UNIVERSITY OF CALIFORNIA SAN DIEGO

Software-Hardware Co-design for Processing In-Memory Accelerators

A dissertation submitted in partial satisfaction of the

requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Minxuan Zhou

Committee in charge:

Professor Tajana Šimunić Rosing, Chair
Professor Farinaz Kushanfar
Professor Steven Swanson
Professor Dean Tullsen
Professor Jishen Zhao

2023

The Dissertation of Minxuan Zhou is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

This thesis is dedicated to my parents who have always been there for me without any reservation.

EPIGRAPH

To benefit from the lessons of history, architects must appreciate that software innovations can also inspire architects, that raising the abstraction level of the hardware/software interface yields opportunities for innovation, and that the marketplace ultimately settles computer architecture debates.

— John L. Hennessy and David A. Patterson, *A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development*, Communications of the ACM 62.2 (2019): 48-60.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Tajana Rosing for her support all the way along my Ph.D. I always learn a lot from her brilliant vision, hard work, and great personality. I am grateful for the help from the committee on my thesis and final defense. In addition, I must thank Professor Dean Tullsen for his help and guidance in my early research time. Then, I want to thank all my colleagues in SEELab throughout these years, including Mohsen Imani, Yeseong Kim, Saransh Gupta, Behnam Khaleghi, Joonseop Sim, Yunhui Guo, Daniel Peroni, Anthony Thomas, Michael Ostertag, Justin Moris, Weihong Xu, Jaeyoung Kang, and many others. They are always welcoming, friendly, and eager to help others and share ideas, creating a perfect atmosphere for my PhD life. The help from other collaborators in and outside UCSD was critical to the success of this work. To name a few, I would like to thank Professor Kevin Skadron, Lingxi Wu, Professor Jishen Zhao, Xiao Liu, Professor Kevin Eliceiri, Bing Li, Chris Wilkerson, Rosario Cammarota, Weifeng Zhang, Guoyang Chen, and others. I would also like to thank all funding sources including NSF, SRC CRISP, SRC PRISM, and DPRIVE program. Other than research, I feel lucky to have many friends who make my life in San Diego enjoyable. Most importantly, I would like to thank and show my love to my family for their love and support all the time. I must express very profound gratitude to my parents, Caiying and Shucheng, for giving me everlasting love and unfailing support throughout my whole life. I want to thank my lovely partner, Mingyue, for being my great companion who always provided me with love, support, and help.

Chapter 2, in full, is a reprint of the material as it appears in International Conference on Parallel Architectures and Compilation Techniques, 2021, Minxuan Zhou, Guoyang Chen, Mohsen Imani, Saransh Gupta, Weifeng Zhang, and Tajana Rosing. The dissertation author was the primary author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in The IEEE International Symposium on High-Performance Computer Architecture, 2022, Minxuan Zhou, Weihong Xu, Jaeyoung Kang, and Tajana Rosing. The dissertation author was the primary author of this paper.

Chapter 4, in full, is currently being prepared for submission for publication of the material,

Minxuan Zhou, Yujin Nam, Pranav Gangwar, Weihong Xu, Arpan Dutta, Kartikeyan Subramanyam, Chris Wilkerson, Rosario Cammarota, Saransh Gupta, and Tajana Rosing. The dissertation author was the primary investigator and author of this material.

VITA

2015	Bachelor of Engineering, Beihang University
2017	Master of Science, University of California San Diego
2023	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

M. Zhou, X, Wang, and T. Rosing, 'OverlaPIM: Overlap Optimization for Processing In-Memory Neural Network Acceleration," 2023 Design Automation and Test in Europe (DATE), Antwerp, Belgium, 2023.

J. Kang, **M. Zhou**, A. Bhansali, W. Xu, A. Thomas and T. Rosing, "RelHD: A Graph-based Learning on FeFET with Hyperdimensional Computing," 2022 IEEE 40th International Conference on Computer Design (ICCD), Olympic Valley, CA, USA, 2022

M. Zhou, W. Xu, J. Kang and T. Rosing, "TransPIM: A Memory-based Acceleration via Software-Hardware Co-Design for Transformer," 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea, Republic of, 2022

Y. Wei, **M. Zhou**, S. Liu, K. Seemakhupt, T. Rosing and S. Khan, "PIMProf: An Automated Program Profiler for Processing-in-Memory Offloading Decisions," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2022

M. Zhou, Y. Guo, W. Xu, B. Li, K. W. Eliceiri and T. Rosing, "MAT: Processing In-Memory Acceleration for Long-Sequence Attention," 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2021

Y. Kim, M. Imani, S. Gupta, **M. Zhou** and T. S. Rosing, "Massively Parallel Big Data Classification on a Programmable Processing In-Memory Architecture," 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), Munich, Germany, 2021

M. Zhou, L. Wu, M. Li, N. Moshiri, K. Skadron and T. Rosing, "Ultra Efficient Acceleration for De Novo Genome Assembly via Near-Memory Computing," 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), Atlanta, GA, USA, 2021

M. Zhou, G. Chen, M. Imani, S. Gupta, W. Zhang and T. Rosing, "PIM-DL: Boosting DNN Inference on Digital Processing In-Memory Architectures via Data Layout Optimizations," 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), Atlanta, GA, USA, 2021

X. Liu, **M. Zhou**, R. Ausavarungnirun, S. Eilert, A. Akel, T. Rosing, V. Narayanan, and J. Zhao, "FPRA: A Fine-grained Parallel RRAM Architecture," 2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), Boston, MA, USA, 2021

M. Zhou, M. Li, M. Imani and T. Rosing, "HyGraph: Accelerating Graph Processing with Hybrid Memory-centric Computing," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021

M. Zhou, M. Imani, Y. Kim, S. Gupta, and T. Rosing. 2021. "DP-Sim: A Full-stack Simulation Infrastructure for Digital Processing In-Memory Architectures". In Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC '21). Association for Computing Machinery, New York, NY, USA

M. Imani, S. Pampana, S. Gupta, **M. Zhou**, Y. Kim and T. Rosing, "DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 2020

X. Liu, **M. Zhou**, T. S. Rosing and J. Zhao, "HR3AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing," 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), Lausanne, Switzerland, 2019

M. Zhou, A. Prodromou, R. Wang, H. Yang, D. Qian and D. Tullsen, "Temperature-Aware DRAM

Cache Management—Relaxing Thermal Constraints in 3-D Systems," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 10

M. Zhou, M. Imani, S. Gupta, and T. Rosing. 2019. "Thermal-Aware Design and Management for Search-based In-Memory Acceleration". In Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19). Association for Computing Machinery, New York, NY, USA

M. Imani, S. Gupta, Y. Kim, **M. Zhou**, and T. Rosing. 2019. "DigitalPIM: Digital-based Processing In-Memory for Big Data Acceleration". In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI '19). Association for Computing Machinery, New York, NY, USA

M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing. 2019. "GRAM: Graph Processing in a ReRAM-based Computational Memory". In Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC '19). Association for Computing Machinery, New York, NY, USA

M. Zhou, M. Imani, S. Gupta, and T. Rosing. 2018. "GAS: A Heterogeneous Memory Architecture for Graph Processing". In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '18). Association for Computing Machinery, New York, NY, USA

K. Cheng, Y. Bai, Y. Zhao, Y. Ma, D. Lu, Y. Peng, **M. Zhou**, "HV2M: A novel approach to boost inter-VM network performance for Xen-based HVMs," Journal of Systems and Software, Volume 114, 2016

ABSTRACT OF THE DISSERTATION

Software-Hardware Co-design for Processing In-Memory Accelerators

by

Minxuan Zhou

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Tajana Šimunić Rosing, Chair

The explosive increase in data volume in emerging applications poses grand challenges to computing systems because the bandwidth between compute and memory cannot keep up with exploding data volumes. Processing in memory (PIM) is a promising technology to solve this memory problem by performing some key operations directly in and near memory. There remain several challenges in fully unleashing the power of PIM. Such challenges come from both the software and the hardware sides. On the software side, PIM requires that each operation happens where the data is. As a result, we need to first find the optimal data layout for each application,

prior to running it in PIM. On the hardware side, due to the limited functionality of PIM operations, PIM acceleration may require customized logic to achieve high performance. Software-hardware co-design plays a critical role in order to fully exploit PIM acceleration. There are a number of challenges to PIM-based software-hardware co-design. First, software mapping (data layout) in PIM architecture has an extremely large design space. Second, the hardware customization should have minimum overhead to maximize memory capacity.

This thesis presents several novel techniques for PIM software-hardware co-design. To tackle the challenges of mapping applications to PIM architecture, this thesis proposes a PIM data layout framework that efficiently optimizes data layout of widely-used machine learning (ML) operators onto general PIM architectures. This thesis also presents the software-hardware co-design for both conventional and non-convolution ML models using PIM architecture. The presented optimizations provide at least 3.7$\times$ speedup over conventional PIM acceleration methods. Finally, this thesis proposes a software-hardware co-design for fully homomorphic encryption, which is a challenging and critical application in cryptography, resulting in 30$\times$ speedup and energy efficiency over the existing PIM solutions. The target applications in this thesis cover a wide range of challenging operations and data transfer patterns needed for various emerging computing tasks, shedding light on the software-hardware co-design for future systems with PIM acceleration.

Chapter 1

Introduction

Data volume is exploding with the emergence of data-intensive applications, such as bioinformatics, internet of things, and machine learning. Conventional computer systems cannot ensure good performance of such data-intensive applications due insufficient memory bandwidth between the processor and the memory. Processing in memory (PIM) is a promising technology to solve this problem by performing some operations in memory. The major advantage of PIM over conventional computing is that PIM can exploit the internal bandwidth of memory which is significantly higher than the off-chip bandwidth used for host-memory communication. For example, each DRAM bank can read 1kB data to its sense amplifier in around 30 ns, providing 30GB/s bandwidth [7, 8]. Considering a 16GB HBM2 stack consists of 32 pseudo-channels and each channel contains 8 banks, each HBM2 stack can provide 7.7TB/s bank-level bandwidth, significantly larger than 256GB/s - 512GB/s of its off-chip bandwidth [9]. We can further increase the exploited internal bandwidth by adopting more fine-grained parallelism, like subarray-level parallelism [10], or using bit-serial PIM scheme [11]. Furthermore, PIM is more scalable than conventional computing by simultaneously increasing compute throughput, memory bandwidth, and capacity. The rest of this chapter covers the PIM background and challenges of PIM software-hardware co-design, and provides an overview of thesis contributions.

1.1 Memory System and Processing In-Memory

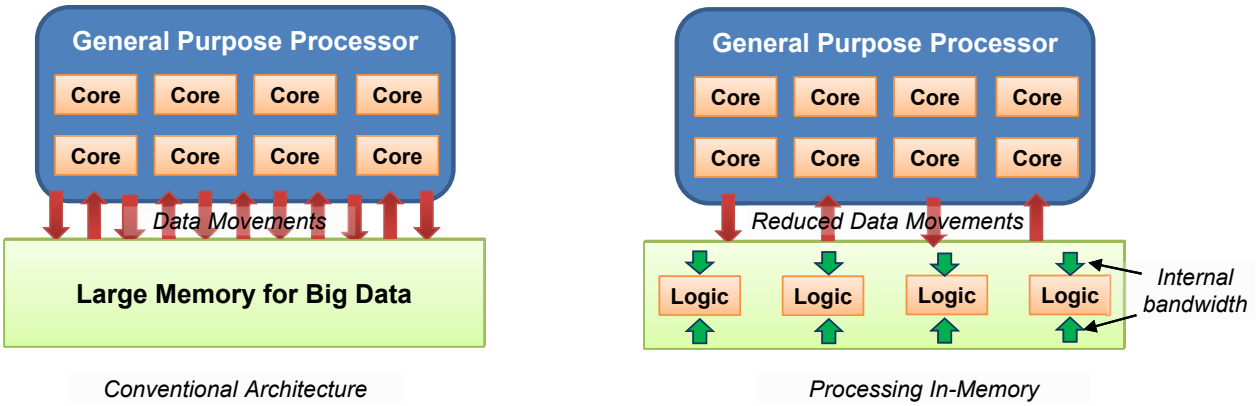


Figure 1.1: The difference between conventional architecture and PIM architecture.

Memory stores data used for computation. To execute a task on a conventional architecture, the computer needs to first load the data from memory to the place where the processing elements (e.g., ALU) can directly access it. However, data loading may incur large latency. For instance, a typical data load of a cache line (e.g., 64B) from dynamic random access memory (DRAM) requires tens of nanoseconds, while ALU only requires 1 or 2 cycles. Such discrepancy motivates computer architects to place fast memory components near processing elements, which is referred to as cache. While on-chip cache provides superior performance over off-chip memory, the density of cache is significantly lower than memory, limiting the capacity of cache that can be placed on the chip. In the case of either a low data reuse rate or a large data volume, off-chip data movements dominate the overall system latency.

The current systems are experiencing an explosive increase in data size for various emerging applications. Conventional general-purpose architectures do not perform well on big data applications due to the limited memory bandwidth. Processing in-memory (PIM) solves such problems by integrating computing logic with memory. Figure 1.1 shows a high-level structure of a conventional (left) and a PIM system (right). Logic-memory integration of PIM provides several significant benefits over conventional architecture. First, PIM reduces the latency and energy consumption required for data movements because of the reduced distance between computing logic and data.

Second, PIM exposes the internal memory bandwidth for computing logic, which can be several to tens of times higher than the off-chip memory bandwidth in conventional architectures. Third, PIM can scale computing throughput, memory bandwidth, and memory capacity simultaneously by simply integrating more PIM computing devices into the system.

1.2 PIM Hardware Technologies

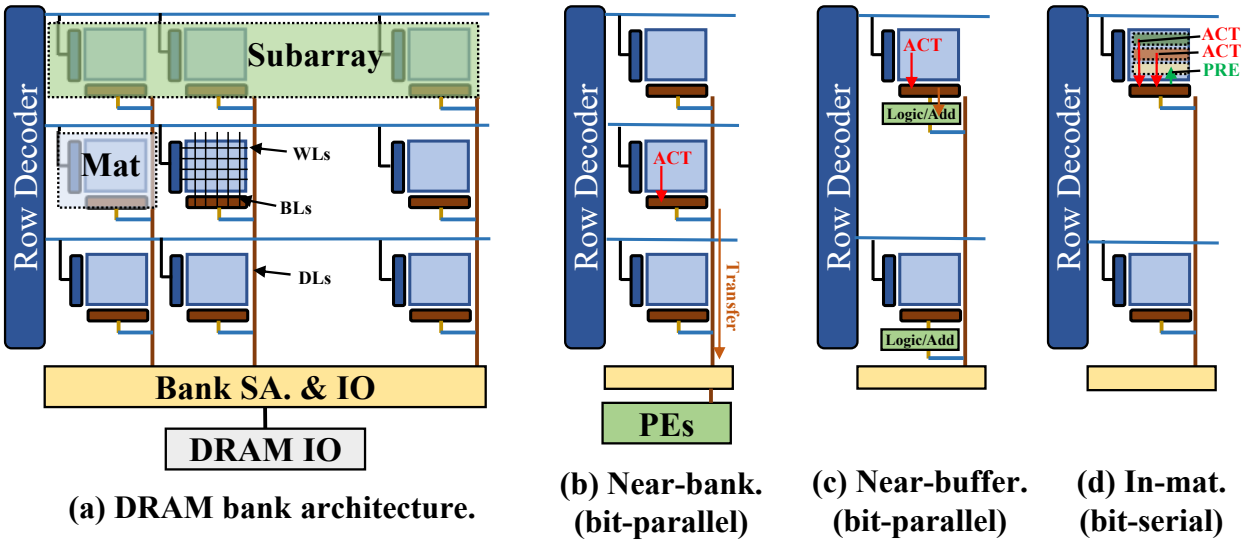


Figure 1.2: Different in-DRAM PIM technologies.

There are various ways of supporting PIM in memory. Figure 4.2 shows different implementations of PIM in a DRAM-based memory as an example. Figure 4.2(a) shows the architecture of a DRAM bank which is the basic hardware component of DRAM. A bank consists of 2D cell arrays and peripherals to transfer data between DRAM cells and IOs. The memory cells are grouped into a set of subarrays, each of which consists of a row of mats. Each mat has local sense amplifiers (row buffers) sensing a horizontal wordline (WL) through a set of vertical bitlines (BLs). Sense amplifiers in mats of a subarray form the subarray row buffer. Upon receiving a DRAM access, the bank activates the corresponding WL in the subarray row buffer and transfers the whole WL to the bank-level sense amplifiers via the data lines (DLs).

There have been several DRAM-based PIM technologies that support operations at different

levels of DRAM. Near-bank processing, as shown in Figure 4.2(b), integrates processing elements (e.g., vector ALUs, RISC-V processor, etc.) near the bank sense amplifiers (SA) and IO [7, 8, 12]. Each bank-level processing element (PE) is customized to fully utilize the data link bandwidth for processing. PEs in different banks run in parallel to fully utilize the internal bandwidth in DRAM.

Figure 4.2(c) shows another type of PIM (near-buffer), which augments the subarray sense amplifiers (row buffers) with compute logic [13]. Due to the constrained chip area, the near-buffer PIM only adopts logic gates, full adders, and shift circuits for multi-bit operations. Therefore, the near-buffer PIM can only process simple operations for each activated WL, and the intermediate results for complex operations should be written back to the cells and read out again for future operations. As compared to near-bank processing, near-buffer PIM support wider input (e.g., 8192b vs. 256b) and can exploit the subarray-level parallelism [10], compensating the long latency of operations. These two types of PIM work on the data with a horizontal layout where each data is stored across multiple BLs in a WL.

The third type of PIM (in-mat), as shown in Figure 4.2(d), uses a vertical bit-serial scheme that lays out data in different WLs of a BL [11, 14, 15]. The bit-serial PIM directly generates the result of computation between different WLs by exploiting the charge-sharing effect of the DRAM mechanism. The bit-serial PIM connects two or more WLs to the sense amplifiers simultaneously using a sequence of activate commands. It then issues a precharge command to a row that will store the values in the sense amplifier after charge sharing. By applying specific voltage and connecting operand WLs to reference WL (e.g., all BLs store "1" or "0"), such activate-activate-precharge (AAP) operation can implement logic operation and majority-based full adder. As compared to near-buffer processing, such in-mat bit-serial processing supports a similar degree of parallelism without introducing costly logic. However, bit-serial PIM requires more activation and precharging commands because each AAP destroys data in operand WLs, requiring row-clone operations [16] to back up the original data. An n-bit multiplication using the bit-serial PIM requires around $7n^2$ AAP operations for 8k values; near-buffer processing requires around $5n$ activation and precharging commands for 8k/n values using carry-save adders, resulting in at least 1.4× higher throughput and

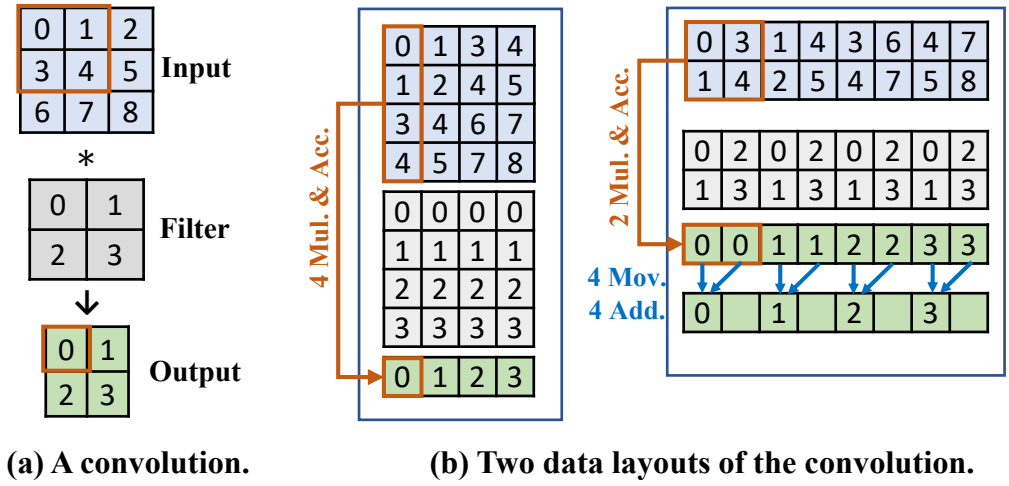
energy efficiency.

In addition to DRAM-based technologies, researchers have investigated PIM architectures with various memory technologies, including SRAM [4, 17, 18], ReRAM [3, 19–21], FeFET [22–24], and NAND Flash [25]. All these technologies share a similar idea of PIM on levels higher than subarray, by integrating compute logic in the peripheral circuit of that level in the memory hierarchy. For the subarray-level PIM, each technology exploits the circuit behaviors of a specific memory device. For example, by applying appreciate voltage to the ReRAM or FeFET cell array, we can get the result of specific computations by sensing the voltage of WLs/BLs based on Kirchhoff's law. For subarray-level computation on 1-bit values, PIM can support arbitrary computations by using universal logic operations. It is also possible to support PIM computations in memory that stores multi-bit value in a cell [19,23,26]. However, the multi-bit PIM computation cannot support arbitrary computation due to limited functionality. In this thesis, we focus on the PIM technologies that use the conventional 1-bit-per-cell data, which is usually referred to as *digital-PIM technology* [27, 28].

1.3 Software-Hardware Co-Design in PIM

Despite significant advantages of PIM, there are challenges from both software and hardware to fully unleash its power. On the software side, PIM enforces each operation to happen in the location of data inside the memory. Therefore, before processing the target application using PIM operations, we need to first determine the data layout of the application. The detailed data layout affects not only the computation latency but also the data transfer pattern as well as the memory usage.

Figure 1.3 shows an example of different data layouts for convolution in a PIM-enabled memory block, which can process a row of data in parallel. The first data layout (left in the Figure 1.3(b)) parallelizes the computations for different output values in 4 columns, requiring 4 multiply and accumulations (MACs). The second data layout (right) provides 2$\times$ more parallelism than the first layout by parallelizing computations of 2 partial results of each output value, only requiring 2 MACs. However, the second layout requires extra data movement and additions to generate the final output



(a) A convolution. **(b) Two data layouts of the convolution.**

Figure 1.3: **Two data layout methods for a convolution.**

values. Furthermore, the two layouts require different memory rows and columns, making the layout problem significantly more complicated.

On the hardware side, due to the limited functionality of PIM operations, PIM acceleration may require customization in the memory device to achieve high performance. For instance, PIM supports extensively parallel arithmetic on vector data but lacks an efficient mechanism to support operations with complex data movements. When using the second layout in Figure 1.3(b), data movement of reduction operation can introduce large overhead, offsetting the performance benefits of higher parallelism of MACs in PIM. The adoption of new customized hardware for the missing functionality can significantly boost PIM performance.

1.4 Data Layout Optimization in PIM for Deep Neural Networks

PIM acceleration is one of the most compelling solutions for DNN acceleration, due to DNN parallelism. PIM-based DNN accelerators require that data be properly laid out to make the best use of the available in-memory operations. However, previous PIM accelerators tend to optimize for a particular DNN dataflow, neglecting the large design space of data layout. Chapter 2 systematically investigates the data layout for PIM DNN acceleration. We propose a mapping framework to represent the whole design space of PIM data layout for general DNN models. Our investigation

shows that an exhaustive exploration of the whole design space for mapping a DNN application to PIM architecture is not computationally tractable. Therefore, we propose a compiler-level optimization, PIM-DL, that finds highly efficient data layouts for PIM DNN acceleration using a two-level dynamic programming algorithm and a heuristic-based search. Our experiments show that DNN PIM solutions created by our PIM-DL provide 3.7× and 4.3× better performance and energy efficiency as compared to the state-of-the-art under the same hardware constraints.

1.5 Software-hardware Co-design for Complex Applications

In Chapter 3 and Chapter 4, this thesis extends the discussion of software-hardware co-design to more complex applications other than conventional DNNs, including Transformers and Fully Homomorphic Encryption. PIM acceleration of these complex applications requires several special considerations on both the software and hardware sides.

1.5.1 Transformers

Transformer-based models are state-of-the-art for many machine learning (ML) tasks. Transformers usually execute slowly because of thier large memory footprint and low data reuse rate, stressing the memory system while under-utilizing the compute resources. Memory-based processing technologies, including processing in-memory (PIM) and near-memory computing (NMC), are good solutions to accelerate Transformers by minimizing data movement and enabling extensive compute parallelism. However, the previous memory-based ML accelerators design software and hardware components for compute-intensive ML models (e.g., CNNs), which may not fit specific characteristics of Transformers. In Chapter 3, we propose TransPIM, a memory-based acceleration for Transformer based on software and hardware co-design using emerging high-bandwidth memory (HBM). At the software-level, TransPIM uses a token-based dataflow to avoid the expensive inter-layer data movements introduced by previous layer-based dataflow. TransPIM also introduces lightweight hardware modifications in the HBM architecture to support PIM-NMC hybrid processing

and efficient data communication for accelerating Transformer-based models. Our experiments show that TransPIM is 3.7× to 9.1× faster than existing memory-based acceleration. As compared to conventional accelerators, TransPIM is 22× to 115× faster than GPUs and provides 2× more throughput than ASIC-based Transformer accelerators.

1.5.2 Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) enables arbitrary computations on encrypted data without decryption, thus providing full security to applications. Unfortunately, FHE computation is orders of magnitude slower than computation on plain data due to the explosion in data size after encryption. Processing in-memory (PIM) is promising to accelerate data-intensive workloads with extensive parallelism. However, FHE is challenging for PIM acceleration due to the intensive long-bitwidth computations and complex data movements. In Chapter 4, we propose a PIM-based FHE accelerator, FHEmem, which exploits processing in-memory technologies with a novel memory architecture customized for FHE operations to achieve high-throughput and energy-efficient acceleration. We propose an optimized processing flow, from low-level hardware processing to high-level application mapping, that fully exploits the high throughput of FHEmem hardware for FHE applications. Our evaluation shows FHEmem provides up to 10.7× higher throughput than existing FHE accelerators under similar power and chip area constraints. As compared to previous PIM acceleration, FHEmem is up to 30× more efficient due to the software-hardware co-design.

Chapter 2

Data-Layout Optimization of PIM

To optimize execution of applications on PIM, it is critical to systematically analyze the design space and to perform detailed performance modeling of different design choices. However, the co-optimization of software and hardware can be extremely complicated because both parts have large design space with complex mutual interaction. One way to address this it to investigate software-level optimizations on a generic PIM architecture model, or to investigate the hardware-level optimization for key operation patterns found in target applications. This chapter discusses our comprehensive optimization framework which comprehensively optimizes PIM acceleration, and shows its impact on deep neural networks (DNNs) as an example application.

2.1 Introduction

Deep neural networks (DNNs) have been used in various application domains, ranging from natural language processing [29, 30], speech recognition [31, 32], to image object detection [33–37]. The size of emerging DNNs has become extremely large due to the need for high accuracy, posing significant challenges to conventional architectures because of limited parallelism and memory wall issues [29, 30, 38–41].

PIM is a promising non-conventional technology to accelerate emerging data-intensive applications by not only reducing the data movement but also increasing computing parallelism. Researchers have used PIM to accelerate neural networks [3, 4, 13, 19, 26, 42] and other applications from a wide range of fields [20, 41, 43–49]. Many PIM-based DNN accelerators rely on the analog domain of resistive memory (ReRAM) [19, 26, 42]. Although such accelerators provide significant improvement in both energy efficiency and performance, they are very inefficient area-wise due to costly peripherals needed for data conversion, have no floating-point support, and face scaling difficulty due to unstable multi-bit cells [42].

Digital PIM (DPIM), enables in-situ computations in conventional digital memory including ReRAM [50,51], DRAM [14,16], and SRAM [17,28]. Since DPIM can directly work on digital data, it does not require costly peripherals for data conversion needed by analog PIM [19, 26, 42]. Several works have shown promising performance and scalability of DPIM-based DNN accelerators [3,4,13].

DPIM acceleration has strict requirements on how data should be placed in memory. Such data layout determines the degree of parallelism as well as other critical factors including memory utilization, and data throughput. The data layout problem in DPIM architectures is different from that in other types of spatial accelerators, which assume hierarchical architecture with separate storage components and processing elements. DPIM architecture only has a large digital memory which combines both computing and storage functionality. DPIM accelerators usually have thousands of large basic components (e.g., 1Mb digital memory block), leading to a large design space for optimization of application data and operation placement.

Most state-of-the-art DPIM DNN accelerators [3,4,13] use an output-parallel layout that allocates separate memory rows for computations of different output elements. Such a layout requires significant data duplication because different outputs usually share a lot of input and filter data. As reported in NeuralCache [4], input loading and filter loading may contribute to 61% latency during the DNN inference. If we simply reorganize the layout to combine every two computations with a shared operand in the memory, we can approximately reduce 25% of data loading at a cost of double computation time. This shows there is a trade-off between computation parallelism and memory

usage for DPIM data layout. There have been several work [1, 2] systematically investigated the design space of mapping one DNN layer to conventional DNN accelerators. However, this has not been done in the DPIM scenario. The design space of DPIM data layout becomes even larger for a category of PIM-based DNN accelerators [3, 19, 26, 42], which pre-allocate memory resources for different program regions (usually layers). These accelerators, which we call static accelerators, introduce additional design dimensions for memory allocation. In this work, we show that the combination of conventional per-layer mapping and DPIM-specific memory allocation forms an extremely large design space in DPIM architecture.

To fully explore the design space of DPIM DNN acceleration, we propose a new mapping framework of data layout. Unlike the state-of-the-art mapping framework [1, 2] for conventional DNN accelerators, the proposed framework considers both general and DPIM-specific dimensions of the design space. The decision of each dimension has an impact on multiple aspects of acceleration including computation cost, memory usage, and data movement pattern. All these aspects determine the overall efficiency of DPIM acceleration. With our framework, we can formulate the design space of data layout problem in DPIM DNN acceleration, which has an exponential complexity with the number of DNN layers and the number of available memory components (e.g., block) in the DPIM system.

The complexity of the design space makes it impossible for an exhaustive exploration using an efficient algorithm. Therefore, we further propose PIM-DL, a data layout optimization framework, to accelerate DPIM DNN inference. PIM-DL utilizes a two-level dynamic programming algorithm and a genetic algorithm based on heuristic search to efficiently find a good data layout based on application and hardware information, that performs better than previous methods. We evaluate PIM-DL on DPIM acceleration for several widely-used DNNs by an open-sourced DNN compiler and a cycle-accurate simulator. Our experiments show that PIM-DL can improve the performance of DPIM DNN acceleration under various architecture configurations by up to 1.9× while using 30% less memory without any hardware change. Furthermore, we apply PIM-DL to several state-of-the-art DPIM DNN accelerators and observe a 2.5× speedup on average. We also explore the

design space of hardware-software co-design, inspired by our data layout experiments, and compare several customized systems with PIM-DL to state-of-the-art DPIM DNN accelerators varying in memory technologies and acceleration modes. Our experiment shows our software-hardware co-design systems provide 3.7× and 4.3× better performance and energy efficiency than corresponding baselines under the same hardware constraints.

Overall, we make the following contributions in this work:

- This is the first work that comprehensively and systematically investigates the data layout problem in DPIM DNN acceleration. We generalize the data layout problem in DPIM using a DNN mapping framework for customized accelerators including DPIM-specific design dimensions which have not been investigated.
- We design compiler-level optimizations for a generic DPIM architecture to generate efficient data layout for DNN inference and dramatically boost performance and memory utilization of general DPIM DNN acceleration.
- Inspired by our experiments on data layout, we exploit the hardware-software co-design to build several customized systems which significantly improve the performance and energy efficiency of state-of-the-art.

2.2 Background and Motivation

This section elaborates on important preliminaries for DPIM-based DNN acceleration.

2.2.1 DPIM Architecture Model

This work targets a general DPIM architecture model which is shown in Figure 2.1. The DPIM architecture consists of multiple tiles, where tiles are connected to each other through an inter-tile network. Each tile contains several DPIM blocks and uses an inter-block interconnect to handle data

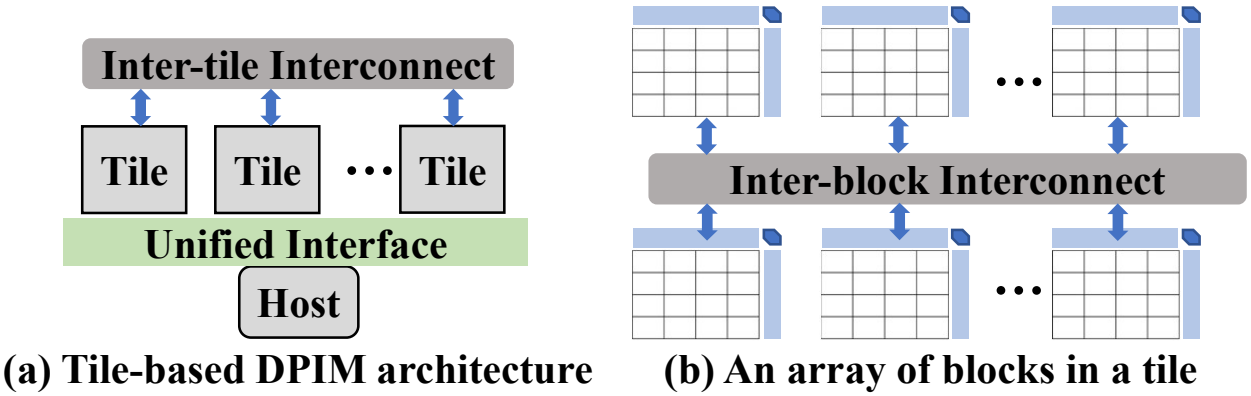


Figure 2.1: Generic DPIM architecture model.

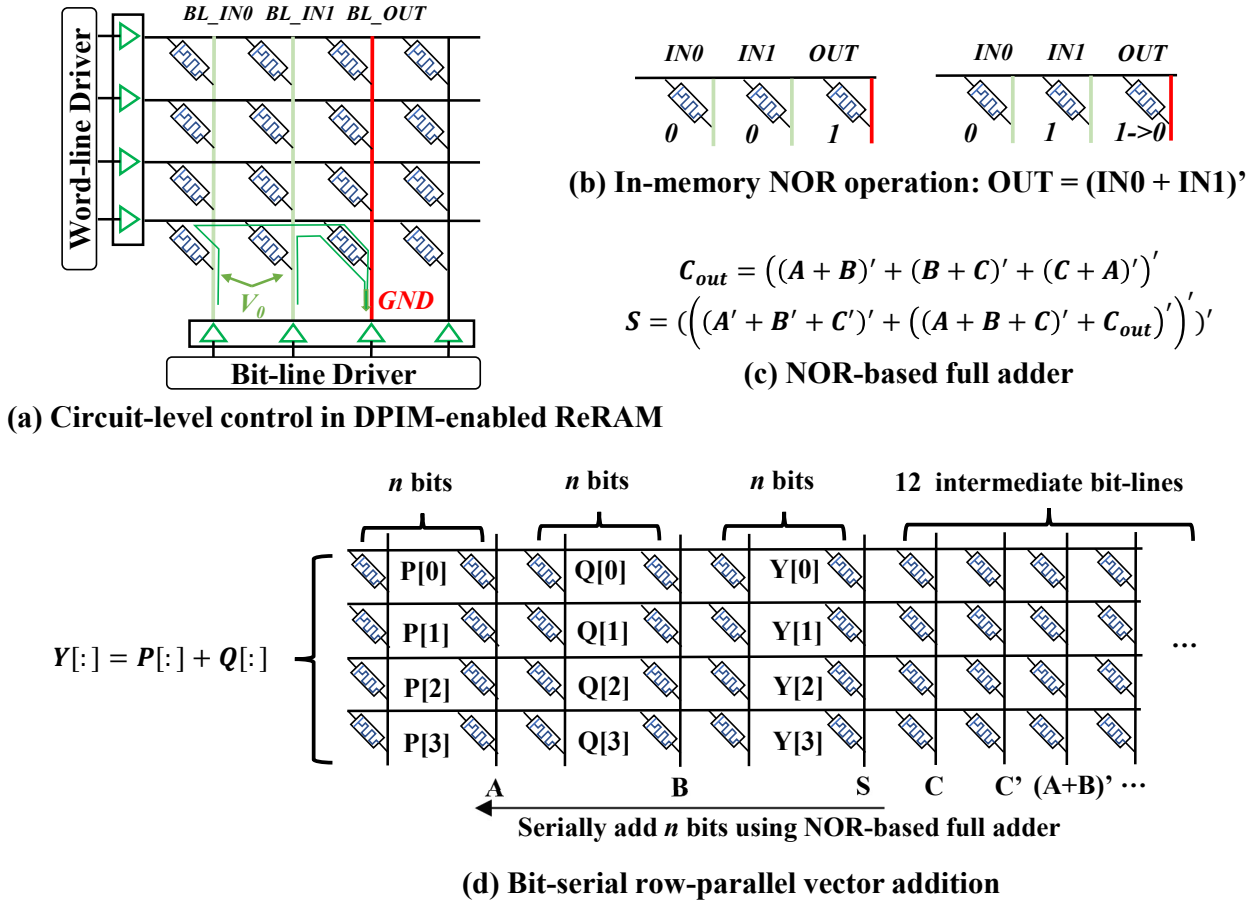


Figure 2.2: Operations in a ReRAM-based DPIM.

movements between different blocks. We can configure this model to emulate state-of-the-art DPIM architectures by customizing hardware characteristics like operation latency and memory structure.

Several recent works have implemented DPIM functionality in various digital memory technolo-

gies including SRAM [4, 17], DRAM [13, 14], and ReRAM [3, 50, 52]. The common backbone of various DPIM technologies is row-parallel bit-serial operation, which exploits the shared bit-line circuits in the digital memory block to process all rows in a bit-line using a single step. When using a universal bit operation, DPIM-enabled memory can support custom computations by sequentially executing multiple bit-line steps. We take an example of a DPIM-enabled ReRAM block to illustrate basic DPIM operations. Figure 2.2(a) shows a memory block that includes a crossbar of ReRAM cells and peripheral circuits for driving bit-lines and word-lines. By applying a specific voltage to the bit-lines, the value stored in a resistive cell may change. This operation can take effect on all word-lines (rows) enabling highly parallel operations. Figure 2.2(b) shows an example of computing NOR operation, a universal operation that can be used to implement custom functions. Figure 2.2(c) shows a NOR-based 1-bit full adder that takes 12 NOR steps.

We can exploit this 1-bit full adder to support multi-bit operations. Figure 2.2(d) shows an example of addition for two vectors with 4 n-bit values. The memory sequentially applies the full addition to each bit of computation by reusing the carrier. In addition to the carriers, we also need to reserve several bit-lines (12 in this case) to store intermediate values like $(A+B)'$. These intermediate bit-lines can be shared across computations. This scheme can process an n-bit vector addition in $12n+1$ steps for all elements. When processing long vectors, this brings a significant performance benefit because of the extremely high degree of parallelism. Other than addition, we can also implement different custom functions including multiplication, subtraction, and division. In addition to integer values, DPIM also supports floating points by separately computing exponent and sign bits [3].

We should note that different memory technologies have different schemes for DPIM operations. For example, ComputeCache [17] and NeuralCache [4] proposed to exploit the sense amplifier in SRAM to sense shared bit-lines between two activated rows. ComputeDRAM [14] utilized specialized sequences of DRAM commands (e.g., row activation and pre-charge) to implement DPIM operations in commodity DRAM chips by only slightly modifying the memory controller. The high-level computing scheme of different DPIM technologies is still row-parallel bit-serial

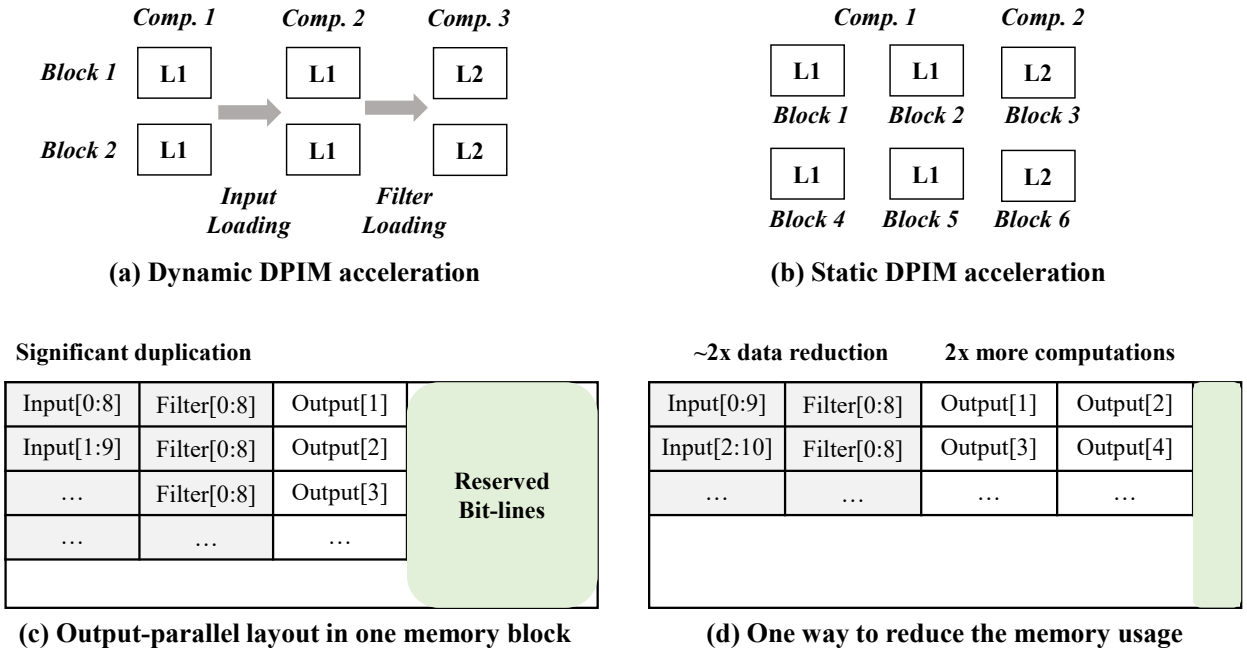


Figure 2.3: DPIM DNN accelerators and data layout.

operation so that we can design several general strategies, including the data layout, for most DPIM accelerators. We refer readers to previous works for more details on the circuit-level design and implementation for different DPIM technologies [3, 4, 13, 14, 17].

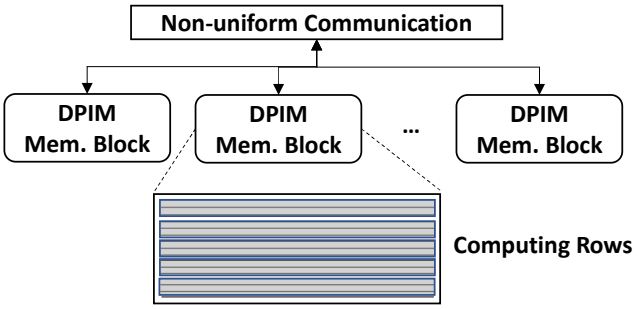
2.2.2 DPIM-based DNN Accelerators

DPIM architecture can emulate a large group of SIMD processing units, which is promising to accelerate DNN applications. We can categorize DPIM DNN accelerators into two groups, dynamic and static, based on the acceleration mode as shown in Figure 2.3(a). On the one hand, dynamic DPIM DNN accelerators allocate all memory resources to process one DNN layer at a time. The weights and inputs of the layer need to be loaded to the memory before computation. After the layer is done with computation, its results are reorganized in the memory to compute the next layer. Even though the dynamic accelerator fully utilizes memory resource for computations, it has a couple of drawbacks. First, weight loading for each layer would consume significant amount of time and energy due to data loading from external memory [4]. Second, processing one layer at a

time cannot support pipelined execution between different layers for more throughput. On the other hand, the static acceleration mode stores weights for the whole DNN model statically and has been widely adopted in many types of PIM DNN accelerations, like analog PIM (e.g., Pipelayer [19], ISAAC [42], Prime [26], etc.) and near data processing (e.g., Tetris [53]). In the static acceleration, a portion of the memory handles computations for a specific layer, and sends results to another portion of the memory handling the next layer. Even though static accelerators require more memory, it avoids weight loading during the runtime and can pipeline the execution of different layers to improve the throughput. Since dynamic acceleration can be represented as a special case of static acceleration, this work focus on the static DPIM acceleration and provide results to both dynamic and static acceleration in Section 2.6.

State-of-the-art DPIM DNN accelerators, both dynamic and static, use the fully output-parallel mapping for all DNN workloads. Figure 2.3(c) shows an example of 1D convolution with a 1*9 filter using the fully output-parallel layout used by NeuralCache [4], which allocates the computation for each output element in a memory row, exploiting row-parallel operations to process all outputs in parallel. However, such output-parallel mapping introduces data duplication that may significantly increase the memory usage. In dynamic accelerators, large memory usage may require the system loads data several times for a layer since that cannot fit all data. This introduces large overheads due to data loading and computing sub-parts sequentially. For static accelerators, this data layout may significantly increase the memory requirement for DNN workloads. To reduce the memory usage, Figure 2.3(d) shows one method that combines each two computations, sharing an operand, to one memory row. This method reduces the memory usage by almost $2\times$ at the cost of $2\times$ more row-parallel operations. There are other design dimensions that we can fine-tune the data layout to achieve better performance by comprehensive exploration.

Recent DNN mapping frameworks, such as Timeloop [1] and Interstellar [2], explore a relatively limited design space of DNN application mapping, with focus on a basic nested loops and primitives such as loop splitting and loop reordering. We refer to them as conventional frameworks. Even though the DPIM architecture is similar to single-level distributed processors (e.g., memory rows),



(a) DPIM accelerator model.

```
# Mapping for channels
for m = [0, M):
  for n = [0, N):
    # Mapping for operations
    for f = [0, F):
      for o = [0, O):
        Output[m][o] += Filter[m][n][f]
                        * Input[n][o+f]
```

(b) An example 1D-Conv layer for mapping.

Figure 2.4: (a) The architectural abstraction of DPIM accelerator using conventional DNN accelerator model [1, 2]. (b) An example of 1D convolution layer.

the conventional frameworks cannot fully explore the design of data layout in DPIM architectures for three reasons. First, the DPIM architecture has a single type of component that acts as both a compute and a storage unit. However, the conventional framework does not consider the data layout for different computation mappings. Second, the conventional framework only explores the design space of a single layer, which cannot support static DPIM accelerators. DPIM accelerators require a holistic framework that consider the global constrains of the architecture when mapping all the layers. Third, the locations of the memory partitions allocated to different layers also have an impact on the overall system performance because of the inter-layer data movements. For example, we may need to move the output of the preceding layer to the input of the succeeding layer. Most accelerators have a non-uniform communication network for all processing units (e.g., memory blocks or rows in DPIM architectures), so the overhead of data movement varies as a function of the allocation scheme. In this work, we propose a novel mapping framework for DNN data layout on DPIM architecture (Section 4.2), and an efficient data layout optimization that finds a good data layout in the large design space (Section 2.4).

2.3 PIM-DL Data Layout Framework

We formulate the data layout of a DNN model with n layers in DPIM architectures using two sets: global layout strategy $L = \{l_i, i \in \{1...n\}\}$, and memory allocation $M = \{m_i, i \in \{1...n\}\}$.

Specifically, l_i is the data layout strategy for layer i. In Section 4.2, we introduce a way to use the conventional DNN mapping framework to represent the layer layout in DPIM. Furthermore, m_i is a set of memory resources allocated for layer i using the layout l_i. As compared to the conventional framework, the data layout of DNN model in DPIM architecture has a significantly larger design space. Assuming the size of design space of conventional framework is S, the design space of L has a size of S^n. For each global layout strategy L, the number of possible memory allocations is $N_m(N_m-1)...(N_m-k+1)$, where N_m is the number of memory resources in the DPIM architecture and k is the number of memory resources required for L. In this work, we use the memory block as the granularity of memory resource allocation.

Our mapping framework for DNN applications enables optimization across the full design space of DNN data layout on DPIM architectures. This section illustrates all design dimensions of the framework which impact different aspects of DPIM DNN acceleration including computing parallelism, memory utilization, and data transfer pattern. We investigate data layouts and corresponding cost models of different DNN mappings through the example 1D-conv layer as shown in Figure 2.4(b). The example convolution has N input channels and M output channels. The filter size is F and the convolution generate O outputs for each output channel. We select convolution as the main example because it is the most complex and time-consuming in a wide range of emerging DNNs. We should note that our analysis is applicable to other layers like fully-connected layer, which can also be represented as a nested loop.

2.3.1 Operation Layout

We first investigate the detailed data layout of mapping a single 1D convolution. Figure 2.5 shows layouts and cost models of 4 basic mappings. We use $parallel\_for$, used in Timeloop [1], to indicate a spatial parallel for a loop in the mapping. As mentioned before, the basic hardware components for resource allocation is memory rows, so that a $parallel\_for$ places all operands for each item in a row and aligns computations for all items in the same bit-lines. In this case, DPIM can process all these items in parallel using row-parallel bit-serial operations. In addition, a normal for

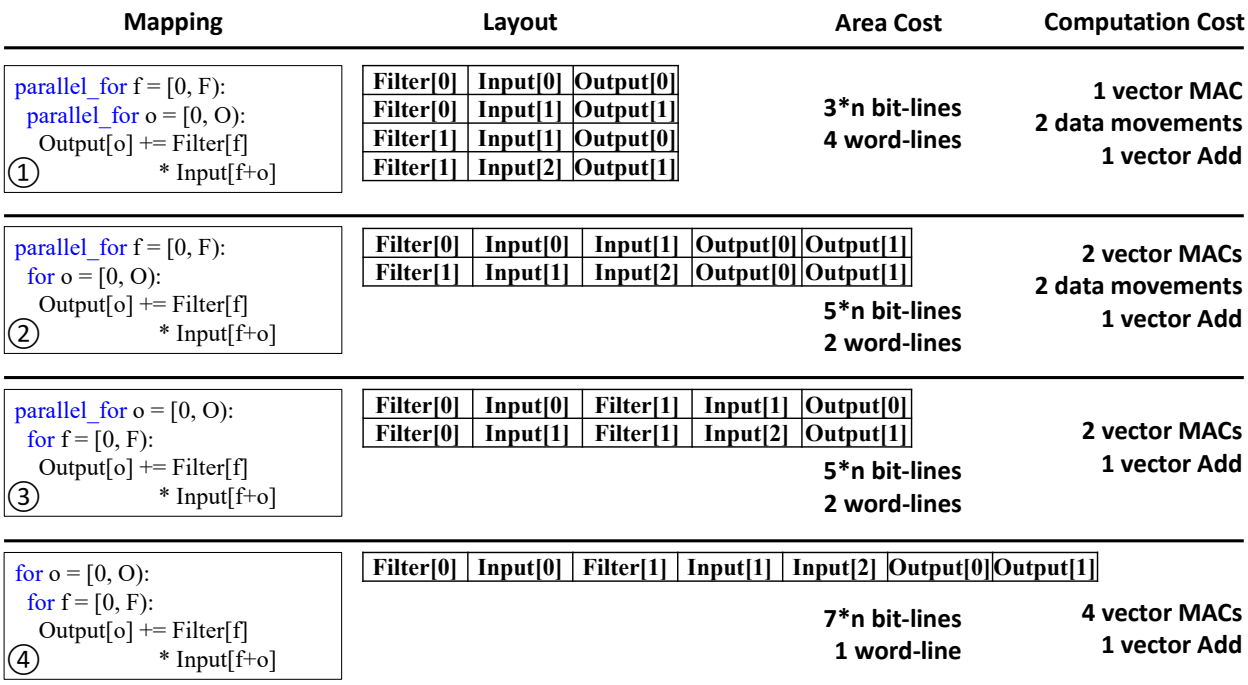


Figure 2.5: Data layouts of different 1D-Conv mappings.

```
# === DPIM Mapping for 1D-Conv ===
# First-level mapping
parallel_for f1 = [0, F1):
  parallel_for o1 = [0, O1):
    # Second-level mapping
    parallel_for o0 = [0, O0):
      for f0 = [0, F0):
        o = o1*O0 + o0
        f = f1 * F0 + f0
        Output[o] += Filter[f] * Input[f+o]
```

Filter[0]	Input[0]	Filter[1]	Input[1]	Output[0]
Filter[0]	Input[1]	Filter[1]	Input[2]	Output[1]

Filter[0]	Input[2]	Filter[1]	Input[3]	Output[2]
Filter[0]	Input[3]	Filter[1]	Input[4]	Output[3]

Area Cost	Computation Cost
5*n bit-lines	2 vector MACs
4 word-lines	1 vector Add

Figure 2.6: Loop tiling with an additional level.

places computations for items in a row, requiring sequential execution. The most parallel mapping is Mapping 1 (Figure 2.5) which parallelizes all computations. However, such straightforward layout requires data duplication to maximize the parallelism. Furthermore, Mapping-1 needs to re-align partial sums to generate final outputs, requiring extra data movements.

One way to reduce the data size is to combine rows with shared elements by folding the loop. For example, Mapping-2 folds computations of 2 outputs in one row for each filter replica, saving memory space for 2 elements. However, such mappings introduce extra vector MACs. We can also avoid data movements by changing the order of the loop as shown in Mapping-3 and Mapping-4. We can further enlarge the design space by add another level of loop in the operation, as shown in Figure 2.6. The example shows we can control the number of parallel computation outputs by adopting different parallel schemes in two levels. According to the data size and architecture configurations, the most efficient mapping might be different. Furthermore, the memory requirement (including the dimension of bit-lines and word-lines) is also important since it influences the constraints for other design dimensions.

2.3.2 Layer Layout

Emerging DNNs usually apply operations on multi-channel data to capture comprehensive features. The result of each output channel is a function of different input channels with corresponding filters. Such channel-level parallelism provides another dimension of the design space for data layout. In the conventional mapping framework, we can explore the layer layout by changing the order of loop. Figure 2.7 shows two example mappings of layer layout to illustrate the cost models in the layer-level.

Output-parallel Data Layout

We can parallelize a convolution layer along the dimension of output channel because computations of output channels are independent. Figure 2.7(a) shows an output-parallel mapping which schedules a $parallel\_for$ for the output channel in the outer loop. This example mapping

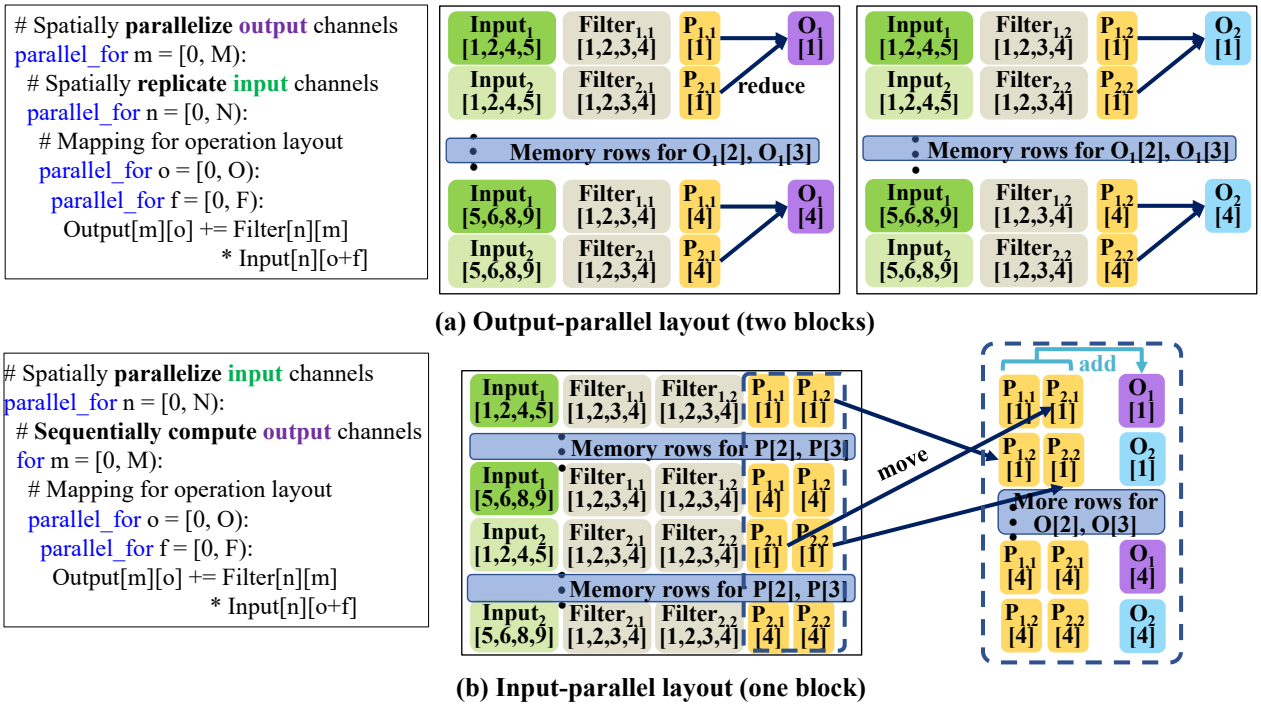


Figure 2.7: Two channel-parallel data layout schemes for convolutions of two input/output channels.

further apply *parallel_for* for all inner loops to fully parallelize convolutions for all output elements. Because of such spatial parallelism, we need extra data movements to reduce partial sums to generate final outputs. For example, we align $Input_1[1,2,4,5]$ and $Input_2[1,2,4,5]$ in the first two memory rows in Figure 2.7(a), and compute partial results of $O_1[1]$ by convolution with $Filter_{1,1}$ and $Filter_{2,1}$ respectively. We should reorganize partial results of each output element in the memory to compute the output element by additions. We can take $log_2 N$ steps to complete such reduction, by moving and adding half of partial sums at each step. Reduction in each memory block can happen in parallel because a typical memory block (e.g., 1K rows) can fit all convolution data for an output element which requires N rows where N is less than 1024 for most current CNNs.

Such output-parallel layout achieves high parallelism while it may require too many memory blocks to fit $N * M * O$ rows. To solve such issues, we can adopt more sequential operations for the operation layout to reduce the number of rows by combining multiple convolutions in the same row. In this case, we can change the inner-loops for operation layout to fold computations inside each

channel. Furthermore, we can break the channel-level loops into more levels and fold computations across channels. Such computation folding comes at the cost of using more bit-lines and sacrificing parallelism.

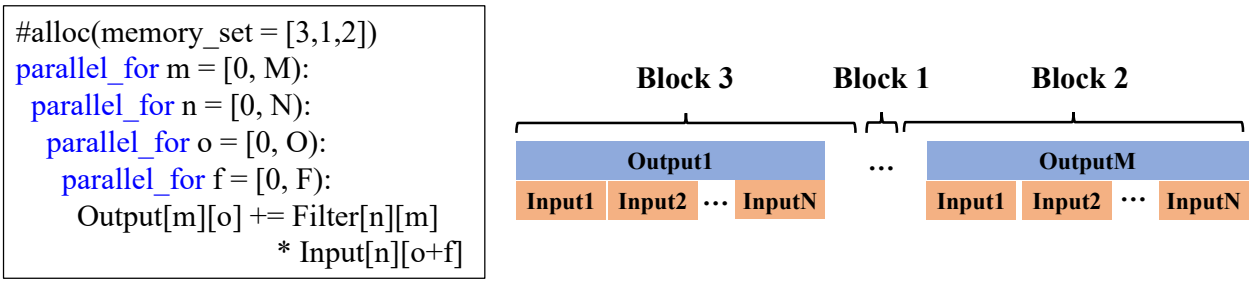
Input-parallel Data Layout

Another way to parallelize a convolution layer is to schedule computations of different input channels in parallel. DPIM architecture can implement this strategy by aligning the computations in different input channels for a specific output element, as shown in Figure 2.7(b). In the input-parallel layout, computations of each input channel generate partial results that need to be summed up with partial results in other input channels to calculate the output results. Since partial results for a specific output channel distribute vertically, we cannot accumulate them directly by PIM operations, requiring data movements to realign them (right part of Figure 2.7(b)).

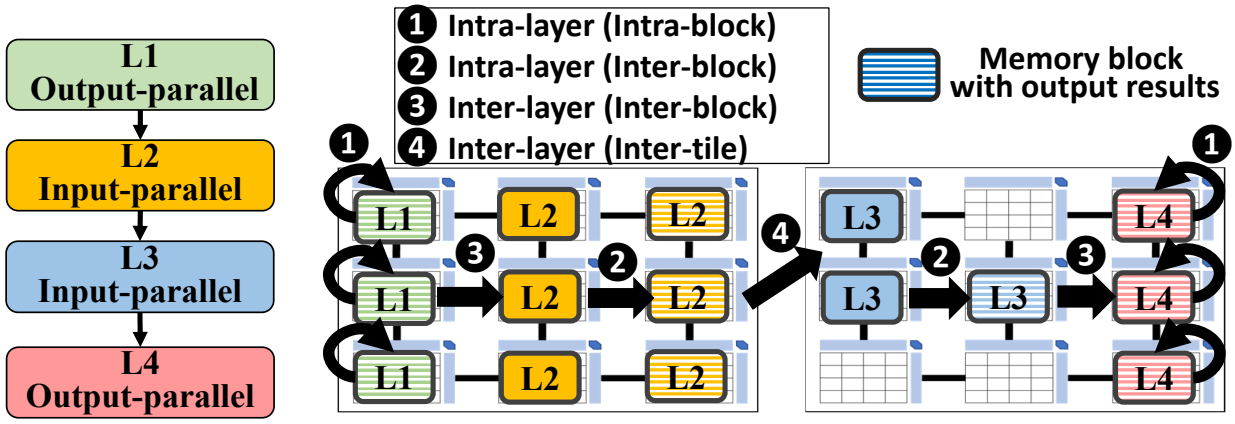
The input-parallel layout requires less number of word-lines (rows) than the output-parallel layout - $N * O$ as compared to $N * M * O$. However, it requires a large number of bit-lines, which is equal to $M * F * b$, to fit all M filters for an input channel to compute partial results in b-bit precision. For example, in a common 3*3 convolution with 8-bit fixed point values, each filter requires 72 bit-lines in the memory; this means a memory block with 1024 bit-lines can only fit 13 filters. If the number of output channels is larger than 13, we have to distribute all filters for an input channel across multiple blocks, causing extra inter-block data movements during the sequential execution. A trade-off we can adopt to improve the parallelism is storing a copy of input in each block so that all blocks can process convolutions and reductions in parallel. Similar to output-parallel layout, we can fine-tune the input parallel layout by changing the loop structure.

Other Design Dimensions

The basic difference between output-parallel and input-parallel layout is the order of loops for input and output channels. As introduced before, we can break these two loops into more loop levels to change the data layout in DPIM. Two examples shown in Figure 2.7 assume loops of



(a) Resource allocation for 1 layer with *memory_set*.



(b) Global layout for 4 layers with different layer layout strategies.

Figure 2.8: Memory allocation and global layout.

operation mapping always in the inner loops. With the conventional framework, it is possible to explore more mappings by shuffling the order of these four loops which can change the order of partial sums aligned in the memory. However, the cost model of layout layer depends on the relative order between input-channel and output-channel loops which determines the layout for partial sums. Therefore, we can customize any layer layout based on the two basic mappings introduced in this section.

2.3.3 Memory Allocation

After determining the detailed layout for one layer, the next step is to allocate memory resource (e.g., memory blocks) for the layer. Considering the DPIM system usually contains a large number

of memory blocks, the memory allocation for a layer also adds a dimension in the design space which indicates the memory blocks we want to allocate for a layer with a given layout. Therefore, we add an *alloc* configuration in the DPIM mapping framework which gets in an ordered set of memory blocks (*memory_set*), as shown in Figure 2.8(a). With the detailed layer layout, we evenly divide all parallel computations to the ordered memory blocks. The figure shows an example of allocating three blocks in the order of (3,1,2) to a layer using output-parallel layout. Based on the mapping shown in the left side of Figure 2.8(a), the layout parallelizes computations of all output channels, where each output channel parallelizes computations using all input channels as shown in the right side of Figure 2.8(a). Inside each output-input computation, the detailed operation layout is defined by the two inner loops in the mapping. Then, the memory allocation for set (3,1,2) evenly divides these computations into three segments, and allocate the first, the second, and the third segment to block 3, 1, 2 respectively.

In static DPIM DNN accelerators, we need to distribute layers across the global memory, introducing the problem of global layout. The global layout can be represented by allocating different sets of memory to all layers. The number of required blocks for each layout depends on its layout strategy (including both operation and layer layout). The global layout introduces various data movements. The first type of data movement is intra-layer, which happens during the layer computation. Both channel-parallel data layouts would cause intra-layer data movements when accumulating partial results. As analyzed in Section 2.3.2, output-parallel layout accumulates partial sums in the same memory block since it aligns all input data for a specific output element in consecutive rows; and input-parallel layout reserves a specific set of blocks to reduce all partial results distributed across different blocks. As shown in Figure 2.8, intra-layer movements of output-parallel layers (L1 and L4) happen inside each block, while those of input-parallel layers (L2 and L3) use the inter-block interconnect. Other than intra-layer data movements, data dependency also happens between different layers. Such dependency results in inter-layer data movements which may use inter-block interconnect or inter-tile interconnect.

To ensure the generality of memory allocation, we can define any memory set with blocks that

Algorithm 1 Tile-level optimization

$Layers[N]$ $Layout[N][N]$ FnFunctionend optimizeLayerGroup(group: layer[n], nBlk: int) $f[:N][:N] = inf$; $f[0,:N] = 0$ $decision[:N,:N] = null$

for $i \leftarrow 1$ n **do** layout t for $group[i]$ $rb = MemoryCost(i,t)$

for $j \leftarrow req\_blk$ $nBlk$ **do**

if $f[i][j] > f[i-1][j-rb] + PerfCost(i,t)$ **then** $f[i][j] = f[i-1][j-rb] + PerfCost(i,t)$
$decision[i][j] = t$ /*Generate the optimized layout for each layer group based on *decision*/

have not been allocated for other layers. However, allocating memory blocks that have long distance with each other (e.g., in different memory tiles) to a layer may introduce large data movement overhead if the layer layout requires inter-block data transfer for some operations (e.g., reduction). Furthermore, allocating long distance memory blocks to layers with data-dependency will also cause large data movement overhead. Since different data movement patterns take various latency (e.g., inter-tile movements are usually slower than inter-block movements), it is important to carefully design the global layout based on both the DNN structure and the architecture configuration. In Section 2.4, we propose a holistic data layout optimization for general DPIM DNN acceleration.

2.4 PIM-DL Optimization

The PIM-DL framework shows DPIM DNN acceleration has a large design space which is usually too large to be exhaustively searched for an optimal data layout. In this section, we introduce an optimization algorithm which holistically optimizes data layout for general DPIM DNN acceleration. PIM-DL optimization includes three steps, tile-level optimization, global optimization, and block allocation, to efficiently find an efficient data layout.

2.4.1 Tile-level Optimization

The goal of the tile-level optimization is to find the optimized data layout for allocating a `layer group` in a tile, where a `layer group` denotes a group of DNN layers allocated to the same tile. However, the number of possible combinations of DNN layers is too large to be efficiently

explored. Therefore, we limit the exploration of single-tile data layout to consecutive DNN layers in the network based on the fact that the bandwidth of intra-tile data bus is much higher than the inter-tile interconnect, making it more efficient to process consecutive layers in a tile. The tile-level optimization searches for the best layout strategy for each set of consecutive layers that fit them in the same tile.

Algorithm 1 outlines the steps of optimizing a layer group with consecutive DNN layers, which is based on dynamic programming [54]. Each state $f[i][j]$ stores the minimum cost of allocating the first i DNN layers using j memory blocks. To calculate each $f[i][j]$, the algorithm considers all possible data layout strategies for each layer by exploring the PIM-DL data layout framework introduced in Section 4.2. For each layout strategy t, we can calculate the number of required memory blocks (rb). We can estimate the cost for adopting layout t for layer i by adding $f[i-1][j-rb]$ and the estimated performance cost of the layout, which includes data loading, computation, intra-layer data movement, and inter-layer data movement.

The data loading cost and the computation cost can be accurately calculated for each layer with a specific layout. These costs mainly change as a function of data dimensions of the DNN layer. For the intra-layer data movement, we cannot directly calculate it because we do not know which memory blocks we allocate to each layer at this point. We approximate this cost by assuming the system has a uniform data bus (e.g., shared bus or fully-connected interconnect) inside each tile so that the intra-layer data movement cost is a function as the size of moved data. In Section 2.4.3, we introduce a way to reduce the estimation error for intra-layer data movement cost by a block allocation method which uses a genetic algorithm to find an efficient block allocation for a specific layer which has a similar intra-layer data movement cost as our ideal assumption. The inter-layer data movement cost is calculated based on the average bandwidth of inter-block network if the input data comes from a layer in the same layer group; otherwise, it is calculated based on the average bandwidth of inter-tile network. Our cost estimation is based on hardware simulation, which is introduced in Section 2.5.

The dynamic programming algorithm runs with all layer groups of consecutive DNN layers

Algorithm 2 Global optimization

$Layout[N][N], tiles[nT]$ $TileLayout[nT]$ $f[0:nT][0:N] = inf$; $f[0][0] = 0$ $decision[0:nT][0:N] = null$

for $t \leftarrow 1$ nT **do**

  for $i \leftarrow 1$ N **do**

    for $j \leftarrow 0$ $i-1$ **do**

      if $f[t][i] > f[t-1][j] + Cost(Layout[j+1][i])$ **then** $f[t][i] = f[t-1][j] + Cost(Layout[j+1][i])$ $decision[t][i] = j$ /*Generate tile allocation for all layers based on $decision$*/

($N(N+1)/2$ in total) and generates the optimized layout strategy for each possible layer group. The exploration results are used by the global optimization algorithm which finds the best allocation for the whole network in the multi-tile architecture.

2.4.2 Global Optimization

The tile-level optimization finds the optimized layout for a specific layer group in a single tile, which can give the optimal layout if the whole DNN can fit in a tile. However, current DNNs for real-world applications are usually large and deep, which may require multiple tiles in static DPIM accelerators. Therefore, we propose a cost-aware algorithm for dividing a large DNN into multiple tiles while minimizing the overall cost considering inter-tile data movements. Algorithm 2 shows the process of global optimization which is also based on dynamic programming. Each state $f[t][i]$ denotes the minimum cost of allocating layer $0-i$ in t tiles. The algorithm calculates the minimum cost from a single tile to nT tiles. For each tile t, the algorithm iterates over all layers in a topologically sorted order to calculate $f[t][i]$. For each $f[t][i]$, the algorithm checks all possible continuous layer group $j-i$ and updates $f[t][i]$ if $f[t-1][j-1] + cost(j,i)$ is less than $f[t][i]$. The $cost(j,i)$ is the minimized cost of allocating layer j to i in a tile, which has been calculated in the tile-level optimization phase. We record layout decisions during the execution, and generate the best layout we found by backtracking from $f[nT][N]$.

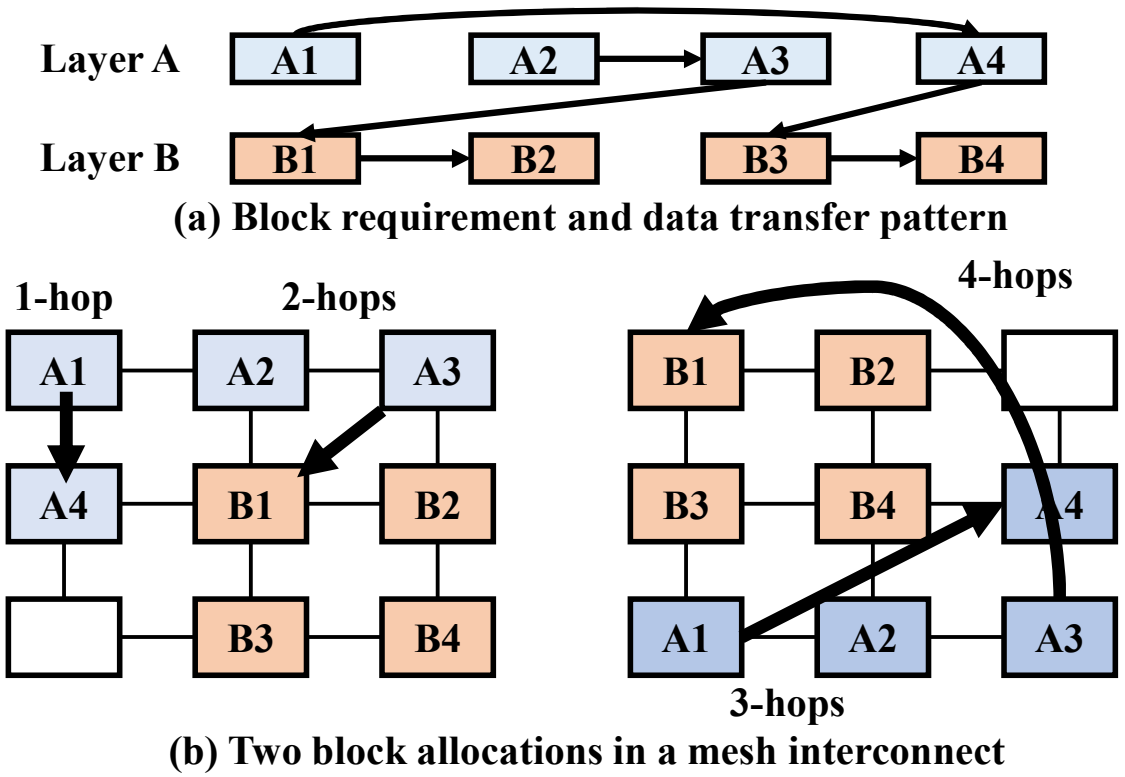


Figure 2.9: Block allocations in a mesh interconnect.

2.4.3 Block Allocation

After global layout optimization, we generate data layout strategy and tile allocation for all DNN layers. However, the previous two steps do not consider locations of blocks allocated to each layer (*memory_set*). For an architecture with a uniform-latency data transfer network in each tile (e.g., shared bus), the block allocation would not impact the overall performance since all inter-block data transfers have the same latency. However, DNN accelerators usually have customized data links with non-uniform latency. For example, FloatPIM [3] adopts a latch-based linear data link which can efficiently handle data movements between consecutive DNN layers. In architectures that use non-uniform data links, different block allocations may exhibit very different performance results. Figure 2.9 shows two block allocation schemes for two connected layers, each of which requires 4 blocks based on the layout. We assume a simple mesh network for inter-block connection. The figure shows the data transfer from A1 to A4 can be directly handled in the mesh interconnect for

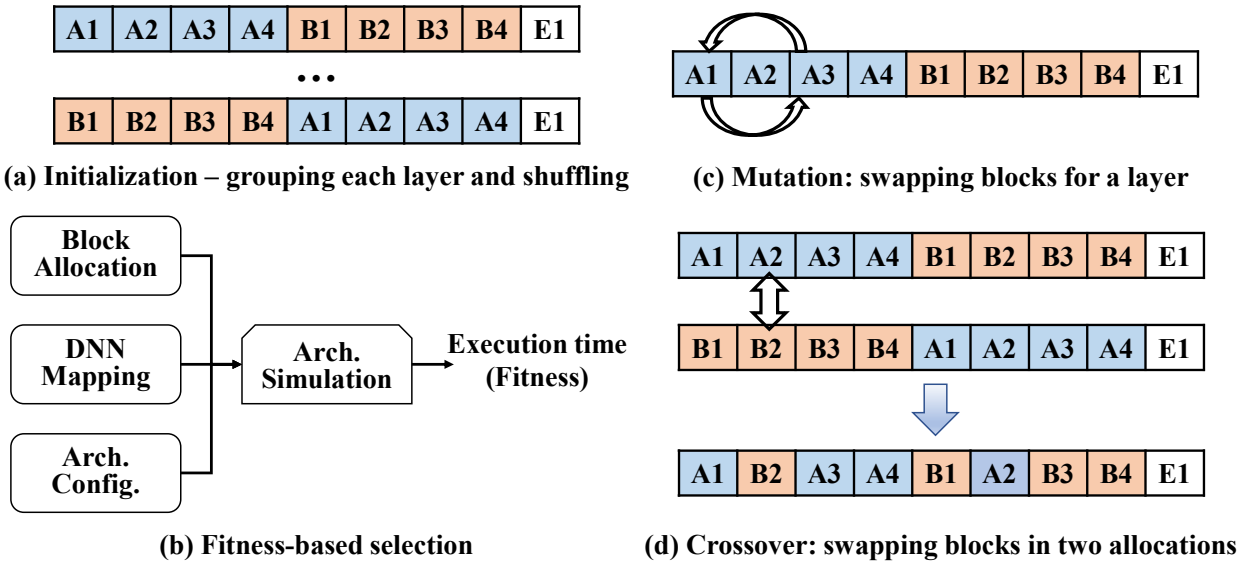


Figure 2.10: The genetic algorithm for block allocation.

the first allocation, while requiring 3 hops in the second one.

For a given interconnect structure, the only way to find the optimal block allocation for a layer group is exhaustive search which is not computational tractable since a tile usually contains hundreds of memory blocks. In order to generate efficient block allocation for general DPIM architectures, we use a heuristic search based on genetic algorithm [55], as shown in Figure 2.10. Specifically, we first encode all blocks allocated to each layer and use a vector as the genetic representation. If a tile is not fully used, we also encode empty blocks (e.g., E1 in Figure 2.10) to keep the length of vector the same as the number of blocks in a tile. During the initialization (Figure 2.10(a)), we first generate allocations that group blocks for each layer together and shuffle the order between different layers. For each generation, we define the fitness as the execution time based on hardware simulation (Figure 2.10(b)). Specifically, our simulator estimates the execution time by taking in the block allocation, the layout strategy, and the hardware configuration. For each generation, the mutation operation randomly swaps two blocks in the same layer and the crossover operation swaps the same position in two allocations (Figure 2.10(c)). We run the genetic algorithm for 3000 generations to find the near-optimal block allocation within reasonable time.

Table 2.1: Hardware Parameters for ReRAM Device.

k_{on}	k_{off}	$\alpha_{on}, \alpha_{off}$	$V_{T,ON}$	$V_{T,OFF}$	x_{on}
$-216.2m/s$	$0.091m/s$	4	$-1.5V$	$0.3V$	0
x_{off}	R_{ON}	R_{OFF}	E_{set}	E_{reset}	E_{NOR}
$3nm$	$10K\Omega$	$10M\Omega$	$23.8fJ$	$0.32fJ$	$0.29fJ$
E_{search}	T_{NOR}	T_{search}	T_{Switch}	V_{RESET}	V_{SET}
$5.34pJ$	$1.1ns$	$1.5ns$	$1ns$	$1V$	$2V$

2.5 Methodology

Compiler Implementation. We implement PIM-DL in Glow, an open-source machine learning compiler for heterogeneous architectures [56]. We instrument the graph lowering engine of Glow for the data layout optimization. We add a new back-end of DPIM in Glow, which generates DPIM operation trace based on the optimized data layout. We then use an in-house cycle-accurate simulator for evaluation.

DPIM Simulation. Our simulation adopts a two-step method which has been widely used in several previous works on emerging architectures [19, 42, 57]. We first model timing and energy parameters for operations on different hardware components using validated circuit-level simulators; and then use these numbers in the simulator to estimate the performance of different architecture configurations. We investigate three widely used memory technologies for DPIM acceleration including ReRAM [3], DRAM [13], and SRAM [4, 17].

The basic ReRAM technology used in this work is the Voltage ThrEshold Adaptive Memristor (VTEAM) model [58] with I_{ON}/I_{OFF} ratio of 10^3. The detailed parameters, including both energy and timing, of the VTEAM model is listed in Table 2.1. We use HSPICE design tool for circuit level simulations to provide timing and energy results for ReRAM operations. The DRAM specification used in this work is extracted from a published datasheet from the industry [59]. We model all CMOS components (including buffers and interconnects) in Cacti [60] at 32nm technology. The interconnect is modeled by Orion 3.0 [61] in 45nm technology, and we scale the results to 32nm technology.

Hardware Configurations. Our experiments cover a wide range of hardware configurations,

Table 2.2: Architectural parameters.

Memory Block		
Organization	#bit-lines (columns)	1024
	#word-lines (rows)	1024
Tile		
Block array	#blocks	256
	Dimension	16*16 by default
DPIM System		
Organization	#tiles	32
	Dimension	1*32 linked by chain
Serial links (Inter-tile)	Bandwidth	160GB/s
	Latency	8-cycle

Table 2.3: Tested DNN models.

Network	**Depth**	**Width**	**#MACs**	**#Para.**
AlexNet [33]	7	1	7.27G	60.97M
DenseNet [36]	120	1	4.87G	25.56M
GoogleNet [35]	21	4	16.04G	7M
InceptionV2 [63]	33	4	12.27G	72.56M
MobileNet [64]	53	1	573.78M	4.23M
ResNet50 [34]	49	2	3.87G	46.72M
Vgg19 [65]	18	1	196.32G	314.12M
SqueezeNet [66]	17	2	861.34M	12.58M

which will be detailed in corresponding sections. However, the high-level organization of tile-based architecture is shown in Table 2.2, which has 32 tiles, and each tile has 256 memory blocks. Each memory block contains 1024 bit-lines and 1024 word-lines, providing a total size of 8Gb. We investigate three different memory technologies in the baseline architecture model, and show the cross-technology results in Section 2.6.3. We model the on-chip NoC (inter-block interconnect) with 128-bit channels and assume 3 cycles for router and 1 cycle for wire as the zero-load delay [53]. We simulate different structures for inter-block NoC and show the performance comparison in Section 2.6.1. The inter-tile network is modeled as SerDes link used by HMC with an average 160GB/s bandwidth [62].

DNN Workloads. We use 8 popular DNN models in our experiments, as shown in Table 2.3,

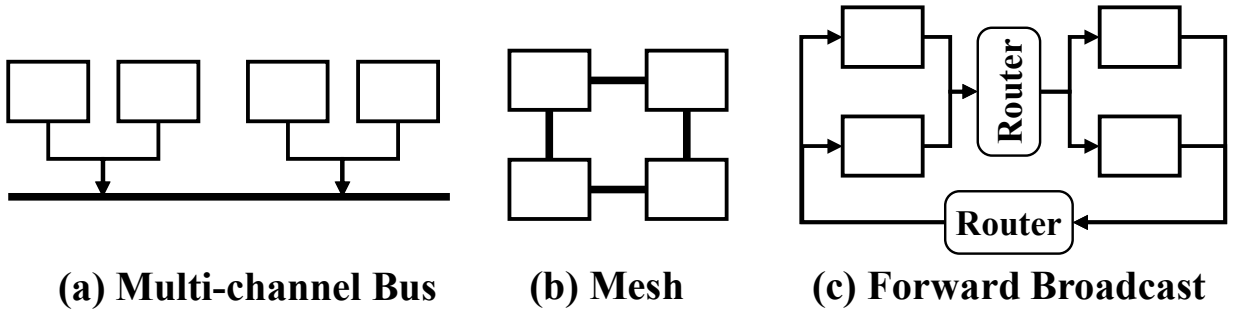


Figure 2.11: Interconnect structures used in our experiments.

including AlexNet [33], DenseNet [36], GoogleNet [35], InceptionV2 [63], MobileNet [64], ResNet50 [34], Vgg19 [65], and SqueezeNet [66]. We test all models on the inference task for ImageNet dataset [67].

2.6 Experiments

2.6.1 Data Layout Optimization

To verify the efficiency of PIM-DL, we compare it to several heuristic-based methods which adopt a fixed strategy for all DNN layers. Furthermore, we conduct such experiment on various architecture configurations to justify the generality of PIM-DL.

Software and Hardware Baselines

All baseline layout strategies are fully parallel for either input-level or output-level. Such strategies are commonly used in previous DPIM DNN accelerators including FloatPIM [3], Drisa [13], NeuralCache [4]. We denote input-parallel and output-parallel schemes as `In` and `Out` in all figures. For each channel parallel method, we show results of three fine-tuned mappings by selecting different degrees of parallelism, denoted as `Max`, `Mid`, and `Min`. We adopt a sequential block allocation for all baselines, where blocks allocated to each layer are placed sequentially in the memory.

All baseline architectures are ReRAM-based static accelerators with different interconnect networks as shown in Figure 2.11. We test 1 uniform interconnect, `Bus`, indicating a global bus shared by all blocks in a tile. Each 8 blocks in a tile share a channel, giving a 512 GB/s total bandwidth which is similar to a HMC chip with 32 vaults [62].

We also test 2 non-uniform interconnects: `Mesh` and `Broadcast` as shown in Figure 2.11. Specifically, `Mesh` is a widely used interconnect in domain-specific accelerators [68, 69]. `Broadcast` is a modified version of interconnect used in FloatPIM [3]. The original interconnect organizes all blocks in a chain to fit the sequential data transfer pattern of DNN workloads. However, FloatPIM assumes each block can process one layer which is different from the generic DPIM acceleration as analyzed in Section 4.2. In our experiments, a DNN layer usually requires multiple blocks, leading to a broadcast data transfer pattern. The `Broadcast` network organizes blocks in different column and support single-direction 1-to-N data transfer between two columns. We group 16 blocks in a column so that each tile has 16 block columns.

For the inter-tile interconnect, all baselines assume a Mesh-like network in these experiments because our experiments show that most widely used inter-tile connection networks, including Mesh, Chain, and Bus, give similar results because data transfer pattern between tiles is simple.

Comparison on Uniform Interconnect

To evaluate the efficiency of different parts in our optimization, we show the performance and memory usage on `Bus` (Figure 2.12) to exclude the effect of block allocation optimization. As shown in the results, the average speedups provided by the data layout optimization is 18.8% as compared to the output-parallel layout used in state-of-the-arts (`Out-Max`). Across different tested DNNs, the data layout found by our optimization framework can improve the performance by 41.0% and 13.6% as compared to the worst and the best heuristic-based methods on average, respectively. The results show that a single heuristic-based data layout cannot provide the optimal performance for all scenarios. Furthermore, the optimization decreases 30.5% memory usage of `Out-Max`. The results show that PIM-DL can always provide the best performance, while heuristics without adaptive layout

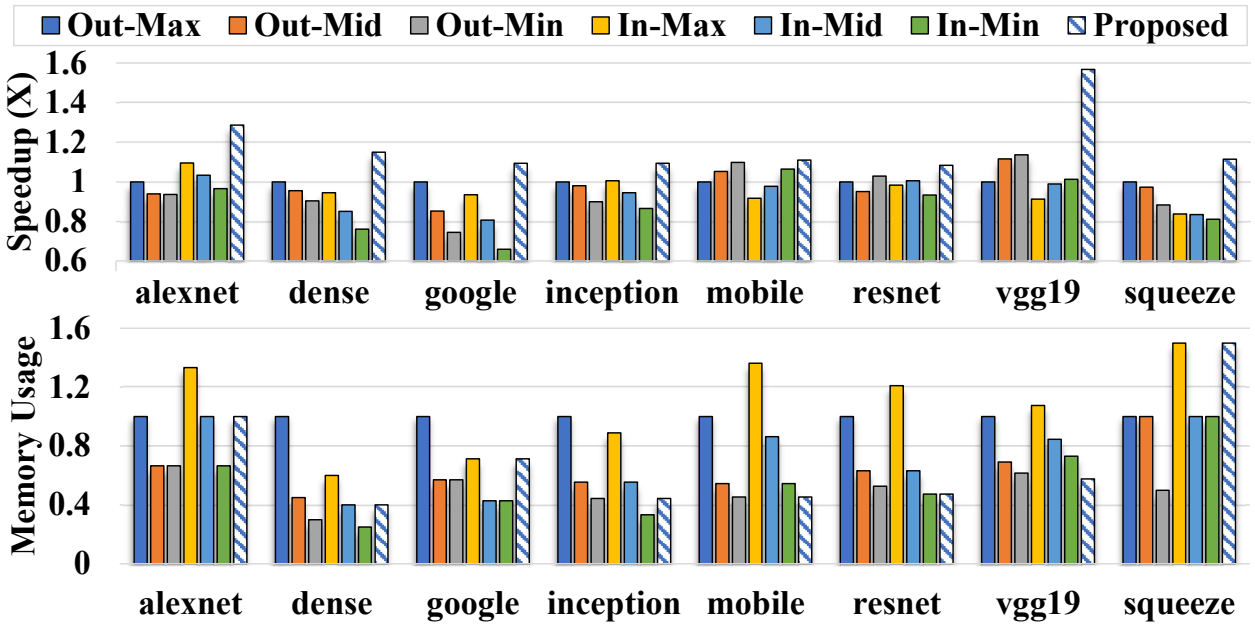


Figure 2.12: Performance and memory usage results of heuristic-based layout schemes and PIM-DL with a `Bus` interconnect for blocks in a tile. The results are normalized to `Out-Max` layout.

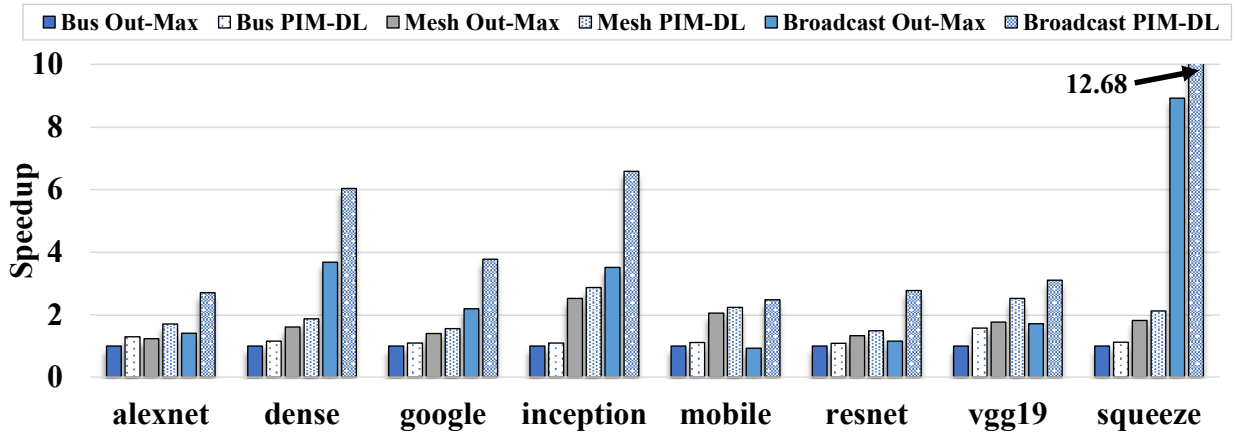


Figure 2.13: The performance of `Out-Max` layout and the optimized layout across interconnect structures.

strategies lead to sub-optimal performance and memory utilization.

Comparison on Non-Uniform Structures

We then compare the data layout optimization with heuristic-based methods on different interconnect structures. Figure 2.13 shows the performance results of `Out-Max` and the optimized layout

found by our framework on three interconnect structures. All results are normalized to *Bus Out-Max*.

We first investigate the performance improvement provided by data layout optimization on non-uniform interconnects. As compared to the fully output-parallel layout, data layout optimization can provide 1.20× and 1.94× speedup on `Mesh` and `Broadcast` respectively. Such results indicate that PIM-DL can improve the performance of DPIM DNN acceleration across a wide range of architectures. Furthermore, PIM-DL provides more speedup on customized architectures.

Such experiment results also show a significant benefit provided by interconnect customization. With the data layout optimization, `Broadcast` is 1.61× faster than `Mesh`. Furthermore, `Broadcast` interconnect requires 81.1% less area as compared to `Mesh` interconnect. Such improvements on area efficiency come from significantly less routers, even though each router takes larger area because of large multiplexer.

2.6.2 Applicability to other DPIM Accelerators

We apply PIM-DL to previous DPIM accelerators by modeling costs of different operations based on the specific architecture design. We first utilize PIM-DL on FloatPIM [3] as an example. Figure2.14(a) shows the percentage of layers using different data layout schemes decided by the optimization. The result shows that over 95% of layers can improve the performance by using fine-tuned strategies. We should note that only 1% layers keep using the original scheme of FloatPIM (fully output-parallel). Figure 2.14(b) shows the normalized time breakdown of the original FloatPIM and our optimization. The result shows that the optimization can significantly reduce the data movement overhead, leading to a 2.5× speedup over the original FloatPIM.

Even though PIM-DL is mainly designed for DPIM DNN accelerators using static computing model, we can still achieve performance improvement by data layout optimization in dynamic DPIM accelerators like NeuralCache [4]. Since the dynamic DNN acceleration exploits the whole memory for each layer, we can still explore different mappings to find the most efficient layout for each layer. Figure 2.15 shows the performance and energy efficiency improvements provided by our exploration, which are 2.6× and 1.5× respectively.

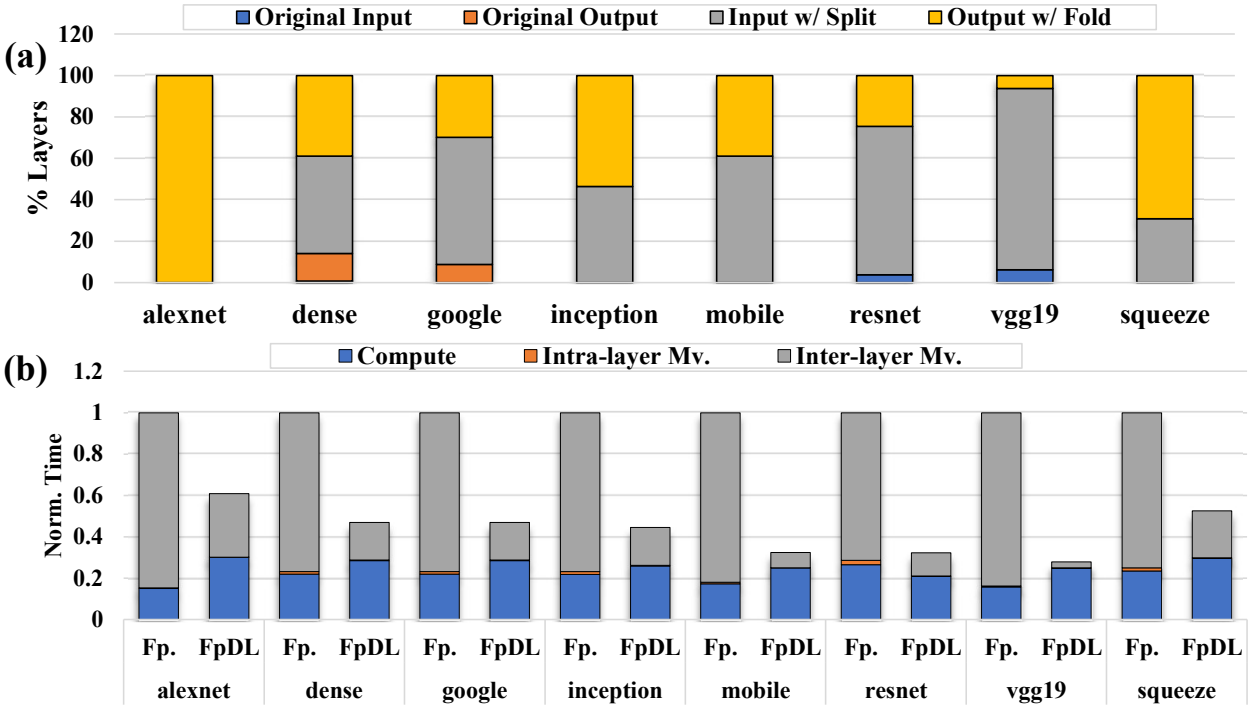


Figure 2.14: (a) The percentage of layers using different layout schemes on FloatPIM [3] with our optimization; (b) Normalized performance breakdown of original FloatPIM (Fp) and FloatPIM with PIM-DL.

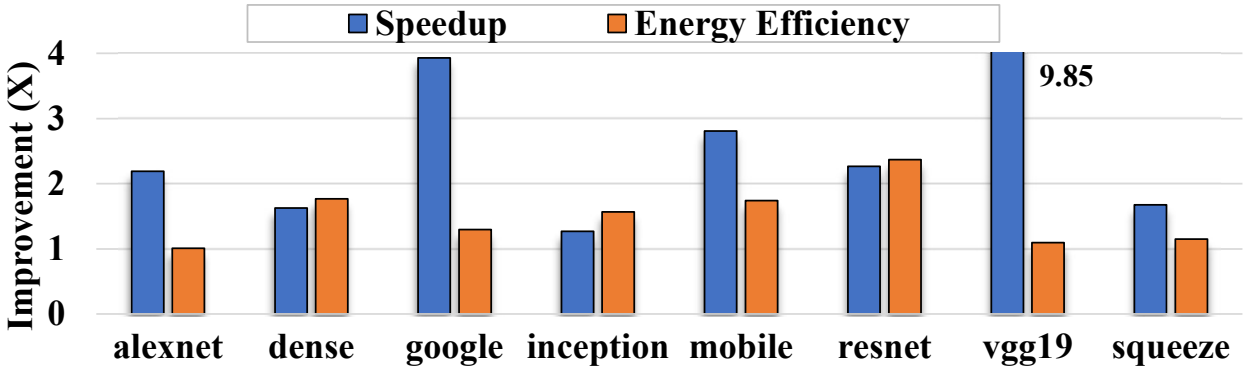


Figure 2.15: Performance and energy improvements on NeuralCache [4].

Figure 2.16 shows detailed layout strategies for all layers in InceptionV2 determined by our optimization. Similar to previous experiments, `In` and `Out` indicate input-parallel and output-parallel layouts respectively. The number from 1 to 5 denotes different fine-tuned layouts based on input-parallel and output-parallel. For example, `OUT-1` is the fully output-parallel layout which is used by NeuralCache [4] for all layers. The result shows none of layers adopts the original strategy. These

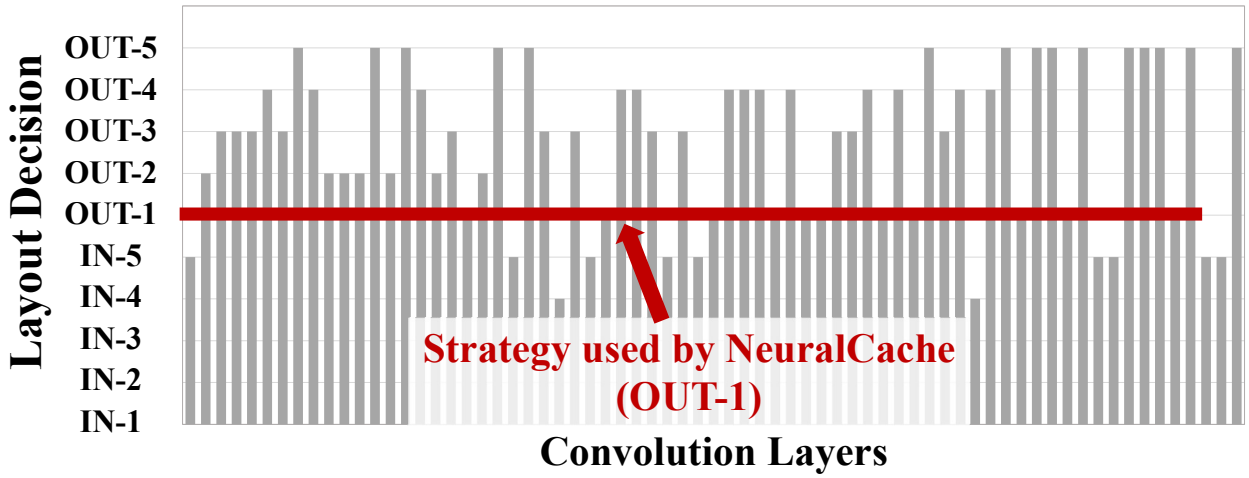


Figure 2.16: PIM-DL data layout of InceptionV2 on NeuralCache [4].

results indicate that PIM-DL is applicable to dynamic DPIM DNN inference.

2.6.3 Data Layout Aware HW/SW Co-Design

Experiments in Section 2.6.1 indicate the software-hardware co-design with optimized data layouts is promising to improve the performance of state-of-the-arts. In this section, we utilize the `Broadcast` interconnect to build customized accelerators based on three widely used DPIM memory technologies: ReRAM, DRAM, and SRAM. We adopt PIM-DL in these customized accelerators and compare the performance with several state-of-the-art accelerators as shown in Table 2.4. For fair comparisons, we use the same memory size for each proposed architecture (SysR, SysD, SysS) as the state-of-the-art using the same memory technology. For static accelerators that cannot fit all layers in the memory (e.g., SysD and SysR), we use a hybrid static/dynamic acceleration and PIM-DL supports this hybrid mode by removing the cross-tile cost in the global optimization. All results are divided by the area of the corresponding systems because of the diversity in system size. All systems run DNN inference workloads with 8-bit fixed-point values. Figure 2.17 shows the average results across tested DNNs and all values are normalized to SysR.

As compared to the previous accelerator using the same memory technology, each of our customized architectures can provide improvements in both performance and energy efficiency.

Table 2.4: DPIM systems for comparison.

Systems	Technology	Mode	Interconnect	Configuration	Area (mm^2)
SysR	ReRAM	Static	Broadcast	32 tiles - 8Gb/tile	40.9
SysD	DRAM	Dynamic	Broadcast	8Gb	31.0
SysS	SRAM	Dynamic	Broadcast	35MB	210.3
FloatPIM [3]	ReRAM	Static	Chain	32 tiles - 8Gb/tile	30.6
Drisa [13]	DRAM	Dynamic	Bus	8Gb	28.5
NeuralCache [4]	SRAM	Dynamic	Bus	35MB	189.8

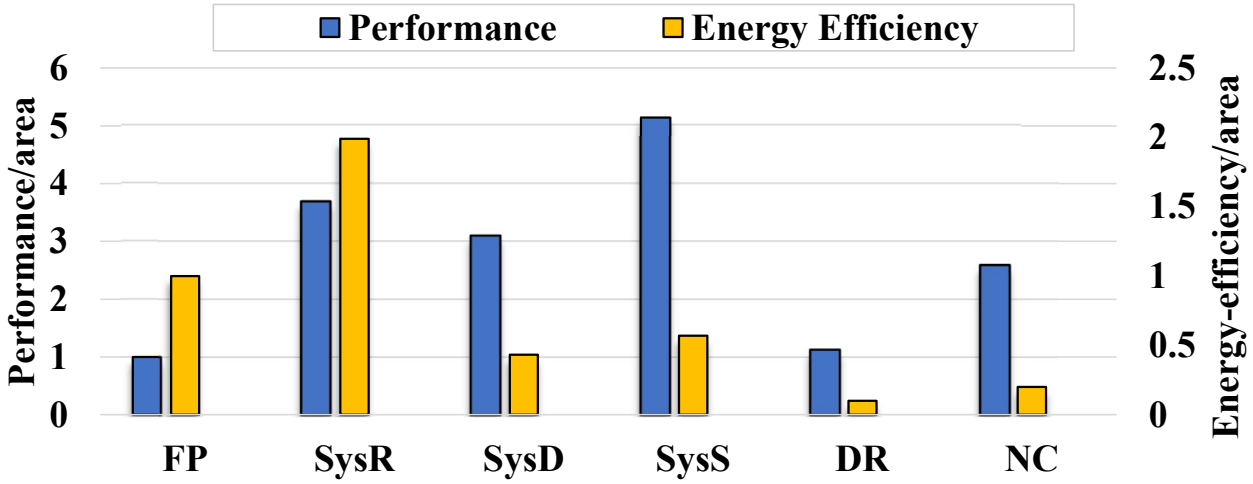


Figure 2.17: Performance and energy results of different systems (averaging across all DNNs). All results are normalized to FP and higher values are better.

Specifically, SysR is 3.7× faster and 2.0× more energy-efficient than FloatPIM [3]; SysD is 2.7× faster and 4.3× more energy-efficient than Drisa [13]; SysS is 2.0× faster and 2.8× more energy-efficient than NeuralCache [4]. These results show that the combination of data layout optimization and interconnect customization can significantly improve both performance and energy-efficiency for different technologies and acceleration modes.

We further compare the results across different memory technologies used in our customized architectures. The normalized performance/area improvements of SysR, SysD and SysS are 3.7, 3.1, and 5.14 respectively; the normalized improvements of energy-efficiency of SysR, SysD and SysS are 2.0, 0.4, and 0.6 respectively. Based on such results, the DRAM-based system (SysD) has the worst performance and energy efficiency because of the low density and the large overhead of PIM-enabled circuit [13]. SysS provides the best performance result, which is 1.4× faster than SysR,

but it consumes 3.3× more energy/area. Such results indicate Non-volatile memories, like ReRAM, would be more efficient than conventional memory DRAM and SRAM technologies because of its high density and energy efficiency. However, we should note that NVM-based accelerators can only support static acceleration mode because it would be too expensive to frequently load weights through time-and-power-consuming write operations.

2.7 Related Work

Memory-centric DNN Accelerators. There are mainly three categories of memory-centric technologies - near-data computing (NDC) [53, 70], APIM [19, 26, 42, 71], and DPIM [3, 4, 13], which have been extensively explored to accelerate DNN applications. For example, Tetris [53] proposes a scheduling and partition algorithm for a NDC-based DNN accelerator to efficiently map the row-stationary DNN dataflow [57] in 3D stacked memory [62] with maximum data reuse. PUMA [72] is a data-flow accelerator which allocates MVM operations in DNNs on a spatial APIM architecture based on compiler optimizations. The approach proposed by Ji et al. [73] maps NN applications into APIM NN chips. However, the architectures targeted in these work are similar to conventional hierarchical spatial accelerators. As illustrated in Section 4.2, the data layout problem in DPIM cannot be fully represented by such mapping convention.

Data-traffic Optimization for DNN. Because of the large data and model size in modern DNNs, the data traffic has become one of the major bottlenecks in various systems [74–78]. HyPar [74] proposes a hierarchical dynamic programming method to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. The cost models of HyPar are based on partitions of different tensors, which are mapped to the accelerator array. AccPar [79] further supports mapping on heterogeneous accelerators. Tofu [75] automatically partitions DNN models across multiple GPU devices to reduce per-GPU memory footprint as well as the total communication cost. MEDNN [78] optimizes the distribution of DNNs on multiple mobile devices. All these works focus on operation-level partitioning across multiple general-purpose processing units, without

further considering data layout. They are orthogonal to the DPIM architectures which require more sophisticated data layout strategies.

2.8 Conclusion

Chapter 2 comprehensively investigates the data layout issues in DPIM DNN acceleration by formulating the conventional DNN mapping as the data layout problem in PIM with detailed cost models. Our efficient optimization algorithm generates good DPIM data layout for general DNN workloads. We conduct several experiments to evaluate the efficiency of proposed data layout optimization and show that PIM-DL provides 3.7× speedup and 4.3× better energy efficiency on a wide range of DPIM DNN accelerators as compared to existing layout strategies.

While our proposed method provides a generic framework to optimize the high-level operators (e.g., convolution, matrix multiplication, etc.) in PIM accelerators, it still does not address all the optimization opportunities of the mapping dimension that consider multiple high-level operators together. The next chapter, Chapter 3, evaluates PIM acceleration of Transformer-based models, and finds that operator-level mapping suffers from several inefficiencies caused by long-latency data movements and insufficient hardware support. Chapter 3 proposes a novel software-hardware co-design for Transformers, including a low data movement processing flow and a lightweight hardware modification in PIM, to tackle these challenges.

Chapter 2, in full, is a reprint of the material as it appears in International Conference on Parallel Architectures and Compilation Techniques, 2021, Minxuan Zhou, Guoyang Chen, Mohsen Imani, Saransh Gupta, Weifeng Zhang, and Tajana Rosing. The dissertation author was the primary author of this paper.

Chapter 3

PIM Acceleration for Transformer

The previous chapter discusses a data layout framework that optimizes software mapping on PIM architectures by exploring the design space of individual operators (i.e., DNN layers) and partially optimizing data transfer between different operators. However, when the cross-layer data movement becomes much more significant, the effect of such an optimization method is bound by the expensive data transfer. Previous chapter only adopts the hardware optimization for intra-memory interconnect networks, not considering the PIM architecture itself for more efficient operations. This chapter investigates an important and emerging application field, Transformer-based models, whose performance is impeded by layer-based mapping as well as insufficient hardware support in PIM architecture. To tackle these challenges, this chapter introduces a novel software-hardware co-design method to design an efficient PIM acceleration for Transformer-based models. This section proposes a novel method of mapping Transformer models onto PIM architecture that minimizes the data movements overhead. Furthermore, this section introduces lightweight hardware customizations in a conventional DRAM product that significantly improves the efficiency of various PIM operations.

3.1 Introduction

Attention mechanism has emerged as a powerful tool to model long-term dependencies in sequential data [80]. Attention-based models, such as Transformer [80] and its enhanced variants [29, 81], have dramatically improved the accuracy of important machine learning tasks, like natural language processing [82], computer vision [83, 84], and video analysis [85]. However, these benefits come at the cost of long execution time due to the large memory footprint and low computation-to-memory ratio. Existing CNN-oriented accelerators are designed for compute-intensive operations (e.g., convolution), making them sub-optimal for processing Transformers [86, 87]. To accelerate Transformer models, there have been several domain-specific accelerators, such as SpAtten [86] and A^3 [87], that offload either the key operation (i.e., self-attention) or the whole Transformer from conventional systems (e.g., GPU). However, these ASIC-based accelerators still suffer from constrained parallelism and limited off-chip memory bandwidth that bound the performance of acceleration.

Memory-based acceleration, including PIM and near-memory computing (NMC), is promising to accelerate Transformer models as it supports extensive parallelism, low data movement cost, and scalable memory bandwidth [4, 13, 14, 52, 68, 88–90]. Although there have been many memory-based neural network accelerators [3, 4, 8, 13, 91, 92], their dataflow and hardware are mainly optimized for compute-intensive CNNs, which may be incompatible with memory-intensive Transformers. Specifically, the dataflows of existing memory-based accelerators [3, 4, 13] are determined in a layer-level granularity, such that either utilize the whole memory to process one layer at a time [4, 13] or allocate mutually exclusive memory resources to different layers [3]. Such layer-based dataflows introduce large non-compute overhead in Transformers because of the large amount of input data and weights that need to be loaded or transferred between layers. On the hardware side, Transformer's complex operations (such as reduction and Softmax), which require both parallel computation and fine-grained intra-memory data reorganization, would be the performance bottleneck for memory-based accelerators. Our experiments (Section 3.2.3) show that the layer-based dataflows spend over

60% of execution time on data movements when accelerating a widely-used Transformer using an in-memory bit-serial accelerator on emerging high bandwidth memory (HBM). Furthermore, the reductions using bit-serial row-parallel PIM operations take around 30% of execution time, exhibiting a much lower compute throughput than other PIM arithmetic operations. The results show both software and hardware issues can significantly limit the efficiency of memory-based Transformer acceleration.

In this work, we propose TransPIM, a software-hardware co-design based on an emerging commodity memory, high bandwidth memory (HBM), that utilizes memory-based acceleration technology to accelerate Transformers. In the software, instead of allocating memory for different layers, TransPIM adopts a token-based dataflow that assigns memory resources for computations across different layers based on the input tokens. With the token-based dataflow, each memory bank processes and stores the intermediate results related to a specific set of tokens. By doing so, TransPIM can significantly reduce the amount of data loaded or transferred across layers because the increased locality of intermediate data from different layers improves data reuse rate. TransPIM only requires data movement for computing the cross-memory (i.e., between different sets of tokens) information which can be handled efficiently by exploiting the large internal memory bandwidth of HBM.

Even though the token-based dataflow significantly improves the throughput by reducing the data movement overhead, the software-level solution cannot resolve the inefficiency of complex operations in Transformer. Existing memory-based accelerators supports either PIM [15] or NMC [8, 12]. However, different key operations in Transformers do not share similar patterns that can be efficiently processed by a single type of memory-based technology. Furthermore, the original data path in HBM heavily relies on the shared bus. Therefore, the resource conflict on the shared bus for transferring data may become the bottleneck for applications requiring high internal bandwidth. To solve these issues, we propose an integrated set of hardware in HBM, including near-memory auxiliary computing units (ACUs) and an optimized data communication architecture. Specifically, the ACUs enable the memory to exploit the benefits of both PIM and NMC to efficiently accelerate

different operations. The optimized data communication architecture adds buffers and specialized links in the HBM hierarchy to offload a large number of data movements from the global shared data path, delivering significantly higher memory bandwidth utilization than the original HBM.

In summary, the contributions of this work include:

- Compared to existing accelerators [86, 87] dedicated to attention, TransPIM is the first end-to-end memory-based accelerator that speeds up the entire Transformer inference by exploiting the emerging memory-based acceleration.

- The proposed software-hardware co-design significantly outperforms existing platforms, including GPU, TPU, and ASIC-based accelerators. Specifically, TransPIM is 22.1 $\times$ to 114.9 $\times$ faster than GPUs on various widely-used Transformers. As compared to ASIC-based accelerators, TransPIM provides 2.0$\times$ higher throughput.

- We propose a token-based dataflow for general Transformer-based models to reduce unnecessary data loading by exploiting holistic data reuse. Our results show that the proposed dataflow is 4.6$\times$ faster than the previous method on various memory-based accelerator architectures.

- TransPIM introduces lightweight hardware components in the conventional HBM architecture to support efficient computing and memory operations for Transformers, without impacting the memory density. Our experiments show that TransPIM significantly improves the performance by 3.7$\times$ and 9.1$\times$ over PIM-only and NMC-only architectures.

3.2 Background and Motivation

This section first introduces the background for Transformer and PIM acceleration. Then, this section analyzes the issues in existing memory-based technologies with accelerating Transformer-based models, which motivate a new software-hardware co-design.

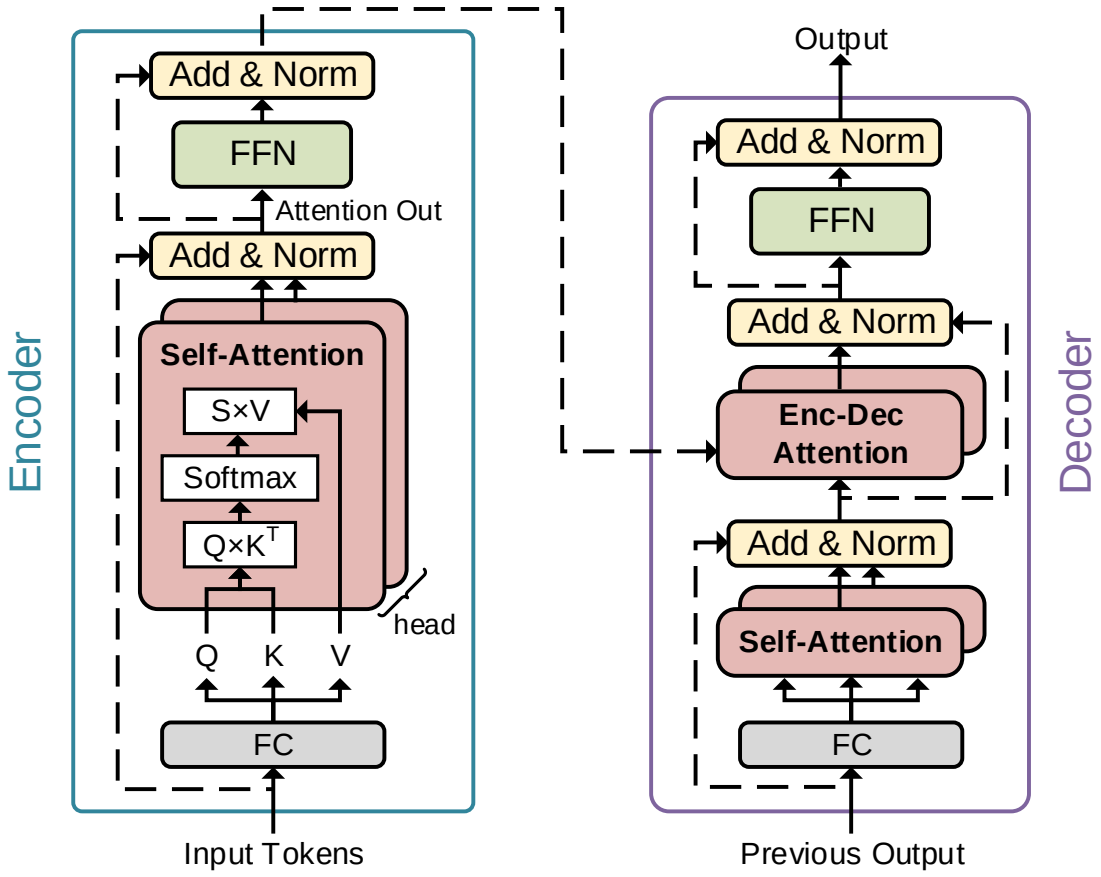
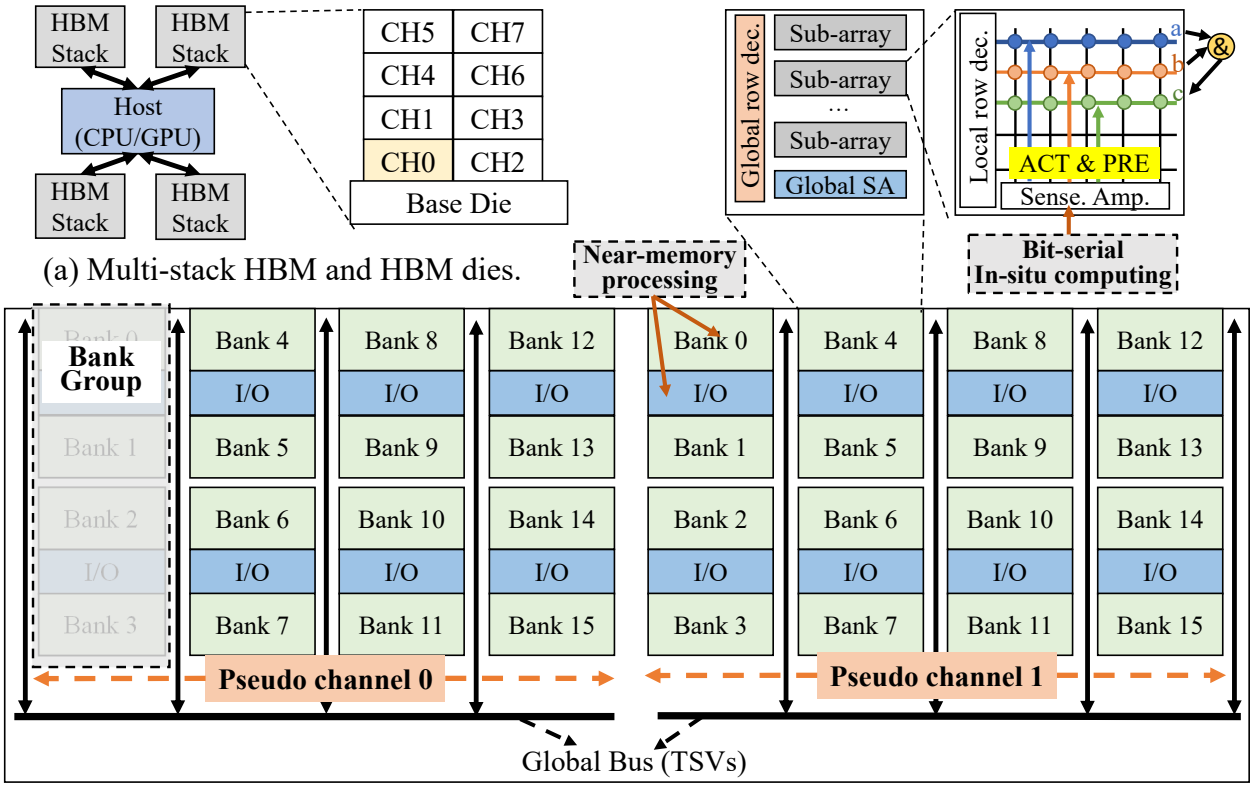


Figure 3.1: Operations of encoder and decoder blocks in Transformers.

3.2.1 Transformer

Transformer has an encoder-decoder architecture [80], as shown in Figure 3.1. Both the encoder and the decoder are constructed by stacking identical blocks. Each encoder block has three sub-layers including a fully-connected layer (FC), a self-attention layer (SA), and a feed-forward layer (FFN). For an input sequence with L tokens, the FC layer gets an embedding matrix of dimension $L \times d_e$ and generates a query matrix Q of dimension $L \times d_q$ and a key matrix K of dimension $N \times d_k$, and a value matrix V of dimension $L \times d_v$ by multiplying the embedding matrix with different weight matrices. For simplicity, we use D to denote d_k, d_q and d_v. Each D-dimension vector in the Q, K and V matrices corresponds to an input token. The encoder block then feeds the Q, K, and V matrices to the SA layer. The key operation in a SA layer is the scaled dot-product attention which computes dependencies between input tokens as $\text{Softmax}(\frac{QK^T}{\sqrt{D}})V$, where $\text{Softmax}(\cdot)$ denotes the Softmax



(a) Multi-stack HBM and HBM dies.

(b) Hierarchical architecture of one channel.

Figure 3.2: Memory-based computing on HBM architectures.

function. The $\frac{QK^T}{\sqrt{D}}$ is defined as the score matrix S. The encoder block feeds the attention output to the FFN layer to generate the block output, which can be used as an input to the next block (either encoder or decoder) or to a task-specific output layer (e.g., classification). Each decoder block also has the FC layer, the SA layer, and the FFN layer to process the output of the preceding layer. Unlike encoder blocks, the input and output of decoder blocks usually contain only one or a few tokens. In addition, the decoder inserts another attention layer that performs attention over previous blocks. Transformer performs many memory-intensive operations [86, 87, 93], which make it suitable for in- and near-memory acceleration.

3.2.2 PIM Acceleration for Transformers

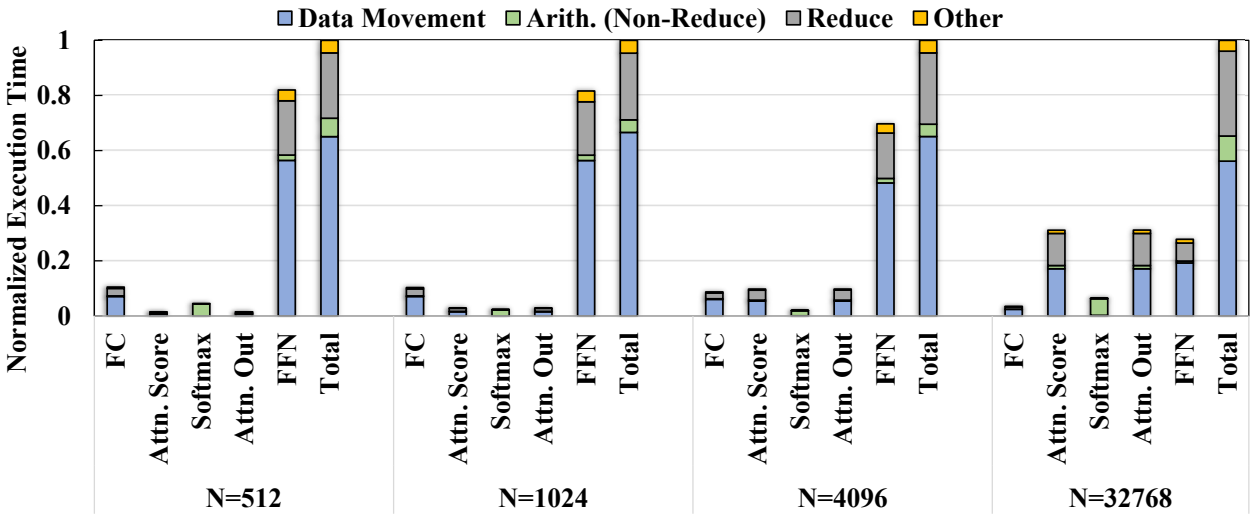
Memory-based computing has been widely used for various memory-intensive and compute-intensive applications due to its extensive parallelism and ability to minimize data movements. The

baseline memory architecture used in this work is the Samsung's high-bandwidth memory (HBM) [8, 9] which has become the state-of-the-art memory solution for various emerging platforms [94–97]. Figure 3.2 shows the overall architecture of 4 HBM stacks. All stacks are connected to the host CPU/GPU for cross-stack communication. Each HBM stack is a 4-high HBM chip with multiple DRAM slices on the top of the base die, connected with many through-silicon vias (TSVs), providing much higher bandwidth and lower access latency than the conventional DRAM.

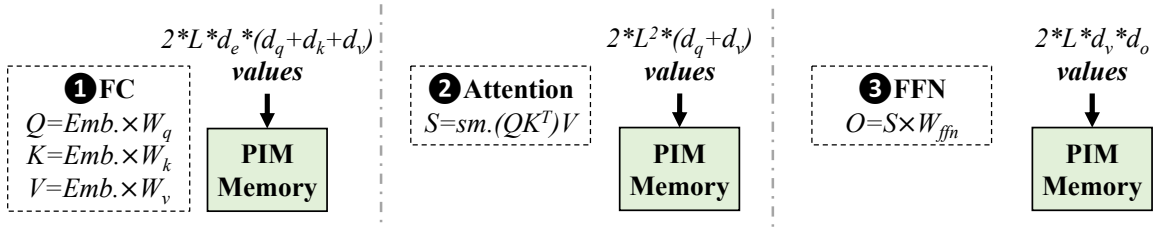
There are two ways to process data in HBM. The first one is near-memory computing (NMC), where compute logic is integrated either in the near-bank I/O or, more aggressively, in the near-subarray circuits inside the memory bank. For example, Samsung recently proposed a new type of HBM called function in-memory DRAM (FIMDRAM) [8], that integrates programmable computing units (PCUs) in the I/O circuits of the memory banks. These non-trivial PCUs take up the chip area for some memory banks, decreasing the memory density. The second technology is processing in-memory (PIM) which supports computing in the DRAM banks (or subarrays) by specialized sequences of activate and pre-charge commands [14] or modified subarray structures [13, 15]. PIM usually requires data to be placed column-wisely and follows a bit-serial row-parallel scheme to process the computation. PIM provides a higher level of parallelism with fewer data movements than NMC, but can only support a limited set of operations with high latency (because of bit-serial processing). NMC supports more general operations but the throughput is limited by the number of NMC processing elements as well as the bandwidth of the data link.

3.2.3 Motivation of Software-hardware Co-Design

Many previous works show that both the software-level scheduling (a.k.a., dataflow) and the hardware design play important roles in neural network acceleration [1, 2, 98]. To maximize the parallelism, existing memory-based DNN accelerators [3, 4, 13, 15] adopt a layer-based dataflow which allocates sufficient memory resources to parallelize computations for different output elements in a layer. The layer-based dataflow requires a whole data loading before processing each layer. We conduct an investigation on the efficiency of existing memory-based acceleration for Transformers.



(a) Latency breakdown of a BERT pre-trained model on PIM (N: #Tokens)



(b) Data loading overhead - Dimension: Emb.← (L, de), Q/K/V/O← $(L, \mathrm{d_{q/k/v/o}})$

Figure 3.3: Challenges of PIM acceleration for Transformers.

We evaluate the latency breakdown of a text classification task using RoBERTa Transformer model on an HBM-based PIM-only system which has 8 8GB HBM stacks. The PIM-only system processes all computations using bit-serial row-parallel operations inside memory subarrays. Figure 3.3(a) shows the profiling results. Figure 3.3(b) shows the size of loaded data for each layer in Transformer when using layer-based dataflow.

Our experiments show that the data movement of the layer-based dataflow takes the majority (around 60%) of the execution time, which is the time for loading and reorganizing data. The long data movement time results from two aspects. First, to maximize the parallelism, the layer-based dataflow needs data duplication in the memory for parallel computations, increasing the amount of loaded data. Second, most parallel data layouts do not exploit the data reuse between neighboring layers. In this case, we need to load all data for the intermediate layers (e.g., the attention layer).

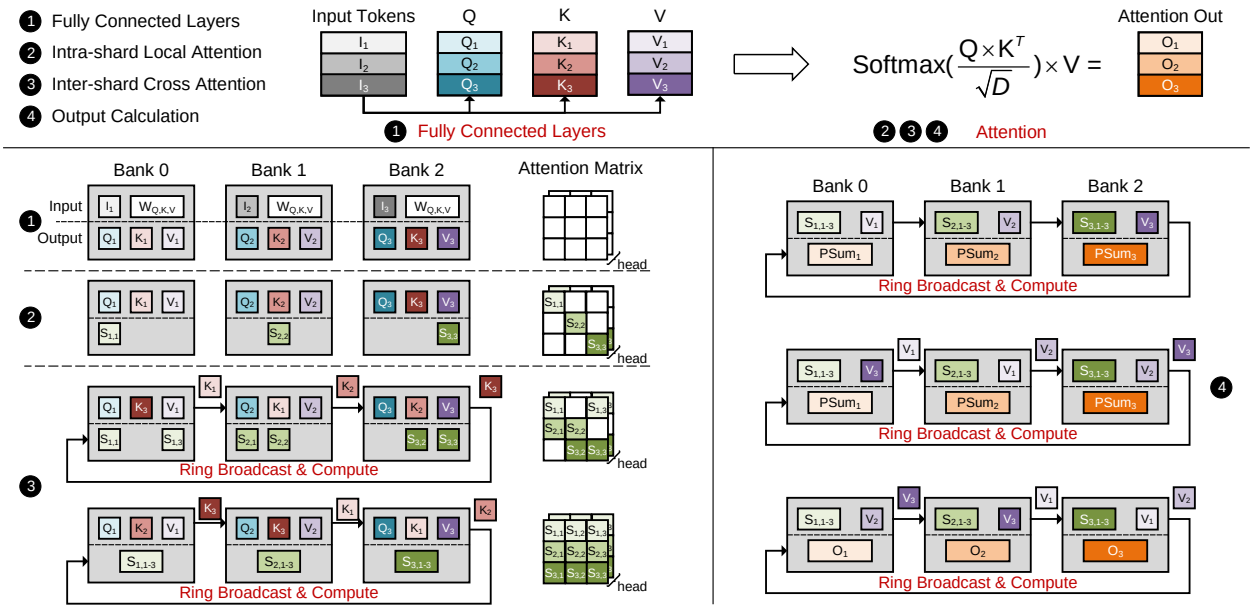


Figure 3.4: Token-based data sharding scheme and the dataflow of Transformer encoder in TransPIM. Banks = 3.

As shown in Figure 3.3(b), the size of computation data for the attention layer grows quadratically with the sequence length. Therefore, minimizing the amount of loaded data is critical to reducing the overall execution time of Transformer. In this work, we exploit the fact that all operations in different Transformer layers are related to tokens in the input sequence, making it possible to improve the data locality by reusing token-related data during the execution.

In addition to the data loading issue, the evaluated PIM-only accelerator does not perform well for some complex Transformer computation primitives, like reduction operations, due to the costly intra-bank (or subarray) data movements for long vectors in Transformer (e.g., $D = 512$). For example, the reduction takes 23% to 32% of time for different sequence lengths, which is significantly larger than other arithmetic operations (4% to 10%). This is because the reduction requires data movements to reorganize the data in the memory for accumulation, degrading the efficiency of PIM operations. Such intra-array data movements introduce more overhead in Transformers than CNNs because the length of vectors for reduction is much longer in Transformers (e.g., 512 for Transformer vs. 9 for convolution with 3×3 kernels).

3.2.4 Key Ideas of TransPIM

Our investigation shows that existing technologies introduce large overhead on Transformers, requiring specific modifications on both software dataflow and hardware support. Therefore, this work proposes to accelerate Transformer via a software-hardware co-design.

Dataflow: TransPIM adopts an efficient dataflow which maps Transformer computation to the memory-based architecture using a token-based sharding mechanism. TransPIM divides the input tokens into different shards and allocates these shards to different memory partitions. In this work, we use memory bank as the basic memory partition for shard allocation. During acceleration, each memory partition processes its associated token shard independently across different layers. As compared to layer-based dataflow, the token-based dataflow avoids the memory traffic for reused data. We also propose an efficient broadcasting algorithm to speed up data movements for dependent data by exploiting the large internal memory bandwidth of HBM.

Hardware acceleration: In the hardware, TransPIM adds lightweight modifications to the conventional HBM architecture, which not only accelerate various Transformer operations but also efficiently support the proposed dataflow. Specifically, TransPIM architecture implements multiple auxiliary computing units (ACUs) within each memory bank to perform vector reduction and Softmax function that cannot be efficiently processed by bit-serial row-parallel PIM operations. TransPIM exploits the benefits of both PIM and NMC to achieve high efficiency and throughput. Furthermore, TransPIM enhances the original HBM data path with specialized data buffers and communication links to support various data manipulations and movements for Transformers.

3.3 TransPIM Dataflow

In this section, we introduce the detailed process of TransPIM dataflow. The underlying architecture is based on compute-enabled HBM as shown in Figure 3.2.

3.3.1 Token-based Data Sharding

The key of TransPIM dataflow is token-based data sharding, which allocates HBM banks based on input tokens. The main benefits provided by the token-based data sharding come from the data reuse across different layers by keeping computations of tokens in the same memory location. We can reduce the data movement cost while exploiting more memory-level parallelism because different banks can handle computations and data movements for allocated tokens independently. After the token sharding, each bank handles computations for its shards throughout the end-to-end Transformer inference. The token-based data sharding is applied to the input tokens before the fully-connected layers of the first encoder block. As introduced in Section 3.2, the input tokens form a matrix with size $L \times D$, where L is the number of tokens and D is the embedding vector dimension. The input tokens are uniformly divided into "shards" along the token dimension and allocated to different memory banks. In the case of N memory banks, $\frac{L}{N}$ tokens are assigned to each bank. Therefore, each bank receives an input matrix with size $\frac{L}{N} \times D$. Figure 3.4 shows an example of distributing 3 tokens into 3 memory banks.

3.3.2 Encoder Blocks

During the Transformer inference, each memory bank performs the computations of FC, attention, and FFN layers for its allocated tokens.

Fully-Connected Layer

The FC layer generates the query (Q), key (K), and value (V) matrices from the input tokens. It involves three matrix multiplications between the input tokens and three weight matrices (W_Q, W_Q, and W_Q). In TransPIM, all weights of the three FC weight matrices are loaded into each memory bank before FC computation. And each bank multiplies the assigned $\frac{L}{N} \times D$ sub-matrix I_i with $D \times D$ weight matrices as illustrated in 1 of Figure 3.4. Then, each bank generates three $\frac{L}{N} \times D$ sub-matrices, Q_i, K_i, and V_i. The Q_i, K_i, and V_i matrices are retained in each memory bank and used

for the following attention layers.

Attention Layer

The computation of attention layer in TransPIM involves three steps: (1) *Intra-shard local attention*, (2) *Inter-shard cross attention*, and (3) *Softmax*.

Intra-shard local attention: The attention scores S are computed by $S = Q \times K^T$. In TransPIM, Q and K matrices are distributed in memory banks. With the local sub-matrices, each memory bank first computes the attention scores between local tokens, as shown in 2 of Figure 3.4). Each memory bank computes the $\frac{L}{N} \times \frac{L}{N}$ partial attention scores in the diagonal of attention score matrix using the local Q_i and K_i from the FC layer. During the intra-shard local attention, each bank i can independently compute the partial attention score matrix $S_{i,i}$ without communicating with other banks.

Inter-shard cross attention: After computing all local attention scores, TransPIM computes other attention scores by moving partial K matrices between different banks. As shown in 3 of Figure 3.4, the inter-shard cross attention consists of multiple ring broadcast and compute steps. To improve the bandwidth utilization, we propose a ring broadcast scheme to transfer K_j data between banks, where each K_j sub-matrix is sent to all banks step by step through an abstract ring of banks (e.g., 0→1→2 ... N→0). In each broadcast step, each bank multiplies local Q_i with the received K_j from a remote bank, generating an $\frac{L}{N} \times \frac{L}{N}$ partial attention scores in the i-th row and j-column of the blocked attention matrix. A total of N ring broadcast and compute steps are required to obtain the entire attention score matrix S. Each bank preserves $\frac{L}{N}$ rows of the attention score matrix, S_i, with shape $\frac{L}{N} \times L$. The performance of inter-shard cross attention heavily depends on the speed of the ring broadcast phase. In Section 3.4.2, we provide an efficient hardware design and a scheduling scheme to fully exploits the internal memory bandwidth of HBM for ring broadcast-based data transfer.

Softmax: The Softmax layer normalizes the exponential of attention scores related to each token. Each bank calculates the Softmax using its local $\frac{L}{N}$ rows of attention scores. There is no data movement between banks thanks to the data locality of attention scores. For multi-head attention

with h heads, there are h attention matrices to calculate. Therefore, the Softmax should be repeated h times to obtain all the results. The naive Softmax requires complex exponential function, reduction, and division. Thus, we design an efficient approach in Section 3.4.1 for calculating Softmax function in the TransPIM hardware.

Self-attention Output and Feed-forward Network

The final step of the self-attention is to multiply the attention score matrix S after Softmax by the V matrix. With the token-based data sharding, each bank stores the partial attention scores and partial V_i matrix. The calculation of self-attention output (4 of Figure 3.4) is similar to inter-shard cross attention. The partial V_i matrix is broadcasted through banks and each bank computes the $\frac{L}{N} \times D$ partial attention out O_i. Feed-forward network (FFN) consists of two consecutive FC layers. In the last step of Figure 3.44, the attention out (input of FFN) has the same token sharding as input FC layers (1). Therefore, each FFN-FC layer has a similar process to the input FC layers.

3.3.3 Decoder Blocks

The key difference between the decoder block and the encoder block is that each decoder block only needs to calculate one new token and attention operations between the new token and the old tokens. Figure 3.5 shows the processing flow for decoder layers which is slightly different from the encoder blocks. In each decoder block, we keep using the data sharding of the preceding block, which can be either the last encoder or the previous decoder. We allocate the last bank to process the FC layers for the new Q, K, and V vectors. In 1 of Figure 3.5, we allocate Bank 2 to process Q_{new}, K_{new}, and V_{new}. At the end of FC layers, TransPIM sends the generated Q_{new} to all other banks to compute attention scores. Besides, K_{new} and V_{new} are concatenated to the old K_i and V_i of last bank. At 2, each bank performs intra-shard local attention using local Q_{new}, K_i, and V_i. At the end of 2, each block obtains and preserves $\frac{L}{N}$ columns ($\frac{L}{N}+1$ for the last bank) of the new score matrix. In the output compute and summation (3), each bank uses its local S_i and V_i vectors to compute partial sum $PSum_i$ of the new attention output O_{new}. This scheme requires one reduction step to generate

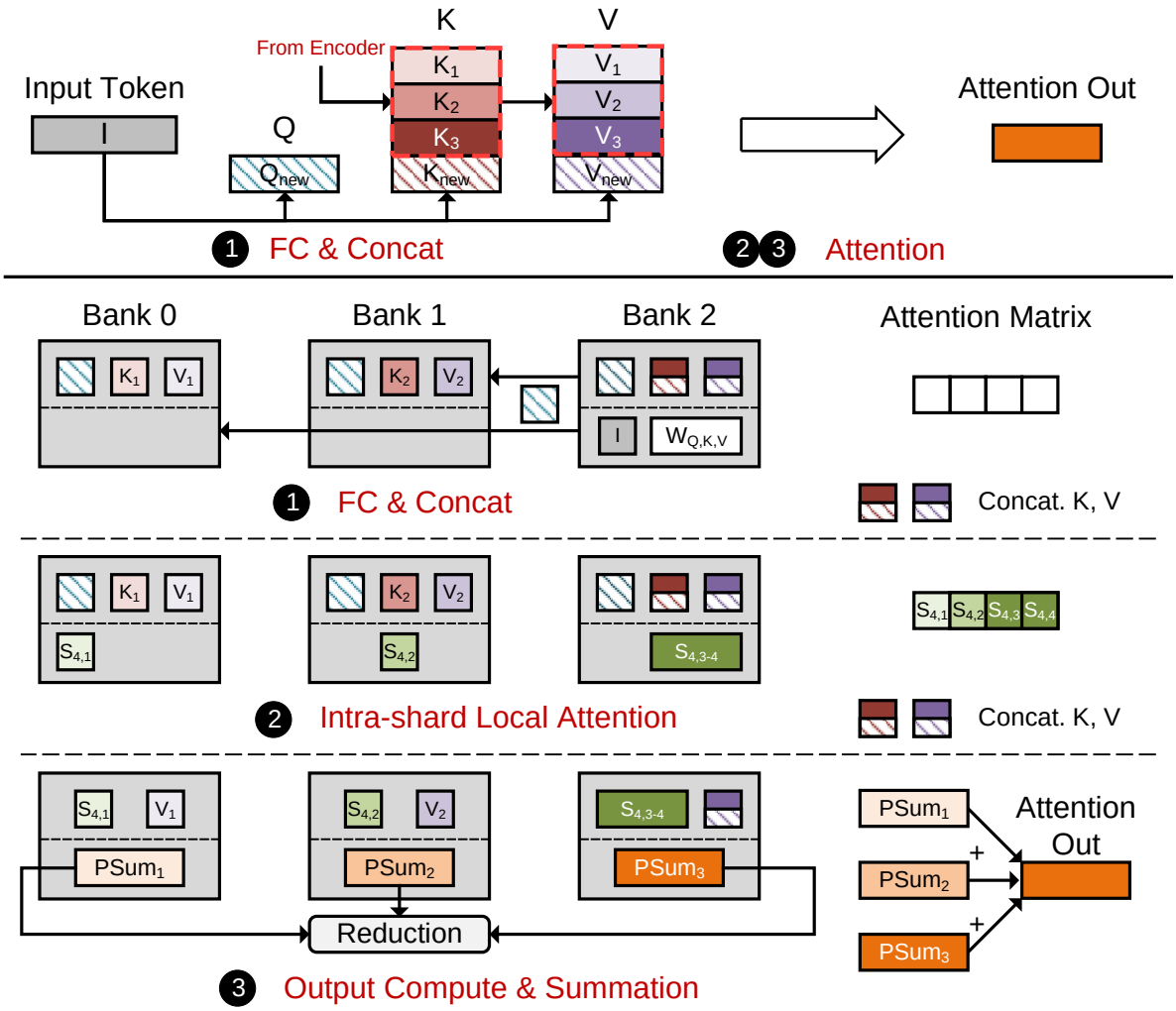


Figure 3.5: Dataflow of TransPIM for Transformer decoder.

the correct output. In Section 3.4.2, we introduce how to efficiently process such reduction in a modified HBM architecture. The decoder scheme supports both encoder-decoder and decoder-only Transformers. The only difference between these two types is whether memory banks pre-store "context" vectors which are K and V vectors from encoder blocks. For each new token, we allocate the bank with the minimum number of tokens to balance computation.

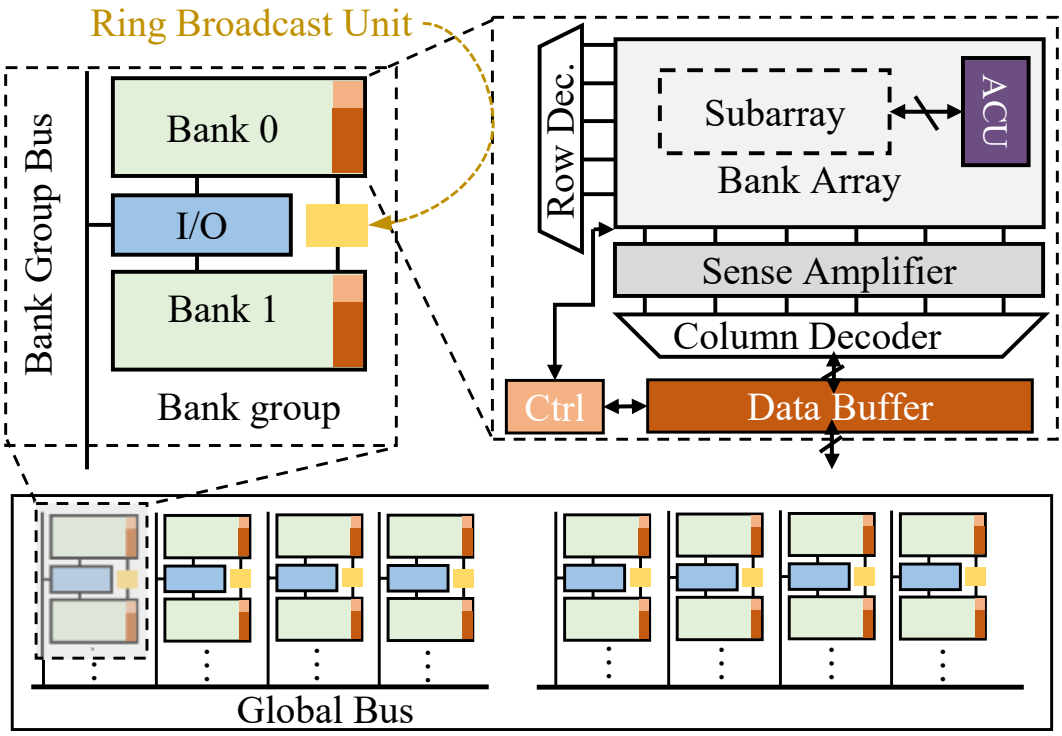


Figure 3.6: Overview of TransPIM hardware based on HBM.

3.4 TransPIM Hardware Acceleration

We propose a new memory-based hardware acceleration to address the challenges of accelerating Transformer models. Figure 3.6 shows the hardware customization in the standard HBM2 structure [99], which has two parts – 1) in-bank auxiliary computing units (ACUs), 2) a data communication architecture with near-bank data buffer and ring broadcast units integrated into the original HBM data path.

3.4.1 Auxiliary Computing Unit

TransPIM adds ACUs to support operations that are not friendly for in-memory bit-serial operations.

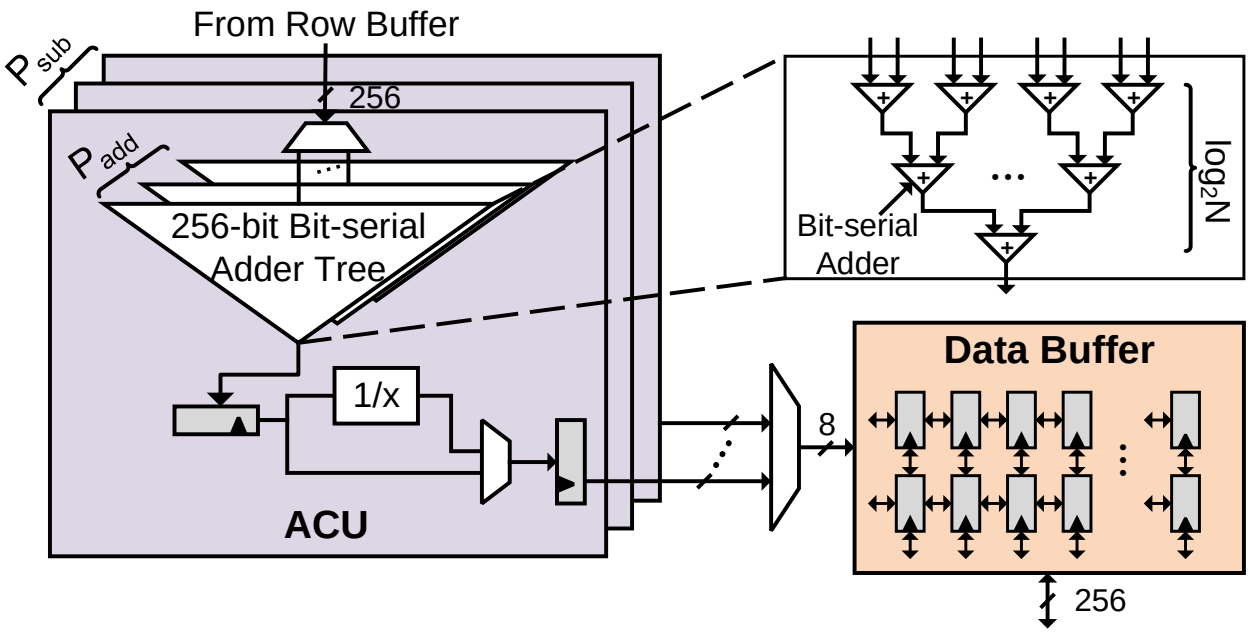


Figure 3.7: Detailed design of ACU and data buffer.

ACU Design

Unlike previous near-bank processing units [8, 12] for conventional row-wise data layout, the proposed ACU works on column-wise bit-serial data in order to support in-memory operations. As compared to previous PIM-only work [13], which modifies the cell array structure and adds extra shifters, the ACU offloads reduction, Softmax, and data broadcast with a more light-weight design.

As shown in Figure 3.7, each ACU is concatenated after the subarray to receive the 256-bit data from the row buffer because it is too expensive to employ arithmetic units to simultaneously process the data of the entire row buffer (e.g., 8Kb). Therefore, the same bit of 256 different numbers in a row is accessed from the row buffer for each memory access to support the bit-serial data layout for in-memory operations. A total of P_{add} 256-bit adder trees are implemented within each ACU to fully exploit the internal bandwidth of memory bank and reduce the DRAM's row activation overhead. Each adder tree is composed of area-efficient 255 bit-serial adders [100] to support the bit-serial data organization and reduce overhead. Besides, the bit-serial adder tree is stage-pipelined. A register and a divisor are implemented to latch intermediate partial sums and compute the reciprocal of row accumulation in Softmax function. Similar to [13], P_{sub} subarrays in each bank are activated

simultaneously to increase the computation parallelism. Thus P_{sub} ACUs are implemented for each memory bank.

When the vector length of point-wise vector multiplication results is less than the width of the adder tree, the ACU reduction can be computed using a single bit-serial adder tree. For b-bit data, a total of b row accesses are required to compute the reduction results. However, the reduced vector length is generally larger than the width of the adder tree. In this case, ACU needs to issue $b \times \lceil \frac{N}{256} \rceil$ row accesses, where N is the vector length (> 256). To speed up the ACU reduction and save energy dissipation, we implement P_{add} bit-serial adder trees in parallel within each ACU. Before precharging and activating a new row, ACU performs P_{add}-time column accesses in the same row and consecutively feeds the data into P_{add} adder trees. Considering that the interval t_{CCD} for DRAM to issue column access commands is much less than row access t_{RC} (i.e. $20\times$ less), the row activation time and latency of reduction decrease to around $\frac{1}{P_{\text{add}}}$ compared to the case with just one adder tree. Moreover, the proposed design trades excessive row activation energy by the register energy. The energy consumed by reduction operation is significantly reduced.

TransPIM Computing

TransPIM supports all computing operations for Transformer by adopting a hybrid in-memory and near-memory computing paradigm with ACUs and data buffers. Specifically, point-wise vector operations are performed in the memory cells since DRAM natively supports such operations with very high parallelism [14, 101]. TransPIM offloads the vector reduction and part of Softmax function from subarray to ACU to improve the efficiency.

Vector Computation: Figure 3.8 (a) shows the data path of vector multiplication in TransPIM. The vectors are organized in a bit-serial format, where b rows of the same bit line are allocated for each b-bit data. For vector multiplication, it includes two steps: point-wise multiplication and reduction. TransPIM calculates the point-wise multiplication in the memory array using existing schemes [14, 15, 101]. We adopt the Boolean majority functions [15] to reduce the computing latency. These PIM operations utilize DRAM timing violations to perform bulk bit-wise operations

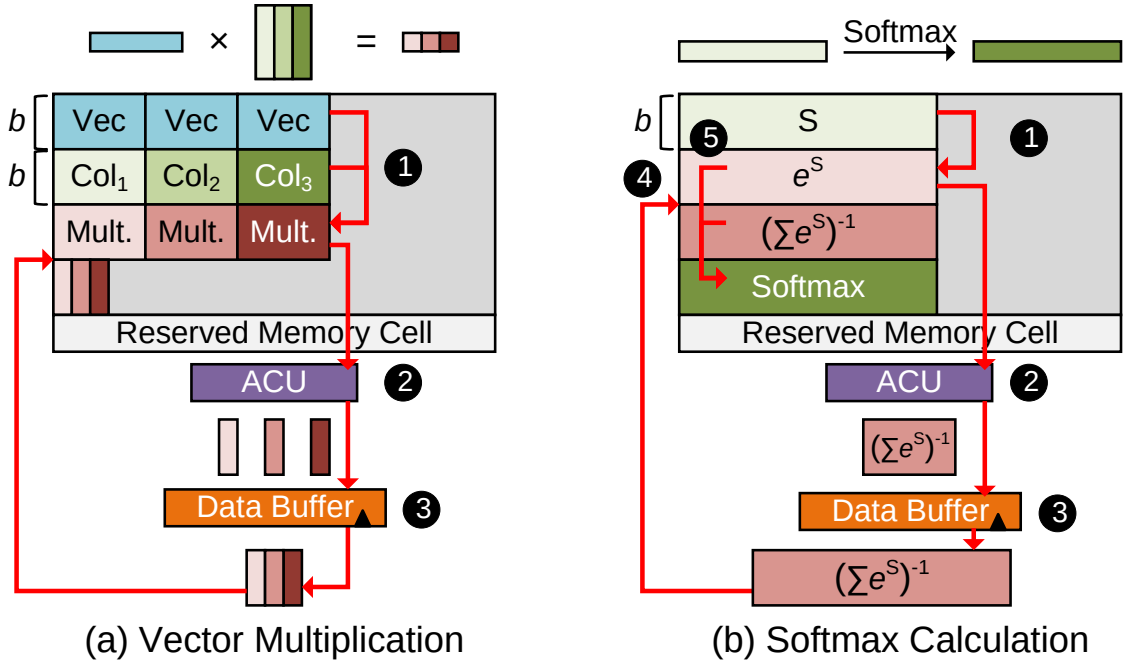


Figure 3.8: The data path of different computations in TransPIM.

in the standard DRAM architecture [99]. The vector has three copies in b rows to parallelize the computation (1). The point-wise multiplications between the replicated vectors and columns of the matrix are computed simultaneously. After point-wise multiplication, a vector reduction is required to obtain the results. The concatenated ACU to the subarray calculates the reduction of point-wise multiplication results (2). The ACU continuously receives data in bit-serial format from the row buffer and temporarily store the reduction results in the data buffer. The final reduction results will be written back to the memory cell as 3.

Softmax Calculation: PIM is unable to directly support the complicated exponential and division operations of Softmax. We resolve this limitation in TransPIM by rewriting Softmax function as $\frac{1}{\sum_{l=1}^{N} e^{S_{i,j}}} e^{S_{i,j}}$, where the reciprocal of row accumulation is moved out of the point-wise exponent. In this case, Softmax becomes the multiplication between the point-wise exponent and the reciprocal of the associated row. The point-wise exponent of attention score matrix S should be first calculated. Then the final Softmax output is the point-wise division between the exponents and the accumulation in the associated row. As shown in 1 of Figure 3.8 (b), the point-wise exponent is approximated using five-order Taylor series expansion, which is computed by PIM multiplication and addition. Then row

accumulation is offloaded to ACU using vector reduction. Meanwhile, the divider in ACU computes the reciprocal of row reduction (2). The single reciprocal value for a row will be replicated 256 copies and written back the memory cell array by the data buffer (3 and 4). Finally, the point-wise multiplication between reciprocal and exponent is computed in memory by PIM operations (5).

3.4.2 Data Communication Architecture

TransPIM dataflow exploits the internal memory bandwidth to reduce the data movement overhead caused by data loading. However, the standard HBM is still insufficient to match the high data parallelism and the internal bandwidth requirement. Even though we can use the bulk in-memory data movement approach like RowClone [16], the internal bandwidth is still limited by the shared data bus. Furthermore, the memory system needs frequent intra-bank and inter-bank data movements for memory-based computations, where data copy and re-organization may significantly downgrade the performance. Thus, we propose a data communication architecture to accelerate various data movements in TransPIM.

Customized Hardware Components

TransPIM introduce two customized hardware components in the HBM architecture for data communication.

Data Buffer: For most intra-bank and inter-bank data movements, we can use the fast parallel mode (FPM) of RowClone [16] to perform fast row copy. However, this approach has two defects. First, it is unable to provide a fine-grained partial copy for a row. Second, the FPM requires the source and destination rows to be located within the same subarray. To overcome the two constraints, we implement a re-configurable data buffer in each bank to manipulate data from ACU or row buffer as in Figure 3.7, realizing more flexible data movement. The data buffer is a configurable 8×256b buffer, consisting of 8 256-bit shift registers, supporting data copy and re-organization. The data buffer can either receive 8-bit (from ACU) or 256-bit data (from sense amplifier).

Ring Broadcast Unit: As illustrated in Section 3.3, TransPIM adopts a ring-based data broadcast

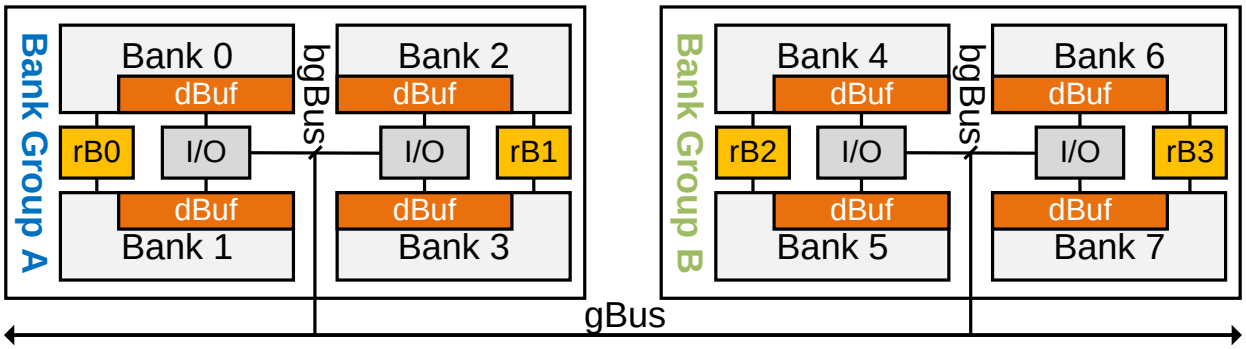
to reduce the data loading overhead for processing attention scores and self-attention output. However, the original HBM cannot efficiently support such ring-based data broadcast because all data transfers in a channel need to use the shared data bus and controller. TransPIM effectively decouples data transfers between different bank groups (or even different banks) by per-bank broadcast units and direct 256-bit link between broadcast units of two neighboring banks, as shown in Figure 3.9(a).

Optimization for TransPIM Data Movement

With the proposed communication architecture, we can accelerate various data movement patterns when running TransPIM dataflow.

Fine-grained data movement: Each data buffer, with the help of its controller, supports fine-grained data copy and duplication in a bank. When fine-grained partial copy is needed (such as 3 and 4 in Figure 3.8 (b)), the data buffer reads data from ACU through 8-bit input and performs data replication. The replicated data is written back to the sense amplifier through 256-bit output in a bit-serial manner. Another advantage of data buffer is its ability to move data between different subarrays without using the shared bus. The data buffer supports parallel accesses by reading 256-bit data from the sense amplifier for each column access cycle. It can cache at most 2Kb data and copy each 256-bit data into the sense amplifier located in a different subarray.

Ring-based data broadcast: Figure 3.9 shows the data movements of ring-based data broadcast (Section 3.3.2) in two bank groups of the TransPIM architecture. As illustrated in Section 3.3, each step of the ring-based broadcast requires all banks to copy data to their next banks in the ring (e.g., $1\rightarrow2\rightarrow3$ ... $7\rightarrow0$ in the figure). If we assume the time of a data copy between two banks is T, the original HBM architecture requires $8T$ because each data copy requires the global bus and controller. For TransPIM architecture, such ring-based broadcast consumes a time of $3T$ as shown in Figure 3.9. In the first step, we use the bank group bus (both BankGroup A and BankGroup B) to perform bank $3\rightarrow4$ transfer. At the same time, we can also copy data from bank $0\rightarrow1$ and $6\rightarrow7$ using ring broadcast links between broadcast buffers. In the second step, we use the bank group bus to transfer $7\rightarrow0$, while using the ring broadcast buffers for $2\rightarrow3$ and $4\rightarrow5$. The two remaining transfers, $1\rightarrow2$



(a) Memory Hierarchy in TransPIM

Time Step	1	2	3
Bank Group Bus A	3→4	7→0	1→2
Bank Group Bus B	3→4	7→0	5→6
Ring Broadcast Buffer	0→1, 6→7	2→3, 4→5	*idle*

(b) Timeline for Ring-based Broadcast

Figure 3.9: The optimizations for ring-based broadcast in the TransPIM hardware. The example shows the data transfers between two bank groups (4 banks per bank group).

and 5→6 can be processed in parallel during the third step. The algorithm can scale to more bank groups with the same time complexity, which is significantly lower than that of the non-optimized architecture.

Token reduction in decoder blocks: As introduced in Section 3.3.3, the output token of each decoder block requires a global reduction for all partial sums distributed in different banks. TransPIM can efficiently reduce all the partial sums in a multi-step parallel way. Specifically, in each reduction step, we separate banks with partial sums into multiple two-bank reduction groups and reduce partial sums of each reduction group by moving partial sums from one bank to another. All reduction groups process the reduction in parallel with PIM operations. TransPIM can efficiently support such data movements by exploiting the internal bandwidth provided by inter-ACU links, bank group bus, and channel bus.

Table 3.1: Architectural parameters for TransPIM

HBM Organization	Channels/die = 8, Banks/channel = 32, Banks/Group = 4, Rows = 32k, Row Size = 1KB, Subarray size = 512×512, DQ size = 256
HBM Timing (ns)	$t_{RC} = 45$, $t_{RCD} = 16$, $t_{RAS} = 29$, $t_{CL} = 16$, $t_{RRD} = 2$, $t_{WR} = 16$, $t_{CCD_S} = 2$, $t_{CCD_L} = 4$
HBM Energy (pJ)	$e_{ACT} = 909$, $e_{\text{Pre-GSA}} = 1.51$, $e_{\text{Post-GSA}} = 1.17$, $e_{I/O} = 0.80$
ACU	Clock = 500 MHz, $P_{sub} = 16$ ACUs/bank, $P_{add} = 4$ Pipelined Bit-serial Adder Tree/ACU, Adder tree width = 256, 3-stage pipelined divider
Buffer	Data buffer: 8×256b, Ring broadcast width = 256

3.5 Experiments

In this section, we describe our experiments that evaluate the benefits of proposed design.

3.5.1 Evaluation Methodology

The hardware characteristics for TransPIM are summarized in Table 3.1. The memory is standard HBM2 [99]. The timing and energy parameters are extracted from the previously published work [102]. Hardware components of TransPIM keep the same area and power constraints as the original HBM. The HBM area is estimated using the analytical tool CACTI-3DD [60] on 22nm technology node. We assume up to 8 HBM stacks are connected to a host CPU through the silicon interposer. The host-HBM bandwidth is 256GB/s [102].

We implement TransPIM using Verilog HDL and synthesize the design on Synopsys Design Compiler using 65nm library. The synthesized design is placed and routed using Synopsys IC Compiler. Moreover, clock gating is applied to save energy dissipation. $P_{add} = 4$ bit-serial adder trees are implemented in each ACU. The constant divider to calculate $1/x$ is three-stage pipelined to satisfy the timing constraints. In order to match the rate of column access time $t_{CCD} = 2$ ns, the ACU is configurated to run at 500 MHz clock frequency. The obtained area and power data of ACU are scaled to 22nm to match the memory technology. We consider the process difference between logic and DRAM using the similar method in previous work [13], where DRAM process incurs around 50% additional area overhead to the logic process.

The implementation results of TransPIM are given in Table 3.2. The 4-parallel bit-serial adder

Table 3.2: Overhead breakdown of TransPIM.

Unit/Bank	Area (um^2)	Power (mW)
Adder Tree	59432.1	25.1
Divider	3055.6	0.7
Data Buffer	2660.4	3.8
Ring Broadcast	337.9	0.2
Others	828.5	2.9

TransPIM	Area (mm^2)
8GB HBM2	53.15
Overhead	2.15
Memory Access	**Energy (pJ/op)**
ACU	0.384
Buffers	0.869

tree takes up 88% of the overall area. Each memory bank of TransPIM is equipped with $P_{\text{sub}} = 16$ ACUs. The total 512 ACUs consume about 2.15 mm^2, incurring 4.0% area overhead to the original DRAM architecture, far less than the 25% threshold of area overhead [12], hence avoiding DRAM density loss.

Simulation

We implement an in-house simulator to model the detailed performance and energy characteristics for TransPIM and all PIM baselines. The front-end of the simulator utilizes the TensorFlow interface which extracts the workload formation for the simulation. The backend simulator is a modified version of Ramulator [103]. We insert additional commands to the simulator for TransPIM to simulate the run-time behaviors of workloads for a given DRAM configuration. The architectural configuration of HBM and timing/energy parameters are shown in Table 3.1.

Hardware Baselines

GPU&TPU: The GPU platform is Nvidia RTX 2080Ti. We measure the GPU power using `nvidia-smi`. We also include a single Google Cloud TPUv3 with eight cores [104] as a baseline. We used JIT-compiled TensorFlow models and calculated the average latency from the second iteration to neglect graph compilation overhead.

Near-bank processing (NBP): Newton [12] is used as the near-memory baseline which is a near-bank processing technology in HBM2E-like DRAM offloading most operations for machine learning model to the near-bank logic. Since the NBP baseline already modifies the bank-level logic,

we enable the broadcast buffer, which handles intra-memory data movements, in the NBP baseline for a fair comparison. We assume the same HBM architecture for the NBP baseline as the one used by TransPIM.

Original PIM: The original PIM architecture is the basic HBM architecture with only the support for in-memory bit-serial operations using the specialized memory controller with modifications to the subarray as suggested by previous works [15]. We also assume the same HBM architecture for the PIM baseline as TransPIM.

Workloads

In this work, we evaluate two widely used Transformer models, RoBERTa [105] and Pegasus [106], for various important NLP tasks including text classification (IMDB) [107], summarization (Pubmed [108] and Arxiv [108]), and question-answering (TraviaQA [109]). The classification and question-answering tasks only have encoder blocks while the summarization tasks have both encoder and decoder blocks. We also evaluate a decoder-only task, language modeling (LM), using GPT-2-medium model [81]. All workloads are implemented using TensorFlow 2 with XLA.

3.5.2 TransPIM Performance

We evaluate the efficiency of TransPIM by comparing it with GPU and various memory-based architectures with either layer-based or token-based dataflow. We denote each system as "dataflow"-"architecture" (e.g., `Token-TransPIM`). For sensitive analysis, we test one more architecture configuration of TransPIM which disables broadcast units and data buffers for communication – denoted by "NB", while "Buf" denotes architectures with broadcast units and data buffers. Figure 3.10 shows the performance and the energy efficiency of all architectures as compared to the GPU baseline. All memory-based systems use 8 HBM stacks with a total capacity of 64GB. The performance is measured as the execution time per batch because workloads with short token lengths (e.g., IMDB and TriviaQA) may not fully utilize the memory for just a single batch. The GPU system runs with the maximum batch size supported for each workload. The energy efficiency is measured as

GOP/J of different systems. All values are normalized to the GPU baseline. All baselines run with a precision of 8-bit for FC and FFN layers which is sufficient for Transformer models [110]. We use 16-bit for Softmax to support a range of exponential.

Comparison to GPU/TPU: The proposed system (`Token-TransPIM`) is 22.1× (8.7×) to 114.9× (57.4×) faster than GPU (TPU). TransPIM shows less significant performance improvement on IMDB because the number of tokens is too small for the PIM system to fully exploit the parallelism because the token-based sharding requires each bank to process at least one token. For the workloads with more tokens, the token-based scheduling can saturate the compute capability of PIM system, fully exploiting the parallelism of PIM operations. As for the energy efficiency, TransPIM is 138.1× (39.5×) to 666.6× (376.7×) more energy efficiency than the GPU (TPU). Similar to the performance results, TransPIM achieves much better efficiency when running workload with long token sequence. The energy efficiency improvements result from the fast execution and the reduction of data movements.

Comparison to previous memory-based acceleration: As compared to previous PIM-only acceleration (layer-allocation), TransPIM with the token-sharding is 9.6× faster. If the PIM-only acceleration also uses the token-sharding processing, TransPIM is still 3.7 × faster. Furthermore, TransPIM is 4.2 × and 1.3× more energy efficient than the PIM-only acceleration with layer-based dataflow and token-sharding respectively. Such results show that TransPIM improves the performance and the energy efficiency of previous PIM acceleration by both software-side and hardware-side customization.

As compared to the NBP architecture, TransPIM is 9.1× and 6.4× faster with token-sharding and layer-based dataflow respectively. However, TransPIM is not more energy-efficient than the NBP baseline with the same dataflow (around 0.2% less). This is due to the large amount of energy consumed by bit-serial in-situ operations which require a lot of parallel row activation and pre-charge operations for all memory subarrays.

Comparison to ASIC: The previous ASIC designs, A^3 [87] and SpAtten [86], adopt pruning techniques to reduce the computation complexity and mostly focus on accelerating the self-attention

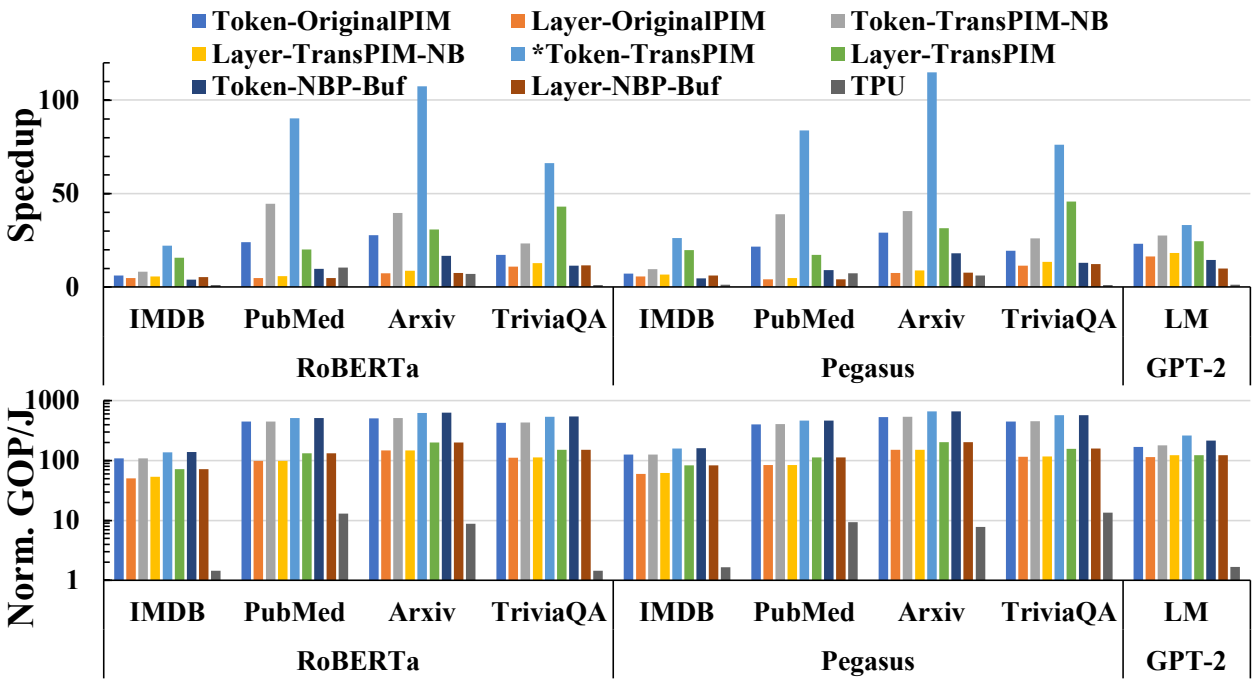


Figure 3.10: Performance and energy efficiency result.

layers. TransPIM targets lower data movement overhead and higher computing efficiency for the end-to-end execution of Transformer models. Since previous ASIC counterparts neglect the area of memory, we assume all systems use the 8GB HBM as the memory. The additional area of TransPIM is 2.15mm^2 for each 8GB HBM chip, which is close to A^3 (2.08mm^2) and $SpAtten_{1/8}$ (1.55mm^2). SpAtten [86] reports a 35× end-to-end performance improvement for generative stage (decoder) in GPT-2 model as compared to GPU. As a comparison, TransPIM achieves 83.9× and 114.9× speedup on two similar workloads (PubMed and Arxiv with Pegasus). Furthermore, TransPIM yields an average throughput of 734 GOP/s which is around $2.0-3.3\times$ of the peak throughput of A^3 (221 GOP/s) and SpAtten (360 GOP/s). The gain comes from three aspects. The token-based data sharding avoids redundant data movement, thus improving the computation efficiency. Moreover, the high data parallelism of in-memory and near-memory computing provides higher peak performance. The optimized data path of TransPIM exploits the large internal bandwidth of HBM to reduce the data movement overhead. In comparison, the performance of ASIC is constrained by limited computing resources and off-chip memory bandwidth.

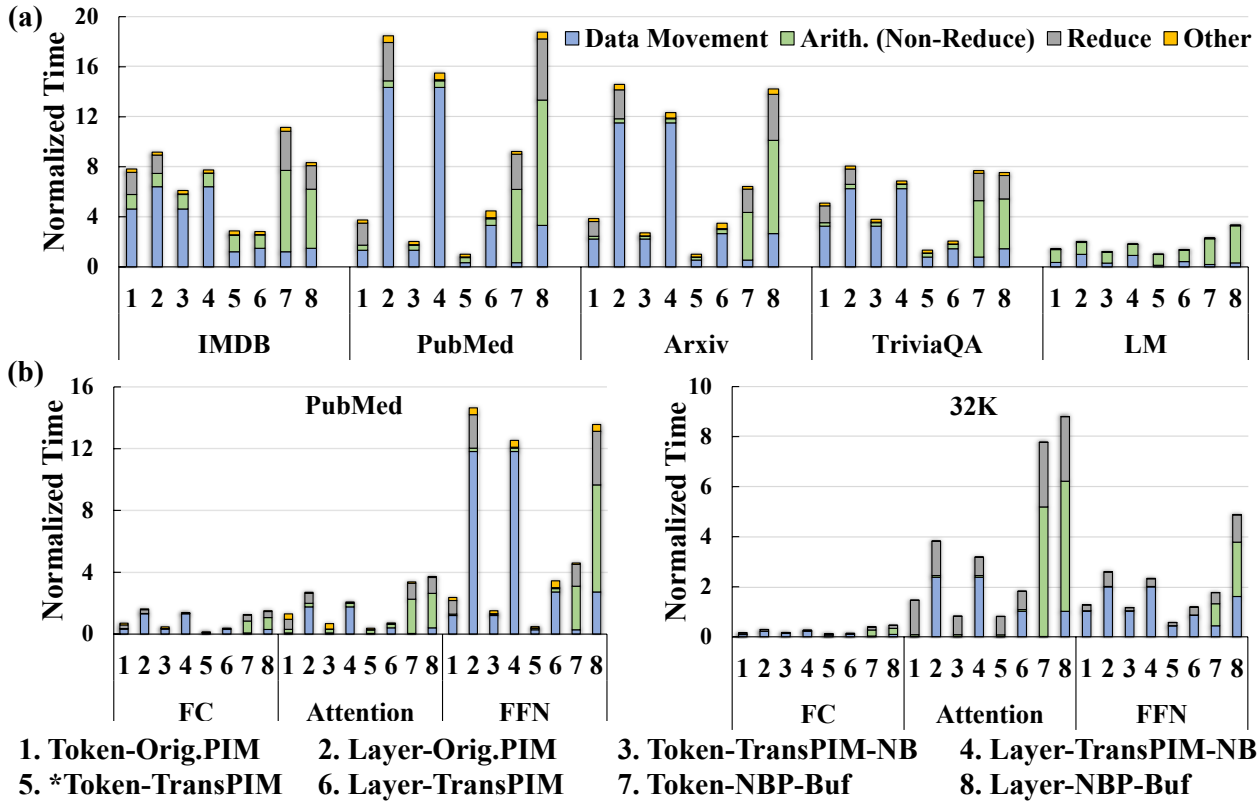


Figure 3.11: Performance breakdown of different systems: (a) overall breakdown, and (b) layer-wise breakdown.

Decoder-only model: For decoder-only workload (GPT2-LM), TransPIM is 1.4× faster and 2.1× more energy-efficient than the second-best system (`Layer-TransPIM`). Both speedup and energy efficiency improvement of TransPIM over other systems become less than other workloads. This results from the fact that the decoder-only model only processes 1 token in each iteration, requiring much less data loading for in-memory computations than encoder-based models.

3.5.3 Detailed Performance Analysis

We also investigate the detailed breakdown of operations for all memory-based systems, as shown in Figure 3.11. The figure shows the breakdown of four important categories of operations including the data movement (loading and intra-memory copy), non-reduction arithmetics, reduction, and other operations including reads and stores.

Improvements over previous acceleration: As compared to the PIM-only system, TransPIM significantly reduces the overhead of data movement because of the efficient data path (18.2× and 4.1× improvements for layer-based or token-sharding dataflow). Furthermore, the customized ACU of TransPIM effectively accelerates the costly reduction operations, where TransPIM spends 35.3× and 56.1× less time on reduction than PIM- and NBP-only systems. As compared to the NBP baseline, the performance improvement of reduction operation becomes even larger because the NBP baseline has a much lower degree of parallelism. The limited parallelism of the NBP baseline significantly increases the latency of other arithmetic operations as compared to the PIM implementation which is 13.2× faster.

Effect of token-sharding: The breakdown also sheds light on the performance benefits of token-sharding. For all systems, adopting token-sharding reduces the data movement latency by 4.8×, 4.5×, and 4.5× respectively. Such improvements depend on the workloads, where we observe 1.3×, 10.1×, 5.0× and 1.9× improvement on IMDB, PubMed, Arxiv, and TriviaQA respectively. Such results show that the token-sharding works better in large workloads (longer sequence) than small workloads because the data loading time of layer-allocation schemes increases quadratically with the sequence length for the attention layers. For the token-sharding, the size of moved data only increases linearly.

Effect of data movement optimization: While the token-sharding dataflow can significantly reduce the data movement overhead, TransPIM can further reduce it through the customized data path with broadcast and copy buffers. As compared to TransPIM without buffers, such customized data path provides a 4.1× reduction on the data movement.

Layer-wise breakdown: Figure 3.11(b) shows the layer-wise breakdown results of summarization using Peagasus for two workloads – PubMed (4K) and a synthetic data with 32K sequence length. All results are normalized to the total time of the proposed `Token-TransPIM` system. Token-based data sharding reduces the data movement overhead in FC and FFN layers because it requires less data duplication for computation (but less parallelism) than layer-allocation dataflow. In attention layers, TransPIM significantly reduces the data movement overhead because of the high bandwidth

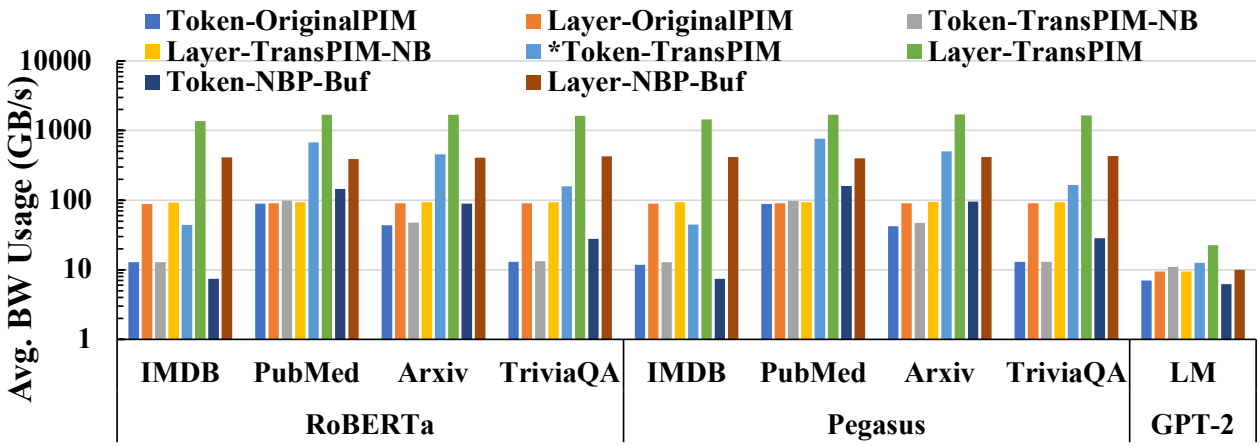


Figure 3.12: Average bandwidth usage.

utilization of ring-based broadcast as well as reduced data movement using token-sharding.

Resource Utilization: We use the percentage of time spent on computations to measure the utilization of memory banks. As shown in Figure 3.11, `Token-TransPIM` has an average 45.8% utilization, which is 1.5× higher than `Layer-TransPIM` (30.8%) because token-based dataflow significantly reduces overhead of data movement. However, `Token-OriginalPIM` and `Token-NBP` provides higher compute utilization, which are 47.7% and 89.5% respectively. This results from the extremely slow computation in PIM-only and NBP-only solutions. Figure 3.12 shows the average bandwidth utilization, which is the size of reading and writing data divided by the latency. The systems using layer-based dataflow consume more bandwidth than systems with token-based dataflow. For example, `Layer-TransPIM` has up to 1699 GB/s average bandwidth usage while `Token-TransPIM` only has up to 762 GB/s. Considering the overall latency, the result shows that layer-based dataflow requires much more data movements than token-based dataflow. Even though our 8-stack HBM system provides enough bandwidth ($BW_{aggregated} = 8 \times 256 = 2TB/s$), layer-based dataflow may become bandwidth-bound when increasing the workload size or decreasing the system bandwidth. On the hardware side, the usage of data buffer and ring broadcast in TransPIM increases the bandwidth usage of a specific dataflow because of low latency, showing that TransPIM's buffer architecture is always effective.

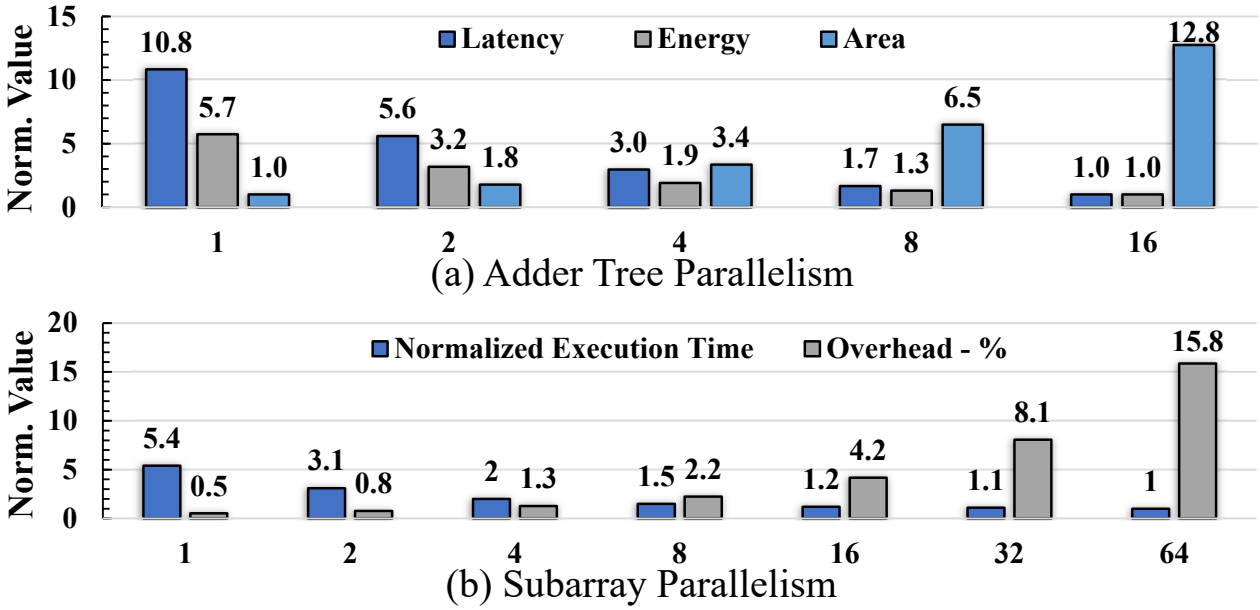


Figure 3.13: Design space exploration results for ACU.

3.5.4 Hardware Customization Exploration

The parallelism of bit-serial adder tree and data buffer size are the two design parameters of ACU. We need to explore different parameters to find the best tradeoff between additional overhead and resulting performance. We conduct a design space exploration on BERT model by varying the parallelism of adder tree P_{add} from 1 to 16. The results are depicted in Figure 3.13. The increased adder tree parallelism increases the accessed columns per row activation, thus reducing the number of repeated activated rows during vector reduction. As a result, the latency and energy consumed by vector reduction decrease by at most 10.8× and 5.7×, respectively. Besides, ACU reduces a large part of DRAM access energy by register access energy as shown in Figure 3.13(a). Finally, we choose $P_{\text{add}} = 4$ as the optimal parallelism for the adder tree in ACU since it keeps a good balance between additional ACU area and performance.

We can enable higher parallelism by simultaneously activating P_{sub} subarrays in a bank, where each subarray contains one independent ACU. However, adding more ACUs in a bank increases the area overhead. Figure 3.13(b) shows the execution time and area overhead when adding different ACU numbers in a bank. Adding one ACU for each subarray (parallelism = $P_{\text{sub}} = 64$) only increases

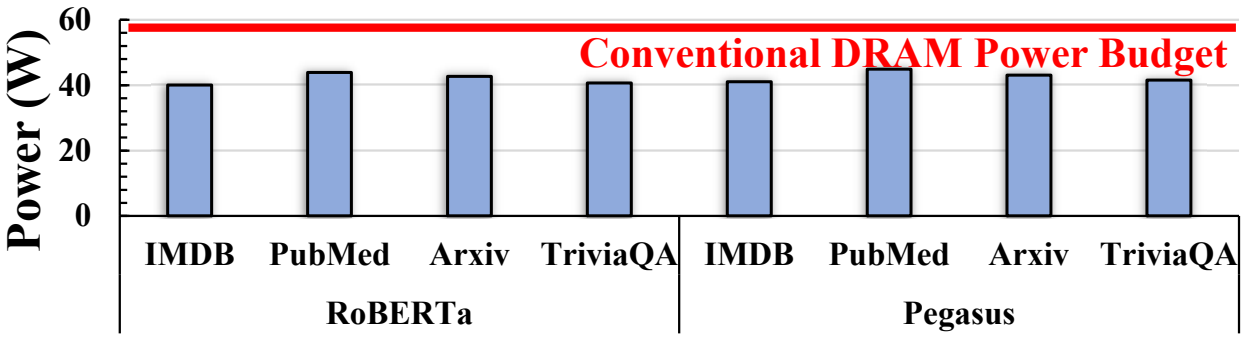


Figure 3.14: Power consumption of TransPIM on various sequence lengths.

the performance by 5.4 × while introducing 15.8% area overhead. We choose $P_{sub} = 8$ to well balance overhead and performance.

3.5.5 Power Analysis

We estimate the power consumption of TransPIM for tested workloads, as depicted in Figure 3.14. Pegasus models on TransPIM dissipate around 2% more power than RoBERTa models under the same sequence length. As the input sequence increases from 128 (IMDB) to 4096 (PubMed), the power of these two models increases about 4W, which is resulted from more computations. Overall, the consumed power of TransPIM is still below the 60W power budget of conventional DRAM system [102]. Thus, TransPIM satisfies the thermal constraints of conventional and TransPIM can be integrated into existing commercial DRAM system without additional modifications in terms of power and cooling.

3.5.6 Scalability

As shown in previous work [111–113], Transformer would become significantly challenging for longer sequences. Memory-based acceleration is promising to provide scalability by simultaneously increasing the memory bandwidth (with low memory access latency) and the compute parallelism. Figure 3.15 shows the speedup when using more HBM stacks for processing workloads with different sequence lengths. The speedup is averaged across all workloads. The result shows that TransPIM

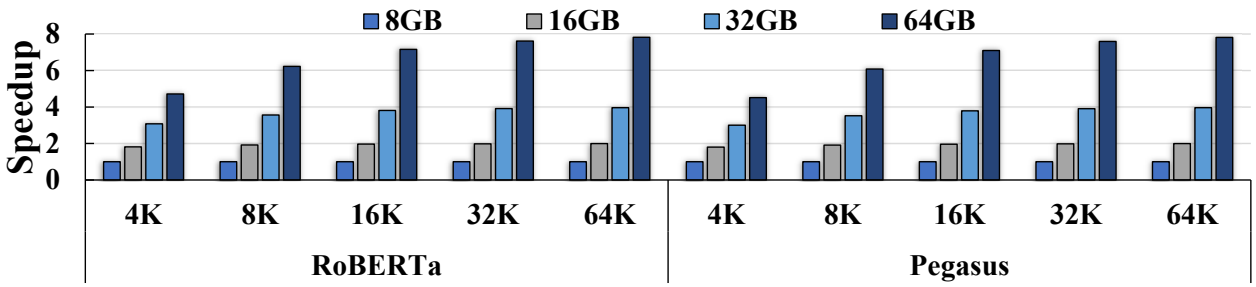


Figure 3.15: Scalability of TransPIM when increasing the sequence length.

provides good scalability (almost linear) for long sequence workloads which saturate the compute capability of HBM. As the GPU-based solution is bounded by the sequence length due to the limited memory capacity, our experiments indicate that TransPIM is a promising solution to extend the applicability of transformer models for long-sequence applications.

3.6 Related Work

Transformer accelerators: A GPU-based serving system and runtime called TurboTransformer is proposed in [114] to process long sequences through maximizing the utilization of computing and memory resources. But the scalability of TurboTransformer is poor since it is unable to support multiple GPUs. In contrast, TransPIM can be easily scaled up by stacking multiple HBM chips to yield larger memory space and support longer sequence lengths. SpAttn [86], A^3 [87], and GOBO [110] are state-of-the-art ASIC processors dedicated for the acceleration of attention module. Both A^3 [87] and SpAttn [86] implement sorting units to prune redundant heads and shrink the memory footprint, thus adapting to the limited on-chip buffer size. SpAttn [86] and GOBO [110] propose low-precision quantization and pipelined architectures to improve the efficiency. Besides, approximate Softmax computation is used in [87]. But GOBO and A^3 are unable to natively support the end-to-end acceleration of the entire Transformer. Moreover, they need to load data from off-chip memory before computation. The off-chip memory bandwidth would become the bottleneck for memory-intensive layers of the Transformer. Instead, TransPIM avoids costly off-chip data transfer by keeping all the data in memory.

PIM accelerators: Various PIM accelerators [3, 8, 12, 13, 91, 115] have been proposed to reduce the overhead of massive data movement as well as support high data parallelism. Newton [12], FIMDRAM [8], and McDRAM [115] adopt the similar near memory architectures and horizontal data organization in memory banks. They cascade bit-parallel arithmetic units to the DRAM bank to perform matrix-vector multiplication. However, the complicated bit-parallel and bulky buffer incur large overhead and decrease memory density [8]. To reduce the overhead of arithmetic units near memory, BFree [91] stores lookup tables that are compatible for computation in memory cells. However, the lookup table requires fine-grained optimization to save the consumed space. Previous in-memory accelerators [3, 4, 13] also optimize the complex reduction. Drisa [13] adds extra shifters in the subarray while NeuralCache [4] relies on the cache I/O peripheral to reorganize data multi-step hierarchical reduction. These two methods either significantly increase the area overhead or introduce large I/O latency. FloatPIM [3] supports reduction by organizing reduction data in a bit-serial way to avoid extra data movement. But this scheme sacrifices parallelism in Transformer which usually has long vectors for reduction. Different from the previous work, TransPIM combines the advantages of in-memory and near-memory computing. The proposed ACU is bit-serial to minimize the overhead of peripheral circuits. Compared to existing PIM accelerators, the proposed PIM-NMC combined computing paradigm provides better efficiency without affecting the memory density. MAT [116] is a PIM-based processing framework for attention-based machine learning models on long-sequence input. It breaks the long-sequence input into segments with various sizes and processes segment in a pipeline manner. However, MAT only targets a single encoder block, different from TransPIM which accelerates the whole Transformer.

3.7 Conclusion

Chapter 3 proposes TransPIM, an end-to-end accelerator for Transformers based on emerging HBM architecture. TransPIM adopts a software-hardware co-design principle to accelerate various Transformer models. As compared to previous accelerators, TransPIM significantly reduces the

overhead of data loading by exploiting the data locality in computations associated with input tokens. TransPIM also includes lightweight hardware modifications in HBM to improve the hardware efficiency of computation and data communication. Based on our evaluation on various Transformer applications, TransPIM is 68.9× and 16.6× faster than GPU and TPU respectively. As compared to prior software-hardware co-design PIM accelerators [11, 12, 88], TransPIM is 9.1× [12] and 9.6× [11] faster. Furthermore, the high-throughput PIM processing of TransPIM enables 2× to 3.3× higher throughput than state-of-the-art Transformer ASIC accelerators.

This thesis has so far introduced several software-hardware co-design technologies in PIM for various ML applications. Even though ML applications feature widely used operators, there are still many types of operations that are required by other emerging applications. Chapter 4 expands on the types of operations accelerated in PIM by introducing a software-hardware co-design of PIM acceleration for fully homomorphic encryption (FHE), which is an important technique to protect data privacy by allowing arbitrary computation on encrypted data without decryption.

Chapter 3, in full, is a reprint of the material as it appears in The IEEE International Symposium on High-Performance Computer Architecture, 2022, Minxuan Zhou, Weihong Xu, Jaeyoung Kang, and Tajana Rosing. The dissertation author was the primary author of this paper.

Chapter 4

PIM Acceleration for Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) enables arbitrary computations on encrypted data without decryption, securing many emerging applications. Unfortunately, FHE computation is orders of magnitude slower than computation on plain data due to the explosion in data size after encryption. Unlike the ML learning applications discussed in the previous chapters, FHE features various FHE-specific compute and data dependency patterns, resulting from the complex mathematical representation of data. Even though PIM can accelerate data-intensive workloads with extensive parallelism and high internal bandwidth, FHE is challenging for PIM acceleration due to the intensive long-bitwidth computations and complex data movement. This chapter proposes a PIM-based FHE accelerator, FHEmem, which exploits processing in-memory technologies with a novel memory architecture to achieve high-throughput and energy-efficient acceleration. We propose an optimized processing flow, from low-level hardware processing to high-level application mapping, that fully exploits the high throughput of FHEmem hardware for FHE applications.

4.1 Introduction

The data explosion leads to an increasing trend of cloud-based outsourcing. Extensive outsourcing significantly increases the risk of sensitive data leaking, necessitating data encryption for protection. Fully homomorphic encryption (FHE) is an emerging technology that enables computations on encrypted data without user interference [117–128]. FHE provides end-to-end data security during the outsourcing, including data transfer and computation, without any requirements for the underlying system and hardware, such as AMD SEV-SNP [129], Intel SGX [130], etc. However, FHE is several orders of magnitude slower than plain data while requiring large memory footprint [6, 131–133]. The inefficiency of FHE results from the data and computation explosion after encryption. Even though FHE can encrypt a vector into a single ciphertext [120, 124, 134], the ciphertext size is still large that includes two or more high-degree polynomials (e.g., 2^{17}) with long-bit coefficients (e.g., $>$1000 bits).

Such issues motivate researchers to develop customized accelerators that provide 4 orders of magnitude speedup over conventional systems [6, 131, 132, 135–137]. However, existing accelerators are still significantly bounded by the data movement even with large and costly on-chip scratchpads [6, 131, 132]. As shown in Figure 4.1(a), each homomorphic multiplication ($HMul$) requires 98MB to 390MB working set for $LogN = 15$ to $LogN = 17$. In Figure 4.1(b), we analyze the memory bandwidth required by different numbers of number theory transform units (NTTUs) under 3 data loading scenarios. Our investigation shows simply doubling the throughput of existing accelerators (1K to 2K) may require over 3TB/s of off-chip bandwidth. Previous accelerators adopt large on-chip storage, up to 512MB, to hold the large working set of FHE computation. However, such large on-chip storage can still suffer from frequent off-chip data transfers due to cache misses on large FHE data. Therefore, it is challenging to achieve both high compute throughput and high memory bandwidth on conventional architectures for FHE applications.

In this work, we exploit the processing in memory (PIM) acceleration for FHE, which is promising to support extensive parallelism with high internal memory bandwidth [3, 4, 11, 14,

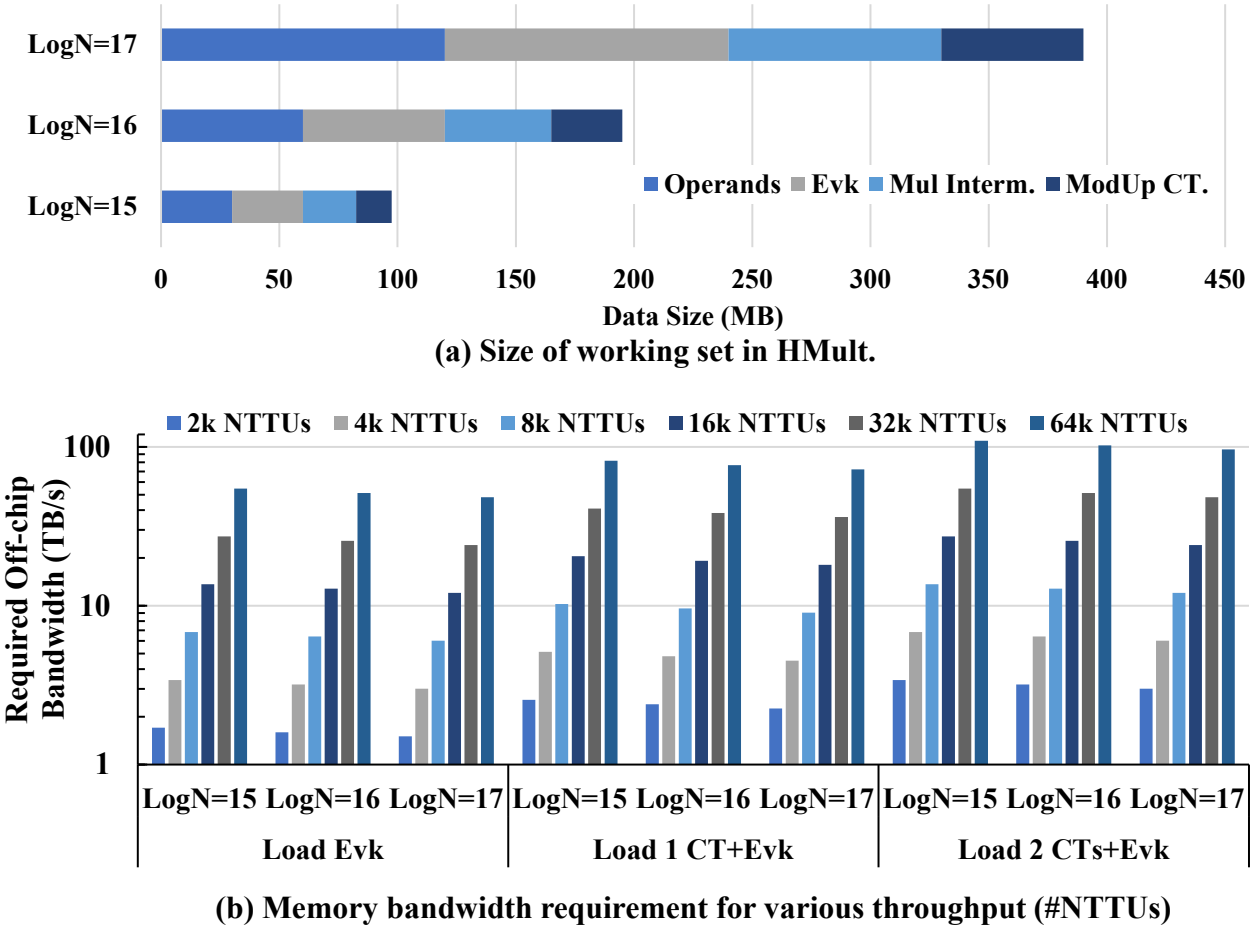


Figure 4.1: The memory bandwidth requirements when varying the on-chip throughput (#NTTUs). We assume L=30, LogQ=1920.

15, 20, 26, 42, 68, 101, 136, 138]. There are several types of PIM architectures that support PIM in different levels in the memory architecture, including subarray-level [11, 13–15, 101], bank-level [7, 8, 12, 92, 139], and channel-level [44, 68]. These PIM technologies adopt high-throughput processing elements that fully exploit the internal memory links that provide higher bandwidth than the off-chip data path. Even though the high parallelism and bandwidth of PIM potentially fit the data-intensive parallel computations of FHE, there exist several challenges that cannot be easily solved by existing PIM-based architectures. First, FHE works on high-degree polynomials with long-bit coefficients and is multiplication-intentive. Such long-bit multiplication is challenging to all existing PIM technologies. For near-bank PIM, the throughput is limited by bank IO width. Furthermore, even though the bank-level PIM can adopt highly efficient multipliers, reading the data

out from the memory cells is still energy-consuming. The bit-serial subarray-level PIM can exploit a significantly large number of internal links (e.g., bitlines). However, the number of operations required by subarray-level PIM may increase quadratically with the operand bit-length. These energy-consuming operations run in a lock-step manner, significantly increasing both delay and energy. Our evaluation shows such PIM technologies fail to provide comparable throughput and energy efficiency to the state-of-the-art FHE accelerators (Section 4.2.4). Integrating more functionality in the sense amplifiers can solve the problem, but introducing significant modifications in conventional DRAM structures [13]. The second challenge of PIM acceleration for FHE is the complex data movement patterns, including the base conversion and number theoretic transform (NTT), that the existing memory architecture cannot efficiently support.

To tackle these challenges, we propose FHEmem, an accelerator based on a novel high-bandwidth memory (HBM) architecture optimized for the efficiency of processing FHE operations. FHEmem introduces a novel near-mat processing architecture, which integrates compute logic near each mat without changing the area-optimized mat architecture. FHEmem exploits the existing intra-memory data links, with careful extensions based on the practicability, to enable efficient in-memory processing of various challenging FHE operations. Furthermore, we propose a software-level framework to map FHE programs onto FHEmem hardware. We propose a load-save pipeline that can fully utilize the memory for FHE programs to support high-throughput computing with minimum data loading overhead.

We summarize the contributions of this work as follows:

- We propose an FHE accelerator with a novel near-mat processing that supports high-throughput and energy-efficient in-memory operations. Our design efficiently exploits the existing data paths with practical modifications in DRAM that introduce relatively lightweight overhead as compared to prior high-throughput PIM solutions.

- We propose an FHE mapping framework that generates a load-save pipeline and data layout that maximizes the utilization of FHEmem for general FHE programs.

- We rigorously explore and evaluate different design dimensions of FHEmem to balance the performance, energy efficiency, and chip area. As compared to state-of-the-art FHE ASICs [5, 6], FHEmem achieves 4.0× speedup and 6.9× efficiency improvement.

4.2 Background and Motivation

4.2.1 Fully Homomorphic Encryption

We focus on CKKS scheme [120] which is widely used in many application domains because it supports real numbers and SIMD packing [34, 140–143].

Basics of CKKS: We define the polynomial ring $R = \mathbb{Z}[X]/(X^N + 1)$, where N is power of 2. We denote $R_q = R/qR$ for residue ring of R modulo an integer q. The ecurity parameter λ sets the ring size N and a ciphertext modulus Q. For each plaintext message, $m(X)$, encryption $\texttt{Encrypt}(m(X), s(X))$ generates a ciphertext $c = (b(X), a(X))$, where $b(X) = a(X) \cdot s(X) + m(X) + e(X)$, $a(X)$ is uniformly sampled from R_Q, and $s(X)$ and $e(X)$ are sampled from a key/error distribution respectively. Each ciphertext can pack up to $N/2$ real numbers to support SIMD processing on all packed numbers [120, 144]. The original modulus $Q = \sum_{l=1}^{L} q_l$ of a ciphertext decreases with homomorphic multiplications by a rescaling process that reduces the modulus by a q_l each time. The technique to recover the ciphertext level is bootstrapping [145].

Arithmetic Operation: Given two ciphertexts $c0, c1 \in R_{q_l}^2$, where q_l is the modulus at level l, the polynomial operation can homomorphically evaluate the arithmetic for plaintexts. The homomorphic multiplication (`HMul`) between two ciphertexts is complex: $c0 * c1 = (I_0, I_1, I_2) = (c0_a c1_a, c0_a c1_b + c1_a c0_b, c0_b c1_b) \in R_{ql}^3$, where I_2 is encrypted under the secret key s^2. This requires a re-linearization operation with an expensive key-switching process on I_2 (Section 4.2.1). The rescaling is applied during the relinearlization, using the divide and round operation: $\texttt{ReScale}(C) = \lfloor \frac{q_{l-1}}{q_l} C \rceil (mod\ q_{l-1})$ to rescale the ciphertext as well as the modulus.

Rotation: CKKS supports homomorphic rotation which rotates the plaintext vector by an

arbitrary step. The rotation is implemented by Galois group automorphism [146] which consists of mapping on each coefficient a_i: $\sigma_k(a_i) \rightarrow (-1)^s a_{ik\ mod\ N}$,where k is an odd integer satisfying $|k| < N$ and $s = 0$ if $ik\ mod\ 2N < N$ ($s = 1$ otherwise). Each automorphism σ_k implements a `Rotate`(δ) which rotates the plaintext by δ. Each automorphism also requires a `KeySwitch` after each `Rotate`.

NTT and Residue Number System: Number Theoretic Transform (`NTT`) is a widely used technique to optimize polynomial multiplications [147]. `NTT` transforms two input polynomials of a multiplication, a and b, to `NTT(a)` and `NTT(b)`. We can calculate $NTT(a*b) = NTT(a) \odot NTT(b)$, where $\odot$ denotes element-wise multiplication with $O(N)$ complexity, where N is the polynomial degree. An inverse NTT (`iNTT`) can transform $NTT(a*b)$. The complexity of `NTT` and `iNTT` is $O(NlogN)$, faster than the original polynomial multiplication with $O(N^2)$ complexity. Residue number system (RNS) is a technique that avoids computation on large values. We adopt the full-RNS CKKS in this work [148].

Key Switching: Key switching is the most expensive high-level operation in CKKS [145]. The key of key switching is the multiplication between the input ciphertext c and the evaluation key evk. To avoid overflow on modulus $Q = q_0 q_1 ... qL$, evk has a larger modulus PQ with the special modulus $P = p_0 p_1 ... p_k$. Thus the first step is to convert c with modulus Q into a ciphertext with PQ by a base conversion (`BConv`):

$$BConv_{Q,P}(a_Q) = ([\sum_{j=0}^{L} [a[j] * \hat{q}_j^{-1}]_{q_j} * \hat{q}_j]_{p_i})_{0 \leq i < k} \tag{4.1}$$

`BConv` requires the data in the original coefficient domain. We need to apply an `iNTT` on the data before `BConv`. `BConv` features an all-to-all reduction between different q_j and p_i residual polynomials. We convert the `BConv` result back to the `NTT` domain to efficiently process the multiplication with evk. The algorithm converts the result with modulus PQ back to modulus Q using `BConv`. Recent advanced CKKS schemes exploit a configurable $dnum$ value to factorize the modulus Q into $dnum$ moduli to increase higher multiplication level [145].

4.2.2 Memory Issues of FHE Accelerators

FHE features large polynomial operations and complex data dependency caused by `(i)NTT` and `BConv`. Recent works [6, 131, 132, 135, 149, 150] have proposed customized accelerators for FHE. Even though these accelerators achieve up to 4 orders of magnitude speedup over CPUs, they suffer from limited memory bandwidth. For example, previous work [132] observed that the excessive usage of high-throughput function units might be a waste - it would be cost-efficient to determine the throughput of on-chip processing elements based on the available memory bandwidth. Such memory issues result from the large data size required for each FHE operation. Existing FHE accelerators adopt large on-chip storage (180MB for SHARP [5], 256MB for CraterLake [6], and 512MB for BTS [132]/ARK [135]) to reduce the frequent off-chip data loading. However, such large on-chip storage may still be insufficient for FHE, as shown in Figure 4.1. For large FHE parameter settings, on-chip storage may only store working sets of one or two $HMul$, leading to frequent off-chip data loading when locality is low. The analysis in Figure 4.1(b) shows 2k NTTUs require at least 1.5TB/s when only loading *evk*, and the bandwidth requirement goes up to 3TB/s when the accelerator needs to load both *evk* and two operands. 3TB/s is expected to require 3 HBM3 stacks [151]. If we increase the throughput to 64k NTTUs, which can fully parallelize operations for $LogN = 17$, the bandwidth requirement can be as high as $100TB/s$. Considering it is challenging to significantly increase either the memory bandwidth or the on-chip storage, processing in memory can be a promising alternative.

4.2.3 In-DRAM PIM Technologies

This work focuses on DRAM-based PIM technologies which support larger capacity than SRAM [4, 152] and lower latency than non-volatile memories [3, 19, 26]. Figure 4.2(a) shows a DRAM bank which is the basic hardware component in DRAM. A bank consists of 2D cell arrays and peripherals to transfer data between DRAM cells and IOs. The memory cells are grouped into a set of subarrays, each of which consists of a row of mats. Each mat has local sense amplifiers (row buffers) sensing a horizontal wordline (WL) through a set of vertical bitlines (BLs). Sense

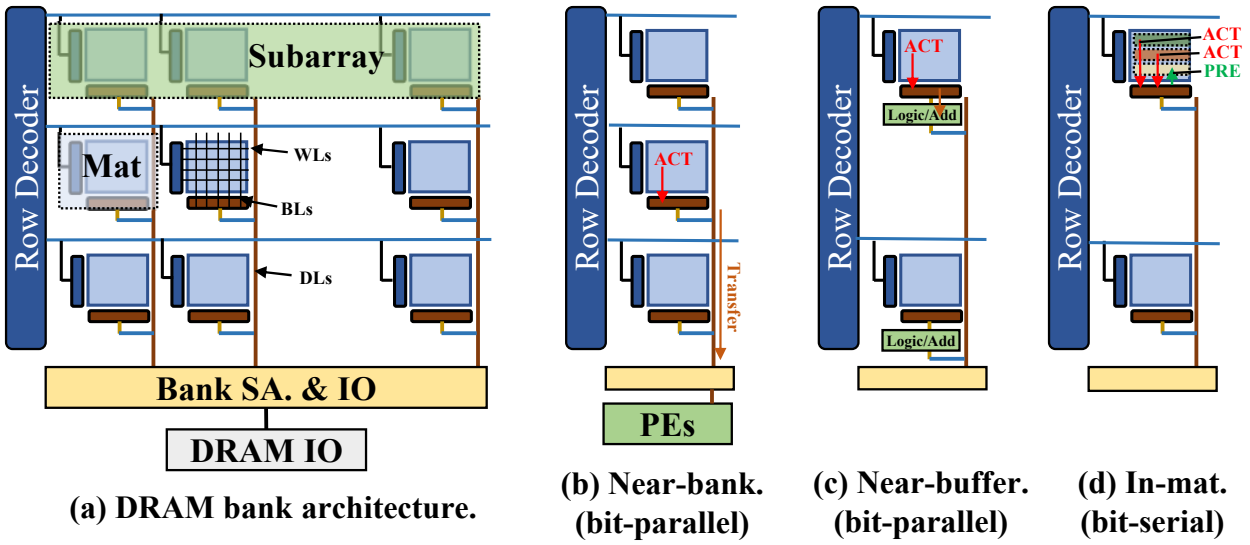


Figure 4.2: Different in-DRAM PIM technologies.

amplifiers in mats of a subarray form the subarray row buffer. Upon receiving access, the bank activates corresponding WL in subarray row buffer and transfers the whole WL to bank-level sense amplifiers via data lines (DLs).

There are several DRAM-based PIM technologies that support operations in different levels of DRAM architecture. The first technology is near-bank processing, which integrates processing elements (e.g., vector ALUs, RISC-V processor, etc.) near the bank SA and IO [7, 8, 12]. Each bank-level PE is customized to fully utilize the data link bandwidth for processing. PEs in different banks run in parallel to fully utilize the internal bandwidth in DRAM. The second type of PIM augments the subarray sense amplifiers (row buffers) with compute logic [13]. Due to the constrained chip area, the near-buffer PIM only adopts logic gates, full adders, and shift circuits for multi-bit operations. As compared to near-bank processing, near-buffer PIM support wider input (e.g., 8192b vs. 256b) and can exploit the subarray-level parallelism [10]. These two types of PIM work on the data with a horizontal layout where each data is stored across multiple BLs in a WL. The third type of PIM uses a vertical bit-serial scheme that lays out each data in different WLs of a BL [11, 14, 15]. The bit-serial PIM directly generates the result of computation between different WLs by exploiting the charge-sharing effect of the DRAM mechanism. Such in-mat bit-serial processing does not introduce significant modifications in DRAM. However, the bit-serial computation is slow and

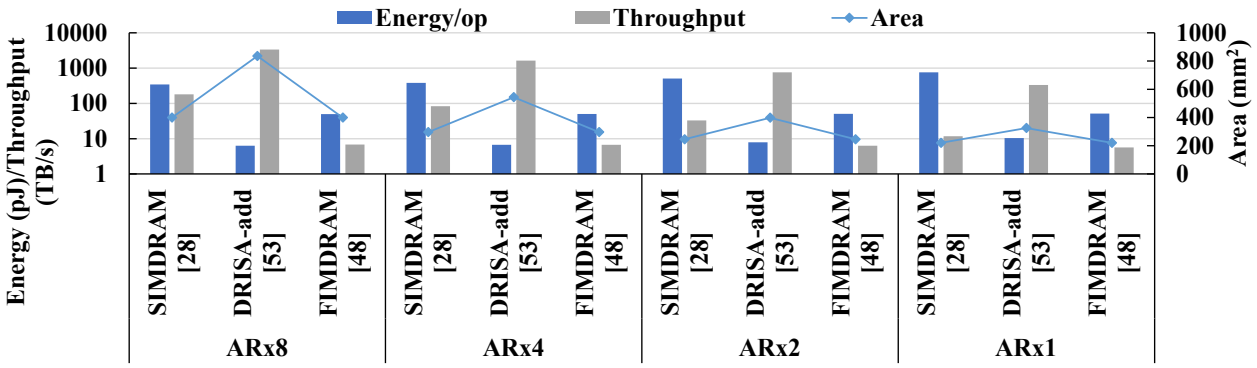


Figure 4.3: Throughput and energy efficiency of 32-bit multiplication using different PIM technologies (32GB).

power-consuming where an n-bit multiplication using the bit-serial PIM requires around $7n^2$ DRAM activations for 8k values.

4.2.4 Challenges of FHE acceleration using PIM

Even though existing PIM solutions can exploit the large internal DRAM bandwidth for high-throughput computation, FHE is still extremely challenging for PIM acceleration.

Long-bit multiplication

As introduced in Section 4.2.1, the basic data structure of FHE is high-degree polynomial, whose coefficient can exceed 2^{1000}. The RNS-decomposed polynomials still have at least 28-bit coefficients due to the limitation of modulus selection [6, 120]. As the complexity of PIM multiplication significantly increases as the bit-length increases, PIM (especially bit-serial in-mat) may suffer from long latency and high energy efficiency due to costly row activations and precharges. One way to improve the performance and energy efficiency of PIM is by increasing the aspect-ratio (AR) of DRAM mat [153, 154]. A high AR mat has fewer WLs (rows) and shorter BLs than a low AR mat. Shorter BLs can significantly reduce the latency and energy of activation and precharge. For instance, ARx4 mat (128 rows) has half the cycle and consumes 33% less energy than ARx1 mat (512 rows) [13, 153]. Furthermore, increasing the AR also increases the number of subarrays in a

bank, leading to a higher degree of parallelism. The downside of high AR is the large area overhead caused by more sense amplifiers and peripherals.

Figure 4.3 shows the throughput and energy efficiency of different PIM technologies for 32-bit multiplications on a 32GB HBM2E-based architecture (Section 4.5). We evaluate three existing PIM architectures, FIMDRAM [8, 92], DRISA [13], and SIMDRAM [11], that represent near-bank, near-buffer, and in-mat bit-serial PIM respectively. The result shows FIMDRAM and SIMDRAM provide 6.8TB/sand 180.6TB/s throughput while consuming 49.8pJ and 342.9pJ energy for each operation using ARx8 memory. As a reference, the recent FHE accelerator [6], which adopts 150k 28b multipliers, can provide 1PB/s of peak throughput while consuming only 4.1pJ for each multiplication, indicating both FIMDRAM and SIMDRAM are not promising for FHE. DRISA [13] provides over 3PB/s throughput and consumes 6.32pJ for each operation in ARx8 architecture. However, DRISA [13] requires a significant change in the DRAM architecture, incurring around 100% area overhead in high-AR architectures. Furthermore, manufacturing DRISA has significant challenges as the modified sense amplifiers cannot easily be aligned with area-optimized bitlines. To tackle these challenges, FHEmem adopts a novel near-mat processing that integrates compute logic near mat while keeping the mat structure intact, incurring less area overhead than DRISA. Even though the theoretical throughput and energy efficiency of FHEmem are lower than DRISA, our experiments show that higher throughput may not effectively improve the performance due to the data movement (Section 4.5). Overall, FHEmem is a more efficient processing paradigm than prior PIM solutions.

Data Transfer Patterns

Another critical challenge of FHE for PIM is the variety of data transfer patterns existing in different FHE operations. Specifically, the `BConv` requires the data movement between different RNS polynomials, followed by coefficient-wise operations; `(i)NTT` and automorphism require data permutation across coefficients of each RNS polynomial. Unfortunately, the conventional memory IO cannot efficiently handle such complex data movement patterns. For `BConv`, each output RNS

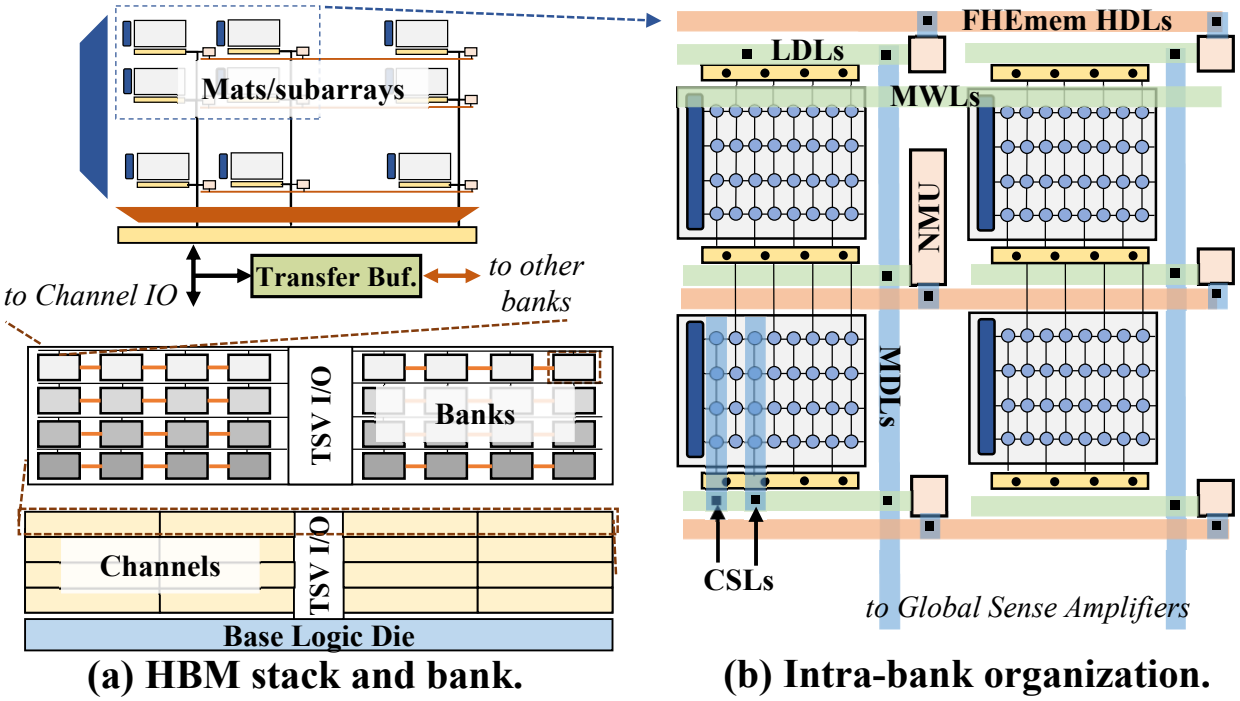
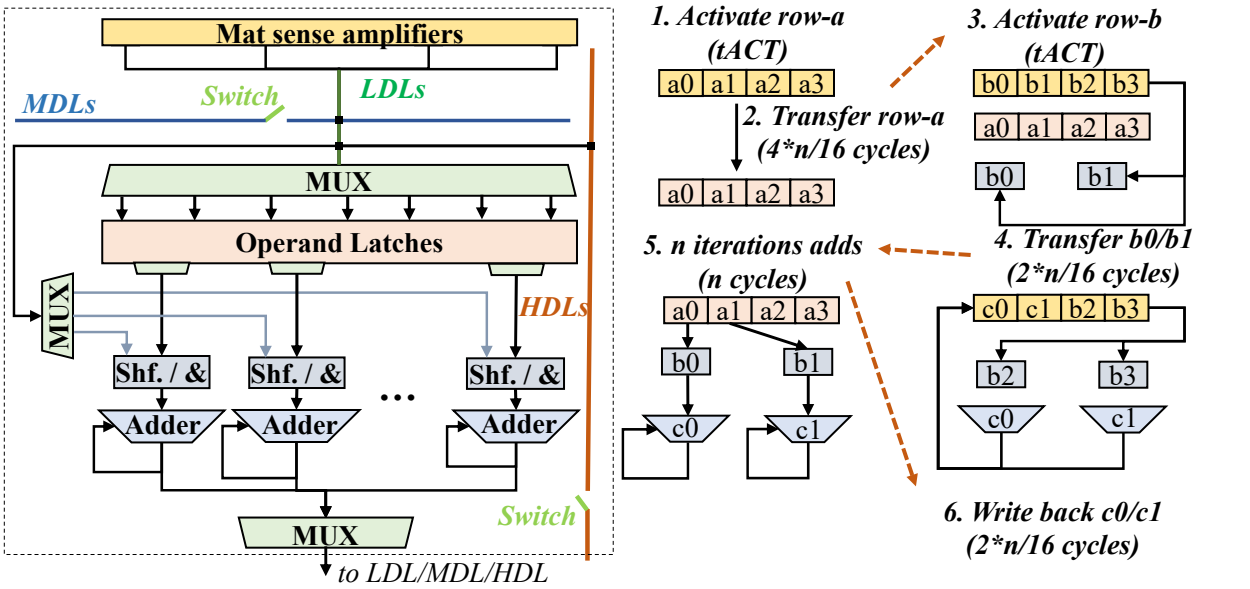


Figure 4.4: The hardware architecture of FHEmem.

polynomial has dependencies with all input RNS polynomials. As each RNS polynomial is large (e.g., 512KB for LogN=64 with 64-bit coefficients), we need to distribute RNS polynomials over different memory banks. In conventional memory, such inter-bank data movements take up the shared bus of each channel, leading to large data movement overhead. For `(i)NTT` and automorphism, the data movement exhibits a fine-grained pattern within a polynomial. For PIM processing, the coefficient-wise permutation requires permutations between BLs which is not supported in the current memory architecture. FHEmem supports these FHE-specific data transfers efficiently at a relatively low cost by exploiting existing intra-memory data links and adding additional links to the less-dense metal layer in DRAM, without introducing complex permutation networks.

4.3 FHEmem Hardware Architecture

The high-level architecture of FHEmem is based on high-bandwidth memory (HBM), as shown in Figure 4.4. Specifically, each HBM stack consists of one base die and multiple DRAM dies in a



(a) NMU architecture and connection. **(b) 4 64b multiplication using two adders.**

Figure 4.5: NMU architecture and NMU-based PIM.

3D structure. All DRAM dies are divided into multiple channels, each connecting to DRAM part through an independent set of through silicon vias (TSVs). Each channel consists of several banks, and the detailed internal structure of the bank is introduced in Section 4.2.3. FHEmem adopts a new near-mat PIM architecture that modifies the DRAM bank architecture to support high-throughput computations while utilizing available DRAM internal links for various FHE operations. The key customized components of FHEmem include near-mat units (NMUs), horizontal data links, and inter-bank connection.

4.3.1 Near-mat unit

FHEmem supports in-memory computations by connecting each mat to a near-mat unit (NMU) via the local data lines (LDLs). Each NMU consists of full adders, shifters, AND logic, and latches to compute FHE operations, shown in Figure 4.5(a). In addition to the mat sense amplifiers, NMU can also receive data from other NMUs via inter-NMU connection. The design of NMU differs from previous PIM-logic integration, DRISA [13], in three ways. First, NMU places all customized logic outside of the mat, which is optimized for area efficiency. On the contrary, DRISA [13] integrates

logic and latches with the mat sense amplifiers and bitlines, which may incur significant changes to the area-optimized mat. Second, DRISA [13] integrates logic to all bitlines in a mat, causing a large area overhead while only gaining moderate performance benefits due to the unbalanced compute and data movement [13]. In FHEmem, we explore the processing throughput of NMU under different architecture configurations and observe the most efficient design does not adopt the maximum throughput (Section 4.5). Third, NMU in FHEmem can support permutations required by FHE using multiplexers in the data path.

To compute FHE arithmetic (i.e., modular arithmetic), NMU requires several steps in mat, data links, and NMU. Figure 4.5(b) shows an example of processing 4 64b multiplications in an NMU with 2 64b adders. For generality, we denote each mat row can store N n-bit values (N=4 and n=64 in this example), and NMU has M n-bit adders. First, the mat needs to activate an operand row (1) and transfer the row to the row-size operand latches (2). Next, the mat activates the second operand row (3) and starts transferring M-value blocks to the shifter and AND logic (4). When both operands are ready, the shifter and AND logic will first generate a partial product using the second operand (b0 and b1) and bit masks of a specific bit of the first operand (a0 and a1 in the latches). NMU takes n cycles to compute an n-bit multiplication (5). After processing an M-value block, NMU writes the result back to the mat (6) and loads the next M-value block for processing (7). Similar to DRISA [13], NMU only needs two row activations for each vector processing but requires serial data transfers via LDLs. NMU can serially write back values in a different order to support permutation.

4.3.2 Inter-NMU Connection

To efficiently process various FHE data transfer patterns, FHEmem enables data transfer between NMUs in both horizontal and vertical directions. In the vertical direction, FHEmem utilizes the master data lines (MDLs) which connect all NMUs in the same mat column. In the horizontal direction, FHEmem adds extra data links, horizontal data links (HDLs), to each subarray (a row of mat). The HDLs in each subarray support the same bitwidth as the MDLs in each mat column (i.e., 16-bit). For both directional links, we add small isolation transistors (switch in Figure 4.5) [155]

to serve as switches that can disconnect each link at a certain point. NMUs separated by the off switches can transfer data independently, significantly improving the bandwidth for intra-bank and intra-subarray data movements.

We note that DRAM process only adopts 3 metal layers, with 1 layer (M1) for bitlines (vertical), 1 layer (M2) for LDLs and master word lines (MWLs) (horizontal), and 1 layer (M3) for column select lines (CSLs) and MDLs [102]. M1 layer is the only fine-pitch (low energy efficiency) layer, optimized for the area of bitlines in a mat. M2 and M3 support more energy-efficient wires with larger area overhead (i.e., 4x pitch of M1). M2 is fully occupied by CSLs and MDLs to support a 256b data path (16b per mat) for each subarray. M3 is less dense than M2 because each subarray only has 1 MWL (instead of 256b of MDLs/CSLs). In FHEmem, the additional horizontal data lines are placed in M3 which incurs less pressure in the routing. Furthermore, HDLs connect NMUs which are placed outside of dense mats (DRAM cell arrays). These characteristics make HDLs and NMUs practical using conventional DRAM technology.

4.3.3 Inter-bank Connection

As introduced in Section 4.2.1, FHE has dependencies between RNS polynomials of a ciphertext during `BConv`. To store a ciphertext, which can consist of tens of RNS polynomials, we need to distribute these RNS polynomials over multiple banks, and each BConv requires a large amount of inter-bank data transfers because of an all-to-all dependency. For such inter-bank data movements, a naive way is to use the conventional channel-level data bus. Unfortunately, such centralized data movements fail to match the throughput of in-memory computations, significantly slowing down the acceleration. Furthermore, the cost of fully-connected interconnect can be prohibitively high. In order to support the inter-bank communication with satisfactory performance, while introducing reasonable overhead in HBM, we propose a partial chain interconnect network between banks inside a channel. The partial chain network connects neighboring banks in each bank group, using 256b-wide transfer links and per-bank transfer buffers. Specifically, the transfer buffer in each bank can communicate with the local master data lines (MDLs) and the transfer links to the neighboring

Table 4.1: FHEmem NMU commands

Command	Operands	Description
nmu_ld	sa_id, sa_col, latch_col	load from SA to NMU latches
nmu_st	sa_id, sa_col, add_id/latch_col	store from NMU latch to SA
nmu_hmov	sa_id, sa_col, direction, stride	horizontal data movement across NMUs
nmu_vmov	src_sa, dst_sa, sa_col	vertical data movement across NMUs
nmu_set(_sa)	sa_id, sa_col, latch_col, add_id	set up operand for add (shift &AND)
nmu_add	sa_id	start addition
nmu_pst	sa_id, sa_col, latch_col_vec	store different NMU latches to SA

banks. We add two 256b transfer buffers in each bank to support seamless transfers between banks. The customized links and buffers enable parallel inter-bank data transfers across different banks in a channel, avoiding sequential transfers via the original channel IO. When transferring a whole row between two neighboring banks, the source bank drives 256b data blocks from the selected subarray SA via MDLs to the transfer buffer, which sends data blocks to the transfer buffer of the destination bank via either the customized links (neighboring banks) or the original channel IO (non-neighboring banks). The destination bank writes data blocks to the selected subarray SA via MDLs.

4.3.4 FHEmem Controllers

FHEmem needs modifications in bank/subarray controllers to support its various functions. FHEmem requires the extra logic for subarray-level parallelism [10], including the extra address latches in each subarray row decoder, and the bookkeeping logic in the bank controller for the status of all subarrays. Since the bank controller sends all subarray-level commands, it can overlap the in-memory computation, normal memory access, and intra-memory data transfers.

The subarray controller needs to support several new commands for NMU processing, as shown in Table 4.1. These subarray-level commands control the same behaviors of all mats/NMUs in a subarray, except *nmu_pst* which stores different latches in different mats back to SA (used for automorphism). For NMU loading and storing, FHEmem supports the flexibility of selecting columns in NMU latches, adder latches, and sense amplifiers, enabling permutation. The horizontal data

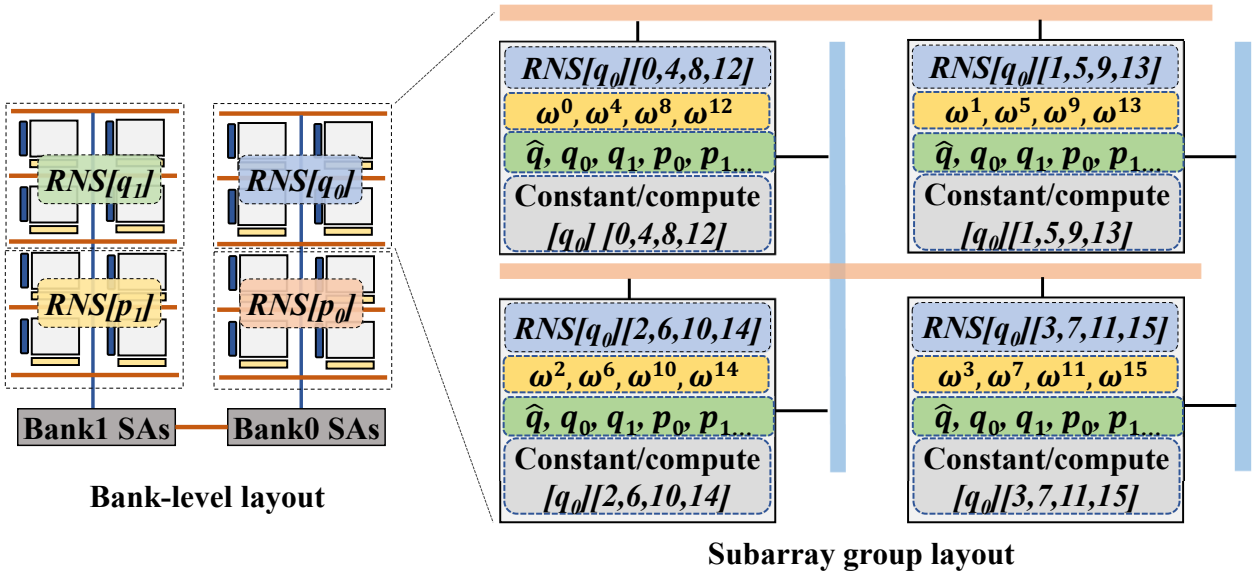


Figure 4.6: The data layout in FHEmem.

movement has predefined patterns, defined by *direction* and *stride*, to support NTT data movements. The vertical data movement has more flexibility to transfer data between two subarrays. Before issuing the *add* command, FHEmem requires a *setup* instruction to load data from mat sense amplifiers to the corresponding shift/AND logic, indicating whether to use shift/AND and the bit position of shift/AND.

4.4 Processing FHE in FHEmem

The proposed FHEmem hardware supports high-throughput computations and data movements. The next challenge is to efficiently utilize FHEmem for FHE applications. This section introduces the data layout and processing flow of FHEmem for basic FHE operations. We propose several algorithm and software optimizations to generate optimized mappings of FHE applications onto FHEmem.

4.4.1 FHEmem Data Layout

In PIM acceleration, data layout is critical that determines the detailed computation and data

movement. Figure 4.6 shows the optimized data layout in FHEmem. Each ciphertext contains a group of RNS polynomials, each of which is a vector of N b-bit integers. To exploit the high throughput of PIM, we distribute RNS polynomials of a ciphertext, including the original RNS terms (RNS_{ql}) and the special terms (RNS_{pk}) used for key switching, across multiple banks using a round-robin method. The figure shows an example of allocating two original RNS terms and two special RNS terms in a bank.

Layout in subarray groups

We divide subarrays into subarray groups, which are basic memory partitions for polynomials. Specifically, each subarray group contains a continuous set of subarrays (e.g., 2 subarrays) which is a 2D array of mats (e.g., 2×2). The 2D distribution allow FHEmem to balance the inter-mat data movements during various FHE operations (esp. NTT). FHEmem distributes coefficients of a polynomial across the mat array in an interleaved way, similar to previous work [132], to efficiently support automorphism (Section 4.4.5). In our setting, each subarray group contains 16 subarrays (16 × 16 mat), requiring each mat use 32 rows to store 256 64-bit coefficients.

Layout for computation

For a computation using two RNS polynomials, we align both polynomials in the same column in a subarray. Each subarray group reserves rows for operand polynomials used for computation, including key-switching keys, constant polynomials, and other ciphertexts. If a subarray group computes with two polynomials in different subarray groups, the memory issues data movements that may happen inside a bank or across different banks.

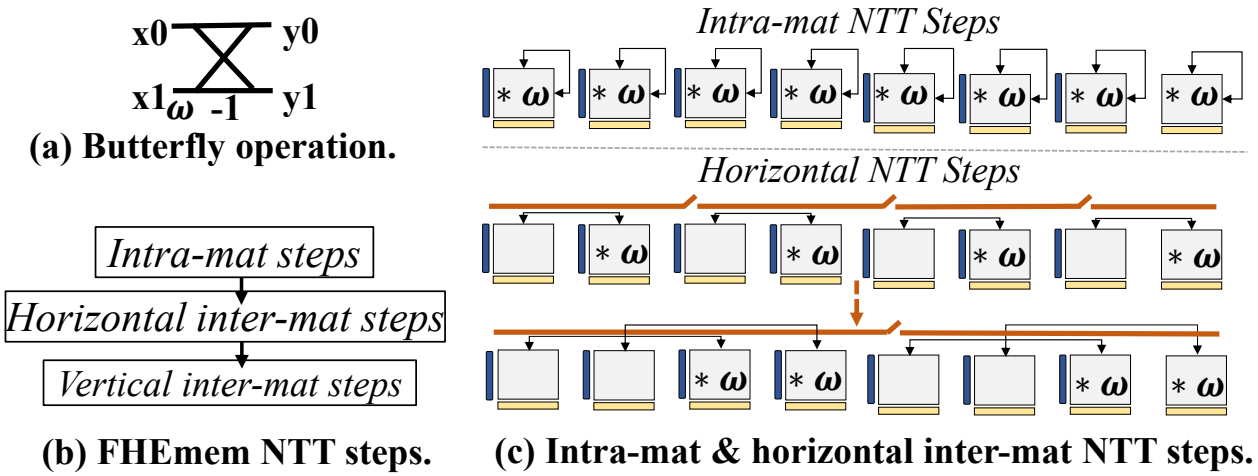
Layout for constants

In addition to RNS polynomial, we need to allocate rows for several constants for FHE operations, including `(i)NTT` twiddle factors, moduli, scaled inverse moduli, etc. To avoid duplicating twiddle factors across NTT steps that requires large memory, we store the vector of twiddle factors which

contains N-th roots of unity, in the same order as the polynomial coefficients. Before each (i)NTT stage k, FHEmem first dynamically computes the twiddle factor ω^{ik} for coefficient i by multiplying the twiddle factor in the previous stage with ω^{i}. For moduli, we keep one copy in each mat in the subarray group, so that each NMU can load the corresponding value independently during the computation.

4.4.2 Algorithm-optimized modular reduction

Each FHE arithmetic is followed by a modular reduction. We exploit the Montgomery algorithm [156] requiring two multiplications, one addition, and one subtraction. NMUs in FHEmem process n-bit multiplication using n serial additions. Therefore, we exploit algorithm optimization to significantly reduce the latency and energy for modular arithmetic. Specifically, we select moduli that are friendly to serial computations while satisfying security requirements and (i)NTT. We exploit the moduli selection technique proposed in previous works [157, 158] that select moduli has the form of $2^{b} \pm 2^{sh_1} \pm 2^{sh_2} \pm .. \pm 2^{sh_{h-2}} \pm 1$, where h is called hamming weight. By using a modulus with a hamming weight of h, we only need to issue h additions during the multiplication, hence reducing the addition steps from n to h. The hamming weight optimization only applies to computations with constant, including the multiplication with modulus and the multiplication with reduction factor in Montgomery reduction. The advantage of Montgomery reduction over Barrett reduction [159] is that both reduction factor and modulus in Montgomery reduction can have a low hamming weight, and it only requires single bit-length computation. We note that prior FHE accelerators [6, 131] also adopt a similar optimization that customized the modular multiplier for Montgomery-friendly moduli. However, their modular multipliers cannot process computations using moduli with different characteristics (e.g., required by other applications). FHEmem provides more flexibility by using addition as the basic computing step.



(a) Butterfly operation. **(b) FHEmem NTT steps.** **(c) Intra-mat & horizontal inter-mat NTT steps.**

Figure 4.7: NTT support in FHEmem.

4.4.3 FHEmem NTT

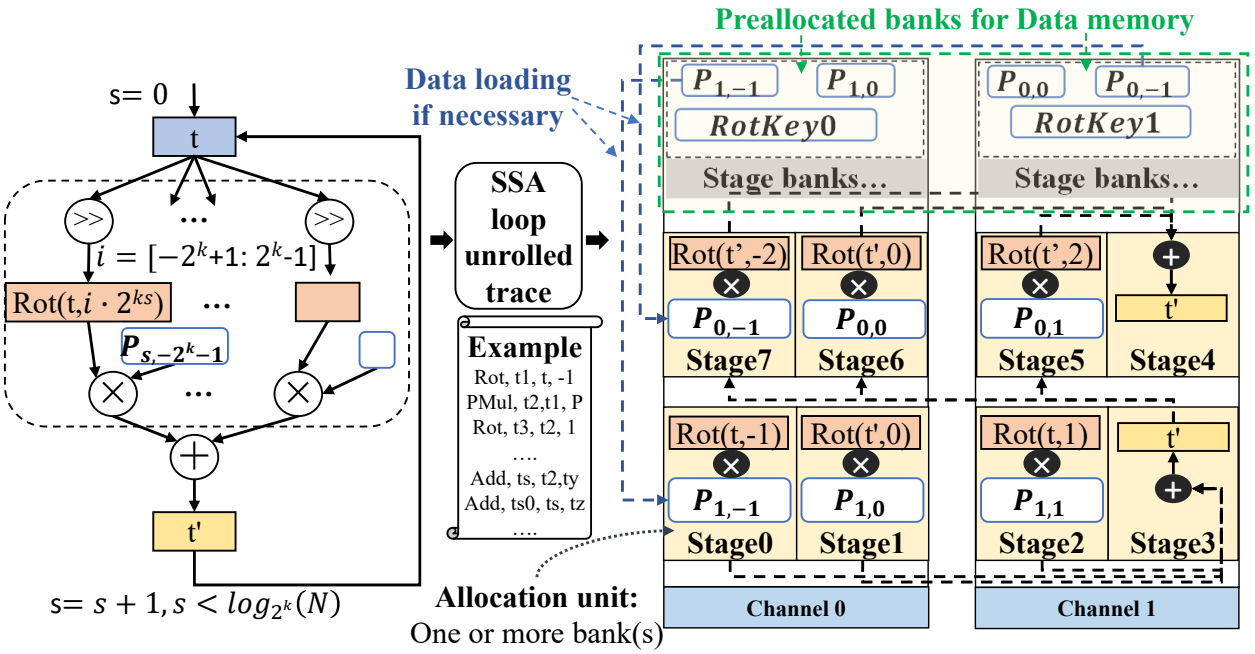
`NTT` executes several steps of permutation and computations. Each `NTT` step requires several butterfly operations (Figure 4.7(a)) on pairs of coefficients, that multiply the coefficients with the twiddle factors, permute the coefficients, and then update the coefficients. FHEmem processes each (i)NTT operation in three stages, intra-mat, horizontal inter-mat, and vertical inter-mat, depending on the butterfly stride of each (i)NTT step (Figure 4.7(b)). For intra-mat steps, where coefficients of each butterfly operation are in the same mat, NMUs in a subarray group independently process computation and permutation. The horizontal inter-mat steps exchange the coefficients between mats in the same row, for which FHEmem uses HDLs for efficient data transfers, as shown in Figure 4.7(c). Specifically, FHEmem turns on/off the switches of NMUs on HDLs, where the connected segments can independently transfer data. Data transfers using the same connected segment are scheduled sequentially. Therefore, as the number of connected segments changes over (i)NTT stages, the transfer latency of different (i)NTT stages varies. The vertical inter-mat NTT steps are processed similarly to the horizontal steps but using MDLs. The key novelty of FHEmem on (i)NTT operations is that FHEmem does not introduce complex butterfly networks in the memory. Instead, FHEmem exploits the existing DRAM internal links (i.e., MDLs and LDLs) with efficient customizations (i.e., NMU, switches, HDLs).

4.4.4 Base Conversion

BConv is a costly but frequent operation in FHE. As introduced in Section 4.2, to generate each special RNS polynomial of `BConv` (with modulus pk), each input RNS polynomial with modulus qi first multiplies $[\hat{qi}^{-1}]_{qi}$ and $[\hat{qi}]_{pk}$. Such partial products are reduced to each special RNS polynomial pk. FHEmem parallelizes multiplications in different subarray groups with different input polynomials. To reduce the partial products, FHEmem first accumulates partial products in the same bank using NMUs and MDLs because partial products of different polynomials in a bank are aligned either in the same subarray group or in different subarray groups in the vertical direction. Therefore, the intra-bank accumulation can be processed in an adder-tree manner by exploiting switches in MDLs. FHEmem processes the final reduction in the bank of the output polynomial, requiring data transfers of partial products from all other banks. FHEmem handles such inter-bank data movements using the customized interconnection (Section 4.3.3). To parallelize the computation, each bank processes different output polynomials simultaneously. FHEmem determines the optimized schedule based on the number of banks used for the ciphertext, the number of input/output RNS polynomials, and the underlying interconnect structure.

4.4.5 Automorphism

Automorphism is a process that permutes the coefficients of a polynomial by using Galois group. FHEmem supports automorphism based on the observation from BTS [132]: the interleaved coefficients (Section 4.4.1) in the same tile (mat in FHEmem) will be mapped to a single tile after automorphism. FHEmem further extends this idea to interleave coefficients in one more dimension, memory row, where the column c of row z of a mat (x,y) stores coefficients with the indices $cN_xN_yN_z + zN_xN_y + yN_x + x$. With such coefficient mapping, the automorphism only requires three steps: permutations in each row, vertical inter-mat permutation, and horizontal inter-mat permutation. FHEmem can handle the first step in NMU and the last two steps using MDLs and HDLs respectively.



(a) Example DAG in *CoefToSlot* step. **(b) In-memory pipeline (first 8 stages, k=1)**

Figure 4.8: In-memory pipeline generation.

4.4.6 Application Mapping Framework for FHEmem

FHEmem can provide large throughput when fully utilizing memory. However, it is not trivial to map a full FHE program to FHEmem hardware with high utilization. Thus, we propose a mapping framework that generates data layout and scheduling in a pipeline manner that can fully utilize the memory to process multiple input data in parallel.

Framework Overview

Figure 4.8 shows an example pipeline for the CoefToSlot step in CKKS bootstrapping. The input of our framework is an intermediate representation extracted from the real FHE program. Our framework generates a trace of FHE operations (e.g., HMul, HAdd, and HRot) in the static single-assignment (SSA) form while unrolling all loops. Our framework then divides the operation trace into multiple pipeline stages. The example shows the first 8 pipeline stages for CoefToSlot in a simplified HBM model with 2 channels. After computation on a stage, the allocated memory needs to transfer data to the memory that processes the pipeline steps with a data dependency. Therefore,

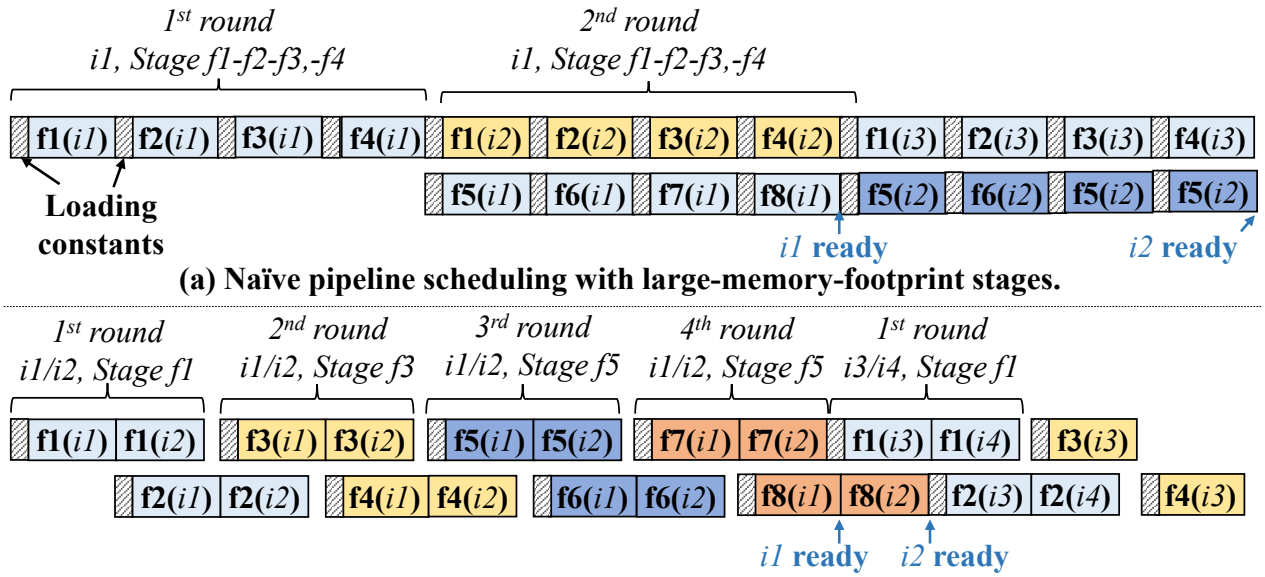


Figure 4.9: Load-save pipeline optimization in two memory partitions.

the latency of each pipeline stage includes loading time, computation time, and transfer time. Our framework aims to minimize the latency of the bottleneck stage in the pipeline.

Memory Allocation for Pipeline Stages

Our framework allocates each stage to a basic allocation memory unit, whose size is determined by the FHE parameter setting including the polynomial degree, ciphertext scaling factor, and ciphertext moduli. In Figure 4.8, the basic memory unit is one bank. To process a stage, we need sufficient memory to support the data layout shown in Section 4.4.1 for the input and output data. Extra memory is needed for constant data (e.g., evk and plaintexts). The ideal case is each allocation unit can hold all data for the stage. When a memory allocation unit cannot hold all data, we store constant data in a reserved memory location called data memory. When data is stored in the data memory, all operations (across all stages) requiring the data need to dynamically load it. We reduce the memory footprint by storing data in one place, instead of duplicating them in different memory locations.

Table 4.2: Architectural parameters.

HBM configuration	8-high HBM2E (16GB/stack)@10nm
Memory organization	#banks/pseudo-channel=8, #pseudo-channels/stack=32
Bank specification	64MB, row_size=1kB, 512*512 mats
Data transfer	inter-bank NoC = 256-bit
Timing (ARx1)	$tRRD : 2ns, tRAS : 29ns, tRP : 16ns, tFAW : 12ns$
Energy @10nm (ARx1)	$row\_act : 413pJ$, $pre\_gsa : 0.69pJ/b$ $post\_gsa : 0.53pJ/b$, $IO : 0.77pJ/b$

Load-save Pipeline

A naive way of pipeline generation is to divide the FHE program into n stages, where n is the number of available allocation units in memory. However, for large applications, each stage may require a large memory footprint for operations, leading to frequent constant loading. As shown in Figure 4.9(a), a stage has 4 operations, and every operation needs to load constants because memory is occupied by the constants of previous operations. In FHEmem, we propose a load-save pipeline by dividing operations into fine-grained stages with a small enough footprint. The fine-grained stages are allocated to different memory in a round-robin manner, requiring multiple rounds to process all stages for an input. In each round, the memory only creates a pipeline with part of the program (f1 and f2 in the first round in Figure 4.9(b)). It runs through a batch of input with only 1 data loading at the beginning of each round. Each memory loads the next round of stages when the current input batch is completed. The load-save pipeline minimizes the data loading while still fully utilizing the memory for computation.

4.5 Experimental Setup

4.5.1 Hardware Evaluation

Memory Technology: The basic architecture of FHEmem is similar to HBM2E [9, 99]. The system configuration is shown in Table 4.2. Each HBM2E stack has 16GB with 16 physical channels. We scale energy and power values from 22nm used in previous work [102] to 10nm (shown in

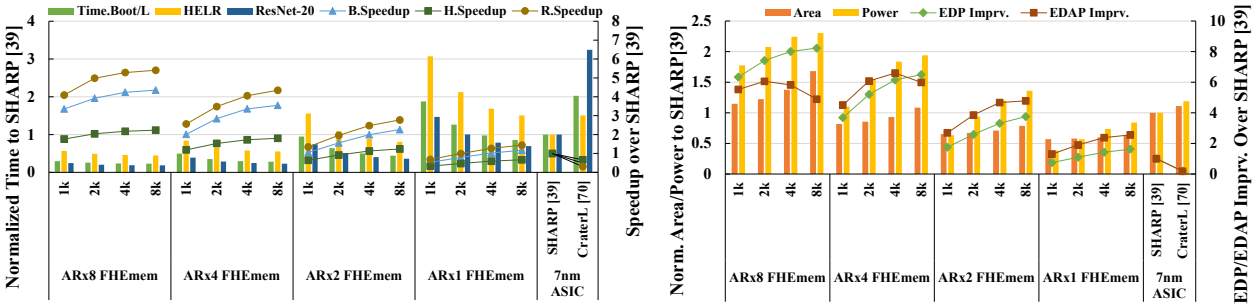


Figure 4.10: Efficiency comparison across different FHEmem configurations and CMOS-ASIC accelerators [5, 6]. All values are normalized to SHARP [5].

Table 4.2), based on the recent HBM2E technology [9]. We follow the method of Vogelsang [160] to calculate the scaling factors for energy. We assume 16 physical channels on a stack are connected by a crossbar on PHY where each bidirectional link is 64-bit wide (bisection bandwidth=64GB/s per stack). We also add stack-stack links for scaled-up systems, commonly used in memory-centric architecture [68, 161–163]. FHEmem has two HBM2E stacks to support 32GB memory. We exploit the remaining signaling links on HBM2E for stack-stack connection so the inter-stack bandwidth is also 256GB/s.

Hardware Modeling: To evaluate the area and power of customized components, we synthesize our design in 45nm technology using Nangate Open Cell Library. We model all other CMOS components (including buffers and interconnects) in Cacti [60] at 32nm technology. We scale all values to the 10nm technology with the scaling factors calculated from previous work [164]. We estimate the delay, power, and area overhead of integrating CMOS-ASIC and DRAM technologies based on the difference in number of metal layers and complexity of the customized logic [12,13,165].

Simulation: We generate FHE operation traces from software implementations of different workloads, and our mapping framework optimizes the trace and generates PIM instructions for simulation. Our in-house simulator adopts a cycle-accurate trace simulation based on the standardized DRAM latency constraints, similar to Ramulator [103]. We model control logic at different levels in the DRAM hierarchy.

4.5.2 Workloads

Logistic Regression (HELR) [140]: This workload has 30 iterations of homomorphic logistic regression where each iteration trains 1024 samples with 256 features as a batch. The multiplication depth is deep, requiring several bootstrappings.

ResNet-20 [34]: The ResNet-20 is a homomorphic neural network inference for one CIFAR-10 image classification. The network is deep with multi-channel convolutions, matrix multiplications, and approximated ReLU function.

Bootstrapping [145,166]: We evaluate the bootstrapping algorithm using a similar framework as previous work [135, 145], which requires 15/30 levels of bootstrapping. We adopt the minimum-key method used in previous work [135] to reduce the rotation keys used in bootstrapping.

4.5.3 FHE Parameters and Evaluation

We evaluate the efficiency of FHEmem on a 128-bit security FHE parameter setting chosen from Lattigo [167], where $logN = 16$, $L = 23$, $dnum = 4$, and $logPQ = 1556$, similar to prior accelerators [132, 135]. Each polynomial is decomposed into 40-61 bit RNS terms, where FHEmem allocate 64-bit for each coefficient. For all workloads, we measure the time using the maximum time across all pipeline stages, indicating the amount of time that we can finish an input when the pipeline is full. In addition, we consider the number of pipelines that can be processed simultaneously in the system when the program (only the bootstrapping workload) cannot fully utilize the memory capacity (e.g., 32GB).

4.6 Evaluation

4.6.1 Comparison to Previous FHE Accelerators

Figire 4.10 compares FHEmem with two state-of-the-art ASIC FHE accelerators (CraterLake [6], and SHAPR [135]). We explore two design choices of memory organization that play important roles

in the performance, power, and area of FHEmem: the aspect ratio of DRAM mat (AR), and the width of adders in a subarray (if each NMU has 2 64-bit adders, a subarray with 16 mats has 2k-width adders). As discussed in Section 4.2.4, high-AR architecture has better performance and energy efficiency than low-AR architecture, but incurring significant area overhead. The width of adders determines the performance of arithmetic computing - wide adder designs support faster computing while requiring a larger area than narrow adders. For the chip area of prior ASIC accelerators, we add the area of 32GB HBM2E ($2\times110mm^2$) for a fair comparison.

Performance

FHEmem shows superior performance over prior FHE accelerators. Specifically, ARx8-8k (lowest EDP) is 4.4× (8.8×), 2.2× (3.4×), and 5.4× (17.5×) faster than SHARP [5] (CraterLake [6]) on bootstrapping, HELR, and ResNet-20 respectively. ARx4-4x (lowest EDAP) is 3.4× (6.8×), 1.7× (2.6×), and 4.1× (13.2×) faster than SHARP [5] (CraterLake [6]) on bootstrapping, HELR, and ResNet-20 respectively. The less significant performance improvement in HELR results from the low portion of bootstrapping which is significantly optimized by adopting the minimum-key optimization [5, 135].

Efficiency

We then compare the power and area efficiency of FHEmem and ASIC accelerators. As compared to SHARP, ARx8-8k improves EDP and EDAP by 8.2× and 5.1×. However, ARx8-8k requires 1.6× larger area and 2.3× higher power than SHAPR. ARx4-4k improves EDP and EDAP of SHAPR [5] by 6.2× and 6.9×, with 0.9× area and 1.8× power consumption. ARx2-2k is a configuration that provides the best performance while using less area (0.65×) and power (0.94×) than SHAPR. For performance, ARx2-2k is 1.56×, 0.92×, and 1.96× faster than SHAPR, leading to 2.59× and 3.96× EDP and EDAP improvement.

Analysis of FHEmem Benefits

As compared to ASIC accelerators, FHEmem provides a higher throughput due to the efficient in-memory computation and large intra-memory bandwidth. For instance, ARx4-4k FHEmem has 16 million 64-bit adders. Considering the cost of DRAM row activations, data transfers, subarray-level parallelism, and 500MHz frequency of additions, the effective throughput of ARx4-4k for 64-bit multiplication is around 637.61 TB/s. During multiplication, the adders consume most of the energy because row activation energy is amortized for the entire row, and data transfer is energy-efficient due to the short wire length. Therefore, the energy efficiency of FHEmem computation is similar to the modular multiplier used by FHE accelerators (slightly higher due to the DRAM-CMOS integration). For the internal bandwidth of NTT, ARx4 FHEmem supports 256-bit link (500MHz) for each of 512 subarrays in a bank. Considering up to half of the subarrays can transfer data simultaneously during NTT, the peak internal bandwidth for NTT is 2048 TB/s in 32GB ARx4 FHEmem. For the slowest NTT step, the internal bandwidth drops by 16× (128 TB/s). As a comparison, SHARP [5] has around 24K 36-bit multipliers running at 1GHz, leading to 221.18 TB/s throughput. Furthermore, the on-chip memory resources in SHARP support 72TB/s bandwidth.

4.6.2 Comparison across different FHEmem Configurations

As shown in Figure 4.10, There is a significant difference in power and area between different FHEmem configurations. For example, the largest FHEmem (ARx8-8k) requires 642.32mm^2 chip area and 218W power, while the smallest FHEmem (ARx1-1k) only requires 223.81mm^2 chip area and 36.24W power. As a reference, the commercial 2-stack HBM2E has a chip area of 220mm^2 [9] and the power budget of a conventional HBM system is 60W [102]. We note that the power budget of conventional HBM is different from the accelerator design targeted in this work, where high power consumption is reasonable if it meets the thermal requirement (e.g., 10W/cm^2/layer [168]).

Based on the results, high-AR FHEmem provides higher performance than low-AR designs because increasing AR can increase both compute and data movement throughput inside a bank.

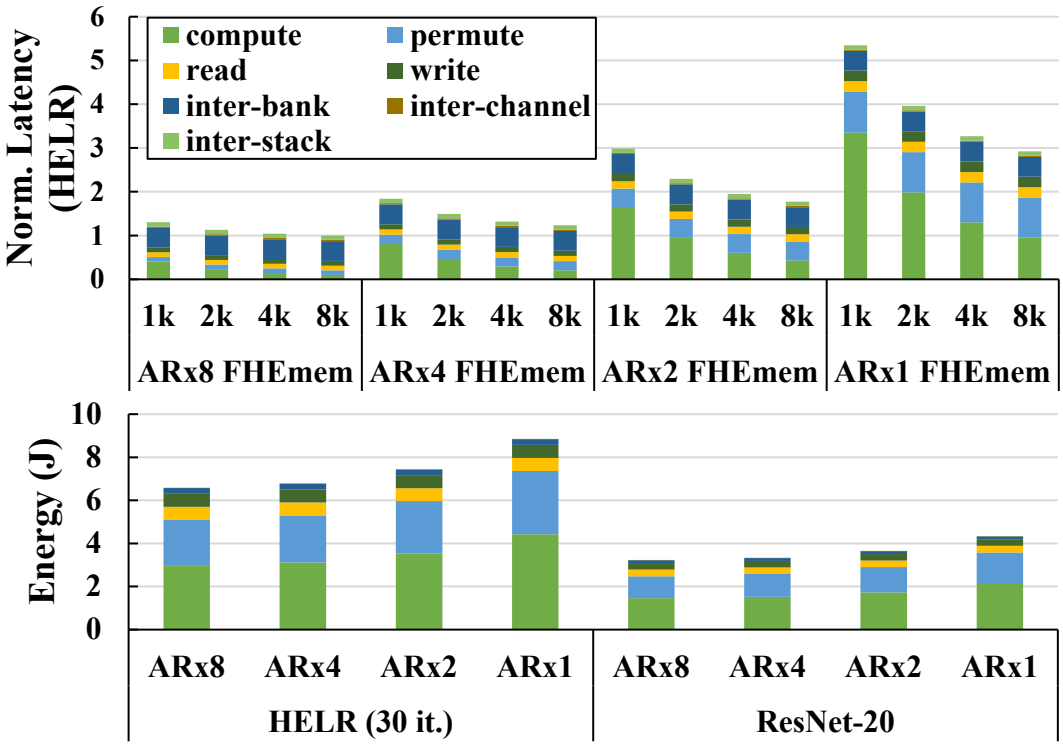


Figure 4.11: The latency and energy breakdown.

For ARx1 and ARx2 FHEmem configurations, doubling AR can provide 1.57× to 1.98× speedup because the execution is compute-bound. For ARx4, doubling AR only provides 1.23× to 1.67× speedup. Increasing adder-width exhibits a similar trend where the effect of increasing compute resources diminishes for high-throughput architectures.

To find the most cost-efficient FHEmem design, we evaluate energy-delay-product (EDP) and energy-delay-area-product (EDAP) for different FHEmem configurations. For EDP, the trend follows the performance, where the largest FHEmem (ARx8-8k) gives the lowest EDP. When considering the area cost, different ARs favor different adder widths. Specifically, ARx8 and ARx4 exhibit the lowest EDAP at 2k and 4k adder-width respectively. The configuration with the lowest EDAP (ARx4-4k) is 1.34× more efficient than ARx8-8k.

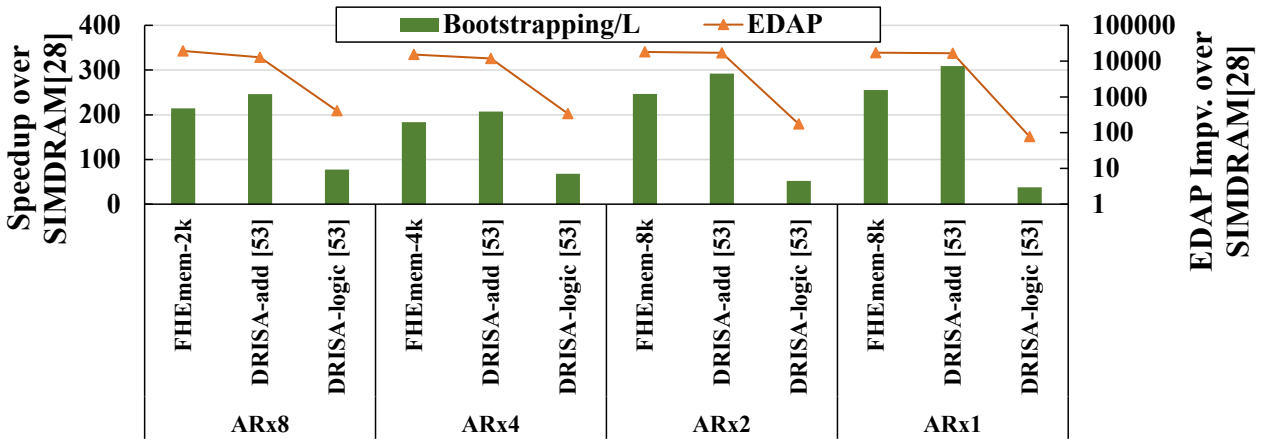


Figure 4.12: Comparison between PIM technologies.

4.6.3 FHEmem Latency and Energy Analysis

Figure 4.11 shows the latency and energy breakdown of different FHEmem configurations. We accumulate all latency values, considering the parallel processing, across all memory banks for the latency breakdown. We divide all operations into 7 categories, including computation (subarray activation/precharge, operand transfer, and addition), permutation (inter-mat transfer), read/write (activation and precharge for data transfers), and inter-bank/channel/stack IO traffics.

The breakdown results provide several key insights of FHEmem. First, in low-AR FHEmem, the latency is dominated by computation and permutation because of the limited throughput of computation and intra-bank data movements. Increasing AR can effectively reduce the latency for both computation and permutation latency. Furthermore, increasing the adder width can effectively reduce the computation latency. However, in high-AR FHEmem, the data movement becomes the performance-dominant operation, especially inter-bank data movements (mainly caused by BConv). This proves the necessity of customized inter-bank links. We analyze the detailed effect of optimizations in Section 4.6.6. For energy consumption, FHEmem consumes most of the energy on computation and permutation, which incurs intensive row-activation and intra-bank data movements.

Table 4.3: Area and power of FHEmem (16GB HBM2E).

ARx4 HBM	**DRAM Cell**	**Local WL Driver**	**Sense Amp**	**Row/Col Dec.s**
Area (mm^2)	56.54	26.15	45.63	0.39
	Center Bus	**Data Bus**	**TSV**	**Total**
	1.56	4.81	13.25	148.33
4K adder	**Horizontal DL**	**Adder&Latches**	**Bank Chain & Buf.**	
Area (mm^2)	14.13	30.43	0.065	
Power/Energy	5.3fJ/b (avg.)	15.86W	0.53pJ/b	

4.6.4 PIM Technologies

Figure 4.12 compares the efficiency of FHEmem to other PIM technologies, including SIMDRAM [11] and DRISA [13]. For a fair comparison, we use the proposed application mapping and customized data links in baseline PIM architectures, while evaluating the difference in processing. We implement an adder-less NMU for permutations in baselines. We select the most efficient (lowest EDPA) FHEmem for each AR. The results show FHEmem is 183.7× to 255.4× faster than SIMDRAM [11], and 2.76× to 6.75× faster than DRISA-logic [13]. Furthermore, FHEmem is at least 19,300× and 47× more efficient than SIMDRAM and DRISA-logic using EDAP. As compared to DRISA-add [13], FHEmem is 1.14× to 1.21× slower from ARx8 to ARx1 because DRISA's adders can directly access the sense amplifiers. However, FHEmem is 1.04× (ARx1) to 1.51× (ARx8) more efficient in EDAP because FHEmem does not introduce area overhead in mat. Furthermore, DRISA is more challenging to manufacture because the large adder area will affect the alignment of cost-optimized bitlines. As a comparison, FHEmem puts all customized logic outside the mat structure.

4.6.5 Overhead Analysis

Table 4.3 shows the area and power breakdown for our customized hardware components of FHEmem based on 1 HBM2E stack (16GB). We exploit the Cacti-7 [60] to generate the area breakdown of HBM and rescale the values to the published work [169]. The table shows a structure of ARx4 HBM and each subarray contains 4k-wide adders. All area values are for a single layer. The

large area mainly comes from the near-mat adders. The horizontal data links use the same technology as the global data lines with the consideration of energy efficiency (4× larger than local bit-lines). We extract the capacitance of material and scaling method from previous work [160] to calculate the energy consumed by data transfer. We note that the target of FHEmem is not a cost-optimized memory product, but a specialized accelerator for emerging applications that require high-throughput computation and/or efficient intra-memory data movements. However, the proposed design is still practical in terms of area and power overhead. First, unlike previous near-subarray PIM [13], FHEmem put the customized logic outside DRAM mat, avoiding the issue of aligning cost-efficient local bitlines. Second, FHEmem can still maintain a normal DRAM working temperature (under 85°C). Previous work [168] shows a 16-high compute-centric 3D memory can tolerate 10W/cm^2 per memory layer to keep DRAM temperature under the 85°C limit with a commodity-server active heat sink. For example, the power consumption and area of 8-high ARx4 4k FHEmem are 173.9W and 367mm^2, resulting in a power density of 5.92W/cm^2 (the highest power density in our exploration).

4.6.6 Evaluation of FHEmem Optimizations

We compare FHEmem with baseline systems enabling a subset of FHEmem optimizations, as shown in Figure 4.13.

Montgomery-friendly Moduli

The Montgomery-friendly moduli can reduce the number of addition steps, improving the computation performance. As shown in Figure 4.13, Base1 is 1.68× and 1.58× faster than Base0 on HELR and ResNet with ARx2-2k architecture. On Arx4 and Arx8 architectures, this optimization only improves the performance by 1.17× and 1.06×. However, Montgomery-friendly moduli can effectively reduce energy consumption by 1.75× because computation is energy-dominant across all FHEmem architectures (Figure 4.11).

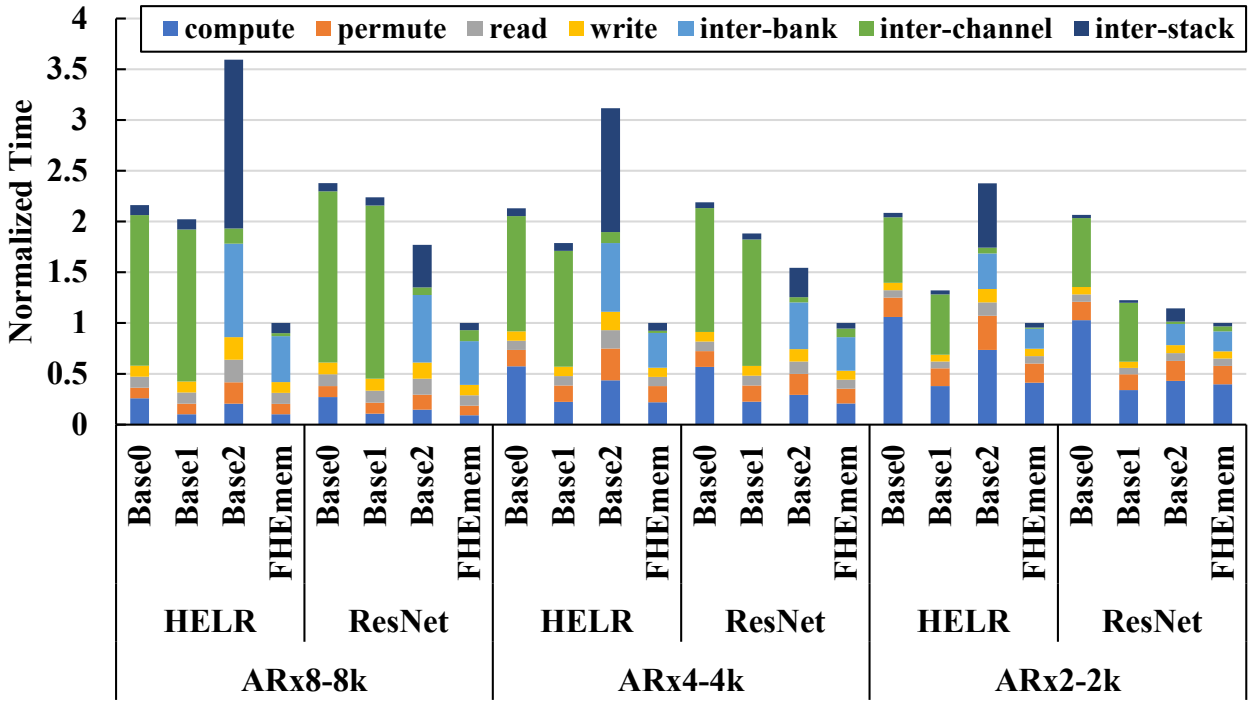


Figure 4.13: Effect of different optimizations including (1) Montgomery-friendly moduli, (2) inter-bank connection network, and (3) load-save pipeline mapping. Base0 uses (3), Base1 uses (1)+(3), and Base2 uses (1)+(2).

Interconnect Network

The comparison between FHEmem and Base1 shows the efficiency of the proposed inter-bank network, where Base1 uses the existing HBM channel IO for all inter-bank data movements. Based on our results, the proposed inter-bank network can improve the performance by 1.31×, 1.86×, and 2.12× on ARx2, ARx4, and ARx8 respectively. The inter-bank network can reduce the latency of related data movement by 3.2× on average.

Load-save Pipeline Mapping

The comparison between FHEmem and Base2 shows the efficiency of the load-save pipeline mapping. For HELR, load-save pipeline mapping improves the performance by 3.59× (1.77×), 3.12× (1.54×), and 2.38× (1.15×) on ARx8, ARx4, and ARx2, respectively, for HELR (ResNet). The significant performance improvement results from the reduction of frequent data loading,

especially loading data from a remote stack.

4.7 Related Work

Several FHE-specific accelerators have been proposed recently in the architecture community [5, 6, 131–133, 135]. CraterLake [6], a followup work of F1 [131], adopts wide vector processor with specialized high-throughput units for `BConv` and on-chip key generation. BTS [132] exploits relatively low throughput function units with large inter-PE crossbar network. ARK [135] uses an algorithm-hardware co-design to significantly reduce the off-chip bandwidth for bootstrapping *evk* and plaintext polynomials. SHARP [5] further improves the performance ARK by using low bit-precision data path (36-bit vs. 64-bit in ARK). However, their technique does not apply to general *evk* and ciphertexts, leading to the same memory issues, as analyzed in Section 4.2. FHEmem exploits the large internal memory bandwidth of memory-based acceleration to unleash the processing throughput in a more area and power-efficient way.

CryptoPIM [133] is a ReRAM-based PIM accelerator with customized interconnect for NTT operations, lacking the support for more general FHE operations. In-storage processing is another promising technology to accelerate big-data applications [170–172]. INSPIRE is an in-storage processing system for private database queries based on FHE by integrating FHE logic (e.g., NTT and permutation) in the SSD channels. INSPIRE only supports small FHE parameters (i.e., N=4096) and its throughput is limited by the number of SSD channels. MemFHE [136] is a ReRAM-based PIM accelerator with the customized data path for bit-level TFHE scheme, not applicable to CKKS and other packed FHE schemes.

4.8 Conclusion

This chapter comprehensively investigates the end-to-end software mapping of FHE applications onto PIM architectures and reveals several key inefficiencies of existing PIM acceleration for FHE.

The proposed techniques in this chapter advance the research on both FHE acceleration and efficient general PIM acceleration using software-hardware co-design, which can be used by future work on highly efficient private computing as well as next-generation memory system design. Our FHEmem is a novel FHE accelerator based on a specialized PIM architecture with software-hardware co-design that is $6.9\times$ more efficient than prior art FHE accelerators [5, 6].

Chapter 4, in full, is currently being prepared for submission for publication of the material, Minxuan Zhou, Yujin Nam, Pranav Gangwar, Weihong Xu, Arpan Dutta, Kartikeyan Subramanyam, Chris Wilkerson, Rosario Cammarota, Saransh Gupta, and Tajana Rosing. The dissertation author was the primary investigator and author of this material.

Chapter 5

Summary and Future Work

5.1 Summary of Thesis

This thesis introduces several software-hardware co-design techniques for efficient PIM acceleration of emerging applications that require both extensive parallelism and high memory bandwidth. Chapter 2 investigates the design space and performance model of different software mappings on PIM architecture and applies it to DNN models. It introduces a method to transform the conventional representation of DNN operators (i.e., nested loop) into a detailed performance and data layout model for PIM acceleration, and then proposes an efficient framework to optimize the extremely complicated problem of mapping a whole DNN model onto PIM architecture. The experiments on various DNN models show that the proposed framework can provide 1.2× to 9/9× speedup on various PIM accelerators [3, 4, 13]. Chapter 2 also explores different interconnect architectures used for PIM acceleration and finds that lightweight modifications in the intra-memory interconnect can further boost the performance of DNN acceleration with data layout optimization.

Chapter 3 expands the discussion of software-hardware co-design to an emerging family of machine learning models, Transformers, which feature different characteristics from conventional DNN models. Transformer-specific characteristics result from the self-attention mechanism and the computations based on large matrices, posing challenges to both existing software mapping methods

and PIM hardware architecture. To tackle these challenges, Chapter 3 proposes a novel software mapping based on the dependency of input tokens and an enhanced PIM architecture with lightweight hardware modifications that are within 4% of conventional HBM for Transformer operations. The proposed software-hardware co-design achieves up to 10× performance improvement over state-of-the-art PIM accelerators [11, 12] with optimized processing flow [88].

To extend the research to non-ML applications, Chapter 4 introduces a software-hardware co-design of PIM for fully homomorphic encryption (FHE), which is a challenging application with complicated operations on large polynomials with high bit-precision coefficients. Experiments on existing PIM architectures lack the hardware support for complex FHE data movement patterns and fail to match the speed of existing FHE accelerators due to the intrinsic inefficiency of PIM for high bit-precision arithmetic. Thus, Chapter 4 proposes a novel type of PIM architecture that uses near-bank processing to improve throughput and energy efficiency of in-memory arithmetic. Near-bank logic also adds support for data permutation inside the memory bank. On the software level, Chapter 4 proposes an optimized end-to-end processing flow for FHE with a pipeline-based application mapping framework to fully utilize PIM hardware for FHE applications. With the proposed software-hardware co-design, the proposed PIM acceleration achieves at least 4× speedup and 7× efficiency improvement over prior-art FHE ASIC accelerators [5, 6].

Even though each chapter conducts research in the context of specific application domain, the proposed software-hardware co-design techniques are general because all investigated operations (e.g., convolution, matrix multiplication, polynomial operations, etc.) are common kernels widely used in many emerging application domains. Furthermore, the proposed techniques maintain the conventional memory interface, making the design very flexible. The proposed techniques provide valuable insights into both software mapping and hardware design for applying PIM acceleration to implement more efficient computer systems.

5.2 Future Work

There are several interesting directions to explore in PIM research. First, most PIM accelerators to date have been very specific to a relatively few applications. A broader set of applications should be explored. Second, how to design PIM-based systems is still an open problem. Third, there is a lack of full-stack system support for PIM. Future work should address these three key directions.

5.2.1 PIM for broad emerging applications

The extensive parallelism and high internal bandwidth of PIM are promising for the acceleration of emerging applications in addition to machine learning and fully homomorphic encryption that are discussed in this thesis. As shown in this thesis, the applications suitable for PIM acceleration should have highly parallel operations with large memory footprints, which are common characteristics of many challenging applications including graph processing, bioinformatics, physical simulation, and image processing. These applications feature various computation and data movement patterns. The PIM designer needs to investigate the software mapping of application operations to the PIM hardware and design an efficient data layout and processing flow that balances the computation and data movements. Furthermore, the innovation of hardware support is also critical in the case that the target application features operations that cannot be efficiently processed by existing PIM hardware. As future systems become more and more heterogeneous, a data center can adopt different types of PIM accelerators (with other types of accelerators) to provide superior performance for various applications.

In addition to the application-specific acceleration, designing a unified PIM architecture that can accelerate applications from several fields is a more cost-efficient research direction. Such research builds on various application-specific PIM accelerators and abstracts a common framework in both software and hardware to support all considered applications with minimized cost. For example, one can extend the tensor-based PIM acceleration (e.g., DNN and image processing) with support for various dependency scenarios in irregular applications (e.g., graph processing) to create

a unified solution for these applications. To design such systems, new domain-specific languages and cost-efficient versatile PIM architectures are both critical.

5.2.2 Heterogeneous systems with PIM

This thesis focuses on the PIM acceleration based on the hardware with a homogeneous technology. However, as discussed in Chapter 1, there have been various PIM technologies providing specific advantages on performance, power, and area. A heterogeneous system with a variety of PIM technologies at different levels in the memory and storage hierarchy with different types of memory devices may provide the most balanced solution for applications. The software stack for the heterogeneous PIM system requires mapping optimization on each of the underlying PIM technologies. Optimizations are needed that determine the execution locations for the applications or parts of an application in a heterogeneous system. On the hardware side, it is an open problem how to design the architecture of such a heterogeneous system, including the technologies, architectural configurations, and the interconnect.

5.2.3 Generic software stack for PIM

The software stack is a key to the success of novel hardware platforms. However, there lacks such a full-stack solution nowadays to enable various PIM technologies in real-world systems. Most current research work only adopts application-specific interfaces to map applications to hardware [53, 68, 88]. The system support built on the conventional system stack only supports the PIM architecture with conventional execution model [7]. The development of a generic software stack for various PIM architectures is critical. As discussed above, it is important to maintain a generic hardware interface that can support different customization in a unified architecture. Such standardization would also help the development of a generic software stack, consisting of the front-end programming interface, the compiler, and the instruction set of architecture. Specifically, the front-end programming interface, such as libraries in conventional languages or domain-specific

languages, enables application developers to exploit the underlying PIM hardware. The PIM compiler can optimize the application code and generate efficient hardware instructions.

Bibliography

[1] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.

[2] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 369–383, 2020.

[3] 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.

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

[5] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[6] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 173–187, Association for Computing Machinery, 2022.

[7] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 231–244, Association for Computing Machinery, 2022.

[8] 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, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops 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, 2021.

[9] C. Oh, K. C. Chun, Y. Byun, Y. Kim, S. Kim, Y. Ryu, J. Park, S. Kim, S. Cha, D. Shin, J. Lee, J. Son, B. Ho, S. Cho, B. Kil, S. Ahn, B. Lim, Y. Park, K. Lee, M. Lee, S. Baek, J. Noh, J. Lee, S. Lee, S. Kim, B. Lim, S. Choi, J. Kim, H. Choi, H. Kwon, J. J. Kong, K. Sohn, N. S. Kim, K. Park, and J. Lee, "22.1 a 1.1v 16gb 640gb/s hbm2e dram with a data-bus window-extension technique and a synergetic on-die ecc scheme," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 330–332, 2020.

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

[11] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, *SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM*, p. 329–345. New York, NY, USA: Association for Computing Machinery, 2021.

[12] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 372–385, 2020.

[13] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 288–301, 2017.

[14] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pp. 100–113, 2019.

[15] M. F. Ali, A. Jaiswal, and K. Roy, "In-memory low-cost bit-serial addition using commodity dram technology," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 155–165, 2019.

[16] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 185–197, 2013.

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

[18] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "14.2 a compute sram with bit-serial integer/floating-point operations for programmable in-memory vector acceleration," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 224–226, 2019.

[19] L. Song *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA'17*, pp. 541–552, IEEE, 2017.

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

[21] C.-X. Xue, J.-M. Hung, H.-Y. Kao, Y.-H. Huang, S.-P. Huang, F.-C. Chang, P. Chen, T.-W. Liu, C.-J. Jhang, C.-I. Su, W.-S. Khwa, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, Y.-D. Chih, T.-Y. J. Chang, and M.-F. Chang, "16.1 a 22nm 4mb 8b-precision reram computing-in-memory macro with 11.91 to 195.7tops/w for tiny ai edge devices," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 245–247, 2021.

[22] J. Kang, M. Zhou, A. Bhansali, W. Xu, A. Thomas, and T. Rosing, "Relhd: A graph-based learning on fefet with hyperdimensional computing," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 553–560, IEEE, 2022.

[23] A. Kazemi, M. M. Sharifi, Z. Zou, M. Niemier, X. S. Hu, and M. Imani, "Mimhd: Accurate and efficient hyperdimensional inference using multi-bit in-memory computing," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2021.

[24] S. Dutta, H. Ye, W. Chakraborty, Y.-C. Luo, M. San Jose, B. Grisafe, A. Khanna, I. Lightcap, S. Shinde, S. Yu, *et al.*, "Monolithic 3d integration of high endurance multi-bit ferroelectric fet for accelerating compute-in-memory," in *2020 IEEE International Electron Devices Meeting (IEDM)*, pp. 36–4, IEEE, 2020.

[25] H.-W. Hu, W.-C. Wang, C.-K. Chen, Y.-C. Lee, B.-R. Lin, H.-M. Wang, Y.-P. Lin, Y.-C. Lin, C.-C. Hsieh, C.-M. Hu, Y.-T. Lai, H.-S. Chen, Y.-H. Chang, H.-P. Li, T.-W. Kuo, K.-C. Wang, M.-F. Chang, C.-H. Hung, and C.-Y. Lu, "A 512gb in-memory-computing 3d-nand flash supporting similar-vector-matching operations on edge-ai devices," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, pp. 138–140, 2022.

[26] 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," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[27] M. Imani, S. Gupta, Y. Kim, M. Zhou, and T. Rosing, "Digitalpim: Digital-based processing in-memory for big data acceleration," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 429–434, 2019.

[28] M. Zhou, M. Imani, Y. Kim, S. Gupta, and T. Rosing, "Dp-sim: A full-stack simulation infrastructure for digital processing in-memory architectures," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 639–644, IEEE, 2021.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[30] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *arXiv preprint arXiv:1906.08237*, 2019.

[31] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[32] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.

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

[34] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30039–30054, 2022.

[35] 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.

[36] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[37] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018.

[38] 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," *SIGARCH Comput. Archit. News*, vol. 42, pp. 269–284, Feb. 2014.

[39] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 609–622, IEEE Computer Society, 2014.

[40] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

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

[42] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[43] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.

[44] 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, 2017.

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

[46] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kusela, 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.

[47] 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*, vol. 38, no. 4, pp. 640–653, 2018.

[48] M. Zhou, M. Imani, S. Gupta, and T. Rosing, "Gas: A heterogeneous memory architecture for graph processing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[49] M. Zhou, M. Li, M. Imani, and T. Rosing, "Hygraph: Accelerating graph processing with hybrid memory-centric computing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 330–335, IEEE, 2021.

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

[51] B. C. Jang, Y. Nam, B. J. Koo, J. Choi, S. G. Im, S.-H. K. Park, and S.-Y. Choi, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Advanced Functional Materials*, vol. 28, no. 2, p. 1704725, 2018.

[52] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Gram: graph processing in a reram-based computational memory," in *IEEE Asia and South Pacific Design Automation Conference*, 2019.

[53] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *SIGARCH Comput. Archit. News*, vol. 45, pp. 751–764, Apr. 2017.

[54] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 2010.

[55] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[56] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[57] 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*, vol. 52, no. 1, pp. 127–138, 2016.

[58] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[59] "DDR3 SDRAM." `https://www.micron.com/products/dram/ddr3-sdram`.

[60] N. Muralimanohar *et al.*, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *MICRO'07*, ACM/IEEE, 2007.

[61] A. B. Kahng, B. Lin, and S. Nath, "Explicit modeling of control and data for improved noc router estimation," in *Dac design automation conference 2012*, pp. 392–397, IEEE, 2012.

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

[63] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[64] 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.

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

[66] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[67] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[68] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.

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

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

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

[72] 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, and D. S. Milojicic, "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*, ASPLOS '19, (New York, NY, USA), pp. 715–731, ACM, 2019.

[73] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 448–460, ACM, 2018.

[74] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 56–68, IEEE, 2019.

[75] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019.

[76] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *arXiv preprint arXiv:1802.04924*, 2018.

[77] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[78] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 751–756, IEEE, 2017.

[79] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerator arrays," in *26th IEEE International Symposium on High Performance Computer Architecture, HPCA*, pp. 22–26, 2020.

[80] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[81] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[82] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[83] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[84] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European Conference on Computer Vision (ECCV)*, pp. 213–229, 2020.

[85] G. Bertasius, H. Wang, and L. Torresani, "Is space-time attention all you need for video understanding?," *arXiv preprint arXiv:2102.05095*, 2021.

[86] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," *ArXiv*, vol. abs/2012.09852, 2020.

[87] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, *et al.*, "A$\hat{3}$: Accelerating attention mechanisms in neural networks with approximation," in *EEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 328–341, 2020.

[88] M. Zhou, G. Chen, M. Imani, S. Gupta, W. Zhang, and T. Rosing, "Pim-dl: Boosting dnn inference on digital processing in-memory architectures via data layout optimizations," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–1, 2021.

[89] M. Zhou, L. Wu, M. Li, N. Moshiri, K. Skadron, and T. Rosing, "Ultra efficient acceleration for de novo genome assembly via near-memory computing," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 199–212, 2021.

[90] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 356–371, 2020.

[91] A. K. Ramanathan, G. S. Kalsi, S. Srinivasa, T. M. Chandran, K. R. Pillai, O. J. Omer, V. Narayanan, and S. Subramoney, "Look-up table based energy efficient processing in

cache support for neural network acceleration," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 88–101, 2020.

[92] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 43–56, 2021.

[93] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "Mnnfast: A fast and scalable system architecture for memory-augmented neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 250–263, 2019.

[94] C.-C. Lee, C. Hung, C. Cheung, P.-F. Yang, C.-L. Kao, D.-L. Chen, M.-K. Shih, C.-L. C. Chien, Y.-H. Hsiao, L.-C. Chen, *et al.*, "An overview of the development of a gpu with integrated hbm on silicon interposer," in *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pp. 1439–1444, IEEE, 2016.

[95] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, "Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 831–840, 2018.

[96] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 116–126, 2021.

[97] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on fpgas: A data analytics perspective," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, 2020.

[98] 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.

[99] "JEDEC Standard JESD235: High Bandwidth Memory (HBM) DRAM," standard, JEDEC Solid State Technology Association, Virginia, USA, 2013.

[100] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Computer Architecture Letters*, vol. 31, no. 05, pp. 377–384, 1982.

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

[102] 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.

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

[104] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA'17*, pp. 1–12, 2017.

[105] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[106] J. Zhang, Y. Zhao, M. Saleh, and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in *International Conference on Machine Learning*, pp. 11328–11339, PMLR, 2020.

[107] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, (USA), p. 142–150, Association for Computational Linguistics, 2011.

[108] A. Cohan, F. Dernoncourt, D. S. Kim, T. Bui, S. Kim, W. Chang, and N. Goharian, "A discourse-aware attention model for abstractive summarization of long documents," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, (New Orleans, Louisiana), pp. 615–621, Association for Computational Linguistics, June 2018.

[109] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 1601–1611, Association for Computational Linguistics, July 2017.

[110] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 811–824, 2020.

[111] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *International Conference on Learning Representations*, 2020.

[112] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, *et al.*, "Big bird: Transformers for longer sequences," *arXiv preprint arXiv:2007.14062*, 2020.

[113] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[114] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "Turbotransformers: an efficient gpu serving system for

transformer models," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

[115] 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*, vol. 37, no. 11, pp. 2613–2622, 2018.

[116] M. Zhou, Y. Guo, W. Xu, B. Li, K. W. Eliceiri, and T. Rosing, "Mat: Processing in-memory acceleration for long-sequence attention," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 25–30, IEEE, 2021.

[117] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[118] K. Matsuoka, Y. Hoshizuki, T. Sato, and S. Bian, "Towards better standard cell library: Optimizing compound logic gates for tfhe," in *Proceedings of the 9th on Workshop on Encrypted Computing; Applied Homomorphic Cryptography*, WAHC '21, (New York, NY, USA), p. 63–68, Association for Computing Machinery, 2021.

[119] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 617–640, Springer, 2015.

[120] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 409–437, Springer, 2017.

[121] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 360–384, Springer, 2018.

[122] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1209–1222, 2018.

[123] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

[124] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[125] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*, pp. 868–886, Springer, 2012.

[126] "PALISADE Lattice Cryptography Library (release 1.11.5)." `https://palisade-crypto.org/`, 2021.

[127] "Microsoft SEAL (release 3.7)." `https://github.com/Microsoft/SEAL`, Sept. 2021. Microsoft Research, Redmond, WA.

[128] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, vol. 15, 2020.

[129] A. Sev-Snp, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, vol. 53, pp. 1450–1465, 2020.

[130] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, (New York, NY, USA), Association for Computing Machinery, 2016.

[131] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 238–252, Association for Computing Machinery, 2021.

[132] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," *arXiv preprint arXiv:2112.15479*, 2021.

[133] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[134] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[135] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1237–1254, 2022.

[136] S. Gupta, R. Cammarota, and T. v. Rosing, "Memfhe: End-to-end computing with fully homomorphic encryption in memory," *ACM Trans. Embed. Comput. Syst.*, nov 2022. Just Accepted.

[137] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, (New York, NY, USA), p. 235–240, Association for Computing Machinery, 2022.

[138] M. Zhou, W. Xu, J. Kang, and T. Rosing, "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1071–1085, IEEE, 2022.

[139] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and S. Hynix, "System architecture and software stack for gddr6-aim," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–25, 2022.

[140] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 9466–9471, 2019.

[141] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *Proceedings of the 39th International Conference on Machine Learning* (K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, eds.), vol. 162 of *Proceedings of Machine Learning Research*, pp. 12403–12422, PMLR, 17–23 Jul 2022.

[142] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *International Conference on Machine Learning*, pp. 812–821, PMLR, 2019.

[143] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," Tech. Rep. MSR-TR-2016-3, February 2016.

[144] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[145] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*, pp. 364–390, Springer, 2020.

[146] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog over-

head," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 465–482, Springer, 2012.

[147] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *International Conference on Cryptology and Network Security*, pp. 124–139, Springer, 2016.

[148] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *International Conference on Selected Areas in Cryptography*, pp. 347–368, Springer, 2018.

[149] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1295–1309, 2020.

[150] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 26–39, IEEE, 2021.

[151] M.-J. Park, H. S. Cho, T.-S. Yun, S. Byeon, Y. J. Koo, S. Yoon, D. U. Lee, S. Choi, J. Park, J. Lee, *et al.*, "A 192-gb 12-high 896-gb/s hbm3 dram with a tsv auto-calibration scheme and machine-learning-based layout optimization," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, pp. 444–446, IEEE, 2022.

[152] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 76–86, 2019.

[153] Y. H. Son, O. Seongil, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing memory access latency with asymmetric dram bank organizations," *SIGARCH Comput. Archit. News*, vol. 41, p. 380–391, jun 2013.

[154] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency dram: A low latency and low cost dram architecture," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 615–626, 2013.

[155] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 568–580, 2016.

[156] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[157] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 56–64, 2020.

[158] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, 2019.

[159] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings on Advances in Cryptology—CRYPTO '86*, (Berlin, Heidelberg), p. 311–323, Springer-Verlag, 1987.

[160] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 363–374, IEEE, 2010.

[161] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 145–155, IEEE, 2013.

[162] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, p. 204–216, IEEE Press, 2016.

[163] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "Syncron: Efficient synchronization support for near-data-processing architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 263–276, IEEE, 2021.

[164] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," *VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, p. m8, 2011.

[165] Y.-B. Kim and T. Chen, "Assessing merged dram/logic technology," in *1996 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, pp. 133–136 vol.4, 1996.

[166] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 587–617, Springer, 2021.

[167] "Lattigo v4." Online: `https://github.com/tuneinsight/lattigo`, Aug. 2022. EPFL-LDS, Tune Insight SA.

[168] J.-H. Han, R. E. West, K. Torres-Castro, N. Swami, S. Khan, and M. Stan, "Power and thermal modeling of in-3d-memory computing," in *2021 International Symposium on Devices, Circuits and Systems (ISDCS)*, pp. 1–4, 2021.

[169] "The ultimate guide to hbm2e implementation & selection." Online: `https://www.rambus.com/blogs/hbm2e/`, Nov. 2020. Rambus.

[170] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "Inspire: In-storage private information retrieval via protocol and architecture co-design," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 102–115, Association for Computing Machinery, 2022.

[171] Z. Ruan, T. He, and J. Cong, "{INSIDER}: Designing {In-Storage} computing system for emerging {High-Performance} drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 379–394, 2019.

[172] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "Genstore: A high-performance in-storage processing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, (New York, NY, USA), p. 635–654, Association for Computing Machinery, 2022.