning [120], biomedical engineering [121], and geographic information systems [122]. Second, the memory-wall [123] impedes its further performance improvement as a result of both the characteristics of image processing workloads and the limited bandwidth provided by the compute-centric architecture. In this project, we develop iPIM, a domain-specific in-DRAM near-bank processing accelerator for image processing

workloads.

[1] 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 [124–126], so it is challenging to design a lightweight architecture supporting diverse image processing pipelines. Second, the design of instruction 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.

To address these challenges of using the near-bank architecture for image processing pipelines, we design the first programmable image processing accelerator (iPIM) and an end-to-end compilation flow based on Halide [127] 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-

[1] ©2020 IEEE. Reprinted, with permission, from Peng Gu, Xinfeng Xie (co-primary author), Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, Yuan Xie. "iPIM: Programmable in-memory image processing accelerator using near-bank architecture." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

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 project 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 $11.02\times$ 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.

4.1 Motivation

4.1.1 Image Processing and Halide Programming Language

Image processing contains heterogeneous pipelines which are wide and deep [127], and it is bound by memory bandwidth on the compute-centric architecture. From the applications' point of view, first, most image processing pipeline stages have low arithmetic intensity (operations per byte) and massive data parallelism for individual pixels, such as elementwise and stencil computations. Second, the whole pipelines are long and heterogeneous (e.g., 23 different stages in local Laplacian filter [128]), and they have complex data dependencies (e.g., resampling and gather in local Laplacian filter). Therefore, from the hardware's point of view, these features make it very difficult to apply pipeline fusion techniques [127, 129–133] to boost the performance. Thus, on compute-centric accelerators like GPU, image processing performance is bound by memory-bandwidth (Sec.4.1.2), and near-bank architecture provides a promising solution.

Although widely-adopted programming languages for image processing like Halide [127] provide optimizations for a wide range of compute-centric accelerators like GPU [134] and FPGA [135], there are no existing solutions for memory-centric accelerators. Halide decouples the algorithm descriptions and the algorithm to hardware mapping, thus programmers can separately describe an algorithm and a schedule. Based on the provided algorithms and schedules, the Halide compiler will synthesize hardware-specific programs. In this work, we propose the first end-to-end compilation framework for image processing applications in Halide on the near-bank architecture by designing novel schedules (Sec.4.3.2) and developing a compiler backend to improve the performance (Sec.4.3.3).

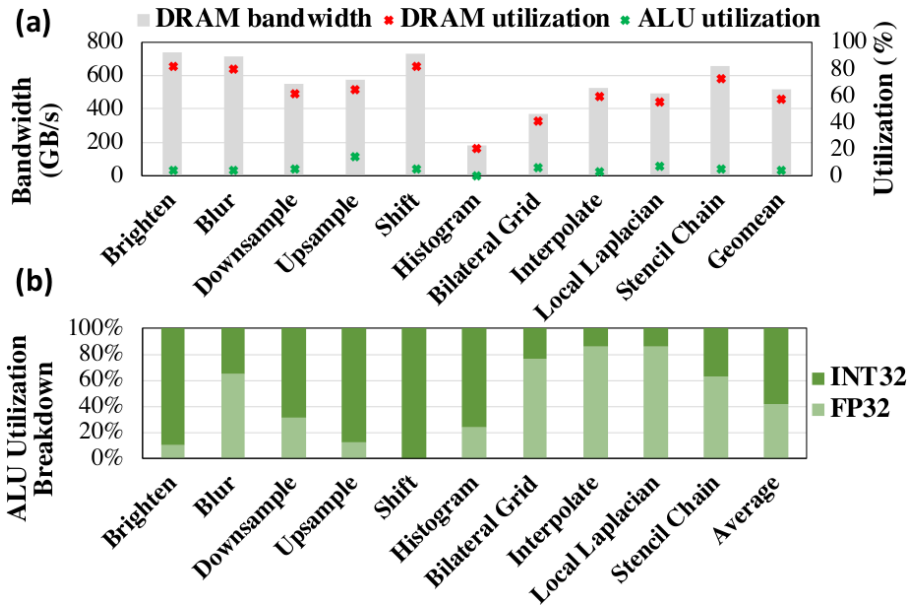


Figure 4.1: GPU profiling results for image processing workloads (Table.4.1).

4.1.2 Image Processing on GPU

First, we find that memory-bandwidth is the performance bottleneck for GPU, which is the current state-of-the-art image processing accelerator [136]. We conduct a detailed profiling of representative benchmarks (Table.4.1) using Halide framework [127] and DIV8K [137] dataset on an NVIDIA Tesla V100 GPU [138]. The measured total DRAM bandwidth, DRAM utilization, and ALU (both FP32 and INT32) utilization are shown in Fig.4.1(a). We observe that these benchmarks exhibit DRAM bandwidth-bound behavior by achieving 57.55% DRAM utilization ($518GB/s$ bandwidth) and 3.43% ALU 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.4.1), which are optimized by Halide pipeline fusion, show little performance improvement compared with single-stage benchmarks (the first 6 in Fig.4.1). 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.4.1(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 [139]. This motivates us to enable architecture support for index calculation in iPIM.

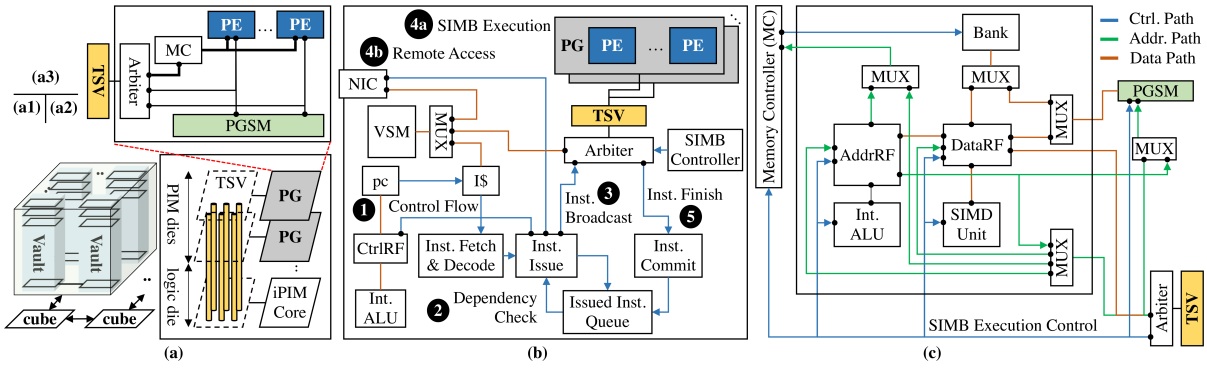


Figure 4.2: iPIM control-execution decoupled 3D-stacking microarchitecture: (a1) 3D-stacking cubes. (a2) A vault. (a3) A Process Group (PG). (b) Components inside an iPIM control core on the base logic die. (c) Components inside a Process Engine (PE) on the PIM dies.

4.2 iPIM Architecture

We introduce the microarchitecture overview in Sec.4.2.1 and describe iPIM's decoupled control-execution scheme in Sec. 4.2.2.

4.2.1 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.4.2(a). First, iPIM consists of multiple cubes (Fig.4.2(a1)) interconnected by SERDES links similar to HMC [140]. Second, one cube is horizontally partitioned into multiple vaults (usually 16 per cube) connected by an on-chip network. Each vault (Fig.4.2(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.4.2(b)), which is the basic unit to execute an iPIM program. Next, one PIM die of each vault contains one process group (PG) (Fig.4.2(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.4.2(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.4.1(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.4.2.2). 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).

4.2.2 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.4.3) 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 can exploit massive bank-level parallelism by programming the bits of $simb\_mask$.

Next, we introduce the detailed pipeline execution of iPIM in Fig.4.2(b) 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*).

4.3 Compiler 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.4.3.1, which includes the design of new schedules for iPIM. Second, we explain the compilation flow in Sec.4.3.2 including the extension of Halide front-end compilation passes and our customized backend for iPIM. Third, we detail the backend optimizations in Sec.4.3.3 for generating efficient iPIM executable programs. Finally, we discuss the system integration of iPIM in Sec.4.3.4 for executing compiled programs.

4.3.1 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. Sec-

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y)
                    + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y)
                    + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
    .ipim_tile(x, y, xi, yi, 8, 8)
    .load_pgsm(xi, yi)
    .vectorize(xi, 4);
```

Listing 4.1: Code example of image blur.

ond, 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 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. For example, the schedule $ipim\_tile(x, y, xi, yi, 8, 8)$ in Listing.4.1 indicates that the image will be partitioned into image tiles (8x8 size). In addition to the partition of the image into tiles, this schedule also indicates the distribution of these image tiles across all PEs. Fig.4.3(a) shows the distribution of these image tiles into different levels in the hierarchy of iPIM. Specifically, image tiles are distributed in an interleaved way to the PEs of the same PG so that they can load adjacent image tiles at the same loop iteration to improve data sharing. The second customized schedule for iPIM, *load_pgsm*(), indicates the usage of shared scratchpad memory at the PG level. For example, the schedule $load\_pgsm(xi, yi)$ in Listing.4.1 indicates that the data of input image needed

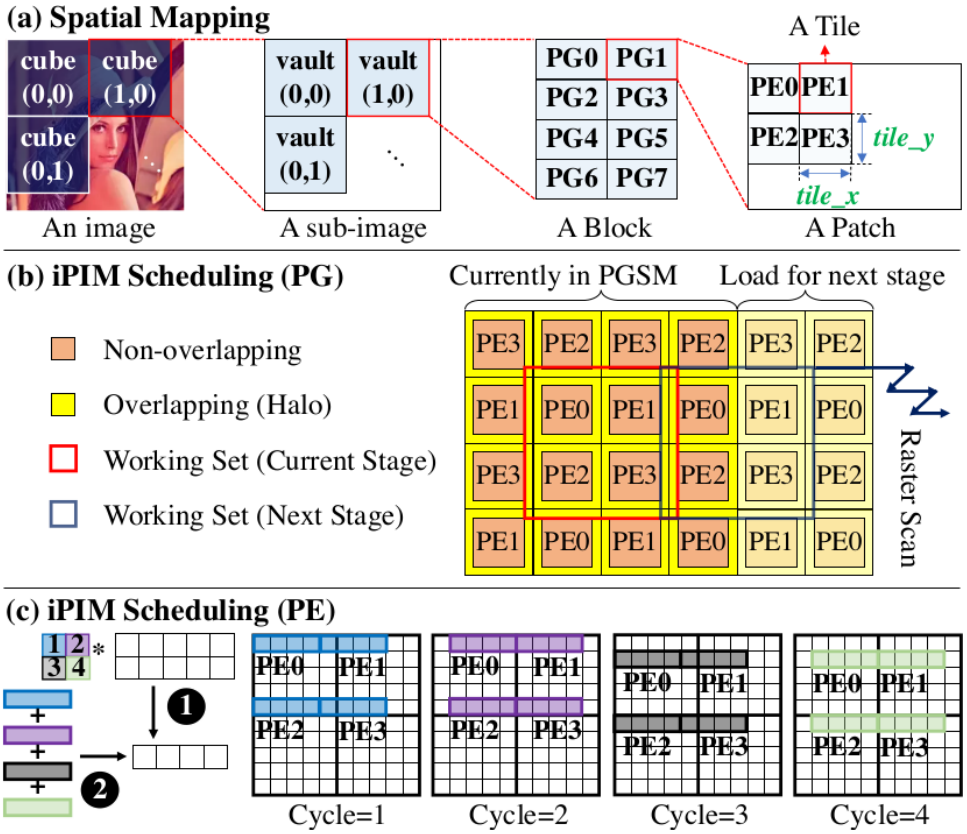


Figure 4.3: An iPIM compilation example of image blur: (a) Spatial mapping. (b) PG-level scheduling. (c) PE-level scheduling.

for computing output along loops xi and yi will be loaded into shared scratchpad memory before using it for the computation. Fig.4.3(b) shows the usage of PGSM according to the specification of *load_pgsm*() in Listing.4.1 at PG-level. After loading data into PGSM, Fig. 4.3(c) shows the temporal scheduling of the computation for each PE including four steps (❷) to load the whole region of input data (❶) for a vector of output pixels. 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. In Listing.4.1, *compute_root*() ensures that the loops

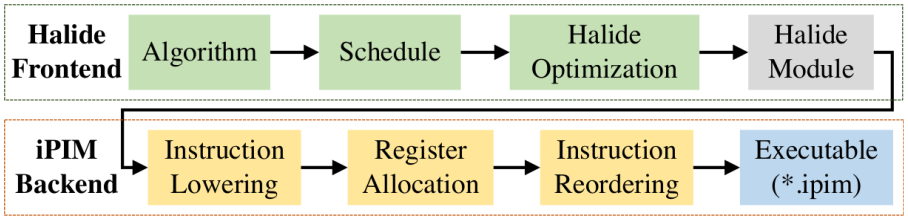


Figure 4.4: The end-to-end compilation flow of iPIM.

along dimensions of the Func *out* will be outermost loops and the stages of computing *blurx* will be fused into the computation of *out*. During code generation, each *compute_root*() implies a kernel function reading input data from and writing output results to DRAM banks. Besides *compute_root*(), 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.

4.3.2 Compilation Flow

As shown in Fig.4.4, 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.

4.3.3 Backend Optimization

In this section, we detail the novel instruction optimizations we developed for the backend. The major goal of the backend in our compilation flow is to generate efficient iPIM executable programs from the input Halide module. The backend decouples this program generation process into two parts, instruction lowering which translates the Halide module into iPIM instructions, and instruction optimizations which improve the performance of the generated programs. We will detail these optimizations into three parts, *register allocation*, *instruction reordering*, and *memory order enforcement*. The effectiveness of our backend optimizations will be quantitatively analyzed in Sec.4.4.4.

Register Allocation: The goal of register allocation is to assign a physical register to each virtual register and avoid the instruction dependency due to the conflict of physical registers. To avoid such conflicts on physical registers, our algorithm is based on the depth-first search on the register interference graph, and it attempts to assign each virtual register from a physical register different from the most recently used one. The input of our algorithm, the register interference graph, is built upon the traditional liveness analysis of virtual registers. After building the register interference graph and converting the register allocation problem into a graph coloring problem, our algorithm tries to avoid the conflict of physical registers rather than solely minimizing the number of physical registers in the allocation. Because the architecture design of iPIM uses simple in-order

Algorithm 4 Instruction reordering algorithm

Input: dependency graph of instructions $G = (V, E)$
Output: a sequence of instructions S
Init the set of ready instructions $R = \emptyset$
for $v \in V$ **do**
  Init $T(v) = 0$
  if $v.degree == 0$ **then**
    $R = R \bigcup \{v\}$
  end if
end for
$N(v)$: the outgoing neighbour nodes of the node v.
$L(v)$: the execution latency of the node v.
for $i = 1$ to $|V|$ **do**
  v^{opt} = Inst with the highest priority for $v \in R$.
  $R = R - \{v^{opt}\}$; $S_i = v^{opt}$; $T(v^{opt}) = i$
  for $u \in N(v^{opt})$ **do**
    $T(u) = max\{T(u), T(v^{opt}) + L(v^{opt})\}$
    $u.degree = u.degree - 1$
    if $u.degree == 0$ **then**
      $R = R \bigcup \{u\}$
    end if
  end for
end for

control core to avoid hardware overheads, the traditional register allocation method could cause the dependency between instructions due to the conflict of physical registers, which further leads to pipeline stalls.

Instruction Reordering: Although the program generated by register allocation is already executable on iPIM, we reorder instructions in the program to maximally exploit the instruction-level parallelism. Because of the instruction issue mechanism of our in-order core, the dependency between adjacent instructions will lead to pipeline stalls. Therefore, the instruction reordering aims to expose instruction-level parallelism to the hardware, which eliminates pipeline stalls and improves the performance. We first build a directed graph where each node stands for instruction and directed edges between nodes represent the dependency between instructions. Then we develop our instruction reordering algorithm which traverses this directed graph in topological order. We associate each node with a timestamp to provide an estimation of its earliest time

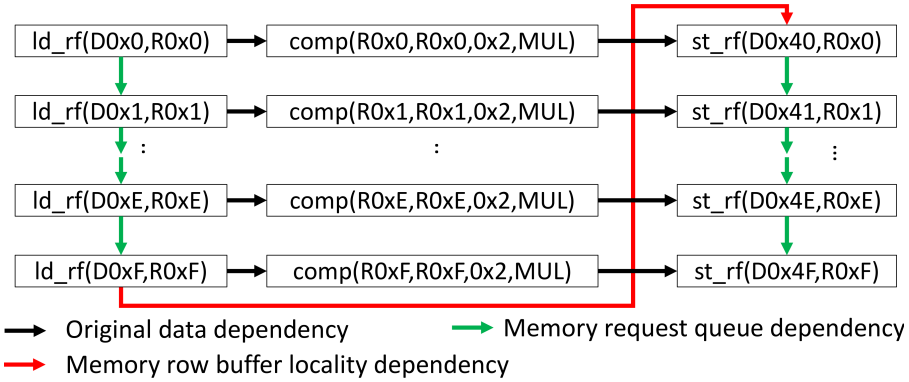


Figure 4.5: Instruction reordering example: image brighten.

ready to be issued ($T(v)$ in Algorithm.4). When there are multiple instructions available at a time step, we will schedule the load instruction with the T smaller than the current time step or the node with the smallest T. After marking the instruction to be scheduled, we update the incoming degree of all its outgoing neighbors, and also their timestamp T. Finally, all instructions are scheduled into the output sequence after iterating through $|V|$ time steps. This graph traversal algorithm is demonstrated in Algorithm 4, and its time complexity is $O(|V|log|V|+|E|)$ where $|V|$ is the number of nodes, i.e. instructions, and $|E|$ is the number of edges in the directed dependency graph.

Memory order enforcement: In addition to data dependency which will block issuing instructions, we also add the dependency for resource conflicts to prevent pipeline stalls due to resource contention on DRAM. In particular, issuing two DRAM load instructions consecutively consumes slots in the instruction queue at the base logic die while the second instruction has to be stalled because of the single memory request queue and a longer DRAM access latency. In some cases, a large number of consecutive DRAM instructions could occupy the whole instruction queue impeding the scheduling of further computation instructions which do not have a dependency on any instruction in the queue. To prevent pipeline stalls due to the lower throughput of the memory request queue, we insert dependency between load instructions and store instructions to defer

the scheduling of consecutive memory instructions. We use the image brighten pipeline as an example shown in Fig.4.5. Since DRAM access latency varies from the case of row buffer hit to row buffer miss, we also add the third kind of dependency to enforce the memory accesses to the DRAM with the same order as they appear in the input program. As shown in Fig.4.5, these two kinds of newly added dependency edges help to avoid pipeline stalls due to DRAM request queue contention and improves the locality of row buffers as it keeps the originally good data access locality on image tiles. After adding these two new kinds of dependency among instructions, the generated instruction dependency graph is passed to the instruction reordering stage.

4.3.4 System Integration

We consider iPIM as a standalone accelerator with a separate address space, which is not a part of the host CPU's system memory. This standalone design can avoid the complexity and overhead of supporting virtual memory [141] and cache coherence [142], which introduces extra communication traffic between the host and PIM accelerator and offsets the benefits of PIM. iPIM can be integrated with the host CPU using a standard bus, such as PCIe [143] and AMBA [144], and can be scaled using off-chip SERDES links similar to HMC [140].

4.4 Evaluation

We first describe the experimental setup and methodologies in Sec.4.4.1. Next, we show the performance, energy, and area results of iPIM in Sec.4.4.2. In Sec.4.4.3, we demonstrate the advantages of iPIM's near-bank design and the effectiveness of decoupled control-execution architecture. In Sec.4.4.4, we show the benefits of iPIM's compiler optimizations by conducting a series of comparative evaluations. In the end, we conclude

Table 4.1: 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) [145]
	Interpolate	It interpolates pixel values using a pyramid of low-resolution samples (12 pipeline stages) [127]
	Local Laplacian	It tone-maps an image and enhances its local contrast using a multi-scale method (23 pipeline stages) [128]
	Stencil Chain	It is composed of a chain of stencil computations (32 pipeline stages) [127]

that iPIM's compiler optimizations are near-optimal by showing the achieved high hardware utilization and instruction per cycle (IPC) number.

4.4.1 Experimental Setup

Benchmark and Dataset Selection. As detailed in Table.4.1, 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 [146], 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 [127, 130, 132, 147], on

the other hand, contain heterogeneous pipeline stages that require the support of programmability. We use DIV8K [137] dataset, which contains over 1500 images covering 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 [11–13] without changing DRAM's core timing. We list the detailed hardware configuration, latency values, energy consumption, and DRAM settings in Table.4.2. 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 [148] with 4 HBM stacks (total $\sim 1199mm^2$), where one HBM stack consumes $\sim 96mm^2$ footprint [149].

Simulation Methodology. We develop a cycle-accurate simulator extended from ramulator [150] 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 [67] 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 [151]. The hardware components of SIMD units and integer ALUs are synthesized by design compiler [152] 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 [11]. For the control core on the base logic die, we adopt an in-order ARM cortex-A5 core [153] to evaluate its area and power. For the GPU evaluation, the baseline image processing workloads are written in

Table 4.2: 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

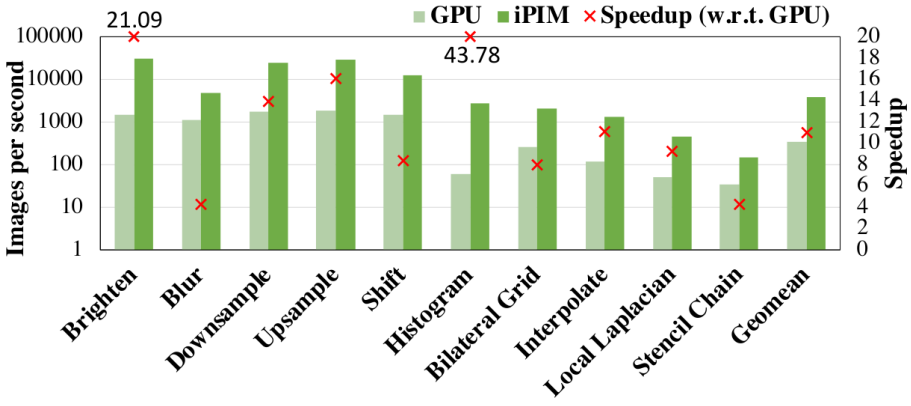


Figure 4.6: Throughput and speedup comparison between iPIM and GPU.

Halide with manually-tuned schedules. The GPU performance and power are measured from *nvprof* and *nvidia-smi*, respectively.

4.4.2 Performance, Energy-efficiency, and Area

Performance. iPIM achieves 11.02× average speedup over the GPU as shown in Fig.4.6. 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.4.4.3). From the software's point of view, this high speedup is achieved through

good compiler optimizations (more analysis in Sec.4.4.4).

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.4.4.4) shows that these two benchmarks have higher computation intensity than other benchmarks, and involve a lot of index calculations which 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.4.7). 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.4.4.3 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.4.7 is approximately proportional to the speedup in Fig.4.6. 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-

Table 4.3: Area evaluation of iPIM components on the DRAM die considering DRAM process overhead.

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

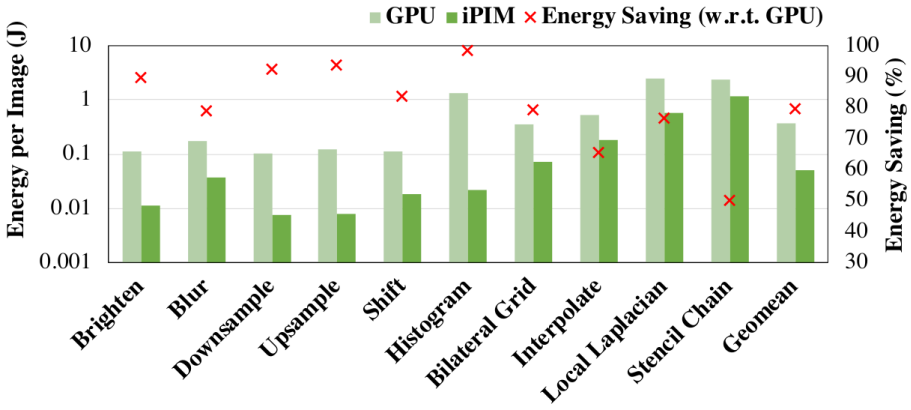


Figure 4.7: Energy comparison between iPIM and GPU.

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 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.4.2(c)), and normalize the total added area to a DRAM die ($96mm^2$ [149]). We show that the added area per DRAM die is small (10.71%) to support programmability according to Table.4.3. Second, we evaluate the area of iPIM's control core on the base logic die (Fig.4.2(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 ($3.5mm^2$ [69]) 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 $105°C$ [149], and we conservatively assume the DRAM dies in our case operates under $85°C$. A prior study on 3D PIM thermal analysis [154] shows that active cooling solutions can effectively satisfy this thermal constraint ($85°C$). Both commodity-server active cooling solution [72] (peak power density allowed: $706mW/mm^2$) and high-end-server active cooling solution [71] (peak power density allowed: $1214mW/mm^2$)) can be used. Also, compared with previous work [154] 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 [155]. 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.

4.4.3 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.4.8. 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

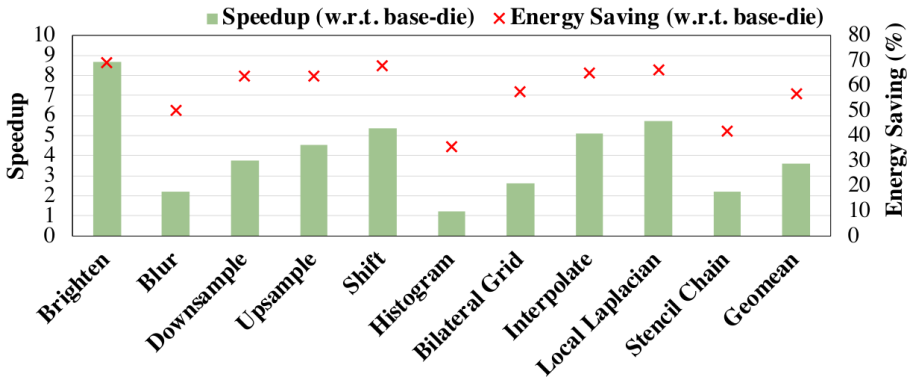


Figure 4.8: Comparison of near-bank and process-on-base-die solutions.

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. 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.4.9. 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 de-

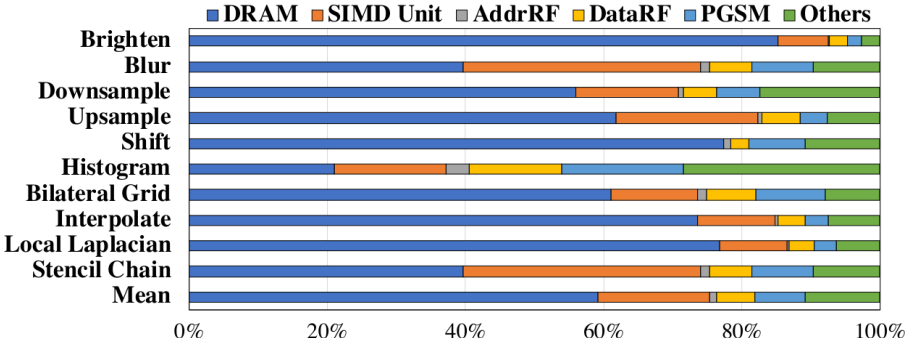


Figure 4.9: Energy breakdown of iPIM programs.

coupled 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%). The low energy consumption of inter-vault and intra-vault data movement is contributed from (1) iPIM's near-bank architecture and (2) localized data movement benefited from the memory hierarchy and compiler optimizations.

4.4.4 Compiler Analysis

Effectiveness of compiler optimizations

We add a set of comparative evaluations to justify the performance benefits provided by iPIM's compiler optimizations (Fig.4.10). We summarize these optimization choices as follows. The register allocation policy determines whether to use the minimum number of physical registers (*min*) or scatter registers to avoid the dependency of instructions (*max*). The instruction reordering option determines whether to reorder the instruction of programs generated by the register allocation stage. The memory order enforcement option chooses whether to add dependency edges on adjacent memory requests or not before sending the dependency graph to the instruction reordering stage. The optimized design (*opt*) adopts the *max* register allocation policy and applies both instruction reordering and memory order enforcement. The naïve baseline (*baseline*1) as-

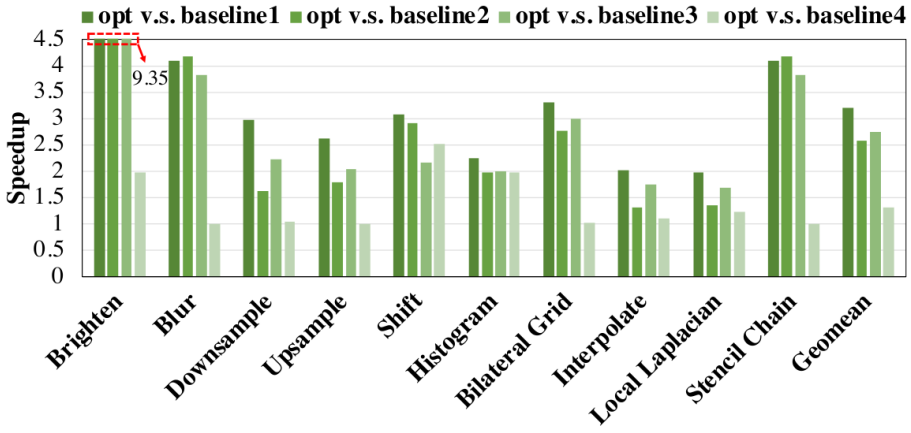


Figure 4.10: Effectiveness of different iPIM compiler optimizations.

sumes the *min* register allocation policy without instruction reordering. All of *baseline*2, *baseline*3, and *baseline*4 have only one different compiler optimization option compared to *opt* . Specifically, *baseline*2 uses the *min* register allocation policy, *baseline*3 does not apply instruction reordering, and *baseline*4 does not enforce memory order. The rest of settings for $baseline2-4$ remain the same as *opt*.

We observe that all of iPIM compiler optimizations provide an overall 3.19× speedup (*opt* v.s. *baseline*1). Further analysis shows that *max* register allocation provides 2.59× speedup than *min* register allocation (*opt* v.s. *baseline*2). This is because iPIM's in-order core does not support expensive register renaming mechanism in the out-of-order execution, and *max* register allocation can optimally eliminate output-dependency and anti-dependency to prevent issue stall of later instructions. Next, instruction reordering provides 2.74× speedup (*opt* v.s. *baseline*3), since it can expose more instruction-level-parallelism by overlapping instructions without dependency. In the end, enforcement of memory-order provides 1.30× speedup (*opt* v.s. *baseline*4). The reason is that it can maximally interleave other instructions with memory access requests and it also improves row buffer locality.

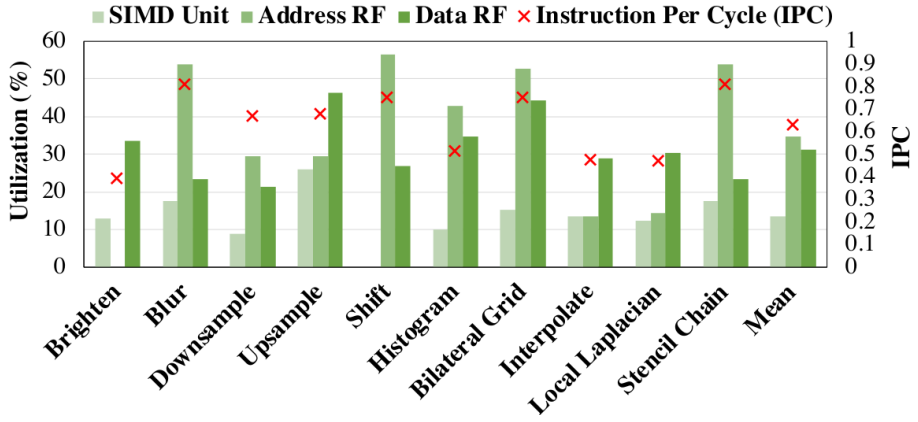


Figure 4.11: iPIM's key components' utilization and IPC.

IPC and Utilization analysis

As shown in Fig.4.11, we provide the IPC of the control cores and the utilization of key hardware components on the PIM dies. First, we observe that the average IPC achieves a very high value of 0.63 after intensive compiler optimizations. This implies that iPIM currently attains near-optimal performance, and further improvement has an upper bound of 1.59× assuming no pipeline stalls. Second, detailed analysis shows that for each benchmark, key hardware components reach very high utilization. For example, benchmarks with intensive index calculations (Blur, Shift, Histogram, Bilateral Grid, and Stencil Chain) realize more than 40% utilization on the address register file. As a conclusion, the high IPC and hardware utilization indicate current compilation flow has well-optimized image processing applications on iPIM architecture.

4.5 Related Work

Image processing accelerators. Previous work has explored Field Programmable Gate Array (FPGA) [130, 131, 147], Coarse Grain Reconfigurable Arrays (CGRA) [156, 157], and ASIC solutions [139, 158] for image processing acceleration. These acceler-

ators target mobile and embedded platforms, where power-efficiency and low latency are the primary goals of optimization. Also, the application scenarios are mostly image streaming applications with a small working set, so spatially distributed data flow architecture is often adopted to map the entire image processing pipeline. To achieve the desired power-efficiency, line buffer [157] is widely used to exploit producer-consumer data locality and fuse pipeline stages, so the intermediate data can stay on-chip without expensive off-chip memory access. In comparison, iPIM focuses on data center and workstation environments, where the complex algorithm pipelines and large working set due to high-resolution images need both high memory capacity and bandwidth. Thus, conventional compute-centric accelerators will suffer from the memory bandwidth bottleneck, while iPIM's near-bank architecture provides abundant bandwidth resources to tackle this challenge.

Process-in-memory (PIM) accelerators. We compare iPIM with previous work using practical DRAM technology without invasive modifications to the DRAM bank's circuitry [91, 159]. The first category of research [125, 160, 161] that tries to integrate processor cores to DRAM dies suffers from large area overhead. iPIM solves this challenge by proposing a control-execution decoupled approach where the control core is placed on the base logic die and shared by execution units closely integrated with each bank. The second category of research adopts 3D process-on-logic-die architecture [25, 57, 58, 69, 103, 162–166], which is bound by the TSV bandwidth available in the base logic die. iPIM evaluation shows 3.61× speedup and 56.71% energy saving compared with this approach. To further improve memory bandwidth, a few work [11–13] employs near-bank architecture similar to this work, but only supports limited fixed functionalities and cannot map the heterogeneous image processing pipelines which have diverse computation and memory patterns. iPIM proposes SIMB ISA and an end-to-end compilation flow to solve this programmability challenge. In addition, some recent

work proposes integrating computation logic on the DRAM DIMM modules to enable low overhead near-data processing [167–170]. While practical, these architecture designs only have small bandwidth improvement over CPUs compared with 3D-PIM solutions. There are also studies exploiting PIM architectures based on non-volatile memory (NVM) technologies [48,49,90,93,171,172]. Compared with these studies, DRAM provides a better write endurance than NVM, which is critical to image processing applications where intermediate results of pipelines need to be written back to memory.

4.6 Conclusion

This project develops iPIM, the first 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, this project develops 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 5

MPU: Memory-Centric SIMT Processor via In-DRAM Near-Bank Computing

This chapter focuses on developing a general-purpose near-bank processing architecture for addressing the memory bandwidth wall of many parallel computing workloads on state-of-the-art GPU platforms. This work further explores near-bank processing architectures under the context of general-purpose computing in addition to our prior studies of application-specific and domain-specific solutions. Although a general-purpose graphics processing unit (GPGPU) with its single-instruction-multiple-thread (SIMT) programming model benefits these workloads by exploiting massive memory-level parallelism and providing DRAM bandwidth higher than traditional CPUs, its further performance scaling is constrained by the "memory bandwidth wall" [123] challenge.

Despite the potential of near-bank computing to provide plentiful memory bandwidth, existing academic research [11–13] and industrial prototypes (e.g., HBM-PIM [15]) focus on developing application-specific or domain-specific architectures. In previous chapters,

our near-bank processing designs, SpaceA and iPIM, also demonstrate such success in SpMV computation and image processing workloads. Although these near-bank processing designs accelerate their target applications through the customization of hardware and software, they lack the flexibility of supporting a wider range of data-intensive workloads. Such a lack of flexibility increases the non-recurring engineering costs for chip fabrication, thus impeding the development and deployment of near-bank processing hardware to address the memory bandwidth wall. As a result, it urges a general-purpose architecture that lowers the non-recurring engineering costs by providing a higher programmability to support various applications without changing hardware.

There are still several challenges for general-purpose near-bank computing in accelerating a number of data-intensive workloads. First, the pipeline of SIMT processors contains complex logic components (e.g., load-store-unit [173]) and large register files [174]. 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 [124–126]. 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 [175]. 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 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.

To address these challenges, we design MPU (Memory-centric Processing Unit), the first general purpose SIMT processor based on 3D-stacking near-bank computing archi-

tecture, and an end-to-end compiler flow supporting CUDA programs [176] 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 project are summarized as follows:

- To the best of our knowledge, we design the first general-purpose 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

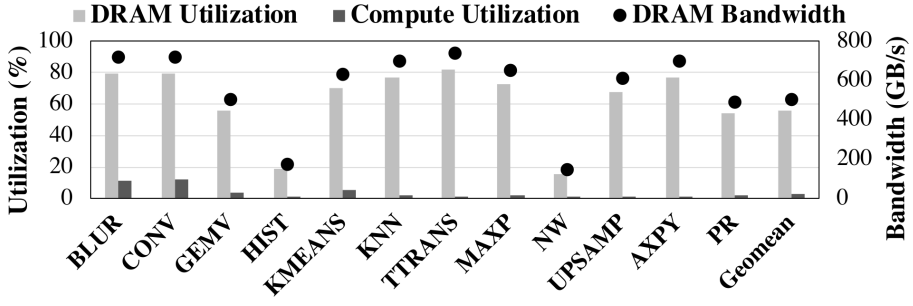


Figure 5.1: Profiling results for data-intensive workloads on NVIDIA Tesla V100 GPU.

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.

5.1 Motivation

5.1.1 GPU Profiling

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 from various application domains including deep learning, bioinformatics, linear algebra, and image processing applications as detailed in Table 5.1. The measured memory bandwidth, bandwidth utilization, and compute utilization of an NVIDIA Tesla V100

GPU [177] are shown in Fig.5.1. 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 [11–13, 50] 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.

5.1.2 SIMT Processor

SIMT model [178] is widely adopted in modern GPUs [179, 180] for accelerating parallel computing programs. The pipeline can be divided into front-pipeline which consists of instruction fetch, decode, and issue stages, middle-pipeline which consists of execution and writeback stages, and end-pipeline which consists of the commit stage. The front-pipeline and the end-pipeline usually involve complex control and logic circuits, such as warp table, SIMT stack, scoreboard, and load-store unit. The middle-pipeline contains arithmetic instruction and memory access instructions. MPU reduces the near-bank overhead by duplicating some parts of the middle-pipeline to DRAM dies, and add corresponding instruction offloading mechanisms. The SIMT model also has some special features that require optimizations. First, the shared memory is extensively used for inter-thread communication inside the same thread block. Placing it in the base logic die may introduce extra communication traffic among the 3D layers. Second, the SIMT scheduling causes warps to access different row-buffers interchangeably, causing a row-buffer ping-pong effect. MPU explores architectural optimizations, near-bank shared memory and multiple activated row-buffers, for these two features detailed in Sec.5.2.3.

5.2 MPU Architecture

First, we discuss MPU's high-level design in Sec.5.2.1. Second, we introduce MPU's hybrid pipeline in Sec.5.2.2. Next, we present two architectural optimizations for SIMT model in Sec.5.2.3.

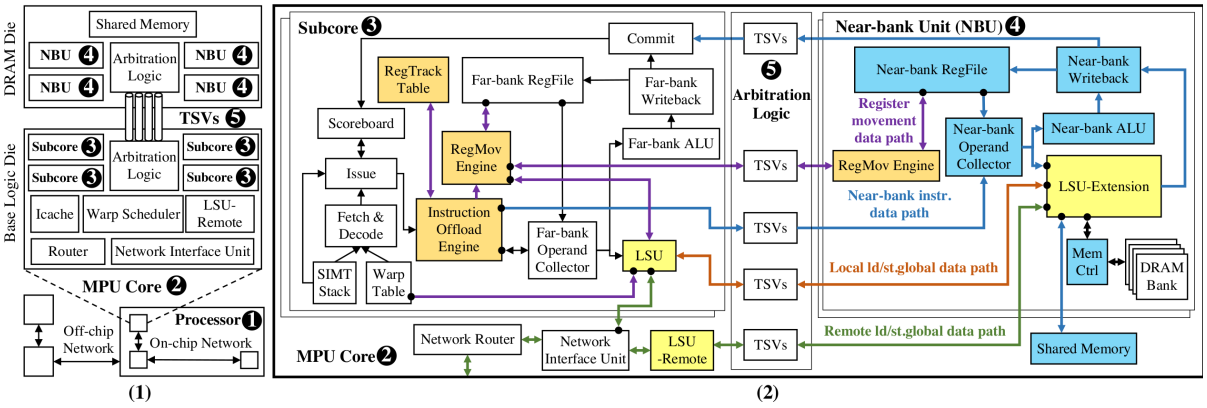


Figure 5.2: (1) MPU architecture overview. (2) Detailed microarchitecture of a MPU core (left: Subcores, middle: TSVs, right: NBUs).

5.2.1 Microarchitecture Overview

From the high-level, MPU adopts a scalable design with many processors (Fig.5.2❶) interconnected by an off-chip network (similar to SERDES links in the HMC [181]) as shown in the bottom of Fig.5.2 (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) [182] (Fig.5.2❺). The base logic die is horizontally partitioned into an array of SIMT cores (Fig.5.2❷) interconnected by an on-chip 2D-mesh network [183].

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.5.2❷), 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.5.2❸), an instruction cache, a warp scheduler, and components for handling inter-core traffic (network interface unit, router, and LSU-Remote). The TSVs (Fig.5.2❺) are evenly divided among the cores (64b 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.5.2❹) and the shared memory. To enable efficient processing in this hybrid architecture (Sec.5.2.2), 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.5.2.3. 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.

5.2.2 Hybrid Pipeline

As illustrated in Fig.5.2 (2), an MPU core (Fig.5.2 (2)❷) adopts a hybrid pipeline design that is split between the base logic die (subcore) (Fig.5.2 (2)❸) and the DRAM die (near-bank unit, NBU) (Fig.5.2 (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 [11, 50], and shared memory can reduce TSV traffic for register movement (Sec.5.2.3).

In addition to the original pipeline components, to assist flexible instruction offload-

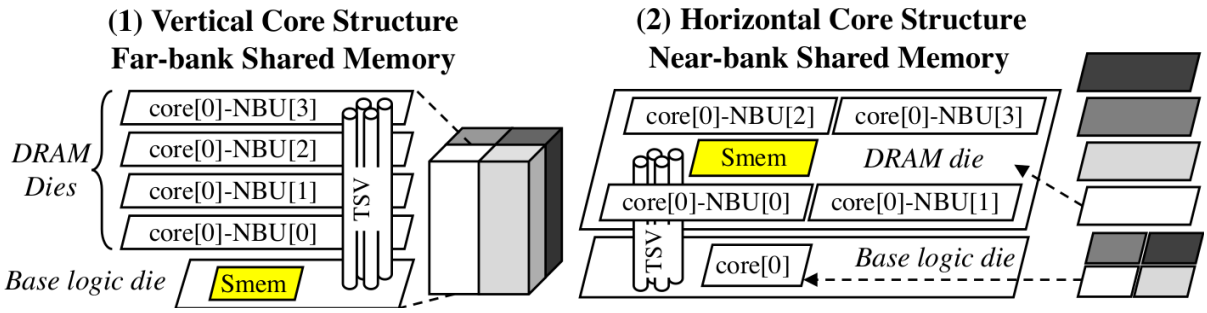


Figure 5.3: (1) Vertical and (2) horizontal core structure.

ing, each subcore also adds an instruction offload engine, a register move engine, and an associated register track table. Besides, the load-store unit (LSU) is modified and augmented to support remote data traffic and near-bank instruction offloading.

5.2.3 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 [175]. 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.5.3 (1)) in the default HMC-style setting [181], 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.5.3 (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.5.4.3, 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) [184] 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.5.4.3, we confirm that this design can decrease the row-buffer miss rate and increase performance for multiple benchmarks.

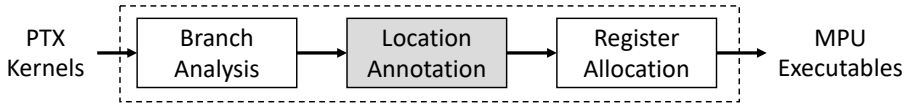


Figure 5.4: The backend of MPU compiler generating MPU executable kernels from PTX kernels.

5.3 Compiler Support

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. Experimental results in Sec.5.4.4 demonstrate that this novel location annotation improves the performance of MPU compared with a default hardware policy.

The end-to-end compilation flow of MPU includes frontend stages and backend stages. In frontend stages, the MPU compiler reuses *nvcc* [185] to compile CUDA programs [175] to generate kernels in Parallel Thread Extension (PTX) [186] ISA which is a kind of intermediate representation of CUDA kernels. Then, the backend generates the MPU hardware executables from the PTX kernels, which includes three main stages as shown in Fig.5.4. Among these three stages, the branch analysis and register allocation stages are common to support the SIMT programming model. 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 [187]. This problem can be formulated as the post-dominator analysis of a control-flow graph representing the program. The register allocation stage analyzes the liveness of each virtual register in the program to build a register interference graph. The allocation of physical registers can then be formulated as a graph coloring problem on this register interference graph.

The location annotation is a novel backend stage to optimize the performance by

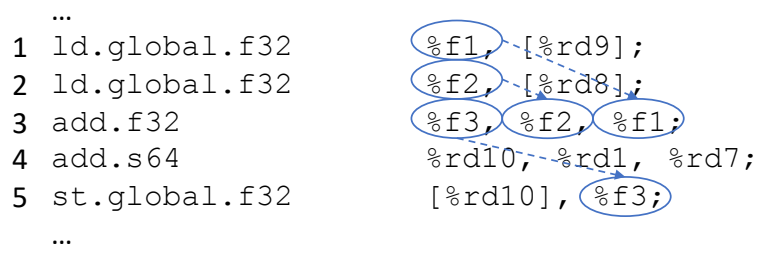


Figure 5.5: A code snippet of PTX instructions with a dependency chain of near-bank registers highlighted.

annotating the location of instructions as either the near-bank NBU or the far-bank subcore on the base logic die. When executing the annotated kernels on MPU, the locations annotated on instructions will be used to finalize the runtime instruction offloading decision. The main idea of this optimization is a heuristic approach based on the static analysis extracting the dependency chains of the control-related, address-related, and value-related registers. First, value-related registers will be annotated as near-bank registers while control-related and address-related registers will be annotated as far-bank registers. Then, instructions from the dependency chain of value-related registers are annotated as the near-bank instructions while the rest of instructions are annotated as the far-bank instructions. For example, Fig.5.5 highlights a dependency chain of value registers. Offloading the compute instruction in line 3 to near-bank compute logic can eliminate the data transfer on TSVs for registers %f1, %f2, and %f3. Finally, the annotated program will be passed to the register allocation stage where registers annotated as different locations will not share the same physical register.

The static analysis of the location annotation is implemented as an iterative algorithm shown in Algorithm 5 to realize the idea of decoupling register dependency chains. Initially, the location of all registers is annotated as U (unknown). The value registers of the global load/store instructions are annotated as N (near-bank) while the address registers of these instructions are annotated as F (far-bank) because of the LSU design.

Algorithm 5 Location annotation algorithm

Input: A kernel with a list of instructions I
Output: A location table L for all registers and instructions.
$L(reg)$: the location of an register reg.
$L(instr)$: the location of an instruction $instr$.
Init the location of all registers to unknown U.
Init the location of all instructions to unknown U.
Init the set of registers $R = \emptyset$.
// Annotate the initial location to address registers, value registers, and
// control registers.
for $instr$ in I **do**
  for $reg \in \{instr.SrcRegs \bigcup instr.DstRegs\}$ **do**
    $R = R \bigcup \{reg\}$
  end for
  if $instr.type \in Instr_{jump}$ **then**
    $L(instr.SrcRegs) = F$
  end if
  if $instr.type == ld.global$ **then**
    $L(instr.SrcRegs) = F$
    $L(instr.DstRegs) = N$
  end if
  if $instr.type == st.global$ **then**
    $L(instr.SrcRegs) = N$
    $L(instr.DstRegs) = F$
  end if
  if $instr.type \in \{ld.shared, st.shared\}$ **then**
    $L(instr.SrcRegs) = N$
    $L(instr.DstRegs) = N$
  end if
end for
// Propagate the location of known registers to others.
while $\forall reg \in R, L(reg)$ does not change **do**
  for $instr$ in I **do**
    for reg in $instr.SrcRegs$ **do**
      if $L(reg) == U$ **then**
        $L(reg) = L(instr.DstRegs)$
      end if
      if $L(reg) != L(instr.DstRegs)$ **then**
        $L(reg) = B$
      end if
    end for
  end for
end while
// Annotate the location of instructions according to the location of their
// destination registers.
for $instr \in I$ **do**
  $L(instr) = L(instr.DstRegs)$
end for

In addition to that, the location of predicative registers is annotated as F (far-bank) because control-related instructions are executed on the far-bank pipeline stages. Then, our algorithm iteratively scans over the program to update register locations. In particular, if the destination register location is known for an instruction, its source registers will follow the same location annotation. If a register is annotated as both N and F from different instructions, it will be annotated as B, meaning that this register could appear on both far-bank and near-bank pipeline stages. This process will finish once the annotated locations of all registers converge. Finally, the location of instruction follows the same location as its destination register.

5.4 Evaluation

In Sec.5.4.1, we introduce the experiment setup and methodologies. In Sec.5.4.2, we show the performance, area, energy, and thermal results of MPU. In Sec.5.4.3, we demonstrate the benefit of MPU's architecture optimizations and the comparison with the prior processing-on-base-logic-die designs. In Sec.5.4.4, we present the effectiveness of the location annotation in our compiler backend optimization.

5.4.1 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 in Table.5.1. 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 [190], or writing CUDA programs in the

Table 5.1: The workloads of the benchmark suite.

Workload	App Domain	Reference	Description
BLUR	Image Processing	Halide [127]	3x3 blur.
CONV	Machine Learning	TensorFlow [28]	3x3 conv.
GEMV	Linear Algebra	cuBLAS [188]	Matrix-vector multiply.
HIST	Image Processing	CUB [189]	Histogram.
KMEANS	Machine Learning	Rodinia [190]	K-means clustering.
KNN	Machine Learning	Rodinia [190]	K-nearest-neighbour.
TTRANS	Linear Algebra	cuBLAS [188]	Tensor transposition.
MAXP	Machine Learning	TensorFlow [28]	Max-pooling.
NW	Bioinformatics	Rodinia [190]	Sequence alignment.
UPSAMP	Image Processing	Halide [127]	Image upsample.
AXPY	Linear Algebra	cuBLAS [188]	Vector add.
PR	Linear Algebra	CUB [189]	Parallel reduction.

Table 5.2: MPU hardware configuration parameters.

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 [150]	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 [67] (J/access)	0.15n/0.27n/1.13n/40.0p/22.2p
Operand_collector / LSU-Extension (J/access)	41.49p/39.67p
TSV [191] / (on)off-chip bus [67, 151] (J/bit)	4.53p/(0.72p)4.50p
DRAM_rowbuffer_policy / DRAM_schedule	open_page / FR-FCFS

same functionality while achieving performance comparable to state-of-the-art libraries, such as cuBLAS [188].

Hardware Configuration. Using the 3D-stacking memory configuration similar to the previous near-bank accelerators [11–13, 50], MPU needs no changes to DRAM's core circuit except the multiple activated row-buffers enhancement [184]. The detailed hardware configuration, latency values, energy consumption, and DRAM settings are presented in Table.5.2. MPU contains 8 processors (total $\sim 926mm^2$) to compare with a Tesla V100 GPU card [148] with 4 HBM stacks (total $\sim 1199mm^2$), where one HBM

stack consumes $\sim 96mm^2$ footprint [149].

Performance Simulation. We build an event-driven simulator from scratch using SimPy [192] to implement the timing model of SIMT cores, DRAM, and interconnect network. This ensures that our implementation and integration of these hardware components are consistent. We use the timing parameters of DRAM banks and other SRAM buffers from cacti [67]. The router latency is from BookSim2 [183]. TSV and on/off-chip buses adopt parameters from previous studies [67, 151, 191]. For the ALU, we use the measured results from PTX instructions [193, 194].

Energy Model. We use state-of-the-art performance tools to extract key power parameters for our hardware components, and our performance simulator generates the trace of hardware events to help with the estimation of dynamic power. In particular, we use the design compiler to get the power values for the SIMT core pipeline based on Harmonica project [6], and we use cacti [67] to evaluate the power of DRAM banks and SRAM buffers. The energy values of routers are from BookSim2 [183] and the power parameters of TSV and on/off-chip links are from prior studies [67, 151, 191].

Area Estimation. We use design compiler [152] to analyse pre-layout area of the vector ALU and the SIMT core pipeline [6]. We use AxRAM's area result [11] 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 [67]. 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 [11]. For multi-row-buffer support, we include the overhead of 128 extra row address latches [184] per memory controller to enable simultaneous activations of 4 subarray row buffers. Our DRAM area overhead estimation method and results are consistent with prior architecture studies of near-bank processing, such as AxRAM [11] and iPIM [50].

Simulator Calibration. We validate our simulator implementation with state-of-

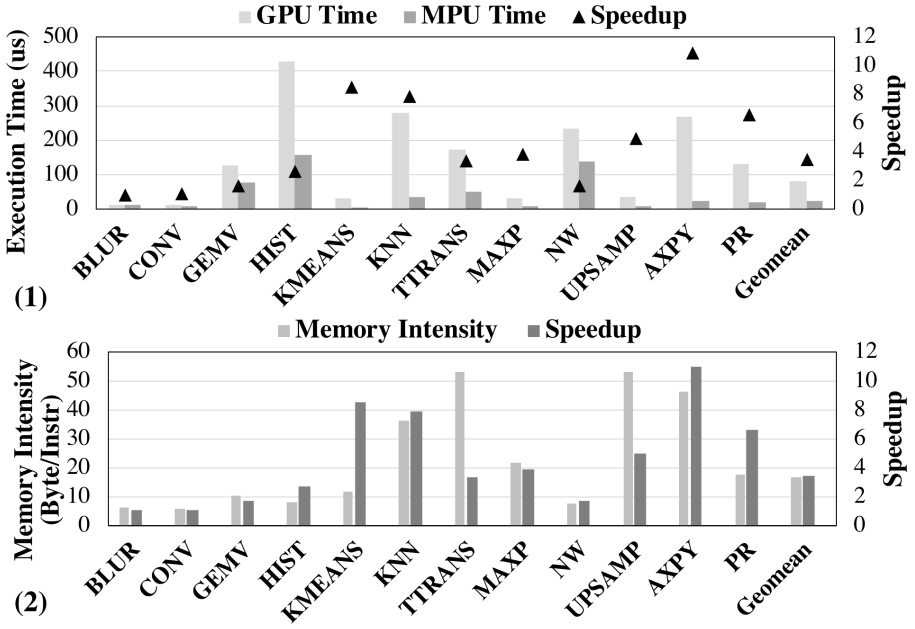


Figure 5.6: (1) Execution time and speedup comparison with the GPU. (2) Workloads memory intensity and speedup.

the-art performance simulators by comparing the simulation results of MPU-Sim and other simulators with the same inputs for key hardware components. In particular, we validate our SIMT cores against the streaming processors in GPGPUSim2 (SM35 TITAN) [195], our DRAM controller against the memory controller in DRAMSim2 (HBM2E) [196], and the network interface against the NoC components in BookSim2 (2D mesh) [183].

GPU Baseline. The GPU performance and power results are collected on real GPU hardware with the help of *nvprof* and *nvidia-smi*, respectively. We include detailed steps to reproduce GPU performance and power on NVIDIA V100 GPU real hardware in the file benchmark/README.md of MPU source code repository.

5.4.2 Performance, Area, Energy, and Thermal Analysis

Performance. MPU achieves 3.45× speedup on average over the GPU as shown in Fig.5.6 (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.5.4.3), and the compiler optimizations for the locations of instructions (Sec.5.4.4).

To further explain different speedup numbers across workloads, we plot the memory intensity (Byte/Instruction) and the speedup of these workloads in Fig.5.6 (2). First, we 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 pipeline units. In Table.5.3, we evaluate the area of added components and normalize the total overhead to a DRAM die ($96mm^2$ [149]). Thanks for our compiler optimizations (Sec.5.4.4) which significantly reduce the near-bank register usage, we shrink the near-

Table 5.3: Area evaluation of MPU components on the DRAM die considering DRAM process overheads.

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

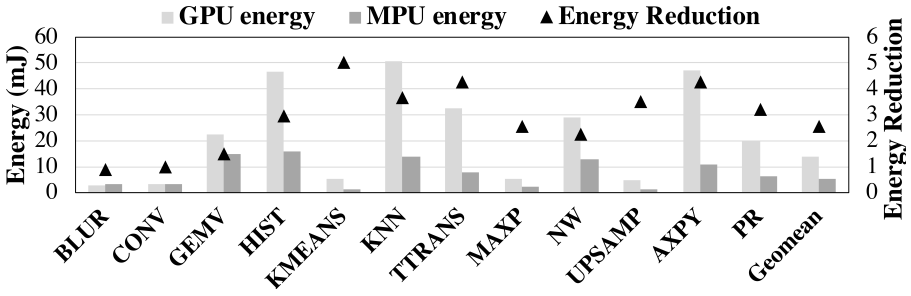


Figure 5.7: Energy and energy reduction comparison with the GPU.

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 [50]. According to the synthesis result of Harmonica [6] 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 ($3.5mm^2$ [69]) 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.5.7). 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

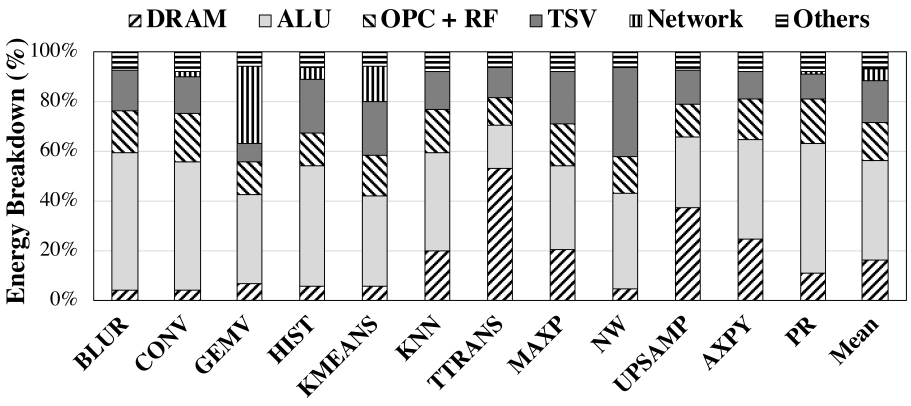


Figure 5.8: MPU energy breakdown.

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.5.7 is approximately proportional to the speedup in Fig.5.6 (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.5.8. 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 great energy reduction.

Thermal Analysis. MPU's peak power is $83W$ per processor considering both

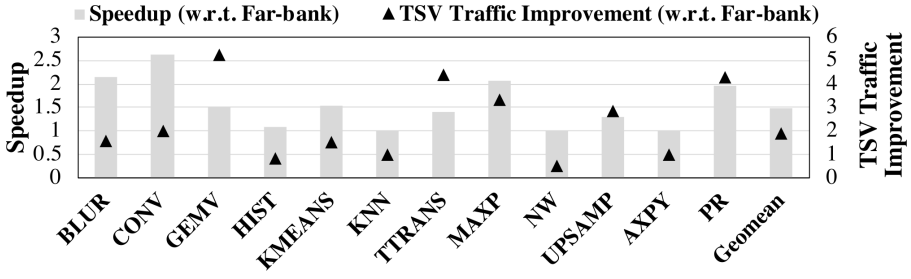


Figure 5.9: Comparison of near-bank / far-bank smem.

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 $105°C$ [149], and we conservatively assume the DRAM dies in our case operates under $85°C$. A prior study on 3D PIM thermal analysis [154] shows that active cooling solutions can effectively satisfy this thermal constraint ($85°C$). Both commodity-server active cooling solution [72] (peak power density allowed: $706mW/mm^2$) and high-end-server active cooling solution [71] (peak power density allowed: $1214mW/mm^2$)) can be used.

5.4.3 Architecture Analysis

Shared memory optimization. To understand the benefit of our near-bank shared memory, Fig.5.9 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 shared memory. This will create a lot of register movement traffic and congest the TSVs.

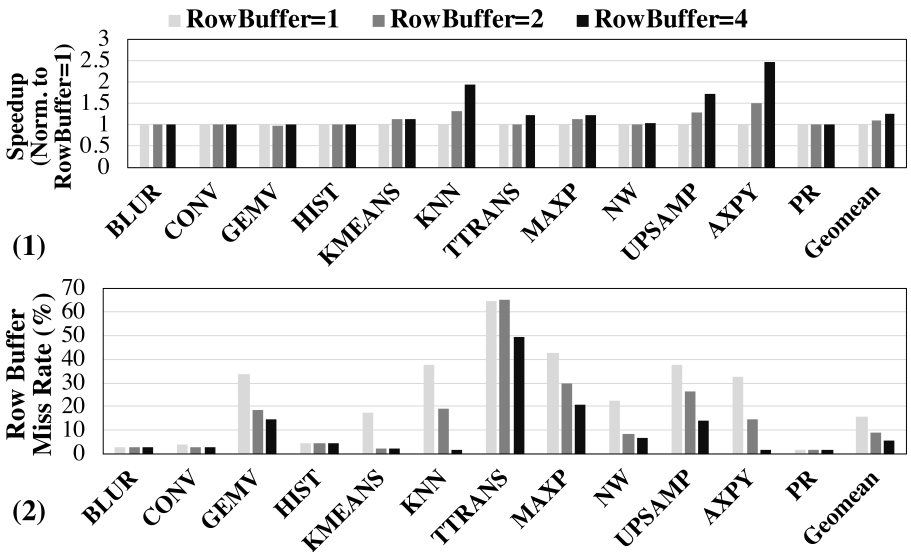


Figure 5.10: Comparison of the number of activated row-buffers on (1) performance and (2) row-buffer miss rate

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.5.10 shows such performance comparisons where the speedup is normalized to a single row-buffer. As shown in the Fig.5.10 (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.5.10 (2) indicates that as we increase activated row-buffer numbers to 2 and 4, the miss rate reduces from 15.60% to 9.20% and 5.45%, respectively. Because more activated row-buffers can effectively

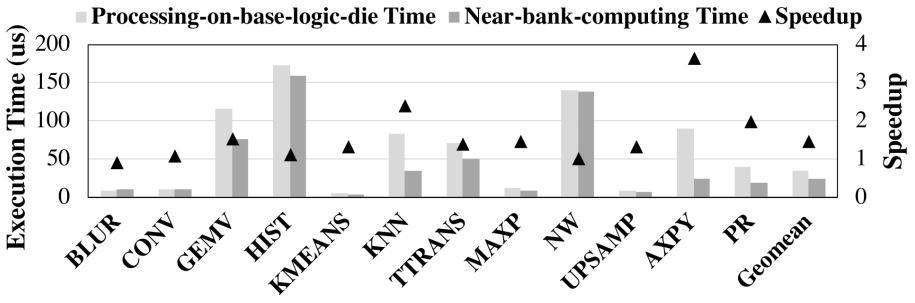


Figure 5.11: Execution time and speedup comparison with the processing-on-base–logic-die solution.

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.5.11 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.

5.4.4 Effectiveness of Compiler Optimizations

We first conduct static analysis according to the iterative algorithm introduced in Sec.5.3 to infer the locations of registers. The breakdown of registers on different locations for all workloads is shown in Fig.5.12. On average, 32.5% registers only appear in near-bank locations, 63.7% registers only appear in far-bank locations, and 3.8% registers could

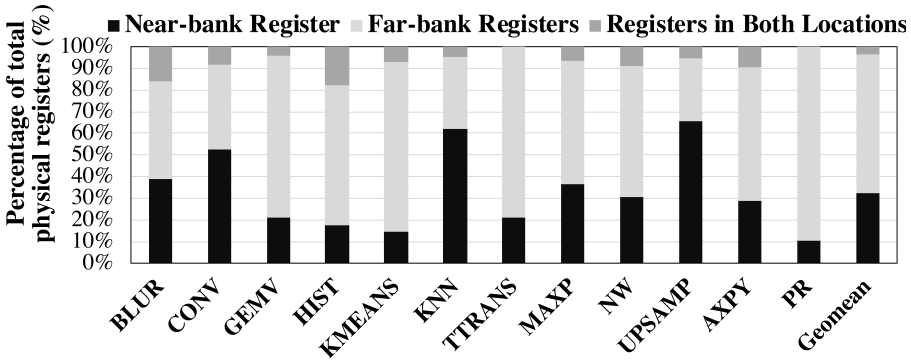


Figure 5.12: Register location analysis.

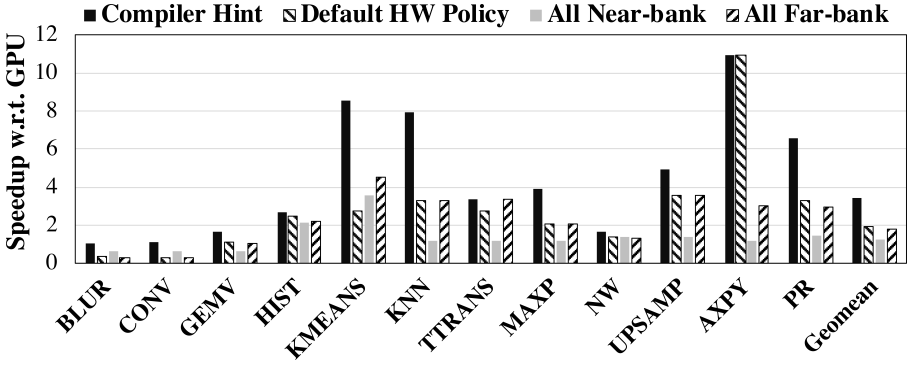


Figure 5.13: Comparison of performance for different instruction location policies.

appear in both locations. Because only registers appearing in near-bank locations need to use the near-bank register file, we effectively shrink the size of the near-bank register file to reduce its area overhead. This breakdown also demonstrates a clear separation of near-bank registers from far-bank registers. Only a small portion of registers could appear in both locations. This clear separation comes from a clear separation of two classes of dependency chains. The first class involves computations on the data value loaded from the DRAM, and the second class involves integer calculations for DRAM addresses and control-flow related variables, such as loop variables. Usually registers for these two classes of dependency chains do not interfere with each other, as registers associated with the first class usually exist in near-bank locations, and registers related to the second class reside on the base logic die. Therefore, for most registers (more than

95%) we can assign them to a certain near/far-bank location to reduce the register file usage.

We further evaluate the performance of different instruction location policies with the GPU as shown in Fig.5.13. On average, using the proposed instruction annotation optimizations, we achieve 3.45× speedup w.r.t. GPU. However, using hardware's default instruction location policy, offloading all instructions to near-bank compute-logic, or offloading all instructions to far-bank compute-logic, we achieve 1.92×, 1.22×, and 1.78× speedup, respectively. Compared with the default hardware policy and both naive offloading strategies, our instruction location annotation is based on the annotated register location. Because of the clear separation of two classes of aforementioned dependency chains, our instruction location annotation assigns most of the computation on data values to near-bank and computation on addresses or control-flow conditions to far-bank. As a result, the register movement traffic on TSVs is minimized, which eventually boosts the performance of programs running on MPU.

5.5 Related Work

General Purpose Near-data-processing Platforms: Pioneering studies [124, 126, 161] attempted to integrate the entire processor on the DRAM die, which incurs considerable area overheads. Compared with them, MPU only places lightweight components in the DRAM dies through a hybrid pipeline, which significantly reduces the overhead. Recently, there are a number of practical general purpose near-data-processing solutions that explore near-cache [197], near-memory-controller [198], near-DIMM [167, 199], and 3D-stacking processing-on-logic-die CPU-style [92, 200, 201] and GPU-style [4, 5, 7, 8, 10] platforms. However, these solutions have several drawbacks. First, they have moderate bandwidth improvement, due to the hierarchically shared bus of the main memory. To

overcome this drawback, MPU unleashes bank-internal bandwidth through near-bank computing. Second, the communication between the host and the near-data logic introduces extra data traffic because of shared memory space, which may offset the benefit of near-data-processing, including fine-grained instruction offloading overhead [4, 92], cache coherence traffic [200], concurrent host access stall [201], and inconsistent data layout requirement [10]. Different from these prior studies, MPU has an independent memory space and supports end-to-end kernel execution, the same as discrete GPU cards [179, 180].

Domain-specific Near-data-processing Accelerators: A large number of previous studies have explored domain-specific accelerators using near-data-processing ideas, including approximate computing [11], image and video processing [50, 202], deep learning [12, 15, 203], graph analytics [103], bioinformatics [169], garbage collection [164], address translation [204] and data transformation [205]. These designs usually adopt domain-specific processing logic, customized data paths, and application-tailored software mapping strategies. The lack of programmability for these accelerators confines them to a niche application market, adding non-recurring engineering costs for silicon manufacturing. In contrast, the SIMT programming model supported by MPU can benefit a wide range of data-intensive parallel programs, and our end-to-end compilation flow greatly eases the burden of programmers.

Analog Process-in-memory Architecture: In addition to the digital near-data-processing solutions, recently there is a surge in researches about analog process-in-memory (PIM) architecture [206, 207]. Different from the digital solution where the memory array and the computing logic are separate, analog solutions modify the memory array to integrate computing functionalities within memory arrays, thus achieving extremely high computation throughput and energy efficiency. However, these designs suffer from analog noise [172], limited write-endurance issues of non-volatile devices [208],

and high overhead of analog-digital converters [209]. Although analog PIM solutions are promising for certain application domains such as neural network [48, 95, 100], they are still challenging for general purpose computing usage. In comparison, MPU adopts commercially available 3D stacking technologies [149, 181] without modifying the DRAM bank's circuit, and this work has demonstrated promising results of MPU on general purpose data-intensive workloads.

5.6 Conclusion

This work proposes MPU (Memory-centric Processing Unit), the first SIMT processor based on 3D-stacking near-bank computing architecture. First, we develop a hybrid pipeline where only lightweight 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 6

MPU-Sim: A Simulator for In-DRAM Near-Bank Processing Architectures

In previous chapters, we have developed three different accelerators based on near-bank computing architectures. In particular, our designs, SpaceA, iPIM, and MPU, have demonstrated that with architectural and software innovations, near-bank computing can benefit different application scopes, including SpMV, image processing, and even a wider range of data-intensive parallel computing workloads. The evaluation of these designs shows that near-bank computing delivers an effective bandwidth not only higher than traditional memory bus but also higher than processing on the base logic die of 3D memory cubes.

[1] Despite its potential benefits to a wide range of data-intensive workloads by providing a significant amount of effective bandwidth, it is challenging to develop efficient

[1] ©2022 IEEE. Reprinted, with permission, from Xinfeng Xie, Peng Gu, Jiayi Huang, Yufei Ding, Yuan Xie. "MPU-Sim: A Simulator for In-DRAM Near-Bank Processing Architectures." IEEE Computer Architecture Letters, vol. 21, no. 1, pp. 1-4, 1 Jan.-June 2022, doi: 10.1109/LCA.2021.3135557.

near-bank processing accelerators. From a hardware's perspective, these accelerators need architectural modifications at the bank-group level to place compute logics near memory banks facing stringent area constraints. From a software's perspective, these accelerators need efficient data locality optimization to exploit the benefits of bank-level bandwidth. It urges an open-source simulator for near-bank processing architectures to study solutions overcoming these obstacles. Although there are several simulators for PIM architectures [103, 210, 211], most of them target processing on base logic die, and focus on the integration with the host system. Moreover, these PIM simulators lack the support for software and hardware features to enable near-bank computing. As a result, it is infeasible to directly use existing PIM simulators for studying the hardware designs and software optimizations of near-bank processing architectures.

In this work, we develop MPU-Sim, an open-source simulator [2] for in-DRAM near-bank processing architectures. MPU-Sim is built for MPU [212], a near-bank single-instruction-multi-thread (SIMT) processor, which supports SIMT programming model for general-purpose data-intensive workloads via near-bank computing. Because MPU is a general-purpose SIMT processor, it includes near-bank compute logics, SIMT processors on the base logic die, and NoC components for inter-vault communication. MPU-Sim supports various hardware designs with the customization of near-bank compute logics and base logic die control units. Moreover, as MPU-Sim takes programs in PTX instructions generated from CUDA programs, it exposes a generic programming interface for users to explore compiler design and optimizations. Our contributions of this work are summarized as follows:

- We develop an open-source simulator, MPU-Sim, for general-purpose near-bank processing architectures.It runs CUDA programs and enables software optimization studies in addition to hardware design explorations.

[2]GitHub link: `https://github.com/GD06/mpu-sim_distribution`

- We conduct calibration studies for key components (processor, DRAM, and NoC) in MPU-Sim with state-of-the-art simulators to validate our simulator implementations.
- We conduct case studies for hardware and software optimization opportunities in near-bank processing architectures with MPU-Sim to demonstrate its potential usage.

6.1 MPU Simulator

We first provide an overview of MPU architecture and software interface in Section 6.1.1. Then, we detail the design and implementation of MPU-Sim in Section 6.1.2. Finally, we introduce the useful auxiliary tools of MPU-Sim for performance analysis and power modeling in Section 6.1.4.

6.1.1 MPU Overview

MPU-Sim is a simulator for MPU [212], a general-purpose near-bank processing architecture based on 3D memory technology. The architecture overview of MPU is shown as Figure 6.1. MPU is composed of several processors connected through off-chip network links. There are several MPU cores inside a processor connected through on-chip network links. Each MPU core includes several subcores, and each subcore is a simple in-order pipeline. Moreover, each MPU core includes several near-bank processing units (NBUs) on DRAM dies where each NBU is associated to one DRAM bank group. There are ALUs inside NBUs, thus the NBU can perform most of the simple arithmetic operations, such as add and multiply. During the execution of MPU kernels, the subcore is able to offload instructions to an NBU for near-bank processing without transferring data to

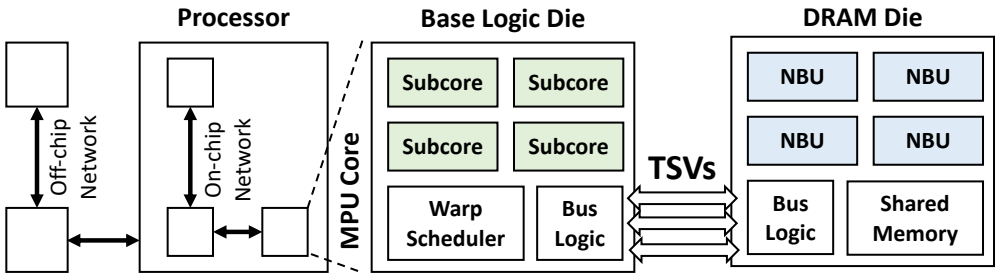


Figure 6.1: The overview of MPU architecture.

compute logics on the base logic die. Because MPU [212] is designed for general-purpose processing, it can be adapted to other near-bank processing accelerators by tailoring control logics in subcores and compute logics in NBUs.

In terms of the software interface, MPU adopts the SIMT programming model to exploit massive bank-level parallelism and address the challenge of control logic and communication overheads. First, there are a large number of lightweight NBUs associated with bank groups leading to a good fit for the SIMT programming model to exploit massive fine-grain bank-level parallelism. Second, the SIMT programming model saves the number of control-logic units by sharing the same control-logic unit among several NBUs, which reduces area overheads. Third, the SIMT programming model reduces the number of instructions communicated between control-logic units and NBUs thanks to instruction sharing among threads from the same thread warp.

6.1.2 Design and Implementation

MPU-Sim focuses on the program execution on MPU by assuming programs and data are ready inside MPU's memory space when the simulation starts. Thus, MPU-Sim does not cover the data transfers between host CPUs and MPU devices, and this design decision is the same as other accelerator simulators, such as GPGPU-Sim [195]. As shown in Figure 6.2, the simulation interface of MPU-Sim takes three inputs, *program*,

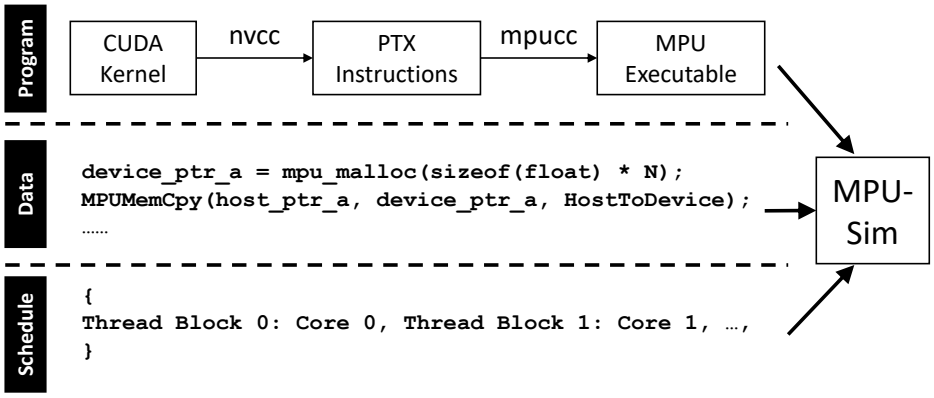


Figure 6.2: The simulation interface of MPU-Sim.

data, and *schedule*. First, MPU supports the execution of PTX instructions, which can be lowered from CUDA programs using *nvcc* without any modifications to CUDA source code. Second, programmers need to allocate data in MPU and transfer data from the host device. Third, MPU-Sim takes a thread block scheduling which specifies the mapping of thread blocks to MPU cores. These interfaces leave a flexibility to explore software optimizations for near-bank processing architectures. For example, instruction offloading optimization can be explored at the compiler backend (*mpucc*), data layout can be customized through the memory allocation, and thread block scheduling can be tailored to study runtime scheduling optimizations.

Listing 6.1 demonstrates an example code snippet of scalar-vector multiplication on MPU-Sim. The function *ScalarVectorMultiply* is a CUDA kernel for the computation. The main function includes memory allocation and memory transfers to set up input data on MPU. Then, it invokes the computation kernel that is running on MPU. Finally, it transfers output data from the memory space to the host memory space. This programming interface is similar to CUDA programming, which eases the burden of writing MPU specific programs from programmers.

Figure 6.3 demonstrates a block diagram of simulation components inside MPU-

```
// CUDA kernel for scalar-vector multiplication
__global__ void ScalarVectorMultiply(float* input,
    float* output, float alpha, int len) {
    int numThreads = gridDim.x * blockDim.x;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = tid; i < len; i += numThreads) {
        output[i] = alpha * input[i];
    }
}

int main() {
    ...
    // Memory allocation on MPU
    mpu_malloc(mpu_input_vec, len * sizeof(float));
    mpu_malloc(mpu_output_vec, len * sizeof(float));
    // Transfer the input data to MPU
    mpu_memcpy(mpu_input_vec, host_input_vec,
               len * sizeof(float), Host2Device);
    // Launch kernel for the computation on MPU
    ScalarVectorMultiply<<<GridCfg, BlockCfg>>>(
        mpu_input_vec, mpu_output_vec, alpha, len);
    // Transfer MPU computation results
    mpu_memcpy(host_output_vec, mpu_output_vec,
               len * sizeof(float), Device2Host);
    ...
}
```

Listing 6.1: Code example of scalar-vector multiplication.

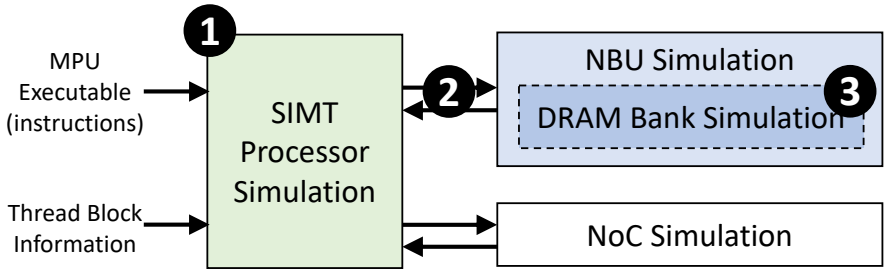


Figure 6.3: A block diagram of MPU-Sim components.

Sim. The SIMT processor simulation takes the program and thread block information as inputs. During the simulation, it offloads instructions to near-bank processing units (NBU simulation) according to the location of each instruction annotated by *mpucc*. It also communicates with hardware components (e.g., memory banks) in other vaults through the NoC interface (NoC simulation). The implementation of MPU-Sim includes three key components: SIMT processors, DRAM, and NoC. Based on these components, MPU-Sim provides a flexibility to study hardware designs, such as customizing compute

logics in NBUs to study near-bank processing accelerators. Additionally, the flexibility of these hardware components can enable the research of other PIM architectures. For example, modifying an NoC mech interconnect structure to a shared bus structure could help generalize our near-bank processing architectures from HMC-based to DIMM-based designs. We conduct calibration studies for these three key components in Section 6.2 by comparing with state-of-the-art simulators.

6.1.3 Simulator Features

Compared with existing open-source HMC-style PIM simulators [103,210,211], MPU-Sim supports new hardware and software features for studying near-bank processing architectures. From the software perspective, MPU-Sim supports the SIMT programming model (CUDA programs) to exploit massive fine-grain bank-level parallelism and reduce control and communication overheads while most of the existing open-source HMC-style PIM simulators are based on OpenMP. From the hardware perspective, MPU-Sim models various architectural components differently for near-bank processing architectures. First, the SIMT processor (Figure 6.3 ❶) adopts a split processor pipeline to enable instruction offloading to NBUs. Second, TSVs between SIMT processors and NBUs (Figure 6.3 ❷) have the arbitration capability as they are shared by multiple subcores and NBUs. Third, DRAM simulation (Figure 6.3 ❸) supports individual bank controls for the flexibility of computation from independent NBUs. These software and hardware features help identify new challenges (case studies in Section 6.3) and enable new research of software and hardware designs for near-bank processing architectures.

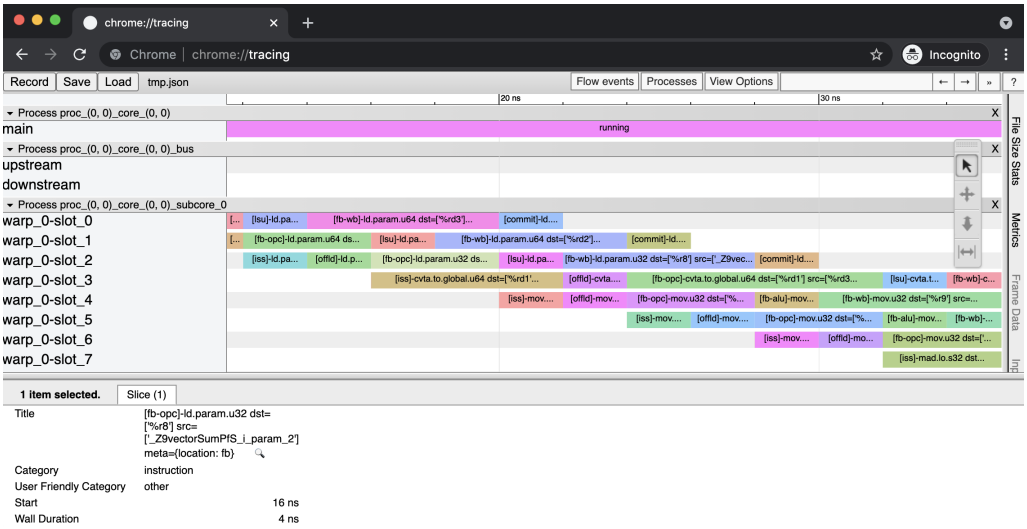


Figure 6.4: An example of performance profiling traces.

6.1.4 Auxiliary Tools

We develop several auxiliary tools in MPU-Sim to help with performance and energy analysis. First, MPU-Sim has a profiling mode that helps with the performance analysis. The profiling mode of MPU-Sim logs detailed performance traces from all hardware components. A detailed performance trace includes detailed hardware events, and each hardware event includes its type, location, start timestamp, and end timestamp. These hardware events can be imported to *chrome://tracing* for visualization. For example, Figure 6.4 shows the snapshot of a hardware trace visualization in *chrome://tracing*. This detailed profiling and visualization helps with the performance analysis, especially analyzing program behaviors and identifying the bottlenecks of hardware components. Second, MPU-Sim outputs the performance counters of each hardware component to help with the energy analysis. We build an energy model that includes the modeling of static energy and dynamic energy. We model the static energy as the static power multiplied by the execution time where the static power is independent from program execution. We model the dynamic energy by accumulating the dynamic energy of each hardware event. For example, the dynamic energy of an NoC link is modeled as the total data transfer

across this link during the program execution multiplied by the energy for transferring a data unit. Another example is the dynamic energy of scratchpad memory. The dynamic energy of a scratchpad memory is modeled as the accumulation of the energy of each read or write transaction during the program execution. With both static and dynamic energy results, MPU-Sim can also output the energy consumption for each hardware component.

6.2 Calibration Studies

In this section, we conduct calibration studies to validate our simulator implementation for the key hardware components in MPU-Sim with state-of-the-art hardware simulators. In particular, we calibrate the processor part of MPU-Sim with GPGPU-Sim [195], the DRAM part with DRAMSim2 [196], and the NoC part with BookSim [183], respectively. Among all studies in this chapter, we evaluate a set of data-intensive workloads from various application domains, including image processing (BLUR, CONV, HIST, UPSAMP), machine learning (CONV, KMEANS, KNN), linear algebra (GEMV, TTRANS, AXPY, PR), and bioinformatics (NW). They are implemented in CUDA and shipped with our simulator repository.

Processor calibration study: We conduct a calibration study to validate the implementation of our SIMT processor by running workloads in both GPGPU-Sim [195] and MPU-Sim, and comparing simulation results. Because the memory hierarchies of GPU and MPU are fundamentally different, we assume an ideal DRAM and NoC during this calibration study. Figure 6.5 elaborates the comparison of simulation results between MPU-Sim and GPGPU-Sim. Because we take GPGPU-Sim as the reference implementation, the simulation error is defined as the runtime difference between GPGPU-Sim and MPU-Sim over GPGPU-Sim's runtime. Because our MPU-Sim does not accurately

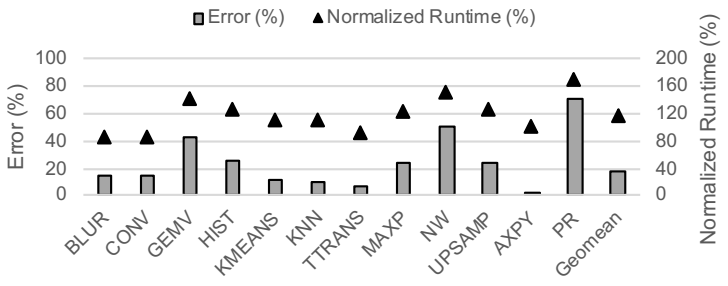


Figure 6.5: The simulation error and normalized runtime of workloads on MPU-Sim compared with GPGPU-Sim.

model shared memory, workloads with intensive shared memory usage in GPU, such as GEMV, NW, and PR, have larger differences (errors) in simulation results. Compared with GPGPU-Sim, MPU-Sim has an average of 17.03% error and the average normalized runtime is 116.05%.

DRAM calibration study: We conduct a DRAM calibration study by comparing the average latency of DRAM requests simulated by DRAMSim2 [196] and MPU-Sim. In particular, we generate two DRAM traces in a random access pattern with 100% and 67% of DRAM read transactions respectively, and feed these two DRAM traces to both DRAMSim2 and MPU-Sim under different DRAM request bandwidth. The average latency of DRAM requests in the simulation is shown in Figure 6.6. It shows that the average latency differences between DRAMSim2 and MPU-Sim are 8.88% and 9.87% for traces with 100% and 67% read transactions. These latency differences come from different DRAM refreshing mechanisms. In particular, DRAMSim2 deploys a per-rank refreshing mechanism while MPU-Sim uses a per-bank refreshing mechanism because DRAM banks in MPU-Sim are independent for the flexibility of simulating near-bank processing architectures.

NoC calibration study: We conduct an NoC calibration study by comparing the simulation results of BookSim [183] and the NoC in MPU-Sim. We implement a uniform random traffic generator to inject single-flit packets for both 4×4 and 8×8 NoCs, where

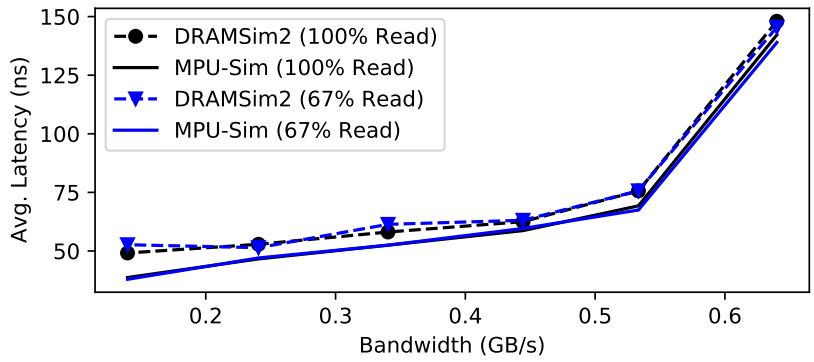


Figure 6.6: The average latency of DRAM requests under different input bandwidth for 100% and 67% read transactions.

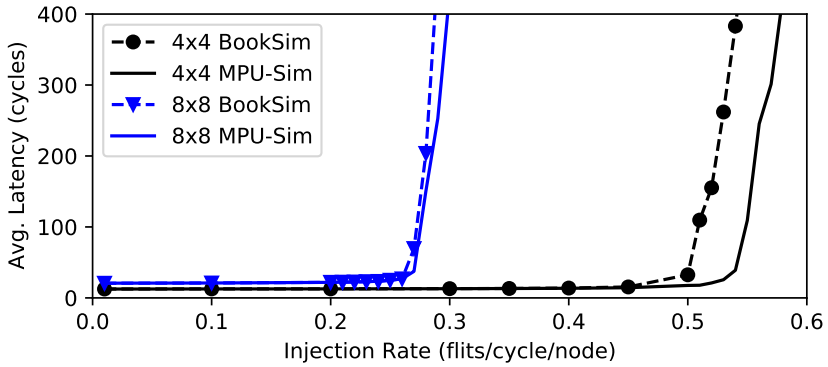


Figure 6.7: Load-latency curves with uniform random traffic pattern for 4×4 and 8×8 meshes.

each router has a three-stage pipeline with look-ahead XY routing and a 4-flit buffer for each input port. Figure 6.7 plots the load-latency curves. The result shows that the zero-load latency differences between BookSim and MPU-Sim are 4.25% and 2.57%, while the throughput differences are 7.95% and 3.21% for 4×4 and 8×8 meshes, respectively.

6.3 Case Studies

In this section, we conduct case studies using MPU-Sim for hardware and software optimization opportunities in DRAM-based processing-in-memory accelerators to demonstrate a potential usage of MPU-Sim. In particular, we conduct two case studies with

MPU-Sim to demonstrate that MPU-Sim helps identify potential hardware and software challenges for near-bank processing architectures. Because of the missing support of critical features, as detailed in Section 6.1.3, existing HMC-style PIM simulators can hardly identify these challenges in the scope of near-bank computing.

DRAM refresh study: In the first case study, we focus on the impact of DRAM refresh on the performance of near-bank processing architectures. In particular, we set up DRAM timing into two configurations, one without DRAM refresh and the other with standard DRAM timing including DRAM refresh. We also consider two different page policies, open page and close page, for DRAM timing. Figure 6.8 shows the normalized runtime of an ideal DRAM without refresh over the standard DRAM. We have several observations from Figure 6.8. First, DRAM refresh degrades the performance of near-bank processing by blocking DRAM requests during refresh periods. In particular, there are 3.26% and 5.88% slowdowns on average for open page and close page policies respectively. Second, the performance degradation of near-bank processing with an open page policy is less than that of a close page policy. Because DRAM refresh increases the number of pending DRAM requests in memory controllers by blocking requests during refresh, it provides more opportunities for an open page policy to reorder requests thus reducing the number of row activation and pre-charge commands. In the open page policy, some workloads, such as TTRANS, even have performance slowdowns when disabling DRAM refresh due to this reason. Profiling results of TTRANS with an open page policy shows that the number of row activation and pre-charge increases 25.23% and 30.45% respectively after disabling DRAM refresh. Third, different workloads are affected by DRAM refresh differently. The performance of some workloads (BLUR, CONV, GEMV, and NW) is less sensitive to DRAM refresh than the others because of their lower DRAM access intensity [212]. In summary, this case study shows that DRAM timing play an important role in near-bank processing and MPU-Sim can be used to study potential hardware

Figure 6.8: The normalized runtime of an ideal DRAM without refresh for both open page and close page policies over the DRAM with actual HBM timing.

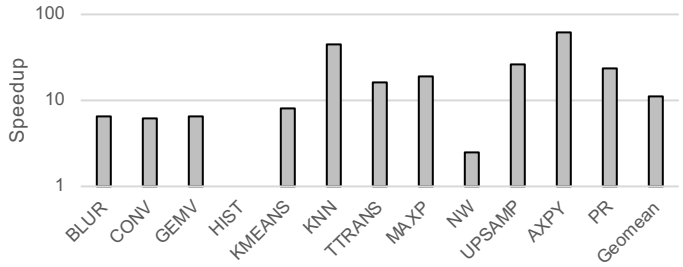


Figure 6.9: The speedup of an interleaved thread block scheduling over the baseline.

optimizations.

Thread block scheduling study: In the second case study, we focus on the impact of data locality on the performance of near-bank processing architectures. In particular, we study the thread block scheduling that maps a thread block ID to an MPU core ID to specify the execution location of each thread block. We compare two thread block scheduling schemes: The baseline scheduling uses a continuous uniform division for thread blocks to MPU cores. The MPU core ID for a thread block i is calculated as $\frac{i}{N}$, where N stands for the number of MPU cores. Then we have an interleaved thread block scheduling, which is aligned to the memory space mapping to DRAM banks. The MPU core ID for a thread block i in this interleaved thread block scheduling is represented as $i \ mod \ N$. On average, the performance speedup of an interleaved thread block scheduling over the baseline is 10.90× as shown in Figure 6.9. This significant speedup results from the huge latency and bandwidth gap between accessing data in local banks and remote

banks. For the histogram workload (HIST), because the number of thread blocks equals to the number of MPU cores, these two thread block scheduling schemes are identical to each other, which result in a speedup of 1.0×. Similar to DRAM refresh study, we found that some workloads (BLUR, CONV, GEMV and NW) are less sensitive to data locality because of a lower DRAM access intensity although the speedups of these workloads are still significant. In summary, the data locality plays an important role in the performance of near-bank processing architectures due to the intrinsic non-uniform memory access, and MPU-Sim can be used as a tool to study potential software optimizations, such as thread block scheduling.

6.4 Related Work

Although prior studies develop HMC-style PIM simulators, such as GraphPIM [103], PIMSim [210], and MultiPIM [211], they lack necessary software and hardware supports for near-bank processing architectures.

From software's perspective, the support for the SIMT programming model is missing from existing open-source HMC-style PIM simulators although there are some in-house PIM simulators modified from GPGPU-Sim. Because of a massive number of DRAM banks, the SIMT programming model can reduce control-logic area overhead and exploit fine-grained parallelism. Thus, MPU-Sim can help with the studies of software optimizations based on the SIMT programming model for near-bank processing architectures.

From hardware's perspective, MPU can enable future research of hardware designs based on near-bank processing architectures. Compared to existing open-source HMC-style PIM simulators, MPU-Sim supports new hardware features with clear abstraction and modularity for bank-level control and datapath, such as individual memory bank controls and TSV arbitration. We detail these architectural supports as follows: First,

existing HMC-style PIM architectures usually directly use off-the-shelf processor designs that are infeasible to be placed near memory banks due to area overheads. MPU uses a split processor pipeline where control logic resides on the base logic die while compute logic resides on DRAM dies for near-bank processing. These new split processor pipeline designs are not supported in existing HMC-style PIM architectures. Second, vault controlled on the base logic die control TSVs in each vault and individual banks have no flexibility to arbitrate for TSV resources. This TSV arbitration is important for near-bank processing because different memory banks could request data from other locations at different timestamps, resulting in traffic contention on TSVs. Third, memory banks within a vault are controlled by the same memory controller in HMC-style PIM simulators. Thus they do not have the flexibility to control each memory bank independently. This independent control is important for near-bank processing because compute logics are designed at the granularity of memory banks.

6.5 Conclusion

In this project, we present MPU-Sim, an open-source performance simulator for in-DRAM near-bank processing architectures. In particular, we introduce the design principles of MPU-Sim and its implementation for key hardware components. Moreover, we conduct calibration studies for the performance simulation of these key hardware components with state-of-the-art hardware simulators to validate the implementation of MPU-Sim. Finally, we conduct two case studies, the DRAM refresh study and thread block scheduling study, to demonstrate the potential usage of MPU-Sim in the future to help with the research of hardware and software optimizations for in-DRAM near-bank processing architectures.

Chapter 7

A Transferable Approach for Partitioning Machine Learning Models on Multi-Chip-Modules

This chapter focuses on developing a novel machine learning-based approach to optimize the workload partitioning problem for multi-chip-modules (MCMs) that have a distributed memory space similar to near-bank processing architectures. Although our previous chapters propose several hardware-specific optimizations for each individual hardware design, some common software optimization challenges remain unsolved. For example, the memory bandwidth and latency gap between local accesses and remote accesses are so significant that the workload partitioning on such a distributed memory space play important roles in performance and energy efficiency. In this chapter, we present our work that targets the partitioning problems of machine learning (ML) workloads on an MCM-based ML accelerator to come up with a novel ML-based solution for this common software challenge of near-bank processing architectures.

In this work, we propose a partitioner that combines reinforcement learning (RL)

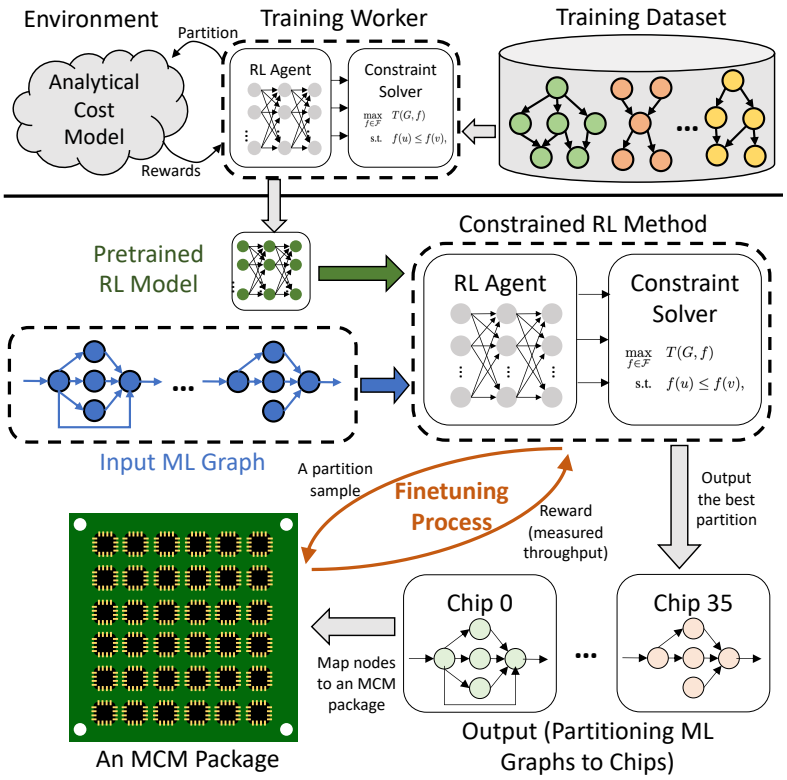


Figure 7.1: The overview of our constrained RL method that includes an offline pre–training on a training dataset and an online fine-tuning to generate an ML model partition. The objective is to optimize throughput or end-to-end latency targeting an MCM package.

with a constraint solver to address the shortcomings of existing solutions. The deep RL engine learns to interact with the dynamic compilation environment and creates an action distribution that is biased towards a balanced partition while the constraint solver takes the distribution and generates partition solutions that respect various constraints (e.g. acyclic dataflow, no skipping chip). The overview of our RL-based method with a constraint solver is shown in Figure 7.1. To further reduce the compilation time, we develop a pre-training and fine-tuning method to generalize the pre-trained policies to unseen input graphs. Our RL-based partitioner shows strong generalization via the pre-training even when using an analytical cost model during the pre-training followed by fine-tuning on real hardware. The use of an analytical cost model saves hardware resource

consumption and reduces pre-training time from a week to hours. During the deployment of our RL-based method, this generalization significantly reduces the number of samples and shortens the compilation time caused by expensive real hardware evaluation. Our contributions are broken down in the following order:

- We define the problem of multi-chip partitioning for MCMs and propose a method that combines the capabilities of deep RL networks and constraint solvers to search for good partitionings in a search space where valid solutions are extremely sparse due to constraints imposed by the hardware architecture.
- Our evaluation for BERT, a production-scale model, on real hardware demonstrates that our RL-based partitioner achieves 6.11% and 5.85% higher throughput than random search and simulated annealing upon its convergence.
- We demonstrate strong transfer learning performance via a pre-training based method. We pre-trained the RL policy on 66 production neural networks from computer vision applications and language models, using an analytical performance model as a reward function. Fine-tuning the pre-trained policy on BERT reduces the search time from more than 3 hours for RL training from scratch to only 9 minutes, while achieving the same runtime performance.

7.1 Motivation

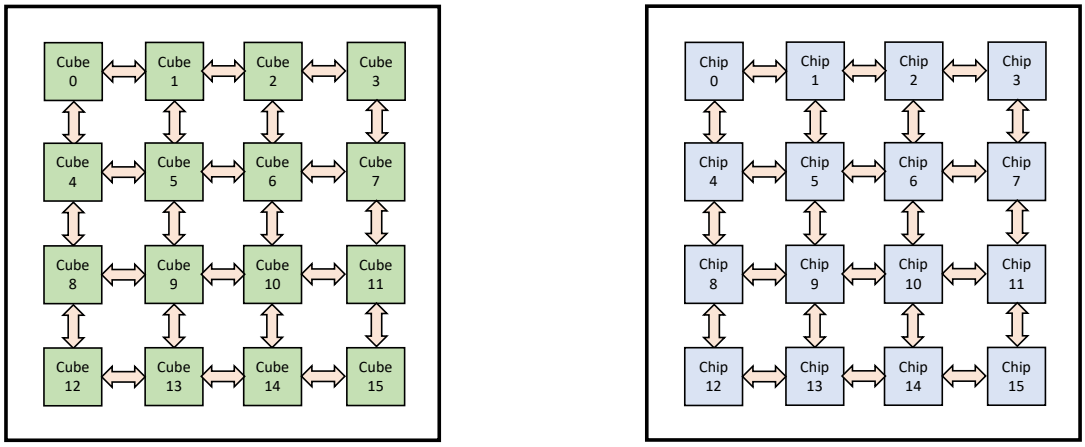
We use an MCM-based ML accelerator as our target hardware platform to study the workload partitioning problem of near-bank processing architectures for two reasons:

First, due to the success of MCMs in real-world industrial prototypes, we can conduct real hardware evaluation while the fabrication of near-bank processing architectures

is still difficult, expensive, and immature. Recent industrial prototypes of ML accelerators, such as Simba [213] and multi-chip TPUs [214], have demonstrated that multi-chip-modules (MCM) can reduce design and fabrication costs while delivering a similar performance and energy efficiency as that of a monolithic chip. Instead of a large monolithic chip, MCM designs are composed of a set of small chips integrated into a package joined by off-chip interconnects [213, 215, 216]. MCM packages reduce the design cost as designing a smaller chip is easier than designing a monolithic large chip, and also have lower fabrication costs since the yield of chips is higher due to a smaller chip area [213].

Second, The memory space abstraction of MCMs is similar to that of near-bank processing architectures. Figure 7.2 presents the architecture comparison between 3D memory based near-bank processing designs and multi-chip-modules (MCMs). Since the memory sizes of individual chiplets in an MCM are significantly smaller than monolithic accelerators, training and inference of large ML models invariably require partitioning the dataflow graph of tensor computations over the chiplets. Multi-chip partitioning is the problem of finding an assignment of operations in a computational graph to chiplets to maximize some performance metrics, typically throughput. The unique hardware characteristics of MCMs make the performance of ML models particularly sensitive to the quality of the partitioning, and therefore finding a good partitioning is an important optimization step in compilers targeting MCMs.

Multi-chip partitioning is challenging for three reasons. First, the specialized hardware architecture imposes constraints that invalidate large parts of the solution space. For example, in a multi-chip TPU package [214], valid partitionings must assign operations to chiplets such that dataflows between chiplets are consistent with their relative positions along a uni-directional inter-ring network (Figure 7.3c). Second, determining whether a partitioning is feasible or not requires executing the subsequent stages in the compilation process. For example, checking whether the peak memory usage for a partic-



(a) 3D Memory-Based Near Bank Processing

(b) Multi-die Chiplet ML Accelerator

Figure 7.2: The architecture comparison between (a) a 3D memory-based near-bank processing design and (b) an MCM-based ML accelerator.

ular placement is less than the available chiplet memory (Figure 7.3f) requires knowledge of the order of scheduling of operations that is only determined at a later compilation pass. Finally, ML compilers usually have stringent time budgets for end-to-end compilation making it harder to find good partitionings in this sparse search space.

There are several existing solutions for the multi-chip partitioning task, such as constraint solvers, compiler heuristics, search-based algorithms, and reinforcement learning. Unfortunately, these solutions can hardly find the optimal partition solution under strong constraints within a stringent time budget in the multi-chip setting. It is a common approach to use constraint solvers in compilers to solve well-formulated optimization problems, such as loop transformations [217]. After encoding the problem as a combinatorial decision optimization, off-the-shelf solvers are then applied to find the solution that minimizes an objective function [217–220]. Unfortunately, some dynamic constraints, such as memory allocation that happens later during the compilation process, are hard to be formulated at the stage of multi-chip partitioning. Moreover, objective functions with a closed-form formulation expressed in a solver logic typically fail to encapsulate the complexity of the MCM system. On the other hand, hand-crafted compiler heuristics,

	CPS	CH	RL	CPS + S	**CPS + RL (Our Work)**
Static Constraints	Yes	Yes	No	Yes	**Yes**
Dynamic Constraints	No	Yes	No	Yes	**Yes**
Requires Close-Form Perf. Model	Yes	No	No	No	**No**
Solution Quality	N.A.	Low	N.A.	Medium	**High**
Time to Solution	N.A.	Fast	N.A.	Slow	**Fast**

Table 7.1: Comparison among Constraint Programming Solver (CPS), Compiler Hueristic (CH), Reinforcement Learning (RL), Search Algorithms with Constraint Solvers (CPS + S), and RL with Constraint Solvers (CPS + RL).

such as greedy algorithms [213] and dynamic programming [221], are frequently used in production compilers. Although they can search for valid partitions efficiently, they often fail to find the optimal placement due to their over-simplification of the performance model. Third, search-based compiler optimizations, such as random search and simulated annealing, can mitigate the problem of inaccurate performance models in real systems by sampling and evaluating partition candidates. However, these optimizations cannot find good candidates in a timely manner when the solution space is overwhelmingly large. In addition, search-based methods do not use prior knowledge to optimize an unseen new graph, thus they always search from scratch resulting in a long search time. Recent work [222] shows that Reinforcement Learning (RL) can be used to efficiently solve graph optimization problems that arise in ML compilers. However, the strict constraints of multi-chip partitioning invalidate most of the decisions in the action space. As a result, conventional RL methods fail due to insufficient valid samples or rewards.

In this work, we propose an RL partitioner working with a constraint solver to address the aforementioned challenges. We summarize the comparison between our work and prior studies in Table 7.1.

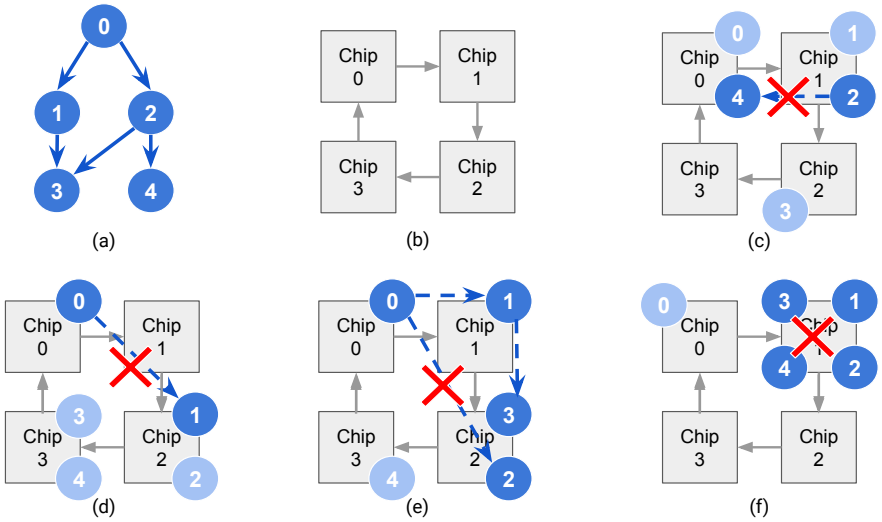


Figure 7.3: Examples of (a) a computation graph representing an ML workload, (b) multiple chips connected by uni-directional links, (c) an invalid partition violating the acyclic dataflow constraint, (d) an invalid partition violating the rule of no skipping chips, (e) an invalid partition violating the triangle dependency constraints, and (f) an invalid partition violating the memory allocation constraint.

7.2 Related Work

Multi-Chip-Module Package: The advance of interconnect and package technology drives the the development of multi-chip-module (MCM) architectures to reduce the design and fabrication cost, e.g., CPU [216], GPU [215], and ML accelerators [213, 223]. Instead of building a large, monolithic chip, an MCM combines small chiplets into a chip package, thus reduces the per-chip complexity and improves manufacture yields. However, coming up with a balanced workload partitioning while minimizing inter-chip communication becomes critical for high workload throughput on an MCM package, as the off-chip communication is both low bandwidth and high latency. Both hardware methods such as active interposers [224, 225] and software methods such as compiler heuristics [213] are developed for the partitioning and mapping problem. We are the first to develop a novel reinforcement learning method tailoring the workload partitioning and placement problem, targeting an edge TPU-based MCM.

Model Parallelism: Power-law in deep learning drives the design of large and complex neural models [226–230]. Model parallelism enables running large models on hardware devices, particularly for edge devices where on-chip memory is scarce. Mesh-TensorFlow [231], GPipe [232], GShard [233] provide various programming primitives to enable the execution of large models on a cluster of hardware accelerators. Pipeline parallelism [232,234] enables model parallelism along the temporal axis, yet still maps computational graphs across multiple accelerators for higher throughput. Although these studies provide computational primitives and conduct heuristic-based optimizations, none of them targets partitioning ML models on an MCM package with hardware constraints. In this work, we develop an automatic model partitioning solution targeting an MCM package.

ML for automatic partition and placement: Reinforcement learning has been used for device placement [222, 235–237] and has demonstrated run time reduction over human-crafted placements and conventional heuristics. Progressive placements [236–238], generate decisions on a per-node basis, so they have difficulty capturing long-distance dependencies over large graphs and are slow in training. Placeto [238] represents the first attempt to generalize device placement using a graph embedding network. GO [222] is the first single-shot method that generates placement decisions for an entire graph and generalizes to unseen data. However, all the above methods do not handle constraints explicitly and can fail when facing strict systems constraints imposed by novel architectures because of an ultra-sparse reward space.

Constrained learning: Differentiable SAT [239] and differentiable optimizer [240] integrate constraints to the end-to-end learning systems, which can learn the logical structure via supervised learning. Constrained Policy Gradient [241] trains neural network policies for high-dimensional control while making guarantees about all policy behaviors throughout training. However, many system constraints can hardly be formulated stat-

ically during the compilation, necessitating an actor to interact with the environment and learn the interacting constraints. None of these studies are tailored for finding optimal solutions in a compilation system with dynamic constraints because of the lack of a closed-form objective formulation. In this work, we leverage the constraint solvers to rule out infeasible partition statically while using reinforcement learning to propose optimal solutions through interaction with a real system.

7.3 Hardware Architecture and Problem Formulation

Hardware specialization provides speedups and higher energy efficiency for ML workloads. However, some specialized hardware impose constraints when mapping the workload during the compilation stage. In this section we introduce the target hardware platform of our problem–a multi-chip TPU [214], then define and formulate the multi-chip placement problem with constraints.

Hardware Architecture: In this work, our target hardware platform is a 36-die multi-chip ML accelerator [214] package with a 1D ring for inter-chip communication. Figure 7.3b shows an example 4-die multi-chip package with only uni-directional links among adjacent chips, and the inter-chip link topology of our multi-chip TPU package is similar to that of Figure 7.3. Each chip has tens of MBs SRAM, and inter-chip links only offer a bandwidth of tens of GB/s. Thus, it is unavoidable to partition production-scale ML models to our multi-chip TPU, and optimizing inter-chip communication is important.

Problem Definition: Denote the set of available chips as $D = \{0, 1, 2, ..., C-1\}$, and a directed acyclic graph $G = (V, E)$ representing an ML workload where V stands for the vertex set of operations and E stands for the edge set of dependencies between operations.

The multi-chip partitioning problem aims to find a function f, which maps V to D denoted as $f : V \mapsto D$, that maximizes or minimizes an objective function. Figure 7.3a shows an example of a computation graph for an ML workload. In this work, we aim to maximize the throughput of ML workloads on ML accelerators. Thus the multi-chip partitioning task can be formulated as Equation 7.1 denoting $T(G, f)$ as the throughput of running the graph G on the hardware with the mapping function f, and $\mathcal{F}$ as the set of all possible mapping functions from V to D.

$$\max_{f \in \mathcal{F}} \quad T(G, f) \tag{7.1}$$

Constraint 1: Acyclic Dataflow Constraint. Since the links among chips are uni-directional, only data transfer from low chip ID from high chip ID is allowed. For example, Figure 7.3c shows an invalid partition where data transfer from chip 1 to chip 0 between node 2 and node 4 is not allowed. Denoting $f(u)$ as the chip ID where the node u is mapped to using the mapping function f, this constraint can be formulated as Equation 7.2.

$$f(u) \leq f(v) \quad \forall (u, v) \in E \tag{7.2}$$

Constraint 2: No Skipping Chips. Since the ML accelerator can pipeline the execution of operators mapped to chips, the backend of the current multi-chip TPU compiler does not allow skipping chips to maximize the throughput. As a part of compilation optimization, we also impose this constraint on this multi-chip partitioning task. For example, Figure 7.3d shows an invalid partition where the chip 1 is skipped without any nodes mapped to it. This constraint can be formulated as Equation 7.3.

$$d \leq f(u) \Rightarrow \exists v \in V, f(v) = d \quad \forall u \in V, d \in D \tag{7.3}$$

Constraint 3: Chip Triangle Dependency. Because the simplified network-on-chip (NoC) routers on the multi-die TPU cannot handle all NoC traffic patterns, direct data dependencies between two chips cannot co-exist with indirect data dependencies between the same chips. For example, Figure 7.3e shows an invalid partition where there is a direct dependency between chip 0 and chip 2 through the data transfer between node 0 and node 2 while there is an indirect dependency chain from chip 0 to chip 1 then chip 2 through the data transfer from node 0 to node 1 then node 3. To formulate this dependency, we define $\delta(d_0, d_1)$ as the length of the longest path from chip $d_0 \in D$ to chip $d_1 \in D$ in the graph G whose nodes are the chips and edges are data dependencies between chips. We then impose this length to be at most one for each direct dependency in the computational graph, as formulated in Equation 7.4.

$$\begin{aligned} \delta(d_0, d_2) &\geq \delta(d_0, d_1) + \delta(d_1, d_2) \quad \forall d_0, d_1, d_2 \in D \\ f(u) \neq f(v) &\Rightarrow \delta(f(u), f(v)) = 1 \quad \forall (u, v) \in E \end{aligned} \tag{7.4}$$

Constraint 4: Dynamic Constraint. In addition to static constraints which can be explicitly expressed in closed-form formulas, there are also constraints from system dynamics. For example, the on-chip memory consumption of a model partition (mapping f) should fit in memory. Figure 7.3f shows an invalid partition that leads to the out-of-memory allocation issue by consuming too much on-chip memory on chip 1. Because these dynamic constraints are not able to be explicitly formulated, we define a boolean function $H(G, f)$ which returns true only if the mapping function f can pass the compilation and hardware evaluation. To ensure the completeness of the problem formulation, we add this boolean function as a constraint in the overall formulas.

Putting It All Together. Combining all static constraints from the hardware architecture of multi-chip TPU and dynamic constraints from the compiler backend, the multi-chip partitioning task that maps a computation graph G to the available set of

chips D can be formulated as Equation 7.5.

$$\begin{aligned}
\max_{f \in \mathcal{F}} \quad & T(G, f) \\
\text{s.t.} \quad & f(u) \leq f(v) && \forall (u, v) \in E \\
& d \leq f(u) \Rightarrow \exists v \in V, f(v) = d && \forall u \in V, d \in D \\
& \delta(d_0, d_2) \geq \delta(d_0, d_1) + \delta(d_1, d_2) && \forall d_0, d_1, d_2 \in D \\
& f(u) \neq f(v) \Rightarrow \delta(f(u), f(v)) = 1 && \forall (u, v) \in E \\
& H(G, f) = True
\end{aligned} \tag{7.5}$$

Although prior studies develop reinforcement learning solutions, strict constraints in Equation 7.5 invalidate most of the possible mappings from the set $\mathcal{F}$. Thus, the reward space is ultra-sparse, making it difficult for the agent to learn an optimal partition. On the other hand, although constraint solvers are good at solving optimization problems, they are not able to handle non-closed form functions. Because the objective function $T(G, f)$ and the dynamic constraint $H(G, f)$ cannot be explicitly formulated, it is impossible to apply constraint solvers directly for solving the problem formulated in Equation 7.5. These shortcomings motivate us to develop our constrained reinforcement learning method detailed in Section 7.4, which combines the reinforcement learning method with the constraint solver to efficiently explore the space of feasible partitions.

7.4 Reinforcement Learning with a Constraint Solver

The overview of our RL-based approach is shown in Figure 7.4. In this section, we will detail our RL framework for generating a chip assignment for nodes in Section 7.4.1, and the constraint solver for generating a valid ML model partition in Section 7.4.2. Then, we will introduce our pre-training pipeline to generalize our RL-based approach in Section 7.4.3.

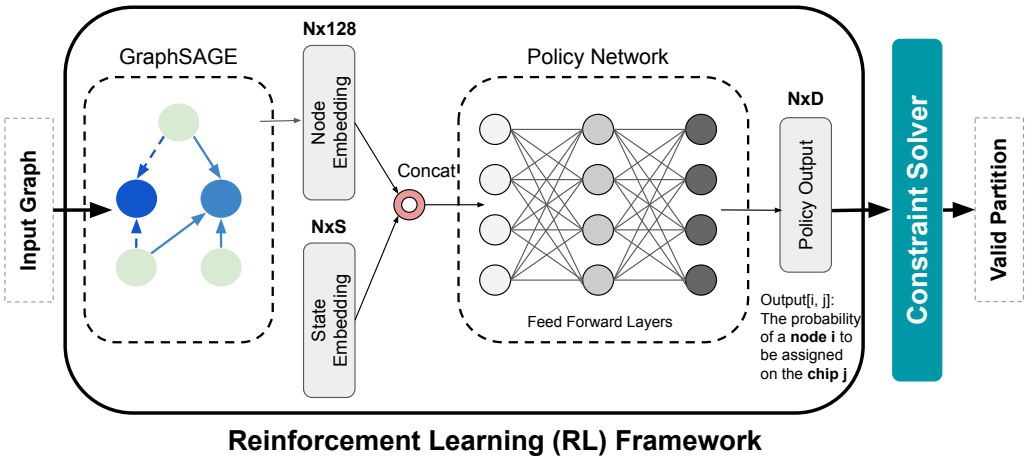


Figure 7.4: The overview of our RL-based method including an RL framework to generate the probability distribution of chip assignments, and a constraint solver to generate a valid partition by sampling according to the policy output from the RL.

7.4.1 Reinforcement Learning

As shown in Figure 7.4, our learnt policy π_θ (policy network) takes the node embedding of a graph, h_G, from a feature network and the current state embedding and generates a probability distribution matrix $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_N]$ where N stands for the number of nodes and the vector $\mathbf{p}_i = [p_{i1}, p_{i2}, ..., p_{iC}]$ stands for the probability distribution of i-th node to C available chips. We adopt GraphSAGE [242] for the feature extraction of an input computation graph to generate h_G, and a fully connected network as the policy network. We train the feature extraction network and the policy network in an end-to-end fashion, using a reward of the execution throughput on the target multi-chip environment.

Let y_i be the action for the i-th node ($y_i \in D$). Ideally, we would like to compute the action distribution of the current node based on the actions of all previous nodes in an auto-regressive manner:

$$p(\mathbf{y}|G) = \prod_{i=1}^{N} p\left(y_i|h_G, y_{i-1}, y_{i-2}, ...\right) \tag{7.6}$$

However, the above is infeasible in practice because the number of nodes can be as large as 10K, and computing the y_i's sequentially can be extremely expensive. To address this issue, we use an iterative but non-autoregressive process as an approximation same as prior work [222]:

$$p(\mathbf{y}^{(t)}|G) = \prod_{i=1}^{N} p\left(y_i^{(t)}|h_G, \mathbf{y}^{(t-1)}\right) \tag{7.7}$$

Although the N sampling procedures are now carried out in parallel within each iteration t, decisions over the N nodes are allowed to mutually influence each other because the process above will be repeated for T times ($T \ll N$). Note the distribution of $\mathbf{y}^{(t)}$ is informed about $\mathbf{y}^{(t-1)}$, the actions made over all the nodes in the previous iteration. At each iteration, a concrete partition solution $\mathbf{y}^{(t)}$ can be sampled from the probability distribution $\mathbf{P}^{(t)}$.

Due to constraints formulated in Equation 7.5, the reward space for the action $\mathbf{y}$ is extremely-sparse. Thus we use the reward of $\mathbf{y}'$ rather than directly using the reward of $\mathbf{y}$ to efficiently learn policy π_θ, where $\mathbf{y}'$ is a valid partition generated from the constraint solver according to $\mathbf{y}$ and partitioning problem constraints. The constraint solver is detailed in Section 7.4.2.

7.4.2 Constraint Solver

The role of the constraint solver in Figure 7.4 is to find a valid partition of the constraint satisfaction problem that follows problem constraints.

The procedure to find a solution is based on the CP-SAT open-source constraint solver [243]. The solver has a variable representing action y_i for each node i and enforces constraints in equations (7.2)-(7.4). Internally, the solver maintains the range of valid values for every variable y_i, called the *domain* of y_i. The procedure to find a solution interacts with the solver by querying the current domain of variables and by manually

Algorithm 6 Constraint solver in SAMPLE mode

Input: A node order $\mathcal{T}$, and a distribution $\mathbf{P}$.
Output: A valid partition $\mathbf{y}'$.
S = init_solver() // Init the constraint solver
$i = 0$
while $i < N$ **do**
  $u = \mathcal{T}_i$
  // Get the current valid domain of the node u.
  $\mathcal{D}_u = S$.get_domain(u)
  y'_u = sample $\mathbf{p}_u$ from $\mathcal{D}_u$
  // Set y_u domain, perform constraint propagation, and backtrack to a previous i if needed
  $i = S$.set_domain(u, $\{y'_u\}$)
end while

setting variable domains. When setting the domain of a variable y_i, the solver internally runs a *constraint propagation* algorithm that recursively prunes the domain of other variables so that they only contain values that are compatible with the new domain of y_i with regards to constraints. If constraint propagation detects an invalid assignment, the solver will backtrack to a previous state, undoing previous decisions.

Our procedure finds a solution by picking the assignment of one node at a time. At each step, it picks a node i and queries the current domain $\mathcal{D}_i$ of y_i from the solver. It then calls the solver to restrict the domain of i to a single value $c \in \mathcal{D}_i$ taken from the domain of y_i. The procedure stops when all nodes are assigned a device.

There are two important factors for our constraint solver to generate a valid partition $\mathbf{y}'$ given an input partition $\mathbf{y}$ and a probability distribution $\mathbf{P}$: (1) the node traversal order, and (2) the strategy to pick a value y_i from the valid domain. First, our constraint solver provides an interface to specify the node order. By default, we generate a random order each time to explore a larger decision space rather than prioritizing a fixed set of nodes that significantly prunes the domain of other nodes. Second, there are two strategies to pick a value y_i from the current valid domain.

- With the SAMPLE strategy, the solver visits nodes according to the input node order. For each node i, it samples a chip ID according to $\mathbf{p}_i$ restricted to the current

Algorithm 7 Constraint solver in FIX mode

Input: A node order $\mathcal{T}$, and a candidate partition $\mathbf{y}$.
Output: A valid partition $\mathbf{y}'$.
S = init_solver() // Init the constraint solver
$i = 0$
while i ¡ 2 × N **do**
  $u = \mathcal{T}_{i \bmod N}$
  if i ¡ N **then**
    $\mathcal{D}_u = S$.get_domain(u)
    if $y_u \in \mathcal{D}_u$ **then**
      $y'_u = y_u$
      $i = S$.set_domain(u, $\{y'_u\}$)
    else
      $i = S$.set_domain(u, $\mathcal{D}_u$)
    end if
  else
    $\mathcal{D}_u = S$.get_domain(u)
    y'_u = randomly sample $\mathcal{D}_u$
    $i = S$.set_domain(u, $\{y'_u\}$)
  end if
end while

domain of the y_i.

- With the FIX strategy, the constraint solver traverses nodes according to the input node order, and assigns y_i to y'_i if y_i is a valid assignment. After the traversal, it repeatedly assigns random chip IDs to remaining nodes (where y_i is invalid) until getting a valid assignment.

The algorithms for SAMPLE and FIX strategy are detailed in Algorithm 6 and Algorithm 7 respectively. In these two algorithms, we rely on the underlying CP-SAT solver (denoted as S) to maintain and update valid domains for all nodes. In particular, the function *get_domain* returns the current valid domain for a node, and the function *set_domain* performs the constraint propagation step to update valid domains for all nodes. The loop index, i, is the number of decisions set by calling *set_domain*. Calling *set_domain* returns the new value of i. In general, this is $i + 1$ but this can also be a lower value if the solver backtracks.

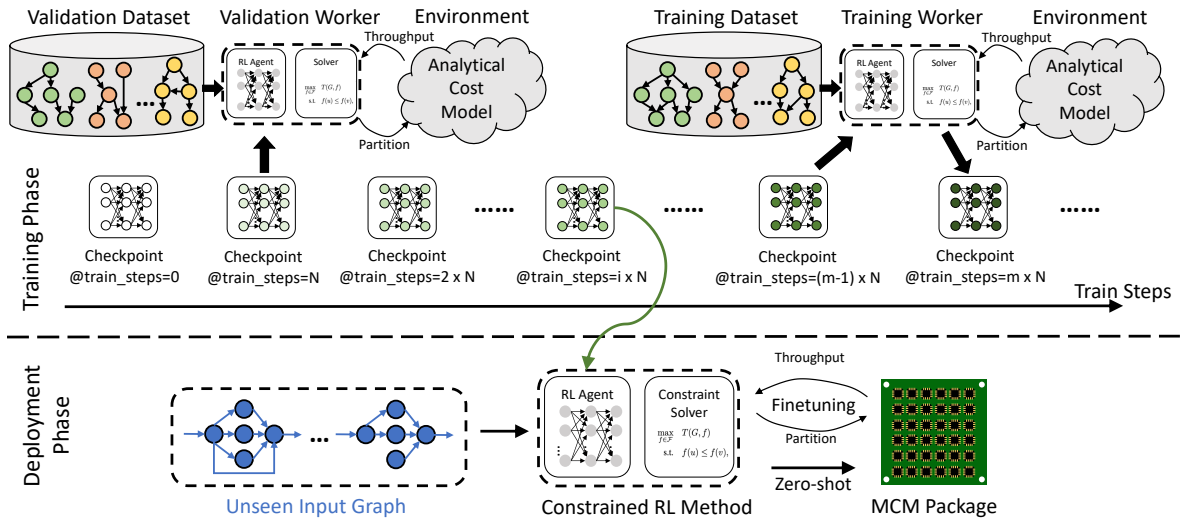


Figure 7.5: The workflow of our pre-training method including a training phase to obtain pre-trained model checkpoints, and a deployment phase using the optimal checkpoint for zero-shot and fine-tuning on an unseen input graph.

As shown in Figure 7.4, RL model outputs both $\mathbf{y}$ and $\mathbf{P}$ so that the constraint solver generates a valid partition $\mathbf{y}'$ according to the input node order and assignment strategy. Because $\mathbf{y}'$ satisfies static constraints, its reward space is denser than that of $\mathbf{y}$, and we use this denser reward space to efficiently train our end-to-end RL models.

7.4.3 Pretraining Pipeline

In addition to the constrained deep RL method introduced in Section 7.4.1, we develop a pre-training based method to generalize the constrained deep RL solution such that it can learn from the training dataset while transferring the learned knowledge to unseen data. As shown in Figure 7.5, the whole pre-training pipeline is composed of two phases: the training phase and the deployment phase.

Training Phase: There are two workers in the training phase: a training worker generating the checkpoints of RL model weights, and a validation worker to evaluate the performance of each model snapshot. In particular, the training worker iterates through

the input graphs from the training set, and periodically generates checkpoints of the RL model weights. Meanwhile, the validation worker conducts a continuous evaluation on graphs from the validation set. The validation worker is warm-started from a pre-trained model checkpoint. During the evaluation, the validation worker conducts a zero-shot prediction and a fine-tuning upon the pre-trained checkpoint, for each graph from the validation set. After iterating through all model checkpoints, the validation worker can pick the checkpoint with either the best zero-shot or fine-tuning performance for deployment.

Deployment Phase: In the deployment phase, we load the optimal checkpoint picked by the validation worker to warm start the RL model. Then the RL model takes a new (previously unseen) graph as the input and directly runs inference (zero-shot). Alternatively, we can fine-tune the RL model for this new graph to capture out-of-distribution data, as detailed in Figure 7.4.

Experimental results in Section 7.5.2 show that our pre-training pipeline is able to generalize the pre-trained solution to a test set of previously unseen graphs.

7.5 Experiments

Our goal is to find a valid and efficient ML model partition to chips while minimizing the search time. First, we introduce the experiment setup in Section 7.5.1. Then, we present the results of our pre-training experiments in Section 7.5.2 to demonstrate the generalization of our RL approach on the test dataset of 16 ML graphs. Finally, we evaluate the BERT model in Section 7.5.3 to show the effectiveness of our RL approach on real hardware.

7.5.1 Experiment Setup

Evaluation Platform: We conduct our experiments on a real hardware platform of multi-chip TPU [214] that includes 36 chiplets in a package. To efficiently pre-train our RL model, reducing pre-training time, and saving hardware resources, our pre-training experiments are based on an analytical cost model. This analytical cost model estimates the latency of running all nodes assigned to each chip, and returns the maximal latency of all chips.

Workloads: We conduct the real hardware evaluation on a production-scale model, BERT [244], that has 2138 nodes and around 340 million (600 MB) parameters. To pre-train our RL model, we use a dataset of 87 ML models from real-world applications, and most of them are from computer vision and natural language processing applications including convolutional neural network (CNN) and recurrent neural network (RNN) models. The computation graphs of these ML models have tens to hundreds of nodes. None of these ML graphs contain a Transformer-like attention mechanism. In our pre-training experiment, we randomly partition these 87 computation graphs into three datasets: a training dataset of 66 graphs, a validation set of 5 graphs, and a test dataset of 16 graphs.

Performance Metric: We evaluate each partition solution and obtain its throughput because a multi-chip TPU focuses more on throughput rather than latency. However, our framework can easily re-target a latency metric. Then we report the throughput improvement over compiler heuristics, such as a greedy algorithm and a random partition, that are usually fast with a time complexity of $O(N)$. We run all experiments on real hardware 5 times and report both mean and standard deviation of throughput improvements.

RL with Constraint Solver: Our default configuration of graph neural network uses 8 GraphSAGE layers, and the size of each layer is 128. For the policy network,

we use a feed-forward network with 2 layers, and the size of each layer is the same as GraphSAGE layers. We use Proximal Policy Optimization (PPO) [245] to train our RL models. We explore various values for hyper-parameters during the training process, such as the number of rollouts, the number of mini-batches, and the number of epochs. We select the optimal hyper-parameter (20 rollouts, 4 mini-batches, and 10 epochs) across all explored settings to report RL results. For this example, we use the FIX mode in the constraint solver, as it outperforms SAMPLE mode.

RL without Constraint Solver: Instead of passing the policy output from RL to the constraint solver, this baseline directly generates partition solutions by sampling based on the output probability distribution $\mathbf{P}$. Our evaluation platform returns a zero throughput when it evaluates an invalid partition. Without the help of the constraint solver, the reward space is extremely sparse as most of the partitions are invalid. As a result, this baseline is not able to find any valid partition in the training process even when the number of samples is sufficiently large.

Traditional Search Strategies: We implement several traditional search strategies working with the constraint solver as comparison baselines. These traditional search strategies include *Random Search (Random)* and *Simulated Annealing (SA)*. We tune these search strategies with the constraint solver empirically to pick an implementation with the best performance. The details of each search algorithm are as follows:

Random Search Strategy (Random) provides a fixed uniform probability distribution, $p(y_i = j|G) = \frac{1}{C}$ for $\forall j \in D$, and the constraint solver works under the SAMPLE mode to generate valid partitions.

Simulated Annealing (SA) starts from the same uniform probability distribution as Random. Each iteration, SA randomly selects a set of nodes, and for each node i from this set, it generates a new random distribution $\mathbf{p}_i$. The constraint solver takes this new probability distribution to generate a valid partition. Based on the evaluation results of

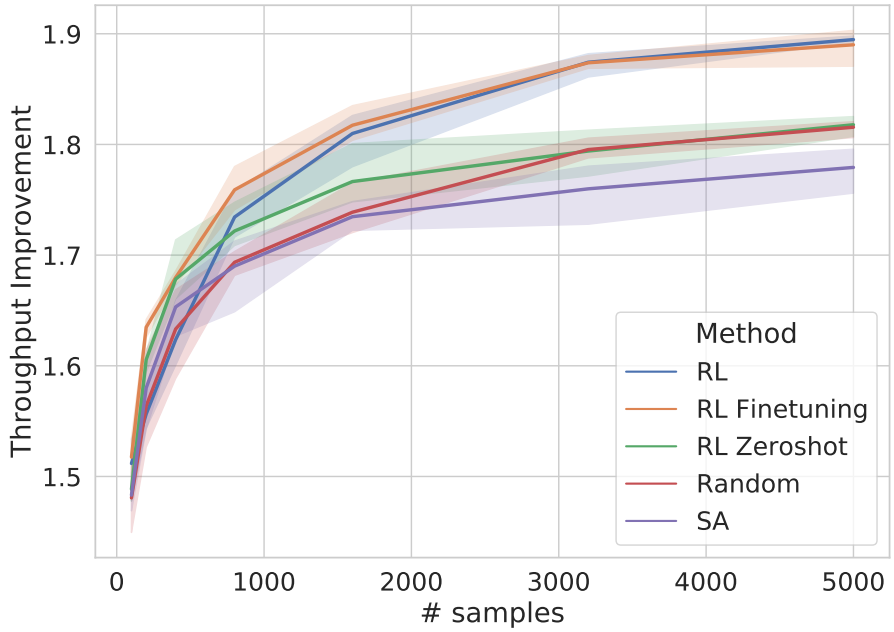


Figure 7.6: The geomean throughput improvement of 16 graphs from test dataset on the analytical model comparison for a random search strategy (Random), simulated annealing (SA), reinforcement learning training from scratch (RL), the zeroshot predictions of the pretrained model (RL Zeroshot), and the finetuning of the pretrained model (RL Finetuning).

Throughput Improvement	$\geq 1.60\times$	$\geq 1.70\times$	$\geq 1.80\times$
Random	305 (1.08×)	915 (0.74×)	3612 (0.41×)
SA	255 (1.29×)	979 (0.69×)	N.A. (N.A.)
RL	330 (1.00×)	676 (1.00×)	1496 (1.00×)
RL Zeroshot	196 (1.68×)	600 (1.13×)	3652 (0.41×)
RL Finetuning	**171 (1.93×)**	**503 (1.34×)**	**1362 (1.10×)**

Table 7.2: The number of samples and the reduction of samples to achieve certain geomean throughput improvement levels. The results 171 (1.93×) indicate that RL Finetuning needs 171 samples and reduces 1.93× samples compared with RL training from scratch to achieve a 1.60× geomean throughput improvement over a compiler heuristic.

this valid partition, SA decides whether to accept the new probability distribution.

7.5.2 Pretraining Experiment

We pre-train the RL model using evaluations from an analytical model. The evaluation with an analytical model is orders of magnitude faster than evaluating the samples on real chips. In particular, we pre-train the RL model on the training dataset (66 graphs) with a total of 20,000 samples. The pre-training process generates 200 checkpoints. Then, we use the validation dataset (5 graphs) to pick the optimal checkpoint with the best average rewards. The pre-training process takes a few hours using the analytical cost model rather than several days compared to using real-chip evaluations. Finally, we evaluate 16 graphs from the test dataset on the analytical model to demonstrate the generalization.

Figure 7.6 presents the geomean throughput improvement over a compiler heuristic of 16 test graphs among RL, RL Finetuning, RL Zeroshot, Random, and SA. An RL-based approach can achieve a better performance by 4.36% and 6.49% compared to Random and SA. Table 7.2 shows the number of samples needed to achieve different throughput gains. It shows that fine-tuning on a pre-trained RL model can reduce the number of samples by up to 1.93x compared to RL training from scratch. Both Figure 7.6 and Table 7.2 show that zero-shot RL (without fine-tuning) can achieve higher throughput in early hundred samples, thus 1.68x fewer samples than RL training from scratch to achieve a geomean of 1.60x speedup, but RL zero-shot does not perform well with more samples. This could result from the different data distributions between the training dataset and the test dataset. It justifies a further fine-tuning on a pre-trained model for out-of-distribution data.

We further evaluate BERT on real chips in Section 7.5.3 to demonstrate a real-world use case.

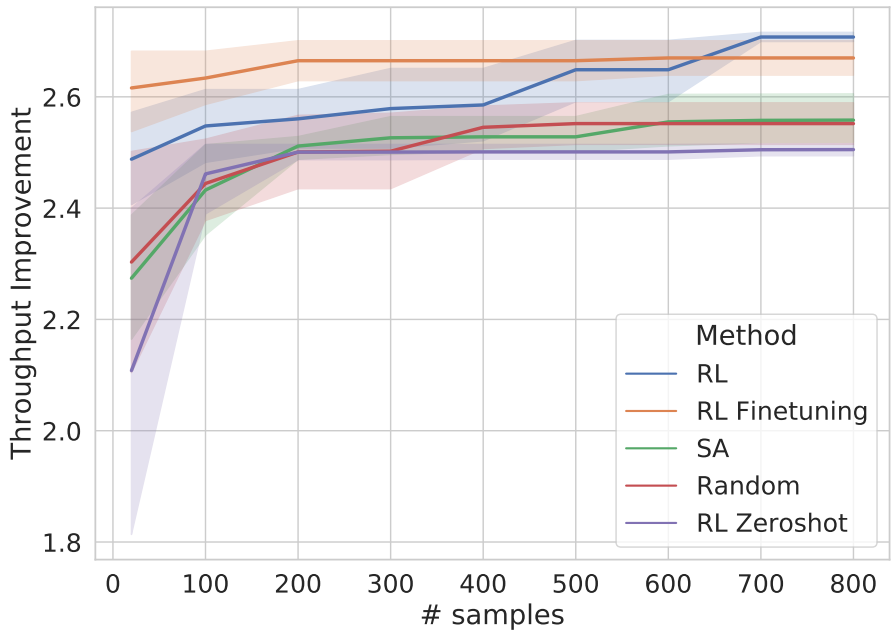


Figure 7.7: The throughput improvement of BERT on real hardware over a greedy heuristic comparing the random search strategy (Random), simulated annealing (SA), reinforcement learning training from scratch (RL), the zeroshot of the pretrained model (RL Zeroshot), and the finetuning of the pretrained model (RL Finetuning).

Throughput Improvement	$\geq 2.55\times$	$\geq 2.60\times$	$\geq 2.65\times$
Random	447 (0.25×)	N.A. (N.A.)	N.A. (N.A.)
SA	576 (0.19×)	N.A. (N.A.)	N.A. (N.A.)
RL	112 (1.00×)	423 (1.00×)	607 (1.00×)
RL Zeroshot	N.A. (N.A.)	N.A. (N.A.)	N.A. (N.A.)
RL Finetuning	**20 (5.60×)**	**20 (21.15×)**	**161 (3.77×)**

Table 7.3: The number of samples and the reduction of samples to achieve certain throughput improvement levels. The results 20 (5.60×) indicate that RL Finetuning needs 20 samples and reduces 5.60× samples compared with RL training from scratch to achieve a 2.55× throughput improvement over a greedy heuristic.

7.5.3 BERT Evaluation

We evaluate a production-scale model, BERT [244], on a real MCM system with 36 chips to demonstrate real system performance. We use a greedy heuristic from the production compiler as the baseline of throughput improvement. Section 7.5.1 details the implementation of our RL method and other traditional search strategies. To generate

sufficient pre-training samples, we adopt the analytical cost model in the pre-training phase, as described in Section 7.5.2.

Figure 7.7 shows that our RL-based approach can achieve 6.11% and 5.85% better throughput than Random and SA respectively, at convergence. Moreover, Figure 7.7 shows that fine-tuning on a pre-trained RL model improves the placement throughput at low sample complexity, compared to RL training from scratch. Unfortunately, RL zero-shot does not work well due to two possible reasons: 1) BERT model is much bigger than the graphs in the training dataset and has a drastically different model architecture compared to models in the training dataset. 2) The difference between the analytical cost model and real-chip evaluations exacerbates the RL environment gap between the pre-training stage and the deployment (test) stage.

Table 7.3 shows that fine-tuning on a pre-trained RL model reduces the number of samples by up to 21.15x for achieving the same throughput gain compared to RL training from scratch. Since the elapsed time of getting a sample takes 26.97 seconds on average, reducing the number of samples from 423 to 20 means a reduction of searching time from more than 3 hours to around 9 minutes. Finally, for a search budget of 10 minutes, the fine-tuning of our RL-based approach can outperform Random and SA by up to 15.18%.

The results demonstrate strong generalization from an analytical cost model based pre-training on training dataset consisting of small production models, to a real-chip environment on large production models such as BERT. The pre-training and fine-tuning method enables the deployment of an RL-based method into production ML compilers.

7.5.4 Analytical Cost Model Accuracy

To justify the use of an analytical cost model during pretraining, we conduct a calibration study on the analytical model. We randomly generate 2000 samples on BERT. Then

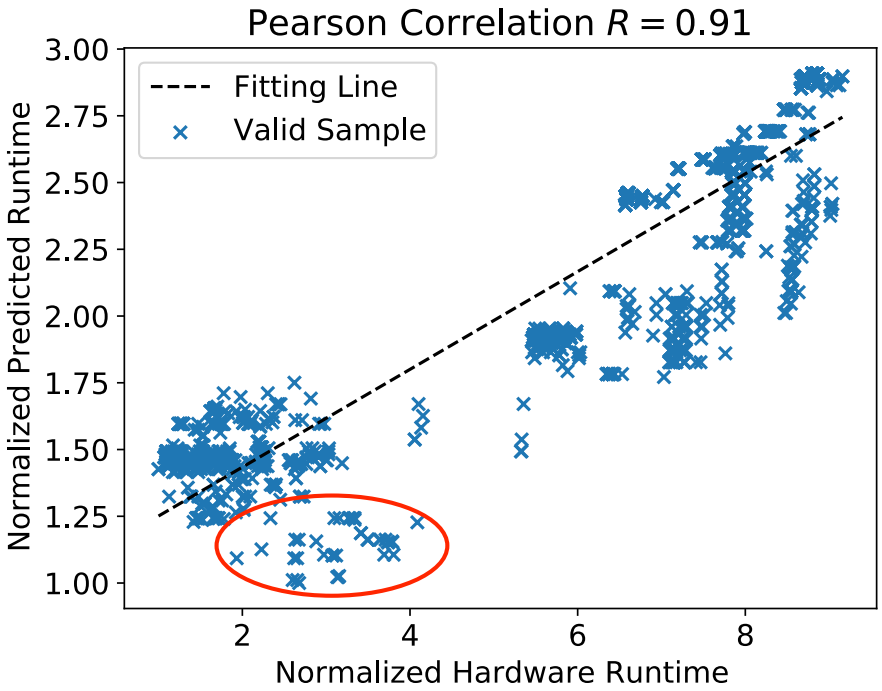


Figure 7.8: The hardware runtime and predicted runtime of valid partition samples for BERT model normalized to the minimal values of all samples respectively.

we evaluate these 2000 samples on real hardware. Finally, we normalize the predicted runtime and measured runtime to their minimal respectively. The runtime is defined as the maximum latency across all chips, which is the reciprocal of the throughput. Figure 7.8 shows the normalized predicted runtime vs. the normalized measured runtime for all valid samples from these 2000 samples. We have three main observations from this calibration study: 1) 13.5% of generated partitions are invalid on real hardware. 2) Some partitions showing lower runtime do not work well on real hardware, such as samples from the red cycle in Figure 7.8. 3) There is a strong correlation (Pearson correlation $R = 0.91$) between predicted runtime and measured runtime.

Observing the critical features of the analytical cost model, we conclude that RL zero-shot does not work well for BERT on real chips because dynamic constraints cannot be captured by the constraint solver (13.5% failures) and there are some false-positive results. However, RL fine-tuning can start from a pre-trained model efficiently due to the

strong correlation between the results of the analytical cost model and that of the real hardware evaluation. The knowledge created during the pre-training with the majority of accurate samples can be still successfully transferred. For example, the knowledge of balanced placement and chip static constraints can be transferred.

7.6 Conclusion

In this work, we develop a deep RL solution working with a constraint solver for ML model partitioning targeting an MCM package. Our method is generalizable to unseen input graphs via a pre-training pipeline. Our evaluation of a production-scale model, BERT, on real hardware evaluation shows that our approach can outperform random search and simulated annealing by 6.11% and 5.85% at convergence. Finally, our RL-based approach is transferable. The fine-tuning on a pre-trained model improves the sample efficiency up to 21.15x than RL training from scratch on BERT placement. This effectively reduces the search time from more than 3 hours to 9 minutes to achieve a throughput improvement of 2.6x, compared to a greedy heuristic in the production compiler.

Chapter 8

Summary

The computation capability increase of contemporary computing platforms, especially with the development of domain-specific architectures, significantly outpaces the growth of main memory bandwidth. This exaggerates the memory bandwidth bottleneck of many data-intensive parallel workloads that play important roles in various application domains. Although processing-in-memory architectures, especially in-DRAM near-bank processing, are promising to deliver a higher effective bandwidth by putting compute logics into DRAM, they face hardware and software challenges to reduce overheads and improve performance.

To address these challenges and facilitate efficient in-DRAM near-bank processing, this dissertation includes six projects that characterize workloads to identify memory bandwidth bottleneck (Chapter 2), develop near-bank processing architectures (Chapter 3, Chapter 4, and Chapter 5), build simulation infrastructures for studying near-bank processing (Chapter 6), and optimize workload partitioning on a distributed memory space by constrained reinforcement learning (Chapter 7). The contributions of this dissertations are summarized as follows:

First, this dissertation starts with a novel benchmarking methodology in Chapter 2.

Neural network (NN) applications are representative workloads that demand high performance and energy efficiency on modern computing platforms. In this project, we develops a holistic benchmarking method from workload characterization to hardware platform evaluations. Our case studies reveal that there are still a number of NN models bounded by memory bandwidth although some commonly used tensor operators are computation intensive, such as matrix multiplication and convolution. Moreover, our evaluations on hardware platforms find that despite the success of neural network accelerators and model compression techniques, it is hard to address memory bandwidth bottleneck unless applying near data processing architectures to provide higher effective bandwidth. These characterization studies and hardware evaluations emphasize the importance of overcoming the memory bandwidth wall for data-intensive workloads.

Second, this dissertation develops an application-specific near-bank processing accelerator in Chapter 3. In particular, we design an accelerator, named SpaceA, to leverage outstanding memory requests to hide the memory access latency to non-local banks. To reduce the memory traffic to non-local banks, we integrate content addressable memory (CAM) buffers in SpaceA to exploit the locality of input vectors. Furthermore, we develop a mapping scheme for SpaceA to distribute the non-zero elements across different banks to achieve workload balance among processing elements (PEs) and to exploit the data locality of the input vector. Our evaluation of SpaceA with the proposed mapping scheme on matrices from real-world applications reveals 13.5x speedup and 87.49% energy saving on average over the GPU baseline with only 4.86% area overhead.

Third, this dissertation develops a domain-specific near-bank processing accelerator in Chapter 4. In particular, 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 ($\sim 10.71\%$). Furthermore, we develop an end-to-end com-

pilation 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.

Fourth, this dissertation develops a general-purpose near-bank processing architecture in Chapter 5. In particular, we design the first general-purpose 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 also 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. Additionally, 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.

Fifth, this dissertation develops a near-bank computing simulator in Chapter 6 that provides an important tool for the hardware and software studies of near-bank processing. In particular, we develop an open-source simulator, MPU-Sim, for general-purpose near-bank processing architectures. It runs CUDA programs and enables software optimization studies in addition to hardware design explorations. Furthermore, we conduct calibration studies for key components (processor, DRAM, and NoC) in MPU-Sim with state-of-the-art simulators to validate our simulator implementations. Additionally, we conduct case

studies for hardware and software optimization opportunities in near-bank processing architectures with MPU-Sim to demonstrate its potential usage.

Finally, this dissertation studies the workload partition problem on a distributed memory space using a constrained reinforcement learning method in Chapter 7. Because the memory space abstraction of multi-chip-modules (MCMs) is similar to that of near-bank processing architectures and we can conduct real hardware evaluation, we select the ML workload partition on an MCM-based ML accelerator as our target problem in this project. In particular, we define the problem of multi-chip partitioning for MCMs and propose a method that combines the capabilities of deep RL networks and constraint solvers to search for good partitionings in a search space where valid solutions are extremely sparse due to constraints imposed by the hardware architecture. Our evaluation for BERT, a production-scale model, on real hardware demonstrates that our RL-based partitioner achieves 6.11% and 5.85% higher throughput than random search and simulated annealing upon its convergence. Moreover, we demonstrate strong transfer learning performance via a pre-training based method. We pre-trained the RL policy on 66 production neural networks from computer vision applications and language models, using an analytical performance model as a reward function. Fine-tuning the pre-trained policy on BERT reduces the search time from more than 3 hours for RL training from scratch to only 9 minutes, while achieving the same runtime performance.

Hopefully, the contributions of this dissertation in benchmarking methodology, architecture designs, software optimizations, and simulation infrastructures pave the way for the future academic research and industrial development of in-DRAM near-bank processing hardware platforms and software systems.

Bibliography

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, H. Dan, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[2] 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.

[3] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, *Top-pim: throughput-oriented programmable processing in memory*, in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 85–98, ACM, 2014.

[4] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, *Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems*, in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 204–216, IEEE Press, 2016.

[5] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, *Scheduling techniques for gpu architectures with processing-in-memory capabilities*, in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pp. 31–44, 2016.

[6] 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.

[7] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, *Toward standardized near-data processing with unrestricted data placement for gpus*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[8] W. Wen, J. Yang, and Y. Zhang, *Optimizing power efficiency for 3d stacked gpu-in-memory architecture*, *Microprocessors and Microsystems* **49** (2017) 44–53.

[9] L. Nai, R. Hadidi, H. Xiao, H. Kim, J. Sim, and H. Kim, *Coolpim: Thermal-aware source throttling for efficient pim instruction offloading*, in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 680–689, IEEE, 2018.

[10] Y. Wu, M. Shen, Y.-H. Chen, and Y. Zhou, *Tuning applications for efficient gpu offloading to in-memory processing*, in *Proceedings of the 34th ACM International Conference on Supercomputing*, pp. 1–12, 2020.

[11] 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.

[12] 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.

[13] 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.

[14] UPMem, *UPMem*, 2020. `https://www.upmem.com/`.

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

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

[17] 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.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, *arXiv preprint arXiv:1704.04861* (2017).

[20] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[21] A. M. Rush, S. Chopra, and J. Weston, *A neural attention model for abstractive sentence summarization*, *arXiv preprint arXiv:1509.00685* (2015).

[22] 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.

[23] Google, "TensorFlow Models." `https://github.com/tensorflow/models`, 2018.

[24] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 269–284, ACM, 2014.

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

[26] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, *Cambricon-X: An accelerator for sparse neural networks*, in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[27] Nvidia, "cuDNN." `https://developer.nvidia.com/cudnn`, 2017.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: A system for large-scale machine learning*, in *12th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), (Savannah, GA), pp. 265–283, USENIX Association, Nov., 2016.

[29] P. Core team, "PyTorch." `http://pytorch.org/`, 2017.

[30] J.-H. Tao, Z.-D. Du, Q. Guo, H.-Y. Lan, L. Zhang, S.-Y. Zhou, C. Liu, H.-F. Liu, S. Tang, A. Rush, W. Chen, S.-L. Liu, Y.-J. Chen, and T.-S. Chen, *BENCHIP: Benchmarking Intelligence Processors*, *arXiv preprint arXiv:1710.08315* (2017).

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

[32] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, *Benchnn: On the broad potential application scope of hardware neural network accelerators*, in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 36–45, IEEE, 2012.

[33] S. Shi, Q. Wang, P. Xu, and X. Chu, *Benchmarking state-of-the-art deep learning software tools*, in *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pp. 99–104, IEEE, 2016.

[34] Baidu, "DeepBench." `https://github.com/baidu-research/DeepBench`, 2018.

[35] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, *Scalpel: Customizing dnn pruning to the underlying hardware parallelism*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, ACM, 2017.

[36] 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.

[37] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, *Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks*, *IEEE Journal of Solid-State Circuits* **52** (2017), no. 1 127–138.

[38] 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.

[39] H. Kung, B. McDanel, and S. Q. Zhang, *Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 821–834, 2019.

[40] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, *Eridanus: Efficiently running inference of dnns using systolic arrays*, *IEEE Micro* **39** (2019), no. 5 46–54.

[41] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, *Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 754–768, 2019.

[42] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, *Bridging the gap between value and policy based reinforcement learning*, in *Advances in Neural Information Processing Systems*, pp. 2772–2782, 2017.

[43] N. Johnston, D. Vincent, D. Minnen, M. Covell, S. Singh, T. Chinen, S. J. Hwang, J. Shor, and G. Toderici, *Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks*, *arXiv preprint arXiv:1703.10114* (2017).

[44] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell, *Full resolution image compression with recurrent neural networks*, *arXiv preprint* (2016).

[45] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[46] S. Ren, K. He, R. Girshick, and J. Sun, *Faster r-cnn: Towards real-time object detection with region proposal networks*, in *Advances in neural information processing systems*, pp. 91–99, 2015.

[47] C. Finn, I. Goodfellow, and S. Levine, *Unsupervised learning for physical interaction through video prediction*, in *Advances in neural information processing systems*, pp. 64–72, 2016.

[48] P. Chi, S. Li, 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 the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.

[49] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie, *Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 733–747, 2019.

[50] 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.

[51] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, *Mlperf inference benchmark*, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, 2020.

[52] C. Ding and Y. Zhong, *Predicting whole-program locality through reuse distance analysis*, in *Acm Sigplan Notices*, vol. 38, pp. 245–257, ACM, 2003.

[53] R. M. Neal, *Bayesian learning for neural networks*, vol. 118. Springer Science & Business Media, 2012.

[54] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, *Weight uncertainty in neural networks*, *arXiv preprint arXiv:1505.05424* (2015).

[55] S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, *Acmc 2: Accelerating markov chain monte carlo algorithms for probabilistic models*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 515–528, 2019.

[56] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, *ACM Transactions on Mathematical Software (TOMS)* **38** (2011), no. 1 1.

[57] 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.

[58] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, *Graphp: Reducing communication for pim-based graph processing with efficient data partition*, in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pp. 544–557, IEEE, 2018.

[59] "NVIDIA cuSPARSE library." `https://docs.nvidia.com/cuda/cusparse/in-\dex.html`, 2018.

[60] S. Beamer, K. Asanovic, and D. Patterson, *Locality exists in graph processing: Workload characterization on an ivy bridge server*, in *2015 IEEE International Symposium on Workload Characterization*, pp. 56–65, IEEE, 2015.

[61] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, *Analysis and optimization of the memory hierarchy for graph processing workloads*, in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 373–386, Feb, 2019.

[62] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, *Saga-bench: Software and hardware characterization of streaming graph analytics workloads*, in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 12–23, IEEE, 2020.

[63] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, *Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 26, ACM, 2013.

[64] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh, *Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi*, in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 136–145, IEEE Computer Society, 2015.

[65] "HMC Specification 2.1." `http://hybridmemorycube.org/`, 2014.

[66] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, *Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube*, in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 66–75, IEEE, 2017.

[67] 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.

[68] S. Galal, O. Shacham, J. S. Brunhaver II, J. Pu, A. Vassiliev, and M. Horowitz, *Fpu generator for design space exploration*, in *2013 IEEE 21st Symposium on Computer Arithmetic*, pp. 25–34, IEEE, 2013.

[69] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, *Tetris: Scalable and efficient neural network acceleration with 3d memory*, in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 751–764, 2017.

[70] M. J. Khurshid and M. Lipasti, *Data compression for thermal mitigation in the hybrid memory cube*, in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 185–192, IEEE, 2013.

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

[72] 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.

[73] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June, 2014.

[74] S. Beamer, K. Asanović, and D. Patterson, *The gap benchmark suite*, *arXiv preprint arXiv:1508.03619* (2015).

[75] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[76] A. Pinar and M. T. Heath, *Improving performance of sparse matrix-vector multiplication*, in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, p. 30, ACM, 1999.

[77] E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[78] J. Mellor-Crummey and J. Garvin, *Optimizing sparse matrix–vector product computations using unroll and jam*, *The International Journal of High Performance Computing Applications* **18** (2004), no. 2 225–236.

[79] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, in *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pp. 1–12, IEEE, 2007.

[80] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in *Proceedings of the conference on high performance computing networking, storage and analysis*, p. 18, ACM, 2009.

[81] J. L. Greathouse and M. Daga, *Efficient sparse matrix-vector multiplication on gpus using the csr storage format*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, IEEE Press, 2014.

[82] D. Merrill and M. Garland, *Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format*, in *ACM SIGPLAN Notices*, vol. 51, p. 43, ACM, 2016.

[83] Y. Nagasaka, A. Nukada, and S. Matsuoka, *Adaptive multi-level blocking optimization for sparse matrix vector multiplication on gpu*, *Procedia Computer Science* **80** (2016) 131–142.

[84] B.-Y. Su and K. Keutzer, *clspmv: A cross-platform opencl spmv framework on gpus*, in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 353–364, ACM, 2012.

[85] X. Xie, D. Du, Q. Li, Y. Liang, W. T. Tang, Z. L. Ong, M. Lu, H. P. Huynh, and R. S. M. Goh, *Exploiting sparsity to accelerate fully connected layers of cnn-based applications on mobile socs*, *ACM Transactions on Embedded Computing Systems (TECS)* **17** (2017), no. 2 1–25.

[86] S. Yan, C. Li, Y. Zhang, and H. Zhou, *yaspmv: yet another spmv framework on gpus*, in *Acm Sigplan Notices*, vol. 49, pp. 107–118, ACM, 2014.

[87] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient sparse matrix-vector multiplication on x86-based many-core processors*, in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 273–282, ACM, 2013.

[88] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, *Cvr: efficient vectorization of spmv on x86 processors*, in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 149–162, ACM, 2018.

[89] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, *Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories*, in *Proceedings of the 53rd Annual Design Automation Conference*, p. 173, ACM, 2016.

[90] D. Fujiki, S. Mahlke, and R. Das, *In-memory data parallel processor*, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1–14, ACM, 2018.

[91] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, *Drisa: A dram-based reconfigurable in-situ accelerator*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, ACM, 2017.

[92] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, *Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture*, in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 336–348, IEEE, 2015.

[93] K. Wu, G. Dai, X. Hu, S. Li, X. Xie, Y. Wang, and Y. Xie, *Memory-bound proof-of-work acceleration for blockchain applications*, in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[94] 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 *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 656–669, ACM, 2018.

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

[96] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, *Time: A training-in-memory architecture for memristor-based deep neural networks*, in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 26, ACM, 2017.

[97] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, *Neurostream: Scalable and energy efficient deep learning with smart memory cubes*, *IEEE Transactions on Parallel & Distributed Systems* (2018), no. 1 1–1.

[98] 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).

[99] 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).

[100] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[101] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, *Snrram: an efficient sparse neural network computation architecture based on resistive random-access memory*, in *Proceedings of the 55th Annual Design Automation Conference*, p. 106, ACM, 2018.

[102] S. Angizi, Z. He, A. S. Rakin, and D. Fan, *Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator*, in *Proceedings of the 55th Annual Design Automation Conference*, p. 105, ACM, 2018.

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

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

[105] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, *Graphh: A processing-in-memory architecture for large-scale graph processing*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[106] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, *Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware*, in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pp. 1–6, IEEE, 2013.

[107] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, *Enabling scientific computing on memristive accelerators*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–382, June, 2018.

[108] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, *Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 347–358, 2019.

[109] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, *Extensor: An accelerator for sparse tensor algebra*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 319–333, 2019.

[110] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, *Alrescha: A lightweight reconfigurable sparse-computation accelerator*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 249–260, IEEE, 2020.

[111] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, *arXiv preprint arXiv:1510.00149* (2015).

[112] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, *Hitnet: Hybrid ternary recurrent neural network*, in *Advances in Neural Information Processing Systems*, pp. 604–614, 2018.

[113] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, *Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, IEEE, 2020.

[114] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, *Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 359–371, 2019.

[115] L. Liu, Z. Qu, L. Deng, F. Tu, S. Li, X. Hu, Z. Gu, Y. Ding, and Y. Xie, *Duet: Boosting deep neural network efficiency on dual-module architecture*, in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 738–750, IEEE, 2020.

[116] Z. Zhang, H. Wang, S. Han, and W. J. Dally, *Sparch: Efficient architecture for sparse matrix multiplication*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, IEEE, 2020.

[117] "JEDEC Standard. High Bandwidth Memory (HBM) DRAM. JESD25A." `https://www.jedec.org/standards-documents/docs/jesd235a`, 2015.

[118] X. L. Li, B. Veeravalli, and C. Ko, *Distributed image processing on a network of workstations*, *International Journal of Computers and Applications* **25** (2003), no. 2 136–145.

[119] Y. Yan and L. Huang, *Large-scale image processing research cloud*, *Cloud Computing* (2014) 88–93.

[120] S. T. Bow, *Pattern recognition and image preprocessing*. CRC press, 2002.

[121] T. M. Deserno, *Biomedical image processing*, .

[122] J. R. Jensen, *Introductory digital image processing: a remote sensing perspective*. Prentice Hall Press, 2015.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] Y. Chi, J. Cong, P. Wei, and P. Zhou, *Soda: stencil with optimized dataflow architecture*, in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[130] 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.

[131] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, *Programming heterogeneous systems from an image processing dsl*, *ACM Transactions on Architecture and Code Optimization (TACO)* **14** (2017), no. 3 26.

[132] 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.

[133] R. T. Mullapudi, V. Vasista, and U. Bondhugula, *Polymage: Automatic optimization for image processing pipelines*, in *ACM SIGPLAN Notices*, vol. 50, pp. 429–443, ACM, 2015.

[134] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, *Automatically scheduling halide image processing pipelines*, *ACM Transactions on Graphics (TOG)* **35** (2016), no. 4 83.

[135] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, *A common backend for hardware acceleration on fpga*, in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 427–430, IEEE, 2017.

[136] 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.

[137] 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.

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

[139] 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.

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

[141] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche, *It's time to think about an operating system for near data processing architectures*, in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pp. 56–61, ACM, 2017.

[142] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, *et. al.*, *Conda: Efficient cache coherence support for near-data accelerators*, .

[143] C. McGinnis, *Pci-sig® fast tracks evolution to 32gt/s with pci express 5.0 architecture*, *News Release, June* **7** (2017).

[144] K.-y. Chae, *Advanced microcontroller bus architecture (amba) system with reduced power consumption and method of driving amba system*, June 19, 2007. US Patent 7,234,011.

[145] 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.

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

[147] 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.

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

[149] 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.

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

[151] 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.

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

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

[154] 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.

[155] 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.

[156] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei, *Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes*, in *International Conference on Field Programmable Logic and Applications, 2005.*, pp. 106–111, IEEE, 2005.

[157] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, *Evaluating programmable architectures for imaging and vision*

applications, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[158] M. Mahmoud, B. Zheng, A. D. Lascorz, F. H. Assouline, J. Assouline, P. Boucher, E. Onzon, and A. Moshovos, *Ideal: Image denoising accelerator*, in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 82–95, IEEE, 2017.

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

[160] 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.

[161] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, *Flexram: Toward an advanced intelligent memory system*, in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, pp. 192–201, IEEE, 1999.

[162] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, *et. al.*, *Google workloads for consumer devices: Mitigating data movement bottlenecks*, in *ACM SIGPLAN Notices*, vol. 53, pp. 316–331, ACM, 2018.

[163] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, *Graphq: Scalable pim-based graph processing*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 712–725, ACM, 2019.

[164] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, *Charon: Specialized near-memory processing architecture for clearing dead objects in memory*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 726–739, ACM, 2019.

[165] J. Li, X. Wang, A. Tumeo, B. Williams, J. D. Leidel, and Y. Chen, *Pims: a lightweight processing-in-memory accelerator for stencil computations*, in *Proceedings of the International Symposium on Memory Systems*, pp. 41–52, ACM, 2019.

[166] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song, *Pim-vr: Erasing motion anomalies in highly-interactive virtual reality world with customized memory cube*, in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 609–622, IEEE, 2019.

[167] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, *et. al.*, *Application-transparent near-memory processing architecture with memory channel network*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 802–814, IEEE, 2018.

[168] M. Alian and N. S. Kim, *Netdimm: Low-latency near-memory network interface architecture*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 699–711, 2019.

[169] 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.

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

[171] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, *et. al.*, *Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 715–731, ACM, 2019.

[172] 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.

[173] 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.

[174] 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.

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

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

[177] 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.

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

[179] J. Macri, *Amd's next generation gpu and high bandwidth memory architecture: Fury*, in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–26, IEEE, 2015.

[180] M. O'Connor, *Highlights of the high-bandwidth memory (hbm) standard*, in *Memory Forum Workshop*, 2014.

[181] 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.

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

[183] 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.

[184] 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.

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

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

[187] 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.

[188] Nvidia, "cuBLAS." `http://docs.nvidia.com/cuda/cublas/index.html`, 2017.

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

[190] 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.

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

[192] 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.

[193] 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.

[194] 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.

[195] 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.

[196] 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.

[197] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, *Livia: Data-centric computing throughout the memory hierarchy*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 417–433, 2020.

[198] M. H. K. E. E. Onur, *Reducing memory access latency via an enhanced (compute capable) memory controller milad hashemi khubaib eiman ebrahimi onur mutlu yale n. patt*, .

[199] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, *Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[200] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, *Lazypim: An efficient cache coherence mechanism for processing-in-memory*, *IEEE Computer Architecture Letters* **16** (2016), no. 1 46–50.

[201] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, *Chonda: Near data acceleration with concurrent host access*, *International Symposium on Computer Architecture* (2020).

[202] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, *Processing-in-memory enabled graphics processors for 3d rendering*, in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 637–648, IEEE, 2017.

[203] B. Akin and A. R. Alameldeen, *A case for asymmetric processing in memory*, *IEEE Computer Architecture Letters* **18** (2019), no. 1 22–25.

[204] J. Picorel, D. Jevdjic, and B. Falsafi, *Near-memory address translation*, in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 303–317, Ieee, 2017.

[205] B. Akin, F. Franchetti, and J. C. Hoe, *Hamlet architecture for parallel data reorganization in memory*, *IEEE Micro* **36** (2015), no. 1 14–23.

[206] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, *The missing memristor found*, *nature* **453** (2008), no. 7191 80–83.

[207] S. Mittal, *A survey of reram-based architectures for processing-in-memory and neural networks*, *Machine learning and knowledge extraction* **1** (2019), no. 1 75–114.

[208] M. Imani, S. Gupta, Y. Kim, and T. Rosing, *Floatpim: In-memory acceleration of deep neural network training with high precision*, in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, IEEE, 2019.

[209] L. Xia, T. Tang, W. Huangfu, M. Cheng, X. Yin, B. Li, Y. Wang, and H. Yang, *Switched by input: Power efficient structure for rram-based convolutional neural network*, in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

[210] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, *Pimsim: A flexible and detailed processing-in-memory simulator*, *IEEE Computer Architecture Letters* **18** (2018), no. 1 6–9.

[211] C. Yu, S. Liu, and S. Khan, *Multipim: A detailed and configurable multi-stack processing-in-memory simulator*, *IEEE Computer Architecture Letters* **20** (2021), no. 1 54–57.

[212] X. Xie, P. Gu, Y. Ding, D. Niu, H. Zheng, and Y. Xie, *MPU: towards bandwidth-abundant SIMT processor via near-bank computing*, *CoRR* **abs/2103.06653** (2021) [arXiv:2103.0665].

[213] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et. al.*, *Simba: Scaling deep-learning inference with multi-chip-module-based architecture*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–27, 2019.

[214] U. K. Dasari, O. Temam, R. Narayanaswami, and D. H. Woo, *Apparatus and mechanism for processing neural network tasks using a single chip package with multiple identical dies*, Mar. 2, 2021. US Patent 10,936,942.

[215] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, *Mcm-gpu: Multi-chip-module gpus for continued performance scalability*, *ACM SIGARCH Computer Architecture News* **45** (2017), no. 2 320–332.

[216] A. Kannan, N. E. Jerger, and G. H. Loh, *Enabling interposer-based disintegration of multi-core processors*, in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 546–558, IEEE, 2015.

[217] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, *Pluto: A practical and fully automatic polyhedral program optimization system*, in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, Citeseer, 2008.

[218] P. Feautrier, *Some efficient solutions to the affine scheduling problem. i. one-dimensional time*, *International journal of parallel programming* **21** (1992), no. 5 313–347.

[219] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et. al.*, *The worst-case execution-time problem—overview of methods and survey of tools*, *ACM Transactions on Embedded Computing Systems (TECS)* **7** (2008), no. 3 1–53.

[220] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, *A statically scheduled time-division-multiplexed network-on-chip for real-time systems*, in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pp. 152–160, IEEE, 2012.

[221] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, *Ios: Inter-operator scheduler for cnn acceleration*, *Proceedings of Machine Learning and Systems* **3** (2021).

[222] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. M. Phothilimthana, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon, *Transferable graph optimizers for ml compilers*, 2021.

[223] B. Zimmer, R. Venkatesan, Y. S. Shao, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et. al.*, *A 0.11 pj/op, 0.32-128 tops, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16nm*, in *2019 Symposium on VLSI Circuits*, pp. C300–C301, IEEE, 2019.

[224] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, *Noc architectures for silicon interposer systems: Why pay for more wires when you can get them (from your interposer) for free?*, in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 458–470, IEEE, 2014.

[225] D. Stow, Y. Xie, T. Siddiqua, and G. H. Loh, *Cost-effective design of scalable high-performance systems using active and passive interposers*, in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 728–735, IEEE, 2017.

[226] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, *Deep learning scaling is predictable, empirically*, *arXiv preprint arXiv:1712.00409* (2017).

[227] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, *Outrageously large neural networks: The sparsely-gated mixture-of-experts layer*, *arXiv preprint arXiv:1701.06538* (2017).

[228] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, *Exploring the limits of language modeling*, *arXiv preprint arXiv:1602.02410* (2016).

[229] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, *Exploring the limits of weakly supervised pretraining*, in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[230] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, 2019.

[231] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, *Mesh-tensorflow: Deep learning for supercomputers*, 2018.

[232] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, *Gpipe: Efficient training of giant neural networks using pipeline parallelism*, *CoRR* **abs/1811.06965** (2018) [arXiv:1811.0696].

[233] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, *Gshard: Scaling giant models with conditional computation and automatic sharding*, 2020.

[234] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, *Pipedream: Generalized pipeline parallelism for dnn training*, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 2019.

[235] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, *Device placement optimization with reinforcement learning*, *ICML* (2017) [arXiv:1706.0497].

[236] Y. Gao, L. Chen, and B. Li, *Spotlight: Optimizing device placement for training deep neural networks*, in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, (Stockholmsmässan, Stockholm Sweden), pp. 1676–1684, PMLR, 10–15 Jul, 2018.

[237] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, *A hierarchical model for device placement*, *ICLR* (2018).

[238] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, *Placeto: Learning generalizable device placement algorithms for distributed machine learning*, *CoRR* **abs/1906.08879** (2019) [arXiv:1906.0887].

[239] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, *Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver*, 2019.

[240] B. Amos and J. Z. Kolter, *Optnet: Differentiable optimization as a layer in neural networks*, 2019.

[241] J. Achiam, D. Held, A. Tamar, and P. Abbeel, *Constrained policy optimization*, 2017.

[242] W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, *NIPS* (2017) [arXiv:1706.0221].

[243] Google, *Cp-sat solver*, .

[244] J. Devlin, M. Chang, K. Lee, and K. Toutanova, *BERT: pre-training of deep bidirectional transformers for language understanding*, *CoRR* **abs/1810.04805** (2018) [arXiv:1810.0480].

[245] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, *CoRR* **abs/1707.06347** (2017) [arXiv:1707.0634].