University of California
Santa Barbara

Memory-Centric Architectures for Big Data Applications

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Jilan Lin

Committee in charge:

Professor Yuan Xie, Co-chair
Professor Yufei Ding, Co-chair
Professor Trinabh Gupta
Professor Peng Li
Professor Jonathan Balkind

December 2022

The Dissertation of Jilan Lin is approved.

Professor Trinabh Gupta

Professor Peng Li

Professor Jonathan Balkind

Professor Yufei Ding, Committee Co-chair

Professor Yuan Xie, Committee Co-chair

November 2022

Memory-Centric Architectures for Big Data Applications

by

Jilan Lin

Acknowledgements

I would like to thank all the people who contribute to this dissertation and all the help that I received during my Ph.D. journey.

First, I want to thank my advisor, Prof. Yuan Xie, for his insightful guidance and generous support throughout my Ph.D. studies. He shows me how to become a good researcher with self-discipline and deep vision. I also learned from him how to better present my research and make myself impressive. Moreover, I am truly grateful to my co-advisor, Prof. Yufei Ding, for her comprehensive efforts in improving my research quality and passing on valuable Ph.D. advice.

Second, during my internship at Alibaba, I thank Dr. Shuangchen Li, Dr. Dimin Niu, and Dr. Hongzhong Zheng for their constructive suggestion and feedback. Also for my internship at Meta, I am sincerely grateful to Dr. Elnaz Ansari and Dr. Edith Beigne for their supportive attitude and useful advice toward my project. I appreciate the knowledge and skills from my industry experiences.

Additionally, I would like to thank my colleagues in the SEAL lab. I enjoyed a great time working and collaborating with them. I really appreciate their contributions to all the publications presented in this dissertation. Especially, I sincerely thank Ling Liang, Liu Liu, Zheng Qu, Tianqi Tang, Abanti Basak, Fengbin Tu, Xing Hu, Lei Deng, and Shuangchen Li. Further, I want to thank Prof. Trinabh Gupta for his deep insight into security and privacy. Our collaboration on the topic of private information retrieval contributes to an important chapter in this dissertation.

Most importantly, I am grateful to my parents, Xuemei Zhang and Xin Lin, who always love me and share their selfless support during my Ph.D. journey.

Curriculum Vitæ

Jilan Lin

Education

2023	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2021	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2018	B.S. in Electronic Engineering, Tsinghua University.

Publications

[C1] **Jilan Lin**, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan 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), pp. 102-115. 2022.

[C2] Liu, Liu, **Jilan Lin (Equal Contribution)**, Zheng Qu, Yufei Ding, and Yuan Xie. "ENMC: Extreme Near-Memory Classification via Approximate Screening." In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1309-1322. 2021.

[C3] Basak, Abanti, Zheng Qu, **Jilan Lin**, Alaa R. Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. "Improving Streaming Graph Processing Performance using Input Knowledge." In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1036-1050. 2021.

[C4] **Jilan Lin**, Shuangchen Li, Yufei Ding, and Yuan Xie. "Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications." In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-9. IEEE, 2021.

[C5] Basak, Abanti, **Jilan Lin**, Ryan Lorica, Xinfeng Xie, Zeshan Chishti, Alaa Alameldeen, and Yuan Xie. "Saga-bench: Software and hardware characterization of streaming graph analytics workloads." In 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 12-23. IEEE, 2020.

[C6] **Jilan Lin**, Shuangchen Li, Xing Hu, Lei Deng, and Yuan Xie. "CNNWire: Boosting convolutional neural network with winograd on ReRAM based accelerators." In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI), pp. 283-286. 2019.

[C7] **Jilan Lin**, Zhenhua Zhu, Yu Wang, and Yuan Xie. "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator." In Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC), pp. 639-644. 2019.

[C8] Zhu, Zhenhua, **Jilan Lin**, Ming Cheng, Lixue Xia, Hanbo Sun, Xiaoming Chen, Yu Wang, and Huazhong Yang. "Mixed size crossbar based RRAM CNN accelerator

with overlapped mapping method." In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1-8. IEEE, 2018.

[C9] **Jilan Lin**, Lixue Xia, Zhenhua Zhu, Hanbo Sun, Yi Cai, Hui Gao, Ming Cheng, Xiaoming Chen, Yu Wang, and Huazhong Yang. "Rescuing memristor-based computing with non-linear resistance levels." In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 407-412. IEEE, 2018.

[J1] Qu, Zheng, Lei Deng, Bangyan Wang, Hengnu Chen, **Jilan Lin**, Ling Liang, Guoqi Li, Zheng Zhang, and Yuan Xie. "Hardware-Enabled Efficient Data Processing with Tensor-Train Decomposition." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 41, no. 2 (2021): 372-385.

[J2] **Jilan Lin**, Cheng-Da Wen, Xing Hu, Tianqi Tang, Chao Lin, Yu Wang, and Yuan Xie. "Rescuing RRAM-based computing from static and dynamic faults." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 40, no. 10 (2020): 2049-2062.

Abstract

Memory-Centric Architectures for Big Data Applications

by

Jilan Lin

Big Data refers to the massive and rapidly growing data in our daily life, which can be very helpful to dig valuable information and make better decisions. However, handling big data workloads poses new challenges in traditional memory subsystems due to the memory wall and power wall issues. Moving a large amount of data from memory to the processor is expensive, which can cause severe performance bottleneck and high energy consumption. Moreover, while the computation capability of modern processors grows fast with Moore's Law, the bandwidth and latency of memory improve much slower due to the I/O's physical constraints.

To address the memory wall challenge for big data applications, this dissertation aims to answer the following two questions: How can we improve the existing memory technology, and what should future memory look like? For the first question, we investigate the approach of near-data processing (NDP). NDP technique adopts custom compute logic near the data storage, which leverages the much higher internal bandwidth and reduces the energy consumption by data movements. Second, we envision the in-memory processing (IMP) technique for next-generation memory. In particular, we study the Resistive Random Access Memory (RRAM), an analog device with tunable resistance. We use RRAM as both storage and computation hardware to process data in-situ and analyze its mapping and reliability issues.

Contents

Chapter 1

Introduction

In 2014, the total digital data ever generated by our devices around the world is 4.4 zettabytes (ZB, 10^{21} bytes). However, with the explosive technology evolution and our increasing reliance on digital services, we now produce 16 ZB data every year [1]. The explosive growth of data brings big opportunities to harvest valuable information and knowledge, and we can leverage the transformative knowledge for various sectors in our society like enterprises, the healthcare industry, and educational services [2]. Among different algorithms in big data applications, deep learning emerges as the most remarkable one. Generally, deep learning models learn hierarchical representations from the dataset and use the abstracted representations to perform tasks including classification [3], language modeling [4], recommendations [5], graph analytics, and so on.

While big data and deep learning are opening up an era of technology revolution, handling data on such a scale poses severe challenges in traditional processor architecture, especially for the memory subsystem. Traditional von-Neumann architectures separate the computation and memory, causing two key issues: *memory wall* and *power wall*. First, the memory scaling is much slower than the computation part, and the memory thus becomes the system bottleneck. In the past decade, the supercomputer's performance

has grown by 33× [6], and the GPU's performance has improved by 35.8× from Kepler architecture [7] to Ampere architecture [8]. In comparison, DRAM's latency has only reduced by 26% [9], and the bandwidth of GPU has only increased by 6.5×. Therefore, the memory wall means the large performance gap between computation and memory caused by the unbalanced scaling. Second, moving data along the way from memory is energy-consuming, resulting in high dynamic energy for the DRAM memory. Studies have shown that accessing one byte of data consumes more than 100× of the energy than computing one-byte data. As a consequence, while the current data center industry consumes 196 to 400 terawatt-hours (TWh) of energy every year [10], it is reported that 25% of them are spent on memory [11].

This dissertation presents a roadmap to future memory systems that overcome challenges brought by the memory wall and power wall. First, we improve the current memory systems by leveraging the near-data processing (NDP) techniques. NDP architectures benefit from higher internal bandwidth and shorter data path by placing the compute hardware much closer to the data. Thus, we are able to bridge the gap between computation and memory. Second, we design the next-generation memory with the in-memory processing (IMP) technique. Specifically, we investigate the Resistive Random Access Memory (RRAM), which can perform in-situ computation within the memory array. This unique characteristic brings up the opportunity to eliminate a large number of memory accesses and greatly improve the system's performance and energy efficiency.

1.1 Opportunities and Challenges

Near-Data Processing. Given that big data applications are increasingly bandwidth-hungry, near-data processing (NDP) technique is getting growing attention to accelerate such workloads [12]. NDP puts customized computation logic beside the data and saves

the system bandwidth and access latency, which can utilize the large bandwidth provided by internal parallelism inside the memory or storage. Previous work has broadly leveraged NDP to accelerate memory-intensive tasks, such as recommendation systems [13] and image processing [14]. Such applications usually show low computation density but require a large number of data accesses. Therefore, we can easily engage the memory/storage with lightweight computation logic to facilitate these workloads.

However, we identify two challenges in current NDP architectures. First, existing NDP architectures generally lack software and hardware co-optimization and thus tend to achieve sub-optimal performance. For example, for the classification tasks that appear in various deep learning models, moving computation directly to NDP would degrade the data reuse and result in higher memory traffics, because the limited on-chip resources in NDP cannot buffer the entire classification weights. Second, most NDP designs focus on machine learning applications, but many other important workloads are overlooked. For example, we use graphs in many domains to abstract the data of interests, such as social networks and financial transactions [15, 16], and graph processing thus demonstrates its importance in modeling real-world problems. Also, another data-intensive workload, privacy-preserving storage and computation, becomes more and more critical to cloud applications, since people are getting aware of their privacy in the services backboned by big data. The goal of this dissertation is to propose a software-hardware co-design approach to maximize the potential of NDP architectures. Particularly, we pay attention to a broad range of big data workloads, including deep learning, graph analytics, and private databases.

In-Memory Processing. While NDP provides a solution to alleviate the bandwidth bottleneck in existing memory systems, the DRAM scaling issue is still a big challenge facing the ever-increasing data size in the future. Therefore, we also investigate the next-

generation memory that is powered by Resistive Random Access Memory (RRAM), which offers the in-memory processing (IMP) capability. RRAM is an analog device, and its resistance is tunable for different states. With the unique resistance-switching character and crossbar structure, RRAM is able to not only store information, but also perform matrix-vector multiplications inside the memory array. This reduces half of the data fetching [17]. As matrix computation is the key operation in many big data workloads, RRAM becomes a perfect memory candidate with computational capability. Also, RRAM provides O(1) computation complexity and ultra-low power consumption [18], making itself a competitive candidate for the next-generation memory device.

Although RRAM has great potential in performance and energy efficiency, using RRAM to accelerate big data applications is not a free lunch. First, previous studies on RRAM characterizations have revealed that current RRAM devices exhibit several reliability issues and non-ideal faults, and we hardly have the RRAM resistance being the exact value expected [19]. Different from using RRAM as a storage device, such faults can accumulate and lead to severe precision loss when using RRAM for computation. Second, lots of computations in big data are sparse, which can hardly benefit from the RRAM accelerators, because the RRAM array stores and computes data in the granularity of matrix. For example, deep learning models are known to exhibit redundancy [20]. However, existing mapping for the sparse model to RRAM is the same as the dense model, and zeros in the sparse model are not skippable. Therefore, there is no performance gain from compression. Therefore, to facilitate the deployment of RRAM-based IMP to real-world applications, this dissertation concentrates on addressing the mapping and non-ideality issues in RRAM devices.

1.2 Contributions

The goal of this dissertation is to address the performance gap between memory and computation. We first introduce the NDP solution to improve existing memory and storage systems for various big data applications. Specifically, we focus on architecting both the software and hardware to fully utilize NDP's benefits. Second, we envision RRAM as the next-generation memory device, which provides the IMP capability to further speed up the computation and lower the energy consumption. We particularly make efforts in designing the mapping and fault-tolerate schemes.

We present three NDP designs, including:

- ENMC, an algorithm and architecture co-designed NDP to support regular-patterned workloads, extreme classifications. ENMC utilizes approximation and hardware specialization to greatly speedup the classification tasks in deep learning.
- G-MEM, a customized memory hierarchy tailored for irregular-patterned workloads, graph analytics. G-MEM re-designs the on-chip memory and leverages NDP technique to support irregular and fine-grained data accesses in graph processing.
- INSPIRE, an in-storage processing architecture that targets larger-scale problem, private databases. INSPIRE engages a protocol and architecture co-optimization approach to facilitate the private query processing.

Moreover, we also propose two IMP designs, including:

- Sparse-oriented RRAM architecture, which designs from both software and hardware perspective to map sparse deep learning model to RRRAM-based accelerator.
- SIGHT, which is a fault-tolerant framework with algorithm and architecture co-design to address the reliability issue in RRAM-based IMP accelerator.

The remainder of this dissertation is organized as follows. Chapter 2 introduces the necessary background and preliminaries for the further discussion. Chapter 3, 4, and 5 propose explore the NDP architectures for various big data applications. Chapter 6 and 7 introduce our IMP design that specifically addresses the mapping and reliability issues. Chapter 8 concludes the dissertation and summarizes our work.

This dissertation comprises our original work published elsewhere in conference and journal papers:

- Chapter 3: Liu, Liu, Jilan Lin (Equal Contribution), Zheng Qu, Yufei Ding, and Yuan Xie. "ENMC: Extreme Near-Memory Classification via Approximate Screening." In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1309-1322. 2021.
  DOI: https://doi.org/10.1145/3466752.3480090

- Chapter 4: Jilan Lin, Shuangchen Li, Yufei Ding, and Yuan Xie. "Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications." In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-9. IEEE, 2021.
  DOI:10.1109/ICCAD51958.2021.9643434

- Chapter 5: Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan 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), pp. 102-115. 2022.
  DOI: https://doi.org/10.1145/3470496.3527433

- Chapter 6: © 2020 IEEE. Reprinted, with permission, from Jilan Lin, Cheng-

Da Wen, Xing Hu, Tianqi Tang, Chao Lin, Yu Wang, and Yuan Xie. "Rescuing RRAM-based computing from static and dynamic faults." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 40, no. 10 (2020): 2049-2062.

DOI: 10.1109/TCAD.2020.3037316

- Chapter 7: Jilan Lin, Zhenhua Zhu, Yu Wang, and Yuan Xie. "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator." In Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC), pp. 639-644. 2019.

  DOI: https://doi.org/10.1145/3287624.3287715

Chapter 2

Technical Background

In this chapter, we introduce the preliminary on memory subsystems, including the basics of DRAM, RRAM, and SSD storage architecture. The content in this chapter is necessary to further understand the following chapters in this thesis.

2.1 DRAM Memory

Dynamic Random Access Memory (DRAM) is usually worked as the main memory in computer systems. The storage unit in DRAM consists of a transistor and a capacitor. We can store one bit of information by charging or discharging the capacitor. For instance, when the capacitor is charged, we assume it represents the logic high ('1'), while a discharged capacitor stores the logic low ('0'). The process of charging/discharging the DRAM cell is called a write operation. Moreover, a read operation is to retrieve the value in the capacitor through the connected transistor.

DRAM Memory Organization. Modern DRAM memory is a passive device controlled by the host processor. It receives and executes commands from the memory controller and then returns back the data. As shown in Fig. 2.1, the host processor is

able to connect to multiple DRAM *channels* to increase the memory bandwidth, and each channel will receive individual access commands. Within a channel, there are multiple DRAM ranks. At a given moment, the host can only issue commands to one rank through the memory bus. For the Dual In-line Memory Module (DIMM) usually seen in the computer, we can have up to four ranks per module. Further, a rank physically consists of multiple DRAM chips that connect to the same command bus. For example, for a DRAM data bus having a width of 64, we need 8 chips with a data width of 8 for each chip (eight ×8 chips). On the other hand, a rank is logically composed of multiple banks, and a bank is distributed among the chips. DRAM banking reduces the latency of memory array access and enables multiple accesses in parallel. Finally, within a bank, there are multiple subarrays, and a subarray could consist of multiple memory mats. A mat is then the crossbar array that has M×N DRAM cells.

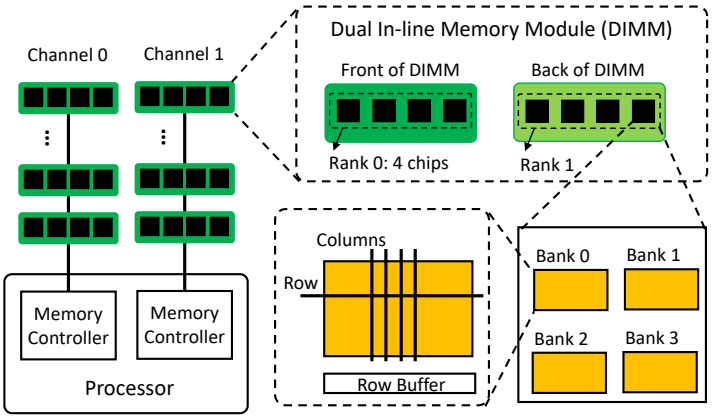


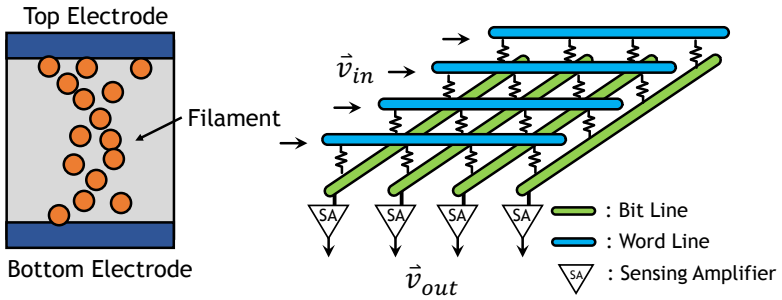
Figure 2.1: An illustration of the DRAM hierarchy.

To address the data in DRAM memory, the host first activates the particular row by specifying the channel ID, rank ID, bank ID, and row ID (we can use row ID to index which mat to access). After receiving this command, the row is read and buffered in the row buffer. Then, the memory controller sends the column ID to address the particular data in the row buffer. After sending the row activate command, the host has to wait

until the row is actually activated. But it can send commands to other ranks or banks to fully leverage the memory parallelism. Moreover, the DRAM read is destructive, meaning activating a row will destroy the information stored in the row. Thus, the host controller needs to issue a precharge command to write back the data in the row buffer.

2.2 RRAM Memory

Resistive Random Access Memory (RRAM) is one of the emerging non-volatile memories, which is also known as the memristor [21]. As shown in Fig. 2.2(a), the RRAM cell is a passive bipolar device and usually applies the metal-insulator-metal structure. The middle insulator showing resistive switching characteristics can be made of various materials such as HfO_2 [22], TiO_x [23], NiO [24], and so on.



(a) (b)

Figure 2.2: (a) An RRAM cell with the metal-insulator-metal structure. The top and bottom electrodes are made of metal and the middle insulator shows resistance switching character. (b) The structure of an RRAM crossbar. The input voltage opens one word line and data are read from bit lines.

The attractive resistive switching property usually comes from the conductive filament between two electrodes [24]. When applying a certain programming voltage to the cell, the filament grows up and connects two electrodes together, which then reduces the cell resistance and makes it conductive. This process is called a SET operation where it tunes the RRAM cell from the high resistance state (HRS, representing "0") to the

low resistance state (LRS, representing "1"). The reversed operation which destroys the filament between two electrodes is thus called a RESET operation. Both SET and RESET are considered as write operations to the RRAM.

RRAM cells can be organized in a crossbar structure for higher area efficiency, as shown in Fig. 2.2(b). When used as a memory device, the row decoder opens one word line according to the address and sends a read voltage, and data are read from selected bit lines through sensing amplifiers. To further improve the density and save the hardware cost, a multi-level cell (MLC) is broadly considered, where one RRAM cell can store several bits of information by tuning it to multiple resistance states [25]. In that case, higher resolution for analog-to-digital interfaces is then required to decode the data.

RRAM-based Computing System With the crossbar structure and resistive switching character, researchers have explored RRAM's potential of in-memory computing [26, 27]. First, we can store an $n \times n$ matrix in an $n \times n$ crossbar. Then, all the word lines are opened simultaneously and we set input voltages with respect to a vector. Therefore, we have the output currents from bit lines being the result of matrix-vector multiplication. The scalar multiplication is done by voltage-conductance multiplication, and the result is accumulated by all the currents within a bit line. Assume that the input voltage vector is (V^i) and the output is (V^o), the computation can then be expressed as in Eq. 2.1:

$$\begin{bmatrix} V_1^o \\ \vdots \\ V_M^o \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,N} \\ \vdots & \ddots & \vdots \\ c_{M,1} & \cdots & c_{M,N} \end{bmatrix} \begin{bmatrix} V_1^i \\ \vdots \\ V_N^i \end{bmatrix} \tag{2.1}$$

$$c_{i,j} = -g_{i,j}/g_s \tag{2.2}$$

where $c_{i,j}$ is the matrix parameter, which depends on the RRAM conductance in the corresponding position (i, j) and the reference conductance g_s in sensing amplifiers as

shown in Eq. 2.2. Since RRAM conductance can only be positive, two crossbars are needed to represent an application matrix with both positive and negative parameters [28].

As this computing mechanism significantly reduces the complexity of matrix-vector multiplication from $O(n^2)$ to $O(1)$, previous work has intensively studied how to leverage it for NN acceleration [26, 29].

2.3 SSD Storage

The general architecture of modern SSD is shown in Fig. 2.3. The SSD storage usually contains a host interface, an embedded processor, DRAM, and flash memory DIMs. The host interface implements the interface protocol, such as SATA or PCI express. When access requests come from the host interface, the embedded processor executes the Flash Translation Layer (FTL) to derive the physical address of the request. The embedded processor can be a RISC processor (such as ARM) that has limited computation capability. The processor further schedules these requests to flash controllers, which control multiple flash channels. The flash controller issues specific commands to the corresponding flash DIM to access the data. The DRAM can be used as a buffer to transfer the data from flash channels to the host.

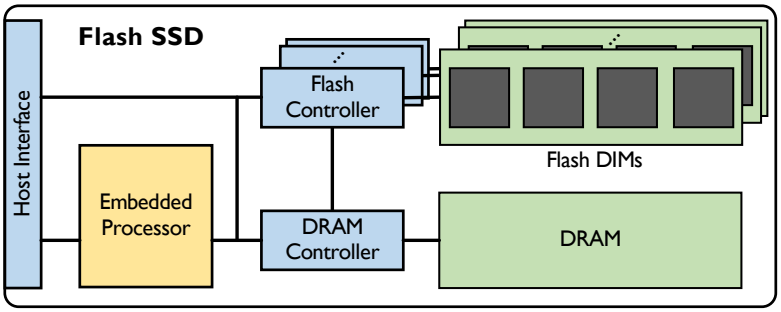


Figure 2.3: The architecture of flash SSD.

SSD storage has been widely used for applications that require large memory capacity, such as industry-level neural network training [30] and large-scale graph processing [31]. However, accessing SSD storage is much slower than memory, and the bandwidth of I/O is very limited. The latency of accessing flash SSD is about 50 μs, while the I/O bandwidth is up to 1.0 GiB/s per PCIe 3.0 lane. Therefore, in-storage processing emerges as a promising solution to overcome this performance gap, which leverages higher (4-8$\times$) bandwidth and low latency internal to the SSD storage [32].

Chapter 3

ENMC: A Near-Memory-Processing Architecture for Extreme Classification

This chapter presents ENMC, the first end-to-end Near-Memory-Processing(NMP) Architecture to address the extreme classification problem through software and hardware co-design. We first introduce the background and motivation for accelerating extreme classification. Then we introduce our proposed algorithm and architecture. Finally, the experimental methodology and results are presented before we conclude the chapter.

3.1 Background and Motivation

Recent advances in many machine intelligence areas, such as natural language processing (NLP) [33, 34, 35], image recognition [36, 37, 38], and recommendation[39, 40, 41], involve tackling the extreme classification problem, where classification category size is extreme large. For example, in the NLP domain, making predictions is basically clas-

sifying the words with high probabilities. Similarly, for image recognition tasks and recommendation tasks, the features generated from hidden neural network layers need to go through the classification layer to output predictions. As shown in Fig. 3.1, extreme classification is the essential component to deal with large-scale problems.

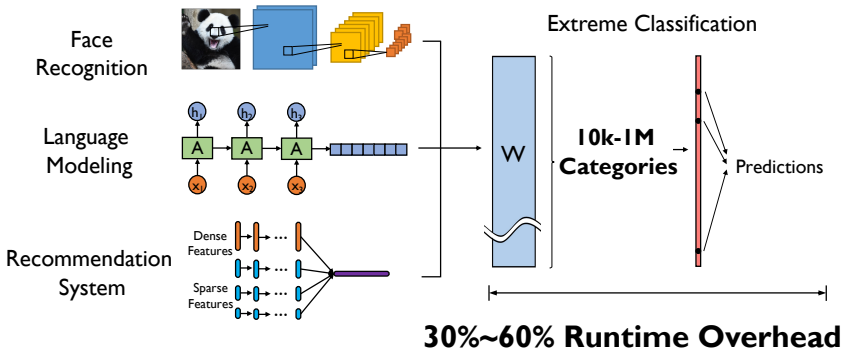


Figure 3.1: Extreme Classification serves as the common component of large-scale Deep Learning applications. The classifier processes hidden representations from application-specific hidden layers and generates predictions as used in recognition, language, and recommendation.

3.1.1 Extreme Classification

The Extreme Classification problem refers to multi-class or multi-label classification with extremely large category volume. Many large-scale NLP and recommendation applications can be modeled as a feature extraction part with an extreme classifier. For example, in NLP applications, the typical sequence-to-sequence modeling consists of a stack of encoders, a stack of decoders, and a final classification layer [34, 42, 43]. Each encoder and decoder is a type of DNN layer, such as Transformer layers [44] and recurrent neural networks [42]. The encoders process input embeddings into hidden representations repeatedly. The decoders that attend over all hidden states from the encoder stack process queries from the previous decoder layer and output decoded hidden vectors. The final classification layer turns the hidden vector from the last decoder layer into a trans-

lated word as in translation tasks or probabilities as in language modeling tasks. The classification layer consists of a large linear layer followed by a softmax layer. The linear layer can be interpreted by performing the inner products of the hidden vector from the decoder stack and a number of weight vectors, which correspond to the target vocabulary size. The softmax function then normalizes the inner products into probabilities.

Also, in large-scale recommendation systems such as commodity product recommendation and webpage recommendation, extreme classification refers to the problem of multi-class prediction [36, 41, 37, 45]. First, the hidden layers, e.g., DNNs, take dense features and sparse features from users as input. Then, the classification layer maps the output of the last hidden layer, usually through softmax normalization, to a probability distribution. For real-world scenarios and next-generation applications, the final classification layer is becoming even more challenging as the computational complexity and memory usage grows linearly with the category size.

3.1.2 Motivation

As classification categories keep scaling in real-world applications, the classifier's parameters could reach hundreds of gigabytes, far beyond the on-chip memory capacity. For large-scale NLP models, the vocabulary sizes are in the range of hundreds of thousands, contributing hundreds of megabytes of data [34, 44]. For recommendation systems, using commodity datasets to solve industry-level problems would require classification on the scale of 100M categories [37, 36], consuming around 190GB of memory.

Due to the large memory footprint of extreme classification, accessing system memory for the classifier's weight data becomes the bottleneck of system performance. We characterize the state-of-the-art Transformer-based language model [46] and show that the final classification layer consumes 50% of overall model inference time. While GPUs

and specialized accelerators can boost the performance of DNN layers [47, 48], they suffer from inter-device data movements when executing the memory-intensive classification layer, as the memory usage exceeds device memory capacity.

3.1.3 Opportunity

The root cause for extreme classification being the bottleneck is the large memory footprint and the low operational intensity. We show in Fig. 3.2(a) that classifiers consume memory in the order of hundreds of megabytes or even gigabytes, far beyond the on-chip memory capacity of modern GPUs or NPUs. The execution time of classification increases linearly with category size and hidden dimensions. From the perspective of DL practitioners and algorithm developers, using a larger vocabulary or category and hidden dimensions is almost always a way to improve model quality. However, increasing memory usage will worsen the memory-bounded execution problem. For recommendation systems, the increasing need for an enormous number of items results in even more challenging requirements to accommodate the classifier.

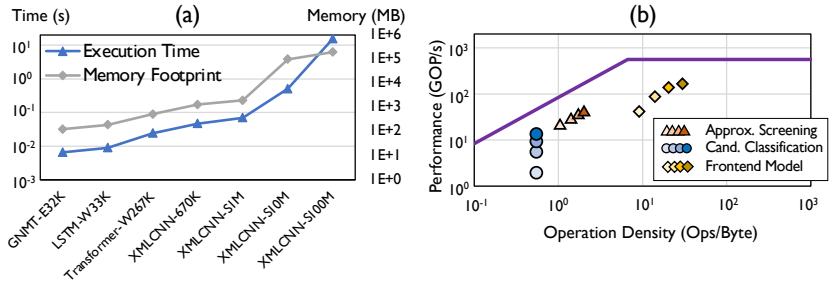


Figure 3.2: (a) The memory footprint and the execution time on CPU of classification layers scale linearly with the number of categories. (b) Roofline analysis of the major components. Darker color indicates larger batch size.

Opportunity of approximation: In extreme classification, outputs from classifier are probabilities. While we should compute all the outputs of the linear transformation using all classifier parameters, many applications require only the probabilities of the

top words. For example, in neural machine translation, we only use the top-K values of softmax-normalized probabilities to select the translated words, where K is the beam search size when applied. Therefore, we could have only the top-K probabilities to be accurate, then have the rest to be approximate, aiming at significantly reduced computations and data accesses. In the next section, we explore the opportunity of using approximation to achieve efficient extreme classification.

Opportunity of NMP: Although approximation can greatly reduce the computation amount in extreme classification, approximate screening is still bounded by the memory bandwidth. As shown in Fig. 3.2(b), we plot the data points for our approximate screening, candidate-only classification, and front-end neural networks in a CPU's roofline model. Both screening and classification exhibit low operation intensity after we eliminate redundant computations and reduce hidden dimensions. Therefore, different from the front-end models that are often bounded by computation capability, approximate screening and candidate-only classification can benefit from the large bandwidth of NMP architectures.

However, existing NMPs often employ a homogeneous architecture equipped with unified floating-point and integer compute units [49, 50, 13]. Our proposed screening method explores a heterogeneous computation pattern that includes a low-precision approximate screening phase and a full-precision candidate-only classification phase. Therefore, our NMP architecture features a dedicated resource management of both phases and a customized pipeline design.

3.2 Design Overview

We propose the first end-to-end solution to address the memory-bound problem of extreme classification with NMP architecture. Fig. 3.3 gives an overview of the proposed

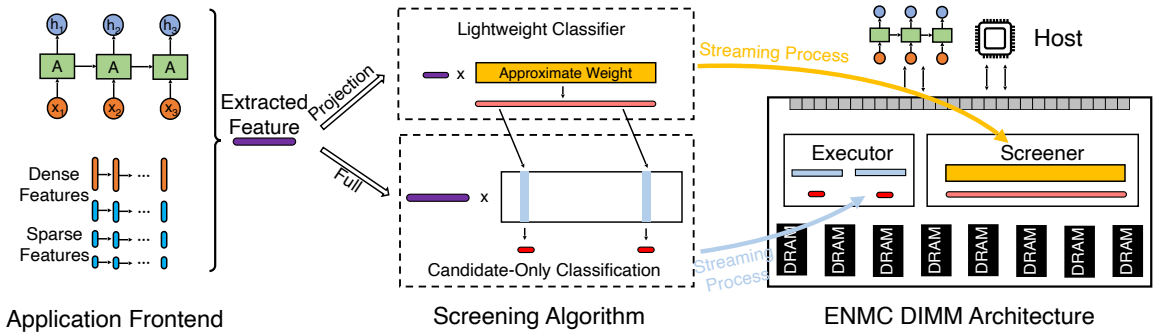


Figure 3.3: The overview of our Approximate Screening algorithm and NMP architecture co-design. Instead of full classification, our co-design essentially performs candidates-only classification, where the candidates are based on the screening method. Our NMP architecture design features a Screener and an Executor to collaboratively process candidates-only classification.

software and hardware co-design. To reduce the overhead of classification, we propose an approximate screening algorithm that directly reduces the required computations and data access involved in linear transformations. As demonstrated in Fig. 3.3, given the extracted feature vectors from the application front-end, a learned lightweight classifier first performs approximate classification to efficiently identify the set of important candidates in the category space. Afterward, the classifier will trigger candidates-only computation to generate accurate classification results, while the rest can directly utilize the approximate results computed from the screening phase. Therefore, a large number of computations and data loading are saved. Our experiments show that the proposed screening method achieves better trade-off for classification accuracy and computation saving, compared with conventional low-rank approximation-based method [35].

To fully leverage the approximate screening method, we further propose the Extreme Near-Memory Classification architecture, namely *ENMC*. Here we highlight the key features of our ENMC design as follows: Firstly, as shown in Fig. 3.3, we deploy a dual-module architecture that contains a Screener module and an Executor module that run in parallel. The Screener performs approximate screening efficiently and predicts

classification candidates in advance. For each candidate selected in a batch, the ENMC controller will generate instructions for accurate computations handled by the Executor. The computing modules are deployed at the rank level such that there is no need to invade the DRAM chips. Secondly, we design the ENMC instruction set to facilitate the workloads accommodation between the host processor and ENMC. It also supports the communications between the Screener and the Executor. We define the instruction format by leveraging the reserved command space so that it is compatible with the commodity DDR interface. Thus, our ENMC DIMM can also support regular memory requests. Finally, we provide the system-level design, including application workflow and program compiler support, to make the ENMC architecture cooperate with the software framework. Our design could be easily extended no matter whether the host processors are CPUs, GPUs, or domain-specific accelerators.

3.3 Approximate Screening

The limited computing capability of NMP logic cannot afford the computations of extreme classification. In other words, the execution of full classification on NMP core becomes the bottleneck. We find that not all computations in classification are useful. In fact, only a small portion of classification results contribute to model predictions. For example, in language modeling tasks, only output probabilities of the most important words need to be accurate. Thus, we propose an efficient approximation method that can estimate the subset of output probabilities that need accurate computations and then populate the rest probabilities with approximate results. Similarly, for other classification-involved tasks, we only need accurate computations for a small number of key candidates and use approximate results for the remaining outputs.

3.3.1 Screening Method Overview

Given a d-dimensional vector ($h \in \mathbb{R}^d$) from hidden DNN layers, where d is the hidden dimension, the softmax classification transforms the hidden vector h to an l-dimensional probability space. We denote the output probability vector as $z \in \mathbb{R}^l$, where l is the vocabulary size. The transformation is essentially matrix-vector multiplication as

$$z = Wh + b \tag{3.1}$$

where $W \in \mathbb{R}^{l \times d}$ is the classifier weight matrix and $b \in \mathbb{R}^l$ is the bias vector. Then, the softmax function normalizes the output vector z into probability distribution as

$$p_i = softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{3.2}$$

where p_i is the i-th element of output probability vector p. The probability vector is then used to perform the next word prediction as in language modeling or translation. While softmax is the most common normalization function used in classification, our method is compatible with other non-linear functions used in classification such as Sigmoid [41].

The memory-intensive transformation is a good candidate for NMP acceleration. However, the computational complexity is not affordable for NMP. Our proposal seeks redundancy in extreme classification and uses low-cost approximated computations to mitigate the computational burden. We introduce a low-dimensional and low-precision screening module that can approximate the original classifier. We first discuss how to reduce computations at inference time given the screening module. After that, we explain the learning process to obtain such a screening module.

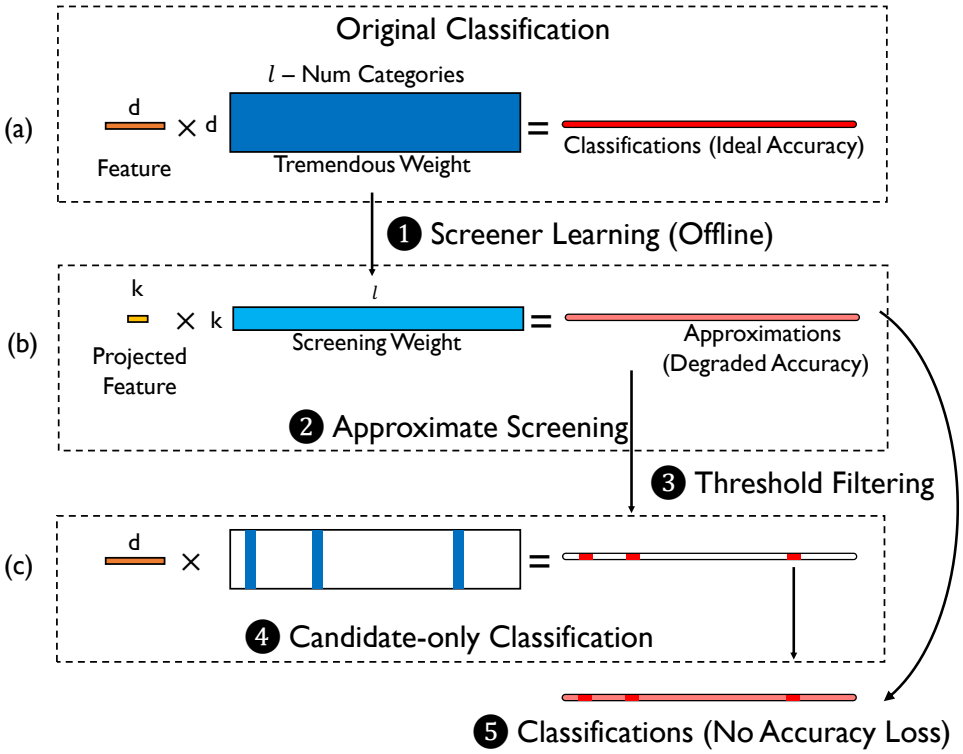


Figure 3.4: Illustration of approximate screening: (1) the screener learns from full classifier at the offline learning phase; (2) the screening step computes approximate results, involving lightweight Screener weights and the projection matrix, and selects candidates among approximate results; (3) the threshold filtering step selects key candidates; (4) only the corresponding vectors in the full classification weights are used to compute candidates-only accurate results; (5) the final results before softmax normalization combine both approximate and accurate results.

3.3.2 Inference Process

As shown in Figure 3.4(a), the standard classification is essentially matrix-vector multiplication followed by softmax normalization. The execution is bounded by accessing W from DRAM modules.

We construct the approximate screening module with a projection matrix P and a reduced-hidden-dimension weight matrix $\tilde{W}$. The initialization of the projection matrix is according to standard sparse random projection [51], and the overhead is negligible (less than 0.1%) compared with classifier weights as the projection matrix P can be represented

in 2-bit format. The process of computing approximate results can be expressed as

$$\tilde{z} = \tilde{W}Ph + \tilde{b} \tag{3.3}$$

where $\tilde{W} \in \mathbb{R}^{l \times k}$ and $P \in \sqrt{\frac{3}{k}} \cdot \{-1, 0, 1\}^{k \times d}$.

Figure 3.4(b) illustrates the process: the d-dimensional hidden vector h is first projected to a lower k-dimensional space, and the low-dimensional vector multiplies $\tilde{W}$ to get approximated output $\tilde{z}$. Compared with full classification, the accessed approximate weight volume is significantly reduced since $k << d$. Furthermore, we can reduce the precision of running the screening module to further reduce accessed data.

After obtaining the approximate results, i.e., $\tilde{z}$, we estimate the importance of all l values and select the most important m values, referred as candidates, that require accurate computations. The estimation can be done with top-m searching or thresholding, where the threshold value can be tuned on validation sets.

Only for the candidates that need accurate computations, our method then need to access full classifier weights W, i.e., a small portion of totally l weight vectors. These weight vectors then multiply with the original hidden vector to produce accurate results for the candidates, as shown in Figure 3.4(c). The final output before softmax function is a mixed vector with approximate values from screening and accurate values from full W.

3.3.3 Learning Algorithm

Here, we discuss the learning procedure to obtain the screening module. The goal for screening is to approximate the classifier well. Therefore, we regard the outputs z from the full classifier as the learning target and train the screening module weights $\tilde{W}$ to fit.

Algorithm 1: Training algorithm for the parameters of the Screener

Data: Batched context vectors $\{h_i\}_{i=1}^{S}$, where $h_i \in \mathbb{R}^d$ from hidden layers; trained classifier weights $W \in \mathbb{R}^{l \times d}$ and bias $b \in \mathbb{R}^l$; projection matrix P.
Result: Screener weights $\tilde{W} \in \mathbb{R}^{l \times k}$ and bias $\tilde{b} \in \mathbb{R}^l$.

```
Initialize projection matrix P ∈ sqrt(3/k) · {−1, 0, 1}^{k×d};
for it ∈ all iterations do
    Compute loss according to Eq. (3.4);
    Update W̃, b̃ with SGD(min Loss);
end
```

The optimization objective function is

$$L = \frac{1}{s} \sum_{s} ||(Wh + b) - (\tilde{W}Ph + \tilde{b})||_2^2 \tag{3.4}$$

where s is the mini-batch size of training samples. During training, the classifier parameters, i.e., W and b, as well as the parameters of hidden layers are fixed and will not be changed. We only update the screening module's parameters $\tilde{W}$ and $\tilde{b}$. The projection matrix P is constructed and initialized before distillation and stays constant during distillation and inference.

Our learning algorithm uses the default training and validation datasets and does not need extra training data. The convergence happens in several training epochs, much faster than the original model training. Algorithm 1 gives the overall training of screening parameters.

3.4 ENMC Architecture

In this section, we introduce the architecture design of the ENMC. We first give a glimpse of the design overview, followed by the microarchitecture details. Then, we present the ENMC instruction set and system-level design.

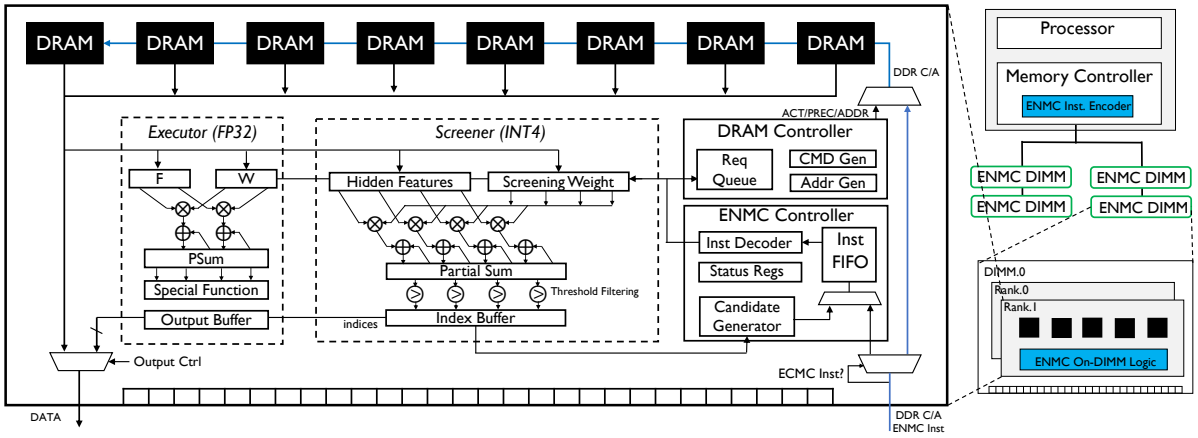


Figure 3.5: The architecture overview of an ENMC DIMM. The ENMC logic is located at each rank to leverage the large rank-level bandwidth. The ENMC mainly consists of a controller to decode instructions, a DRAM controller to generate DDR C/A commands, a screener to perform approximation, and an executor to process full-precision classification.

3.4.1 Architecture Overview

We have exploited the opportunity of eliminating the redundancy in the extremely-large weight and forecasting the classification results with a much smaller overhead using our lightweight screening algorithm. Although the computation bottleneck is alleviated with our proposed approximate screening framework, the tremendous classification dimension is still bandwidth-hungry, and conventional processor-memory systems are hardly able to overcome the memory throughput wall. Therefore, in this section, we further co-architect the near-data processing subsystem, Extreme Near-Memory Classifier (ENMC), to facilitate the processor for computing the extreme classification. The design goal of such near-data architecture is to leverage the large bandwidth provided by rank-level parallelism in a DRAM channel, and we process the classification in data stream through dedicated on-DIMM hardware.

Specifically, we highlight the features of our ENMC design as follows:

First, we deploy a dual-module architecture that contains a Screener module and an

Executor module that runs in parallel. The Screener performs fixed-point screening as described in Section 3.3, and predicts classification candidates in advance. Since the classification weight is low-dimensional and quantized, the Screener is able to process the data in a streaming manner, such that the large rank-level bandwidth can be leveraged. For each candidate found in a batch, the ENMC Controller will generate instructions for further full-precision computations which are completed by the Executor. We put these computation logics at the rank level such that there is no need to invade the DRAM chips.

Second, we design the ENMC instruction set to facilitate workload accommodation from host processors and support the communications between the Screener and Executor modules. We define the instruction format by leveraging the unused address line and data line in the PRECHARGE command to ensure compatibility with the commodity DDR interface. Thus, regular memory requests can also be served with our ENMC DIMM.

Third, we provide the system-level design, including the program compiler support and application workflow, to make the ENMC architecture cooperate with the software framework. Our design could be easily extended to support different scenarios where the host processors could be CPU, GPU, or domain-specific accelerators.

3.4.2 ENMC Microarchitecture

We now introduce the microarchitecture of ENMC. We first present the design overview, followed by the implementation details of each component.

Overview. We put ENMC on the DIMM board between the DRAM devices and the DDR PHY, such that the host processor could interface with ENMC through standard memory channels. Fig. 3.5 illustrates the details of the proposed ENMC architecture. The host processor contains several memory channels, which are deployed as ENMC

DIMMs. The ENMC logic locates at each rank of an ENMC DIMM, and thus enjoys scaling bandwidth offered by a larger number of ranks. The on-DIMM ENMC architecture consists of an ENMC controller, a DRAM controller, and two processing units: the Executor and the Screener. The ENMC controller buffers the instruction from the host processor for approximate screening. It also generates instructions for full-precision computation according to the candidate indices provided by the Screener. Then, it decodes the formatted instructions to generate control signals for data access, computation, and output transmission. The DRAM controller works as a simplified memory controller that processes data access requests in ENMC instructions and generates the standard DDR C/A signals to the DRAM chips. The Screener and Executor take charge of the approximate screening and the full-precision computation as described in Section 3.3.2, respectively. The Screener performs dimension-reduced INT4 computations to efficiently approximate the classifier's output. A preloaded threshold is used to filter out the important candidates based on the approximate results. Apart from floating-point arithmetic, the Executor is also equipped with a special-function unit to process the non-linear activation in the final layer. The two computation modules work in parallel and write results to the output buffer that returns them to the host processor asynchronously.

ENMC Controller. The ENMC controller has two main functionalities: processing the instructions from the host processor (i.e., screening computation) and generating instructions for the Executor (i.e., candidate-only computation). It is made of status register files, an instruction buffer, an instruction decoder, and an instruction generator. The status register files are used for ENMC initialization and store information such as addresses and sizes of input features, vocabulary, and screening weight. It also includes the instruction counter. The instruction buffer is a FIFO, and both the host processor and instruction generator could push instructions into it. The instruction decoder sequentially reads from the FIFO and generates control signals to corresponding ENMC components.

For example, an instruction of accessing a piece of tiled screening weight would result in a read request to the DRAM controller and a select signal to the top DEMUX that chooses the integer weight buffer. Meanwhile, a full-precision computation instruction would lead to a triggering signal to the floating-point MAC array, which reads data from two input buffers and writes results to the partial sum (PSUM) buffer. The instruction generator receives the indices of classification candidates from the Screener (*batch_id*, *candidate_id*), and then reads the constant reg to generate corresponding instruction for candidate-only computation in full-precision.

DRAM Controller. The DRAM controller employs a similar architecture as the host-side memory controller and consists of a request queue, a command generator, and an address generator. The request buffer takes memory requests from the ENMC controller. The command and address generators initiate standard DDR4 C/A signals that are sent to all the DRAM chips. For hardware simplicity, we do not deploy unnecessary features like queue prioritizing, request coalescing, etc.

Screener. The Screener processes the approximate screening phase in the approximate screening algorithm with fixed-point precision. We put two input buffers (feature buffer and screening weight buffer), a fixed-point multiply-accumulate (MAC) array, a partial sum (PSUM) buffer, a threshold filter, and an instruction translator in the Screener. The MAC array performs the screening computation over the two input buffers and accumulates with the intermediate results in the PSUM buffer. After a tiled screening is finished, the data in the PSUM buffer are filtered with a comparator array. The indices of values larger than the threshold are buffered and later sent to the ENMC controller.

Executor. The Executor computes candidate-only classification under full precision. Compared with the Screener, it applies a floating-point MAC array and has an extra special-function unit that performs non-linear activation such as Softmax and Sigmoid.

We also put an output buffer below the special-function unit, which caches both the results from the Screener and the Executor. The output buffer keeps the state of the data with status reg files and notifies the ENMC controller (by pushing a RETURN instruction) when finishing batched/tiled data.

Table 3.1: The ENMP instruction set

ENMC Instruction Set		
Type	Instruction	Description
Initial	INIT reg, data	initialize the ENMC module by writing a particular register

Data Transfer	LDR buffer, addr
	STR buffer, addr
	MOVE buffer1, buffer2
		load/store the quantized feature data into/from the INT4 feature buffer (weight buffer, with specified address addr

Compute	ADD_INT4 buffer1, buffer2 MUL_INT4 buffer1, buffer2 ADD_FP32 buffer1, buffer2 MUL_FP32 buffer1, buffer2	add/multiply the data in two specified buffer buffer1, buffer2
	MUL_ADD_INT4 MUL_ADD_FP32	multiply the data in feature buffer and weight buffer, and accumulate they with the partial sum buffer
	FILTER buffer	filter the data in the specific buffer and write the results to the index buffer
	SIGMOID, SOFTMAX	special functions such as Sigmoid and Softmax that run on specialized hardware for the data in the FP32 partial sum buffer
Control	BARRIER, NOP	synchronization and bubble instruction to let the controller wait for memory accesses, compute operation, data movement, etc.
	QUERY reg	query the value in the specific reg
	RETURN	return the data in the output buffer
	CLR	clear and reset all buffers and registers

3.4.3 ENMC Instruction Set

The design goal of the ENMC instruction set is to make the host processor able to communicate with ENMC DIMM through standard DDR4 memory channels. Inspired by FIRDRAM [52], we issue ENMC instructions from the memory controller with PRECHARGE command combining special addresses and data. For example, accord-

ing to the DDR4 JEDEC specification, for a 4Gb DIMM with 8 ×8 DRAM chips, the row address space consumes 14 bits, i.e., A0-A13 in the C/A bus, and the data bus is 64-bit. Normal PRECHARGE command sets all the row address bits to be low, since no row information is needed. Therefore, an ENMC instruction could be accommodated with sending a PRECHARGE command but turning on the row address signals. Given this insight, we design the ENMC instruction formatted in 13-bit command and 64-bit data that transmits through signal A0-A12 and D/Q bus. With that, we first present the instruction specification and explains the instruction in details. Then, we define the instruction format.

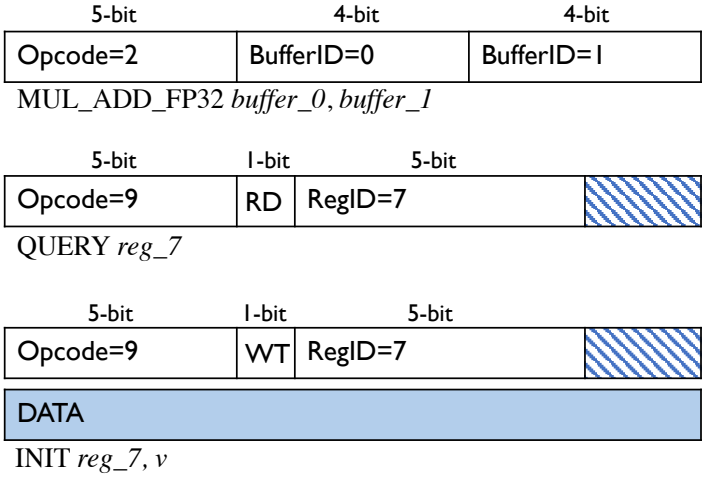


Figure 3.6: Instruction Format

Instruction Specification. As shown in Table 3.1, the ENMP instruction set consists of four types of instructions: Initialization, Data Transfer, Compute, and Control. (a) *Initialization.* The initialization instruction is used to write the status reg files in the ENMC controller, in order to initiate the parameters of a classification task. It specifies which reg to write and the corresponding value. (b) *Data Transfer.* The data transfer instructions are used to access the on-DIMM buffers, such as loading data to the input feature buffer or writing back the results to the PSUM buffer with specific addresses.

Also, we use the instruction MOVE to transfer data in two buffers, such as storing results in the PSUM buffer to the output buffer. (c) *Compute.* The compute instructions corresponds to the computation operations in the two computing units, including ADD, MUL, MUL_ADD, and denotes the operation precision. FILTER instruction is used to filter out the candidates. There are also instruction for special functions such as SOFTMAX and SIGMOID that operate on the PSUM buffer in the Executor. (d) *Control.* The control instructions include BARRIER for synchronization, NOP for stalling, RETURN to send back the output buffer data, and CLR to reset the ENMC. We also design a QUEUE instruction for the host processor to pull the status counters in each component.

Instruction Format. As shown in the Fig. 3.6, a typical ENMC command without data or address takes 13 bits, where the opcode is 5 bits and the rest 8 bits are used to specify which buffer to operate on. For example, Fig. 3.6(a) shows the instruction format for performing multiply-accumulate in the Screener. For the status register accessing instruction, QUERY and INIT shares the same opcode, and we use one bit after opcode to specify the read or write operation, and 5 bits to specify the register index, as shown in Fig. 3.6(b). Moreover, for instructions that involves values (i.e., data or address) that exceeds the length of row addresses, the DQ bus is further utilized. For example, when the host processor tries to write the status reg in the ENMC controller, the command address bus specifies the write operand and the ID of target regwith INIT instruction, and the DQ bus transmits the desired data in burst manner following the ENMC command.

3.4.4 System Design

In this subsection, we further architect the system-level design to facilitate existing software solutions running on the ENMC memory. We first present the programming support that wraps up ENMC instructions into high-level APIs such that a program

could call the ENMC kernels directly. Second, we show the execution flow to demonstrate how the host processor interacts with the ENMC DIMM.

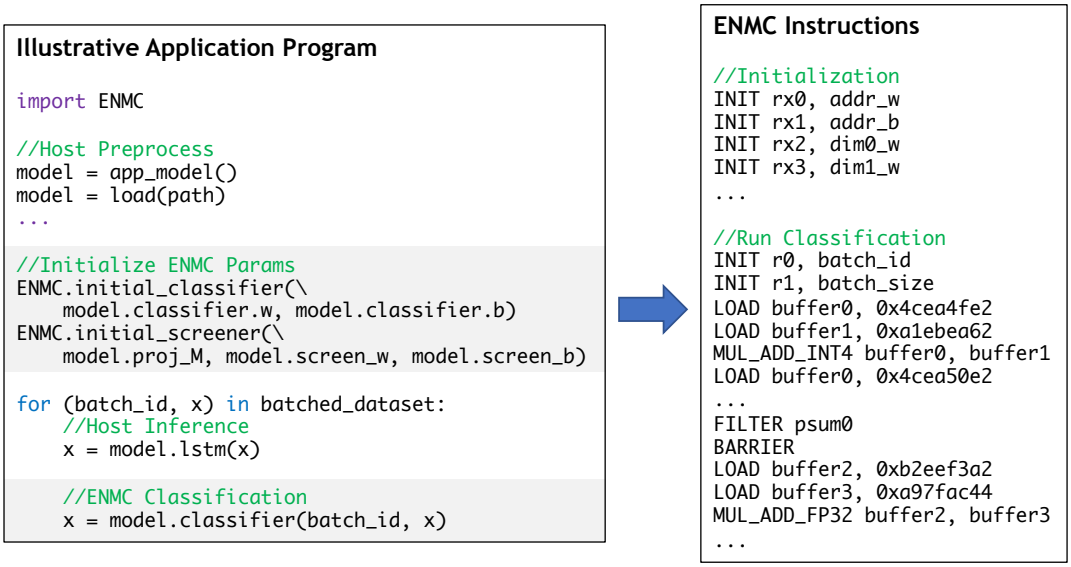


Figure 3.7: An illustrative example of programming support of ENMC. The ENMC APIs are wrapped as high-level function libraries, which are further compiled into ENMC instructions.

Programming Support. Following previous NMP solutions [53, 13], we divide the application code into kernels running on the host processor and ENMC in a heterogeneous manner. Therefore, the host processor calls the provided APIs to offload specific classification tasks. Fig. 3.7(a) shows an illustrative application code in Python style. We wrap up the functions that runs on ENMC DIMM into a Python package, such as initializing the Screener and screening-based classification. Therefore, a programmer could build a machine-learning model transparently using the ENMC package. Inside an ENMC object of classifier, we implement the approximate screening algorithm in the forward function with pretrained projection matrix and screening weight. Furthermore, when translating the applications into ENMC instructions, the compiler tiles the operation with initialized parameters and hardware configurations and executes the instruction in a loop. The ENMC instructions are further packed into a memory request packet and

routed to the memory controller, which transmits them to the ENMC DIMM, as shown in Fig. 3.7(b).

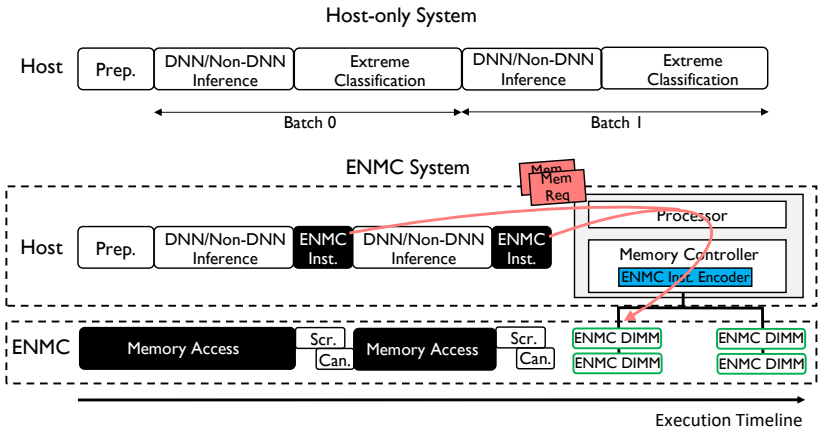


Figure 3.8: The ENMC workflow compared with a host-only system. ENMC offloads the classification tasks to the ENMC DIMMs by sending the instructions as memory requests through the memory controller.

Execution Flow. Fig. 3.8 presents the ENMC workflow compared with a host only system. The execution of front-end feature extraction (DNN-based or non-DNN-based) and the classification can be treated in a decoupled way. To be more specific, the host in the ENMC system is dedicated to run the feature extraction and offloads the classification tasks to the ENMC memory. The ENMC memory works as a regular main memory for data accessing in the first phase, and performs screening approximation and candidate-only classification in the second phase.

3.5 Evaluation Methodology

In this section, we discuss the methodology of evaluating the ENMC co-design, including the implementation details and performance metrics.

3.5.1 Software Evaluation

We implement the approximate screening algorithm on top of existing pre-trained models in the PyTorch machine learning framework [54]. The screening parameters are trained under mean-square-error (MSE) loss using the original training and validation datasets till convergence. Both the input features and the screening parameters are further quantized at inference time. We set the number of candidates, screening parameters size, and quantization precision adjustable for sensitivity studies.

Table 3.2: Evaluated models and datasets.

Dataset	#Categories	Inference Model	Hidden Size	Abbr.
Wikitext-2	33,278	LSTM	1500	LSTM-W33K
Wikitext-103	267,744	Transformer	512	Transformer-W268K
WMT16, en-de	32,317	GNMT	1024	GNMT-E32K
Amazon-670k	670,091	XMLCNN	512	XMLCNN-670K

Workloads. We evaluate our method on different tasks including Language Modeling (LM) [55], Neural Machine Translation (NMT) [34], and product-to-product recommendation [56]. For LM, we use the Wikitext-2 and Wikitext-103 datasets [43] and evaluate on both long short-term memory networks (LSTM) and Transformer networks. For NMT, we use the WMT16 English-to-German dataset and evaluate on Google's Neural Machine Translation System (GNMT) [42]. For product recommendation, we use the Amazon670K dataset [45] and evaluate on a Convolutional Neural Network based model [41]. Table 3.2 lists the applications, the models, and the datasets used in our evaluation, as well as the number of categories and the hidden dimensions. We also synthesize three larger datasets with 1 million, 10 million, and 100 million categories to study the scalability of ENMC (namely S1M, S10M, and S100M). For detailed and reproducible implementation, we will submit our implementation for artifact evaluation and open-source our repository after the anonymous review process.

Baselines. For comparison, we include two other approximation methods for classification: SVD-softmax [35] and FGD [57]. The SVD-softmax method leverages singular value decomposition (SVD) to approximate the classification weight with principle singular values; the FGD method uses graph-based nearest neighbor search to approximate top-k classifications. We implement both baselines in our PyTorch-based framework.

Table 3.3: ENMC Configurations

DRAM Configuration			
Spec	DDR4-2400MHz	DRAM Chip	8Gb×8
Channels	8	Ranks/CH	8
Queue	64-entry	Capacity/CH	64GB
Timing	CL-tRCD-tRP: 16-16-16 tRC=55, tCCD=4, tRRD=4, tFAW=6		
ENMC Configuration			
Tech Node	28nm	Frequency	400MHz
Executor Buffer	256B+256B	Screener Buffer	256B+256B
FP32 MAC	16	INT4 MAC	128

3.5.2 Hardware Evaluation

We implement the ENMC logic in RTL and synthesize it with Design Compiler for hardware parameters including timing, power, and area. We build a cycle-accurate simulator for the ENMC DIMM that interfaces with Ramulator [58] to derive the DRAM timing information. Since the host processor and the ENMC DIMM execute the feature extraction phase and the classification phase separately without complicated feature interactions in between, we simulate a simple host model that only issues ENMC instructions regularly according to the status registers.

Configurations. As shown in Table 3.3, the ENMC DIMM is based on DDR4-2400 specifications. Each rank consists of 8×8 DRAM chips that add to a total capacity of 8Gb. We put 8 memory channels for the host processor, and there are 8 ranks per chan-

nel, contributing to 64GB capacity and 21.3 GB/s bandwidth per channel. In addition, we synthesize our ENMC logic with TSMC 28nm technology, running on the frequency of 400MHz. The two input buffers and accumulation buffer in both Screener and Executor are 256B. We put 64 INT4 MACs and 16 FP32 MACs on each DIMM. For non-linear activations in the executor, we approximate the exponential function with Taylor expansion to the 4^{th} order.

Table 3.4: Comparing ENMC with three NMP baselines, all configured with similar area and power budget.

NMP Designs	Configuration	Est. Area (mm^2)	Est. Power (mW)
NDA [49]	4*4 Functional Units + 1KB Memory	0.445	293.6
Chameleon [50]	4*4 Systolic Array + 1KB Memory	0.398	249.0
TensorDIMM [13]	16-lane VPU + 512B Queue * 3	0.457	303.5
ENMC (Ours)	FP32 * 16 + INT4 * 128 + 256B Buffer * 4	0.442	285.4

Baselines. We compare ENMC with CPU and other NMP architectures, and all of them are equipped with the approximate screening algorithm. The CPU baseline is Intel Xeon Platinum 8280 @ 2.7GHz. It has 28 physical cores and 6 DDR4-2666 memory channels, contributing to a total memory capacity of 512 GB and 128GB/s ideal bandwidth. Three state-of-the-art DRAM-based NMP architectures are also selected for evaluation:

- **NDA** [49] provides a near-data acceleration solution by stacking coarse-grain reconfigurable accelerators (CGRA) with DRAM devices. The CGRA mainly consists of functional units, switches, and memory.
- **Chameleon** [50] is similar to NDA by employing a 2D architecture and focusing on how to integrate the accelerator with commercial DRAM. As Chameleon could work with any programmable compute unit, we put a systolic array as the accelerator core to distinguish it from NDA.

- **TensorDIMM** [13] is a NMP architecure for deep learning applications, especially for recommendation workloads. It leverages the VPU to accelerate the embedding operations in recommendation systems.

For a fair comparison, we configure the ENMC and three baselines with approximately the same area and power budget, as shown in Table 3.4; the control logic and DRAM device controller are excluded.

3.6 Evaluation

In this section, we evaluate the screening method for extreme classification and the micro-architecture of near-memory processing cores. For the method, we show the trade-offs between inference quality and speedup to CPU execution time of full classification. Then, we present the speedup of classification enabled by NMP co-design and the system performance improvements.

3.6.1 Algorithm-level Evaluation

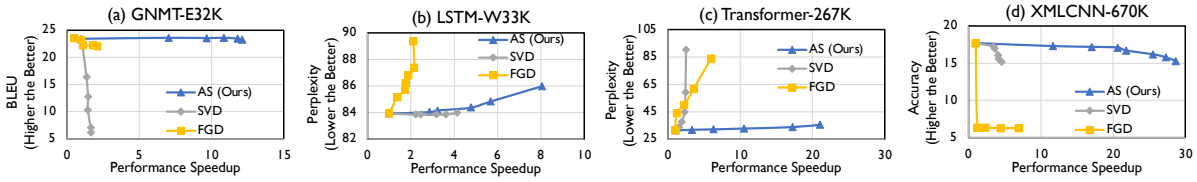


Figure 3.9: Quality vs. Speedup trade-off of Approximate Screening (AS) and two baselines: SVD and FGD.

Overall model quality. We post the hypothesis that extreme classification can afford approximation. Here we provide experimental results to support the hypothesis. Overall, our method can achieve significant computation saving with negligible model quality degradation. We can trade off model inference quality to an acceptable extend for more computation reduction.

As shown in Fig. 3.9(a), compared with using full classification as in NMT tasks, our method can achieve a speedup of 11.8× without any loss in translation quality measured by BLEU score. As for LM tasks, the speedups can reach 5.7× to 6.3× while preserving perplexity results, as shown in Fig. 3.9(b) and (c). Similarly, for product recommendations, our method can achieve a 17.4× speedup with only 0.5% drop in accuracy, as shown in Fig. 3.9(d).

Because of the good approximation that our method achieves, the screening phase can effectively select the key candidates for classification. Using the NMT task as an example, at every decoding step, we want the most likely word or a few words if using beam search. With Approximate Screening, we can identify the key candidates and compute the accurate probabilities of these words for translation, saving redundant computations for the remaining words in the vocabulary. We set the overhead of Approximate Screening to be 3.1% of full classification.

Compared with two other approximation methods, our method achieves a better quality-speedup trade-off, as shown in Figure 3.9. Besides, the computation overhead of SVD-based approximation is 4× more than ours. We infer that the improvement of our method is due to the learning-based approximation and no strong requirement for classifier weights to be low-rank.

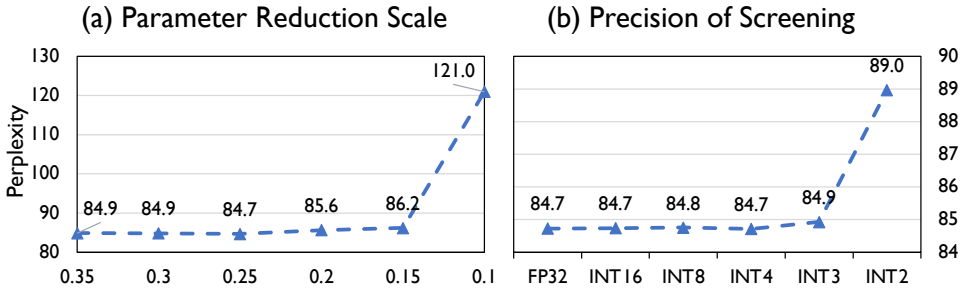


Figure 3.10: Comparing different (a) parameter reduction scales and (b) quantization levels of AS.

Sensitivity on Approximate Screening. Intuitively, better approximation costs

larger computation and data overhead, while achieving better model quality with screening. We show different parameter sizes of the screening module and the corresponding quality. Fig. 3.10(a) shows different parameter reduction scales of the screening module vs. full classifier; we choose the scale to be 0.25 as the good quality preserving. As shown in Fig. 3.10(b), we use 4-bit fixed-point quantization on the screening module as this quantization level maintains approximation as using single floating-point precision.

3.6.2 Architecture-level Evaluation

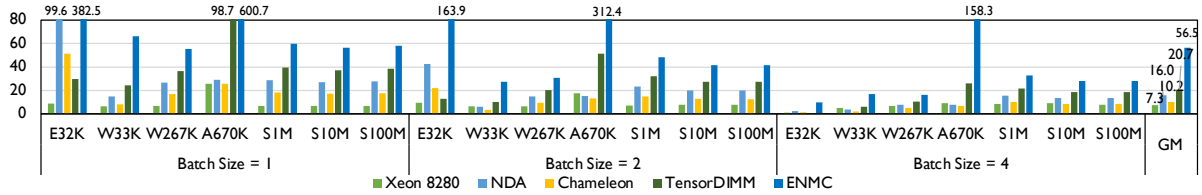


Figure 3.11: The performance results of ENMC, CPU, NDA, Chameleon, and TensorDIMM, normalized to vanilla CPU; all schemes are equipped with approximate screening.

Performance. As described in Section 3.5, we compare ENMC with four baselines. As shown in Fig. 3.11, we take the batch size of 1, 2, 4 and normalize the performance results to the full-classification CPU baseline for each workload, and arrange the results according to the size of classification across the x-axis. Our approximate screening demonstrates a 7.3× performance speedup on average in CPU baseline, and the ENMC offers a total 56.5× speedup over the CPU. Also, 3.5×, 5.6×, and 2.7× averaged speedups are observed when compared with NDA, Chameleon, and TensorDIMM respectively. First, we find ENMC provides significant speedups of 55.5×-600.7× when we do low-latency inference with a batch size of 1, because ENMC processes the inference in a streaming manner over the lightweight classification. The huge performance gain in XMLCNN-670K workload is because we considerably reduce the number of candidates

by 50×. Second, the three NMP baselines benefit from large internal bandwidth and offer 10.2-20.7× speedup over the CPU baseline. However, our ENMC could further boost their performance by 2.7-5.6× with heterogeneous resource management and dataflow customization. This result aligns our assumption that the performance of naive NMP solutions is bounded by the limited on-DIMM buffers and computation resources. Because they employ homogeneous FP32 computation units and hardly meet the throughput requirement in the screening phase. ENMC eliminates the redundant computation and needs only a small portion of FP32 computations. The entire screening phase is processed with lightweight INT4 units in stream.

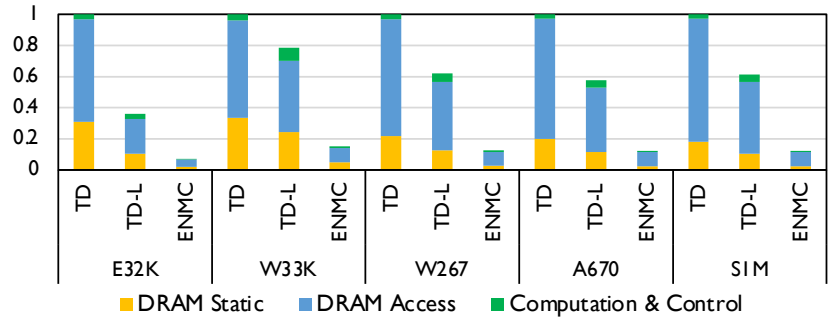


Figure 3.12: Energy breakdown by DRAM static cost, DRAM access, and computation & control logic, normalized to TensorDIMM.

Energy Consumption. We evaluate the energy results of ENMC against TensorDIMM and TensorDIMM-Large for a fair comparison. As shown in Fig. 3.12, we reduce the average energy cost by 5.0× and 8.4× compared with TensorDIMM and TensorDIMM-Large, respectively. Particularly, we breakdown the energy consumed by the DRAM static cost, DRAM access, and on-DIMM computation and control logic. We observe that the significant energy reduction of ENMC comes from two facts: First, the co-designed approximation algorithm greatly reduces the DRAM accesses in ENMC. In ENMC, we perform INT4 and low-dimensional screening during the classification phase, while TensorDIMM and TensorDIMM-Large need to operate over the full classification

weight. Moreover, due to the limited logic-side buffer size, TensorDIMM cannot store the intermediate results in a matrix multiplication operation. Thus, the buffer overflow results in frequent DRAM memory accesses. Second, the reduced execution time leads to the background energy reduction of the DRAM modules. As the DRAM takes a noticeable portion of power for refreshing, ENMC reduces the DRAM static energy consumption by 9.3× and 4.8× compared with TensorDIMM and TensorDIMM-Large.

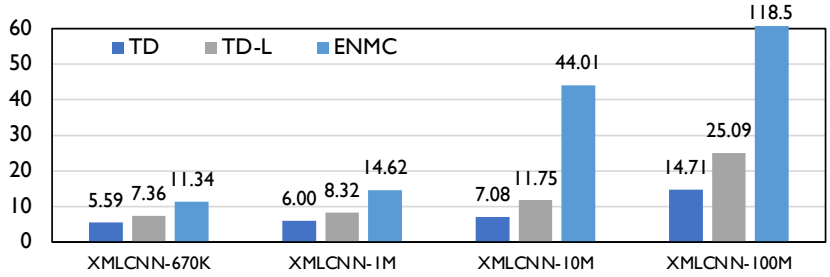


Figure 3.13: The end-to-end performance scalability compared with TensorDIMM and TensorDIMM large.

End-to-End Scalability. We evaluate the scalability of performance considering the end-to-end performance over large synthetic datasets. As shown in Fig. 3.13, we restrict the application to the same front-end model of XMLCNN, and the performance of TensorDIMM, TensorDIMM-Large, and ENMC is normalized to the CPU baseline. For comparison, ENMC achieves 4.7× and 2.9× speedup over TensorDIMM and TensorDIMM-Large. Particularly, for the two smaller datasets, ENMC achieves 2.2× and 1.6× speedups, while for the two tremendous datasets, ENMC achieves 7.1× and 4.2× speedups, compared with TensorDIMM and TensorDIMM-Large, respectively. The excellent scalability of ENMC comes from the fact that the ENMC processes the lightweight classification in stream and does not need to buffer large intermediate results back to DRAM.

Area and Power. Table 3.5 shows the breakdown area and power estimation of ENMC. The total area of ENMC logic is $0.388mm^2$, and the total power is 264.6mW, which are comparable to prior NMP architectures such as RecNMP [53]. Specifically,

Table 3.5: Area and Power Estimation.

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
INT4 MAC	0.013	10.4	**FP32 MAC**	0.145	58.0
Compute Buffer	0.061	56.8	**Control Buffer**	0.053	49.3
ENMC Ctrl	0.035	32.9	**DRAM Ctrl**	0.135	78.0
Total Area 0.442mm^2; Total Power 285.4mW					

the compute unit (INT4 and FP32 MAC arrays) takes 40.8% of the total area and 25% of the total power. The buffers made of register files in the Screener and the Executor compose of 23.5% of the total area and 32.2% of the total power. Finally, the ENMC controller and DRAM controller takes 9.0% and 34.8% of the area, and 12.4% and 29.5% of the power, respectively.

3.7 Conclusion

In this chapter, we address the extreme classification problem with NMP-based software-hardware co-design. We propose an approximate screening algorithm to reduce the computational complexity and memory consumption in classification. We further design a near-memory architecture to utilize efficient candidates-only classification enabled by our screening method. Finally, our approximate screening method achieves 7.3× speedups, and the ENMC architecture further improves the performance by 7.4× and demonstrates 2.7× speedup compared with the state-of-the-art NMP baseline.

Chapter 4

G-MEM: Custom Memory Hierarchy Design for Graph Processing

We have explored the near-memory architecture to accelerate extreme classification. However, the computation and data access in classification workloads are regular-patterned. In this chapter, we take a look into the memory design for an irregular-patterned workload, graph processing. Graph processing participates a vital role in mining relational data. But the intensive but inefficient memory accesses make graph processing applications severely bottlenecked by the conventional memory hierarchy. This chapter focuses on inefficiencies that exist in both on-chip cache and off-chip memory. First, graph processing is known dominated by expensive random accesses, which are difficult to be captured by conventional cache and prefetcher architectures, leading to low cache hits and exhausting main memory visits. Second, the off-chip bandwidth is further underutilized by the small data granularity. Because each vertex/edge data in the graph only needs 4-8B, which is much smaller than the memory access granularity of 64B. Thus, lots

of bandwidth is wasted fetching unnecessary data.

To address the inefficiencies, we present G-MEM, a customized memory hierarchy design for graph processing applications. First, we propose a coherence-free scratchpad as the on-chip memory, which leverages the power-law characteristic of graphs and only stores those hot data that are frequent-accessed. We equip the scratchpad memory with a degree-aware mapping strategy to better manage it for various applications. On the other hand, we design an elastic-granularity DRAM (EG-DRAM) based on the NMP technique. EG-DRAM processes and coalesces multiple fine-grained memory accesses together to maximize bandwidth efficiency.

4.1 Background and Overview

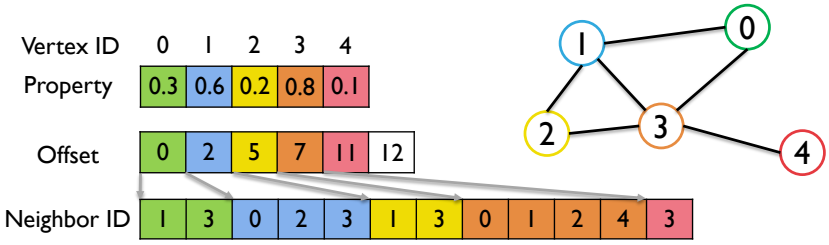


Figure 4.1: A CSR graph formatted with 3 arrays: Property array denotes the value in each vertex; Offset array denotes the index of the vertex's neighbors, which locate at the Neighbor ID array.

In many application domains, we use graphs to abstract the data of interests and excavate information from them, such as social networks, financial transactions, and knowledge databases [59, 15, 16, 60]. Graphs are usually stored vertex by vertex in a dense format, with neighbors attached to each vertex, since the edge connectivity between vertices is very sparse. For example, compressed Sparse Row (CSR) is a common-used data structure to represent a graph [61]. As shown in Fig. 4.1, typically three arrays are used in a CSR-formatted graph. The property array stores the value of each vertex, for

example, the PageRank score in the PageRank algorithm or the depth to the source vertex in the Breadth-First-Search (BFS) algorithm. there is an array storing the properties of all the nodes. The structural information is stored in 2 arrays: Offset array and Neighbor ID array. The offset array records the begin and end indices of the neighbors for each node, which can be found from the neighbor ID array.

CSR is the most compact data layout to represent a graph, but it cannot handle dynamic graphs, in which case graphs are evolving over time. Adjacency list and STINGER [62] that have variable neighbor arrays are used to capture such dynamics. Note that, the fundamental challenges of random and fine-grained access still remain among these data layout variants, requiring a general hardware solution to solve them.

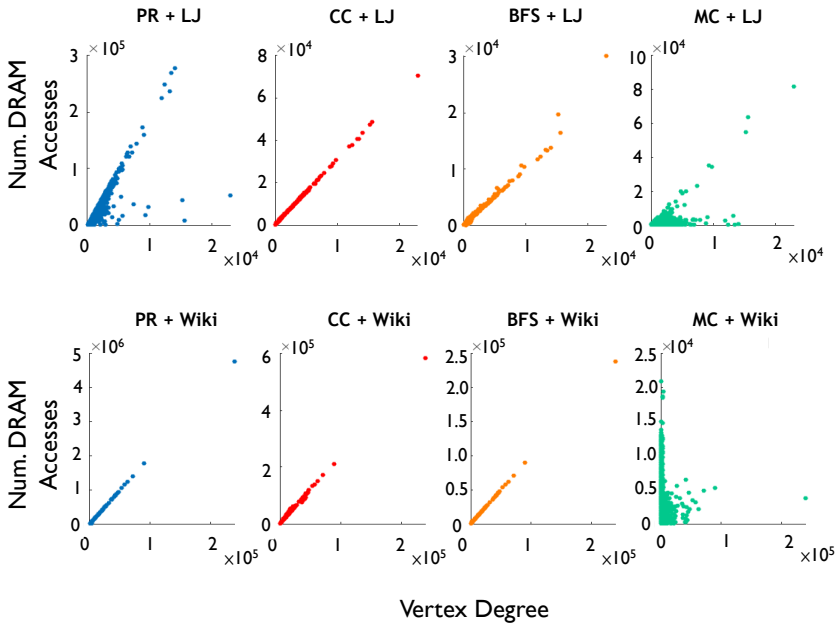


Figure 4.2: The relationship between the vertex degree (shown in x-axis) and the number of memory accesses to it (shown in y-axis). Four graph applications are evaluated on two datasets: LiveJournal and Wiki.

4.1.1 Memory Hierarchy Inefficiencies

In this section, we introduce the characterizations of graph workloads to identify the performance bottleneck in the memory hierarchy.

First, *random access nature severely degrades the performance of on-chip caches.* Lots of work has reported the low cache hits in graph applications [63, 64], since the traditional cache hierarchy is difficult to predict and capture the random access pattern. Typically, cache hit ratio of 10%-20% is observed in the L2 cache, while 30%-60% hit is observed in the LLC [63, 64]. Nature graphs are well-known to be power-law distributed, meaning few high-degree vertices are frequently accessed. As shown in Fig. 4.2, we find the number of accesses to each vertex is literally linearly related to its degree. This gives us hint that the on-chip memory should leverage the native feature from graph datasets.

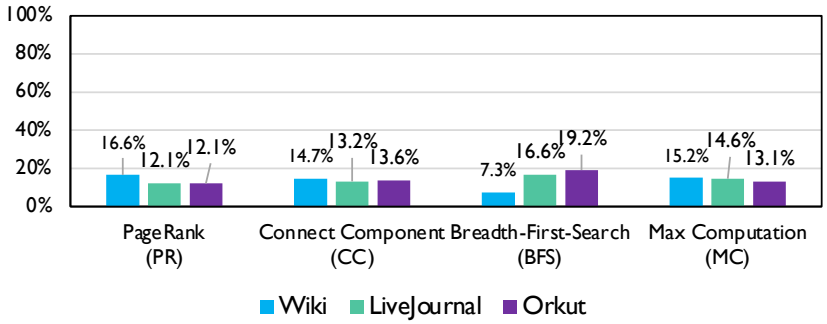


Figure 4.3: The bandwidth efficiency on the 8 graph processing workloads, which is measured as average touched data in a cacheline.

Second, *a cacheline is underutilized due to the fine-grained memory accesses pattern.* Modern DDR4 DRAM is designed to comply with a 64B cacheline in caches. However, previous studies revealed that the granularity of memory accesses in graph applications can be as small as 4B or 8B [65, 66]. To quantitatively evaluate the exact granularity that graph applications demand, we investigate how many portions of the 64B cacheline are actually torched. As demonstrated in Fig 4.3, we extracted the memory traces from various graph applications and derived the percentage of used cacheline by averaging the

touched data size within a time window. Besides, the vertex property data is set to be 4B. We find that even with a large memory window of 128, only less than 20% of 64B data are used. As we mentioned that high bandwidth is essential to the performance of graph processing, this means more than 80% of the memory bandwidth is wasted in fetching unnecessary data.

4.1.2 Related Work

Cache and Prefetcher Architectures: Prior work has broadly discussed the opportunity of graph-specialized caches and prefetchers to improve the data locality for on-chip memory. OMEGA [67] is a distributed scratchpad memory that locates individually beside each core to store high-degree vertices, requiring a fixed data flow and programming model to process the vertex mapping and communication; GRASP [64] extended the existing cache to enable the ability to identify and manage these hot vertices, but the time-consuming graph reordering brings significant overhead and highly limits the end-to-end performance. On the other hand, indirect memory prefetcher (IMP) [68] is designed for applications that exhibit pointer-chasing behavior, which prefetches data indexed by current data in the cache. But IMP could prefetch lots of unnecessary data. DROPLET [63] further improved IMP with dedicated units to identify the property, offset, and neighbor ID data explicitly, and thus fetches those vertices associated with currently cached vertices more precisely. HATS [69] also speculates the vertex data processed in the core and issues prefetches over the entire community of these vertices. Both DROPLET and HATS require a fixed data structure in the application, such that the hardware could directly recognize and operate on the graph data and generate prefetching commands.

G-MEM differs from the prior work by achieving both high flexibility and perfor-

Table 4.1: Comparison of G-MEM's on-chip memory with prior work.

	OMEGA [1]	GRASP [2]	IMP [3]	DROPLET [4]	HATS [5]	**G-MEM (ours)**
Flexiblity	×	✓	✓	×	×	✓
Performance	✓	×	×	✓	✓	✓

mance, as shown in Table 4.1. We put a general-purpose scratchpad as the on-chip memory, which does not require one particular data layout and data flow and could fit into various application scenarios. Instead, we design a data allocation strategy based on a lightweight reordering algorithm to manage the scratchpad from the software level.

Memory Subsystem Design: There are relatively fewer studies on specialized off-chip memory to accelerate graph processing. Dynamic-Granularity Memory Subsystem (DGMS) [70] and Adaptive-Granularity Memory Subsystem (AGMS) [71] leveraged narrowed memory bus to achieve fine-grained memory accesses. As a memory channel contains both the command/address (C/A) bus and data bus, such designs only narrow the width of the data bus but still need the same C/A bus. Therefore, the C/A bus then becomes a huge overhead considering the limited pin fan-out from the chip. We will explain this in detail in Section 3.2. On the other hand, Gather-Scatter DRAM [72] designs a DRAM memory that is able to perform strided gather/scatter memory access. However, as the conventional strided prefetcher does not work for the random memory accesses, this fix-pattern gather-scatter DRAM is not preferable for graph processing workloads.

G-MEM leverages the recent near-data processing (NDP) technology (specifically DRAM-based NDP technology), which puts a special-function unit on the DRAM DIMM to operate data near the memory and meet different application demands [53, 13, 73]. The key idea is to pack the sophisticated and fine-grained memory requests together on the DRAM DIMM and send them back to the processor. Therefore, G-MEM boosts bandwidth efficiency without introducing overhead to the system bandwidth.

4.1.3 Proposal

To overcome the inefficiencies in graph processing and achieve both high flexibility and performance, we present G-MEM, a customized memory hierarchy design for graph processing. We design the G-MEM based on two insights: First, as specialized cache architectures could limit the application flexibility or bring significant hardware cost, a software-managed on-chip memory is leveraged to facilitate access to the hot vertices in a graph. Second, the coarse-grained data in the memory bus comes from multiple DRAM devices in a DRAM DIMM, we could expect finer-grained accesses with fewer DRAM devices. Specifically, we design a coherence-free scratchpad on-chip memory, since the power-law characteristic of graphs is hardly captured by the traditional cache hierarchy. To facilitate the use of the scratchpad memory, we further propose a graph-aware mapping strategy to manage the scratchpad from the software level. In addition, we design an elastic-granularity memory subsystem based on near-data processing (NDP) architecture that has been widely studied in prior work. We make up our elastic DRAM DIMM with off-the-shelf DRAM devices, and each DRAM device provides 8B data access. We equip the DIMM with a gather-scatter unit, which processes and coalesces multiple fine-grained memory accesses together to maximize bandwidth efficiency.

Specifically, the contribution of this chapter includes:

- We design a coherence-free scratchpad as the on-chip memory to facilitate efficient access to the frequent data, coming with a proposed degree-aware vertex remapping strategy to exploit the graph's power law nature.

- We propose an off-chip memory composed of elastic-granularity DRAMs (EG-DRAMs) to coalesce fine-grained accesses and boost bandwidth efficiency. We design the instruction format of EG-DRAM such that the processor could communicate through the standard DDR channel.

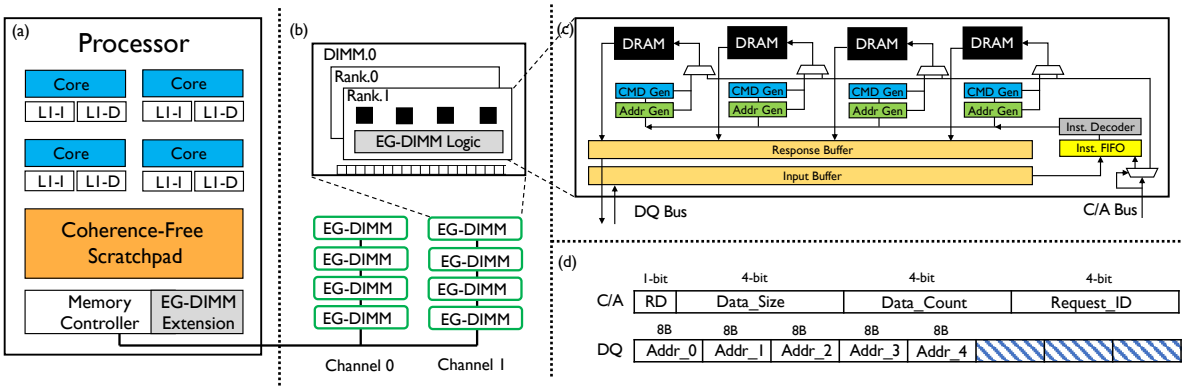


Figure 4.4: The overview of G-MEM design. (a) A multicore process equipped with G-MEM, including a coherence-free scratchpad and extended memory controller. (b) The elastic-granularity (EG) memory channels connected to the processor. (c) The microarchitecture of the elastic-granularity DRAM, which achieves fine-grained access by separating DRAM devices. (d) The instruction format of EG-DRAM, with packing multiple memory requests together.

- We further optimize the extension in the memory controller and the DDR4 protocol, to save the bandwidth demand and area overhead of the EG-DRAM.

- The Sniper-based [74] simulation demonstrates a 2.63× overall speedup over a vanilla CPU, with 1.44× and 1.79× speedup against the state-of-the-art cache architecture and memory subsystem, respectively.

4.2 G-MEM Architecture

In this section, we present the architecture design of G-MEM. We first give an overview of G-MEM, followed by the coherence-free scratchpad design and the degree-aware vertex remapping scheme. We then present the NDP-based elastic DRAM memory.

4.2.1 Overview

Fig. 4.4 presents an overview of the G-MEM architecture, which includes the on-chip coherence-free scratchpad (CF-scratchpad) and off-chip elastic-granularity DRAM

(EG-DRAM). First, we make the scratchpad memory coherence-free by having it share the memory space with DRAM. Therefore, each core can access it with a designated address space without maintaining the coherence between the scratchpad and the DRAM. We further propose a degree-aware vertex remapping scheme to manage the scratchpad from the software level. Second, we design the elastic-granularity DRAM as the off-chip memory. We leverage the near-data processing (NDP) technology to access the on-DIMM DRAM devices individually and achieve fine-grained data access. We also design the instruction format by leveraging the unused address line and data line in the PRECHARGE command to ensure compatibility with the commodity DDR4 interface.

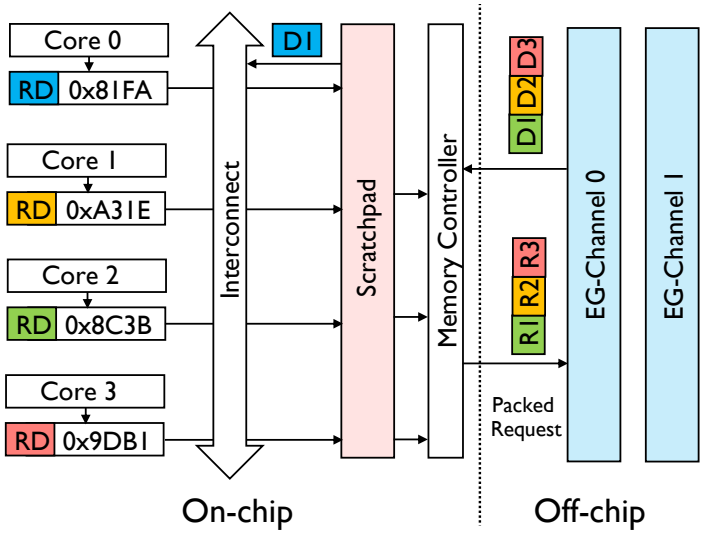


Figure 4.5: The data flow in the G-MEM hierarchy. Four read requests are issued from cores, where one is served by the on-chip scratchpad and the other three go through the off-chip EG-memory.

We give an illustrative data flow in G-MEM hierarchy in Fig. 4.5. Four read requests to different fine-grained data are issued from cores, which transit through the interconnection between cores and scratchpad. The scratchpad identifies that the address of request #1 locates within itself and sends the desired data back. Then, the other three requests are forwarded to the memory controller. With the EG-DRAM ex-

tension unit, the memory controller packs those requests together and sends them to the off-chip EG-DRAM. Thus, three reads could be served with only one memory walk to boost bandwidth efficiency.

4.2.2 Coherence-Free Scratchpad

In this section, we introduce our coherence-free scratchpad that locates in the same memory address space as DRAM, We then present how is the scratchpad memory managed at the software level.

Opportunity of using scratchpad memory. Concerning that some higher-degree vertices are accessed much more frequently than other vertices and hardware prefetchers are unlikely to predict such memory access pattern, a scratchpad memory that is managed at the software level is more appealing to store these high-degree vertices. Also, compared with a large shared cache, the scratchpad memory brings the benefits of easier hardware implementations and no tag access/hierarchy traverse time, meaning a smaller area overhead and lower access latency. A similar idea has been exploited in OMEGA [67], which equips each core with a scratchpad memory to store high-degree vertices by reducing the cache size. However, we here focus on a more general and flexible design and minimizing the traffic between the scratchpad and main memory.

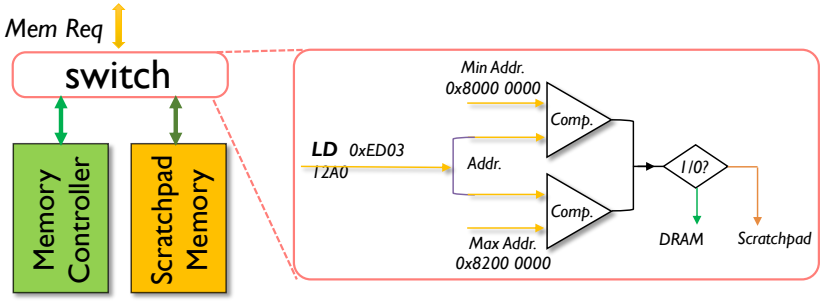


Figure 4.6: Coherence-Free Scratchpad

Avoiding coherency by memory space partitioning. To eliminate the coher-

ence between scratchpad and DRAM and reduce the bandwidth demand, we propose a heterogeneous memory subsystem consisting of both scratchpad and DRAM, meaning the scratchpad shares the same address space with DRAM and works as a fast main memory.

As shown in Fig. 4.6, first, when allocating the memory addresses, the memory management unit records a reserved memory space that is as large as the scratchpad memory. For instance, a 32MB scratchpad memory takes the address from 0x80000000 to 0x82000000, and any data allocated within this address belongs to the scratchpad. Then, when a memory request comes down to the memory controller, a switch determines to either route the request to the scratchpad or bypass it to the DRAM controller only according to the address of this request. Therefore, we make the scratchpad transparent to the program at run-time, and no special control of different data flows is required.

Graph-Aware Scratchpad Management. The system-level management of the scratchpad memory such as memory allocating mentioned above can be achieved by the memory management unit(MMU) [75, 76] where the scratchpad can be treated as another piece of memory in a similar way as the non-uniform memory access (NUMA) in multi-socket CPU [77]. We would like to explain more in detail about the software-level management for the scratchpad memory. For large-scale graphs, we can only expect a small portion of vertices can fit into the on-chip scratchpad memory. As discussed in Section 4.1.1, this small portion is preferable to be the high-degree vertices that are quite frequently accessed. However, the problem is these vertices are not stored continuously and also randomly exist in the node array.

Key idea: We propose a degree-aware vertex remapping scheme to tackle the issue of randomly distributed high-degree vertices. The scheme manages to exchange high-degree vertices to the front tail of the vertex array, such that they would locate in sequential address space in the scratchpad. For example, for a graph with 100 vertices,

the vertex 0-9 are almost high-degree vertex and locate in the scratchpad while vertex 10-99 with relatively lower degrees remain in the DRAM. In addition, since we do not essentially need to sort the vertices but just separate them instead, the algorithm only needs to scan over the vertex array and identify a high-degree vertex by a threshold, so heavy reordering is not required.

As illustrated in Algorithm 2, we maintain 2 vertex pointers, p_{low} and p_{high}, for recording the current low-degree vertex and searching the next high-degree vertex, respectively. Here we rely on some input degree threshold that may be derived from experiences and dataset characteristics to determine whether a vertex is high-degree or low-degree. As we find the expected p_{low} within the scratchpad region and one p_{high} outside the scratchpad, we switch the properties of these 2 vertices and pair them into a map. The algorithm terminates when we go to the end of the scratchpad or the total number of vertices, indicating that we already have enough hot vertices in the scratchpad or we cannot find any more hot vertices. In terms of the algorithm complexity, in the worst case, it takes at most $O(N) + O(E \cdot S)$ time under the condition that we switch out all the vertices in the scratchpad. However, since we are unlikely to face such many high-degree vertices due to the power-law distribution and the size of the scratchpad is relatively small, the computation complexity in normal cases is around $C \cdot O(E)$, which is compatible to the complexity of one graph processing iteration. Note that, existing pre-processing frameworks usually provide techniques like vertex sorting [78]. Our proposed vertex ID remapping strategy can be easily embedded into the pre-processing step at the software level.

Algorithm 2: Degree-Aware Vertex Remapping

Data: Graph $\mathcal{G}\{V, E\}$, degree threshold θ_{low} and θ_{high} , number of vertices in the scratchpad S ($S < |V|$).

```
1  begin
2      Initialize: N = |V|; Vertex pointers p_low = v_0, p_high = v_S; Vertex map V_map;
3      while p_low < v_S do
4          if p_low.degree() > θ_low then
5              p_low++;
6              continue;
7          end
8          while p_high < V_N do
9              if p_high.degree() < θ_high then
10                 p_high++;
11                 continue;
12             end
13             &p_temp = &p_high;
14             &p_high = &p_low;
15             &p_low = &p_high;    // Switch p_low and p_high
16             V_map.insert(p_low, p_high);
17             break;
18         end
19     end
20     # pragma omp parallel for
21     for e in E do
22         if e.source in V_map then
23             e.source = V_map[e.source]
24         end
25         if e.dest in V_map then
26             e.dest = V_map[e.dest]
27         end
28     end
29 end
```

4.2.3 Elastic-Granularity DRAM

In this section, we introduce our design for the elastic-granularity DRAM (EG-DRAM). Additionally, we design the instruction format for accessing multiple fine-grained data in a packed request.

Challenge and opportunity in conventional DRAM. Given that on average 14.0% of one 64B cache line (around 9B) is actually accessed in graph applications as characterized in Section 4.1.1, a memory access granularity of 8B appears more reasonable and efficient. However, the current DDR4 DRAM is burst-oriented and designed to fit with the 64B cache line size [79]. This is then the smallest granularity we can expect per transaction. Even though the DDR4 offers configurable burst length (for example, setting burst length from 8 to 2 to access 16B data each time), this, however, only affects memory controller settings on the processor side but not the DRAM. As a result, the DRAM still sends 16B data and another 48B invalid data back to the controller, meaning that we cannot actually have bandwidth gain from changing the burst. Moreover, 16B granularity seems still larger than what we want of 8B.

The opportunity inspiring our EG-DRAM comes from the DRAM's internal hierarchy. Generally, a DRAM DIMM is organized as *rank*, *chip*, *bank*, *row*, and *column*. When accessing DRAM, we have to specify the addresses and choose which *rank*, *bank*, *row*, and *column* the data is located in, but all the *chips* within the same rank share these addresses. Each DRAM *chip* (or DRAM device) outputs 4- or 8-bit data (what is called x4 chip or x8 chip) every clock edge. With 8 dual-edge clocks of bursting, 4B or 8B fine-grained data could be expected from a single chip.

Elastic-Granularity (EG)DRAM design. The key idea of EG-DRAM is cutting down the number of DRAM chips/devices for each individual memory access. As shown in Fig. 4.4(c), the EG-DRAM mainly consists of the instruction FIFO, instruction decoder,

command (CMD)/address (Addr) generator, and input/response buffer. Multiplexers are also used to bypass the EG-DIMM logic and serve regular memory requests.

Instruction FIFO & instruction decoder: The instruction FIFO receives packed memory requests from both the C/A bus and DQ bus (which is first cached at the input buffer). The multiplexer determines if the coming request is an EG-instruction, such that it pushes the request to the instruction FIFO or directly forwards it to the DRAM devices. The instruction decoder reads from the FIFO and dispatches the request to different DRAM devices separately, such that finer-grained data could be achieved.

Command & address generators: Each DRAM device is equipped with an individual command/address generator. They are finite-state machines that follow the standard DDR4 protocol and work as signal generators to activate the DRAM device. To minimize their area overhead, we simplify and optimize the states within them, which will be discussed in Section 4.2.4.

Input & response buffer: Since the regular DDR4 channel is synchronized but our EG-instruction is processed asynchronously, the two buffers are used to cache the data from/to the DQ bus. The input buffer stores the instruction from the DQ bus, while the response buffer cache the data for a packed request.

EG-instruction format. The goal of designing EG-instruction is compatibility with DDR4 protocol, such that the memory controller can communicate with EG-DRAM through standard DDR4 memory channels. Inspired by FIRDRAM [52], we issue ENMC instructions from the memory controller with PRECHARGE command combining special addresses and data. For example, according to the DDR4 JEDEC specification [79], for a 4Gb DIMM with 8 $\times$8 DRAM chips, the row address space consumes 14 bits, i.e., A0-A13 in the C/A bus, and the data bus is 64-bit. Normal PRECHARGE command sets all the row address bits to be low, since no row information is needed. Therefore, an ENMC instruction could be accommodated by sending a PRECHARGE command but

turning on the row address signals.

Given this insight, we design the ENMC instruction formatted in 13-bit command (line A0-A12) and 64B data (DQ bus). As shown in Fig. 4.4 (d), in the command line, we use 1 bit to denote request type (read or write), while the data size, data count, and request ID are specified with 4 bits respectively. On the other hand, the 64B (under bursting) in the DQ bus are used for the addresses for these requests. It is mandatory that the multiple requests packed in one instruction are the same type and the same data size, such that the returned data could be formatted in a strided pattern. Note that for a write instruction, the DQ bus needs 2 bursts to send both the addresses and data.

4.2.4 Memory Controller Extension and Optimization

In this section, we introduce the extension to the existing memory controller for EG-DRAM. We then optimize the DDR4 commands for better C/A bus efficiency.

Memory controller extension. To encode and issue the EG-instruction, we need to extend the memory controller. The main task of the extension unit is to scan the memory request queue through a fixed window and coalesce the proper requests together as a packed instruction. It leverages the address mapping unit to identify those requests within the same DRAM rank, such that they could be packed simultaneously. Note that, the data type of vertex properties in a graph are fixed for one workload, i.e., they are either *float*, *int*, or *long*. Therefore, finding data of the same size is not difficult.

DDR4 protocol optimization. Two design considerations motivate the optimization for the existing protocol: First, since we currently need the C/A bus to send both regular DDR4 commands (PRECHARGE, ACTIVATE, READ, etc) and EG-instruction, the memory controller may encounter C/A bandwidth bottleneck during the execution. Second, the limited on-DIMM area requires us to minimize the size of command/address

generator, since they are copied individually for each DRAM device and could be a large overhead. Therefore, the goal of the optimization is to reduce the number of commands in the DDR4 protocol, which could save both bandwidth and area.

Opportunity: We find that there are generally 3 steps to access the DRAM: *precharge* (write back the current opened row), *row activate* (open another row), and *column access* (read/write the data), all of which have the corresponding command in the DDR4 protocol. Since the command bus is not double-rate, meaning 3 cycles are needed compared with the 4 data cycles. However, as the graph applications are filled with random memory accesses, we may hardly gain many row buffer hits and the current opened row appears to write back again and again. This means letting the data wait at the row buffer does not bring much benefit and we could save one cycle by precharging the row buffer automatically after the column read/write operations.

Therefore, we re-design the finite-state machine in both the memory controller, by replacing all READ/WRITE commands to READ_AP/ WRITE_AP. Thus, the C/A bus or signal generators only need to send 2 commands for one access. This eliminates the *precharge* command, and only 2 cycles in the command bus are occupied in accessing 4-cycle data, leaving half of the command bandwidth being idle.

4.3 Evaluation

In this section, we present the evaluation results of our G-MEM. We first clarify our experiment methodologies and simulation setup. Second, we present the comparison of G-MEM to prior work, along with the sensitivity studies. Finally, we discuss the area and power overhead in the EG-DRAM architecture.

4.3.1 Methodology

Evaluation Tools: We evaluate the G-MEM based on trace-based and cycle-accurate simulations through the Sniper multi-core simulator [74], with modifications on the memory hierarchy and DRAM subsystem. With the interface provided in Sniper, we model the power consumption and area breakdown with McPAT [80].

Configurations: We configure a 32-thread system with four memory channels, while each thread has 32KB L1-D and 32KB L1-I caches. The scratchpad size is set as 32MB, considering we do not enable L2 cache. Each memory channel consists of 4 DDR4-2666 ranks, and each rank has 8×8 DRAM chips that add to a total capacity of 8Gb. In addition, we synthesize our EG-logic with TSMC 28nm technology, running on the frequency of 400MHz. The input buffer and response buffer have the size of 512B respectively, aligned with the size of maximal requests.

Baselines: Since we targeting general-purpose graph processing, we take the performance of a vanilla 32-thread CPU as the main evaluation baseline. We also compare the coherence-free scratchpad (CF-scratchpad) with the GRASP, a recent domain-specific cache design for graph analytics [64]. Finally, we choose the Gather-Scatter DRAM (GS-DRAM) [72] as the baseline of our EG-DRAM. We clarify that GS-DRAM is not designed specifically for graph processing, and the strided gather/scatter may not be preferable for random accesses. However, GS-DRAM is still closely related to our EG-DRAM, as we both customize the DRAM to collect multiple memory requests in one transaction.

Table 4.2: Three graph datasets that are evaluated among four algorithms.

Datasets	Type	Nodes	Edges
wiki-topcast (Wiki)	Hyperlink	1,791,489	28,511,807
soc-LiveJournal1 (LJ)	Social Network	4,847,571	68,993,773
Orkut	Social Network	3,072,441	117,185,083

Graph Workloads: We use 2 algorithms: PageRank (PR), connected component (CC) from GAP benchmarks [78] (for static graphs), and 2 algorithms: bread-first searching (BFS) and max computation (MC) from SAGA benchmarks [81] (for streaming graphs). These two benchmarks have no fixed programming model and well-optimized codes for multithreading. Thus, they are considered state-of-the-art benchmarks. Three datasets are used from the SNAP dataset collection [82], as shown in Table 4.2. Moreover, GAP processes datasets into the CSR format, while SAGA stores them as adjacency lists.

4.3.2 Performance

In this section, we demonstrate the performance results of G-MEM, including the performance speedups compared with the CPU baseline and prior work.

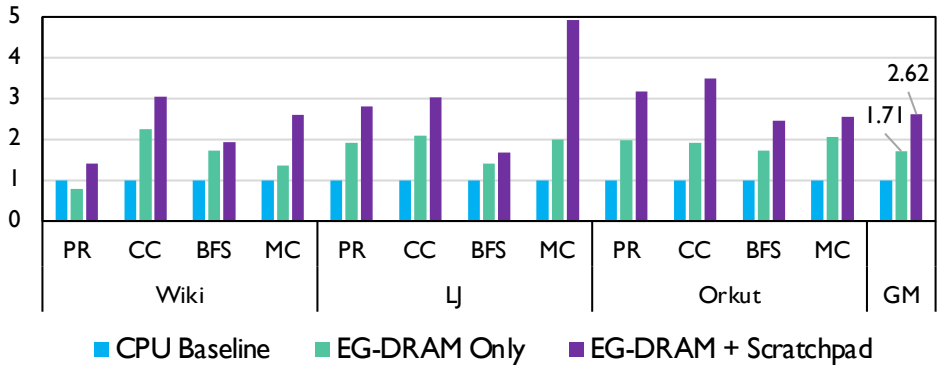


Figure 4.7: The overall performance of G-MEM compared with the CPU baseline. We configure the G-MEM without/with the coherence-free scratchpad.

Overall Performance. Fig. 4.7 shows G-MEM's overall performance results compared with the CPU baseline, where two configurations (without/with the scratchpad) are used to present the performance breakdown of on-chip scratchpad and off-chip DRAM. First, G-MEM without/with the scratchpad achieves 1.71/2.62 speedup on average over the CPU baseline respectively. This indicates both the coherence-free scratchpad and

elastic-granularity DRAM offers considerable speedups to the system. Moreover, we observe that for PR running on Wiki, the EG-DRAM actually degrades the baseline performance. We explain this as a result of the PageRank implementation style and Wiki dataset characteristics. The GAP benchmark implemented PageRank by traversal over all the vertices, whereas other algorithms are active vertices only. This introduces relatively more sequential accesses with parallel threads. On the other hand, the Wiki dataset is reported as a high-skew graph [83], where some vertices have extremely large degrees (>230,000). This could also lead to a sequential access pattern when accessing the neighbors of such vertices.

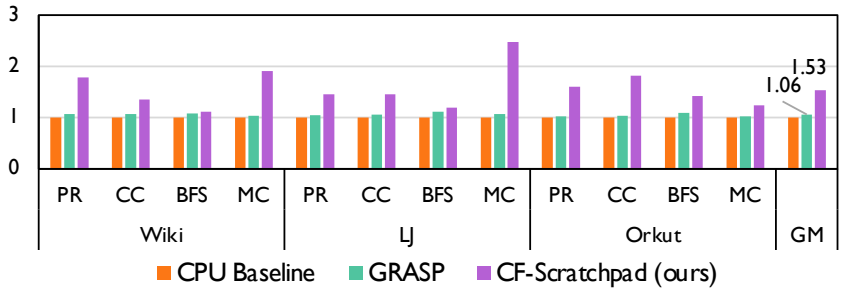


Figure 4.8: The end-to-end performance of CF-scratchpad compared against the GRASP, where both the graph reordering in GRASP and our vertex remapping are included in the performance.

Compared with prior work. As shown in Fig. 4.8, we compare the end-to-end performance of CF-scratchpad with the GRASP, the state-of-the-art cache design for graph analytics [64]. We achieve 1.44× average speedup against the GRASP. Specifically, we find GRASP only achieves 1.06× speedup over the CPU baseline (indeed, GRASP reported 1.04× speedup originally in the paper). This is caused by the large preprocessing overhead included in the end-to-end performance. As GRASP relies on heavy graph reordering algorithms to fit the high-degree vertices into caches, the exhausting reordering time migrates the performance gain from the hardware. Different from GRASP, our degree-aware vertex remapping is threshold-based and lightweight, such that incurring

low overhead to the performance.

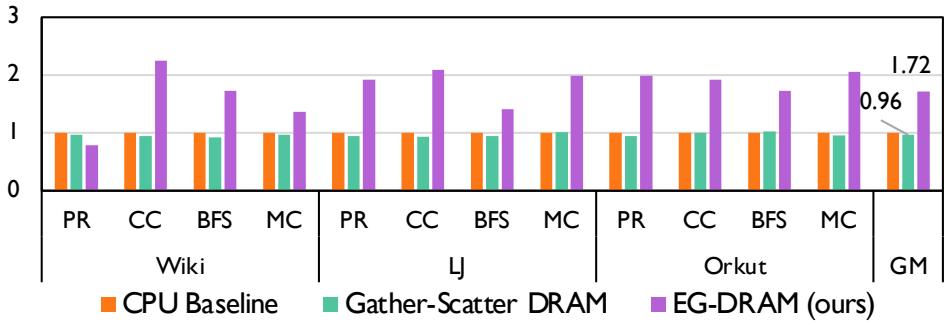


Figure 4.9: The performance of EG-DRAM compared against the GS-DRAM. The near-baseline performance of GS-DRAM is because the strided gather/scatter is not preferred in graph processing.

Fig. 4.9 shows the comparison between GS-DRAM and EG-DRAM, and we achieved 1.79× speedup over the GS-DRAM. As we mentioned before, GS-DRAM may not be suitable for graph workloads. It explores the strided gather and scatter only, hardly coalescing many requests among the randomly distributed data. Therefore, the performance of GS-DRAM is expected to be similar to regular DRAM in the random-access scenario. Our EG-DRAM overcomes this issue by exploiting more complicated on-DIMM logic to serve individual requests from each DRAM device, such that the bandwidth is easily saturated.

4.3.3 Sensitivity Study

In this section, we analyze the sensitivity of G-MEM with different hardware configurations, including the scratchpad size and the DRAM device width.

Sensitivity to different scratchpad size. We use the scratchpad hit ratio to demonstrate the efficiency of the scratchpad, as higher hit ratio leads to reduced off-chip memory traffic. As shown in Fig. 4.10, we vary the size of the scratchpad from 8MB to 64MB and compare their hit ratio. We find that larger scratchpad always outperforms

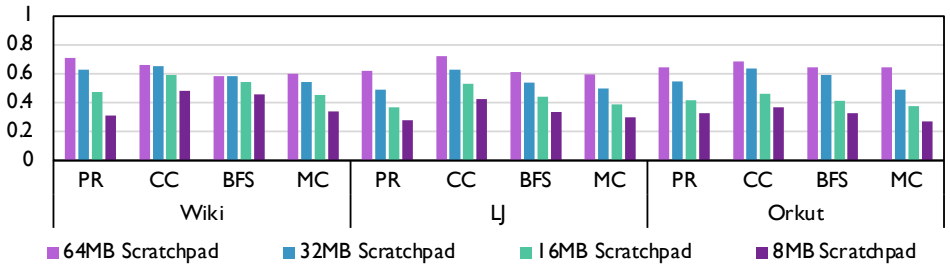


Figure 4.10: The scratchpad hit ratio under different scratchpad size settings, varying from 8MB to 64MB.

smaller ones. With increasing the capacity by 2×, we observe a higher hit ratio increased by 10.3%, 11.5%, and 7.4% for 16MB, 32MB, and 64MB respectively. This indicates that we cannot expect a linear increase in the hit ratio from enlarging the capacity. Considering the trade-off between area and efficiency, we take the size of 32MB as our system configuration.

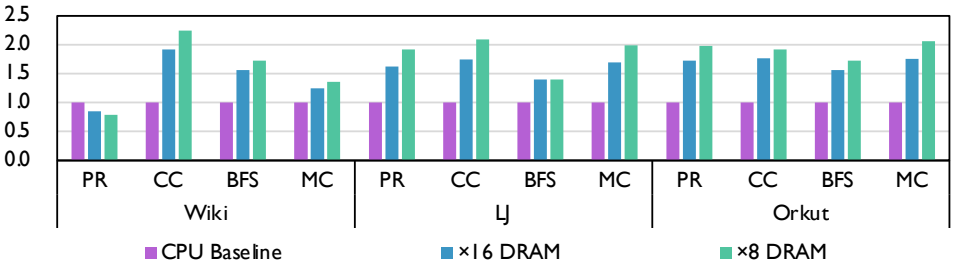


Figure 4.11: The performance results under two DRAM device configurations.

Sensitivity to different DRAM configurations. We configure the DRAM device on the EG-DIMM to ×8 chip and ×16 chip, which results in 8B and 16B accessing granularity for each transaction. As shown in Fig. 4.11, we find that except for the PR on Wiki, the ×8 DRAM demonstrates a better performance than the ×16 DRAM by 11%. For the PR on Wiki, the ×8 DRAM appears less efficient than the ×16 DRAM, which aligns with our explanation in Section 4.3.2: this workload explores more sequential accesses, resulting in a better performance for a coarser-grained DRAM.

4.3.4 Power and Area Breakdown

Table 4.3: Area and power estimation of EG-DRAM's logic.

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
Inst. FIFO	0.017	14.2	**Inst. Decoder**	0.002	2.3
Input Buffer	0.034	28.4	**Response Buffer**	0.034	28.4
CMD/Addr Gen.	0.159	149.6	**Others**	0.001	1.2
Total Area 0.247mm^2; Total Power 224.1mW					

Table 4.3 shows the breakdown area and power estimation of EG-DRAM overhead. The total area of EG logic is $0.247mm^2$, and the total power is 224.1mW, which are quite insignificant considering the area and power of DRAM DIMM are in the order of hundred mm^2 and W [84]. Specifically, the command and address generators take 64.3% of the total area and 66.8% of the total power. The buffers compose 27.5% of the total area and 25.3% of the total power. Finally, the control logic, including instruction FIFO and instruction decoder take 7.7% of the total area and 7.4% of the total power.

4.4 Conclusion

In this chapter, we present G-MEM, a customized memory hierarchy design for graph processing applications. First, we propose a coherence-free scratchpad as the on-chip memory, which leverages the power-law characteristic of graphs and only stores those hot data that are frequent-accessed. We equip the scratchpad memory with a graph-aware mapping strategy to better manage it for various applications. On the other hand, we design an elastic-granularity memory subsystem based on near-memory processing (NMP) architecture, which processes and coalesces multiple fine-grained memory accesses together to maximize bandwidth efficiency. Putting them together, the G-MEM demonstrates a 2.63× overall speedup over a vanilla CPU, with 1.44× and 1.79× speedup

against the state-of-the-art cache architecture and memory subsystem, respectively.

Chapter 5

INPSIRE: In-Storage Private Information REtrieval via Protocol and Architecture Co-design

The previous chapters demonstrated how to apply Near-Data Processing (NDP) technique for existing DRAM memory, and the targeted workloads usually have a small memory footprint. To further explore the scalability of NDP, this chapter investigates the workload that has a much larger footprint and cannot fit into the DRAM memory - private database systems. In this chapter, we first give the background and motivation for accelerating Private Information Retrieval (PIR), which is an important primitive for database privacy. We then introduce our INSPIRE design, the first in-storage processing (ISP) architecture to facilitate private query processing. INSPIRE follows a protocol and architecture co-design approach to reduce the query size and speed up the execution. Finally, we present the evaluation results and conclude this chapter.

5.1 Background and Motivation

Before we dive deep into our design, we introduce the technical background for private information retrieval (PIR), fully homomorphic encryption (FHE), the state-of-the-art FastPIR protocol [85], and performance bottlenecks in FastPIR. Then, the high-level ideas of our proposal are discussed.

5.1.1 Private Information Retrieval (PIR)

With more data being moved to the cloud, database systems have grown quickly to become the backbone of many daily applications [86, 86, 87]. As a result, the demand for user privacy has turned into an increasingly concerning issue: *when accessing the database, can we prevent the server from knowing where parts of the database the user is accessing?* In 1995, Chor et al. introduced Private Information Retrieval (PIR) to address this problem [88]. Subsequent research has extensively studied the broad applications of PIR protocols, including anonymous communication [89, 90, 85, 91], content sharing [92, 93, 94], and business services [95, 96].

There are two lines of PIR protocols: information theoretic PIR (IT-PIR) [97, 98, 99, 88] and computational PIR (CPIR) [100, 101, 102, 85, 103]. IT-PIR protocols replicate the database across multiple non-colluding servers. The client sends different queries to these servers and derives the answer (the desired database record) by combining the responses. IT-PIR protocols achieve information-theoretic security against adversarial attacks [88]. On the other hand, CPIR protocols put the database onto a single server, while guaranteeing security against computationally-bounded adversaries. In this work, we focus on the single-server CPIR because it is more practical than deploying non-colluding servers.

The key insight behind single-server PIR is the *all-for-one* concept. This means that

to retrieve **one** record from a database obliviously, the server should necessarily compute over **all the records** in the database. This is necessary because otherwise the server would learn which record the user is not interested in [104]. The recent breakthrough in fully homomorphic encryption (FHE) [105] has greatly expedited the development of single-server PIR. We discuss the FHE-based PIR protocol with more details in Section 5.1.2 and 5.1.3.

5.1.2 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) Modern PIR protocols rely on Fully Homomorphic Encryption (FHE) to conceal the query information. FHE is a type of encryption scheme that allows generic operations on encrypted data (ciphertext). In the most popular FHE schemes, such as BFV [106, 107], BGV [108], and CKKS [109], the raw data that is encrypted is a vector, and the ciphertext is a polynomial (represented as a vector of polynomial coefficients). Therefore, FHE programs follow a vector programming model, as most FHE operations involve element-wise computation between vectors.

Algorithm 3: FHE Primitives − Client

```
1 Function VecEncrypt(V, pk):
      /* Encrypt a vector V = [v1, v2, v3] into ciphertext C with public key
         pk.                                                              */
2 return C

3 Function VecDecrypt(C, sk):
      /* Decrypt a ciphertext C into a plain vector V = [v1, v2, v3] using the
         secret key sk.                                                   */
4 return V
```

Algorithm 3 and Algorithm 4 present the FHE primitives needed for PIR, for a small example vector containing three elements. At the client side, the function `VecEncrypt` encrypts a raw vector $V = [v_1, v_2, v_3]$ into a **ciphertext** C, where C is a polynomial of

Algorithm 4: FHE Primitives – Server

```
   /* C1=VecEncrypt([v1, v2, v3]), C2=VecEncrypt([w1, w2, w3])          */
1  Function Hom_Add(C1, C2):
   |  /* Cout=VecEncrypt([v1 + w1, v2 + w2, v3 + w3])                  */
2  return Cout

3  Function Hom_Mul(C1, C2):
   |  /* Cout=VecEncrypt([v1 × w1, v2 × w2, v3 × w3])                  */
4  return Cout

   /* C1=VecEncrypt([v1, v2, v3]), W2=[w1, w2, w3]                     */
5  Function Hom_Add(C1, W2):
   |  /* Cout=VecEncrypt([v1 + w1, v2 + w2, v3 + w3])                  */
6  return Cout

7  Function Hom_Mul(C1, W2):
   |  /* Cout=VecEncrypt([v1 × w1, v2 × w2, v3 × w3])                  */
8  return Cout

9  Function Hom_Rot(C1, rk, step = −1):
   |  /* Rotate C1 one step to the left:  cout=VecEncrypt([v2, v3, v1])  */
   |  /* Distinct rotation key rk is needed for different steps (the third
   |     input parameter).                                              */
10 return Cout
```

degree 3 (the same length as V). During encryption, the vector V is usually termed a **message**. It is first encoded into a polynomial, called a **plaintext**, and then encrypted to form the ciphertext. Conversely, the function `VecDecrypt` decrypts a ciphertext to a plain message. The decryption requires a secret key sk that is only known to the client.

The server-side uses three types of FHE operations: `Hom_Mul`, `Hom_Add`, and `Hom_Rot`. The `Hom_Add` and `Hom_Mul` take two ciphertexts as input and return the encryption of element-wise addition/multiplication. Note that these two functions can also take plaintext W as input. `Hom_Rot` is a special operation that rotates the elements in the plain vector according to *step*. The sign of *step* denotes the direction of rotation. For example, with $V = [v_1, v_2, v_3]$, rotating V one step to the left ($step = -1$) will result in an encryption of $[v_2, v_3, v_1]$. For different values of steps, FHE requires different rotation keys rk;

these keys are generated by the client.

FHE Computation Complexity: Adding two polynomials (ciphertexts) is simple, which causes $O(M)$ time complexity with the polynomial degree of M. `Hom_Mul` needs more complicated computation, as multiplying two polynomials requires convolution. Number Theory Transfer (NTT) is widely used to accelerate this computation [110]. NTT is a variant of Discrete Fourier Transfer (DFT), which can transfer the convolution (in the time domain) to the element-wise multiplication (in the frequency domain). The computation complexity of `Hom_Mul` is then $O(NlogN)$ for NTT and inverse NTT. When multiplying two ciphertexts, the resulting ciphertext usually needs to be relinearlized with a sophisticated key switching process [110], during which NTT is also the dominant operation. Finally, `Hom_Rot` requires key switching and data reordering/shuffling. We will discuss this in more detail in Section 5.3.4.

As a remark, in PIR the database content is public and encoded into plaintexts after the NTT computation. Therefore, PIR performs more plaintext-ciphertext multiplications (the ciphertext is the PIR query).

5.1.3 The State-of-the-art: FastPIR

Fig. 5.1 presents the dataflow overview of the FastPIR protocol [85], which contains the following steps for an example database with 4 records $[a, b, c, d]$, where the client wants to fetch the 3^{rd} record c. ① The client generates a query q which is an FHE ciphertext that encrypts a one-hot vector of length 4, where only the 3^{rd} slot in the vector is 1 and others are 0. ② The client then sends the query ciphertext to the server. ③ The server partitions the database into multiple vectors (columns) to facilitate the vector program in FHE. As a result, each record in the database is partitioned into 3 slices. ④ The server performs 3 `Hom_Mul` operations between the query ciphertext and

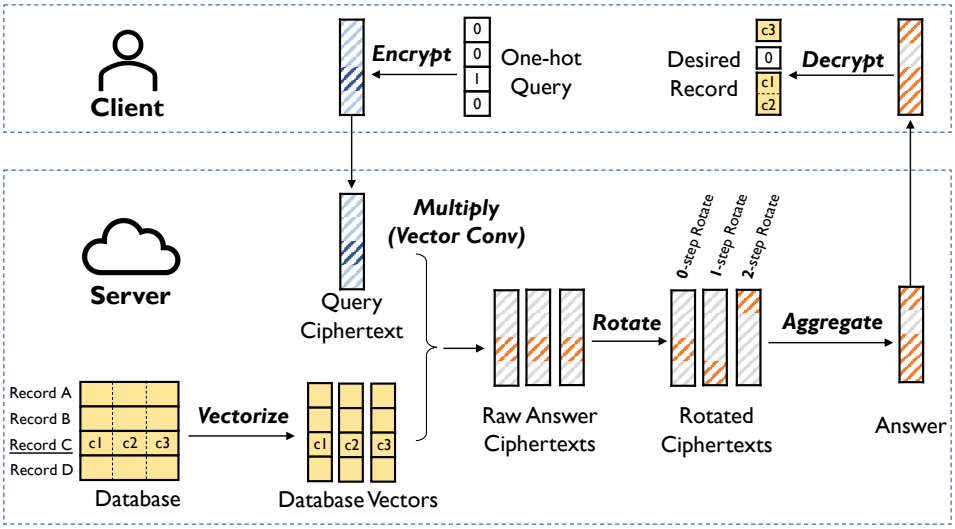


Figure 5.1: Workflow illustration of the state-of-the-art PIR protocol (called FastPIR [85]). To fetch the record **c** from a 4-entry database (containing records **A**, **B**, **C**, and **D**), the client encodes a one-hot vector into a ciphertext. The server performs a reduction computation over the entire database using homomorphic operations. Thus, record **c** is retrieved obliviously, and the server does not know which record is retrieved.

database vectors, resulting in three ciphertexts. ⑤ Through the `Hom_Rot` operation, the server shifts the record slices into different slots in the ciphertexts, which are then added together to achieve a single compact answer. ⑥ The server returns the answer to the client. ⑦ The client derives the desired record with `VecDecrypt`. Note that the record vector could be shuffled in any order, but the client knows the beginning of the record based on the index (three in this discussion).

A key aspect of FastPIR is a tree-based rotation scheme to perform the rotation operations efficiently. Fig. 5.2 shows a more detailed example of this scheme for records with 4 slices each. During the homomorphic multiplication step, a ciphertext is generated for each database column ($[0, 0, c1, 0]$, $[0, 0, c2, 0]$, $[0, 0, c3, 0]$, and $[0, 0, c4, 0]$). These ciphertexts are the leaves of a tree. Then the scheme aggregates the first two leaves (ciphertexts) having the same parent using a rotation with 1 step followed by an add operation. That is, the scheme rotates the ciphertext of $[0, 0, c2, 0]$ into $[0, 0, 0, c2]$, and

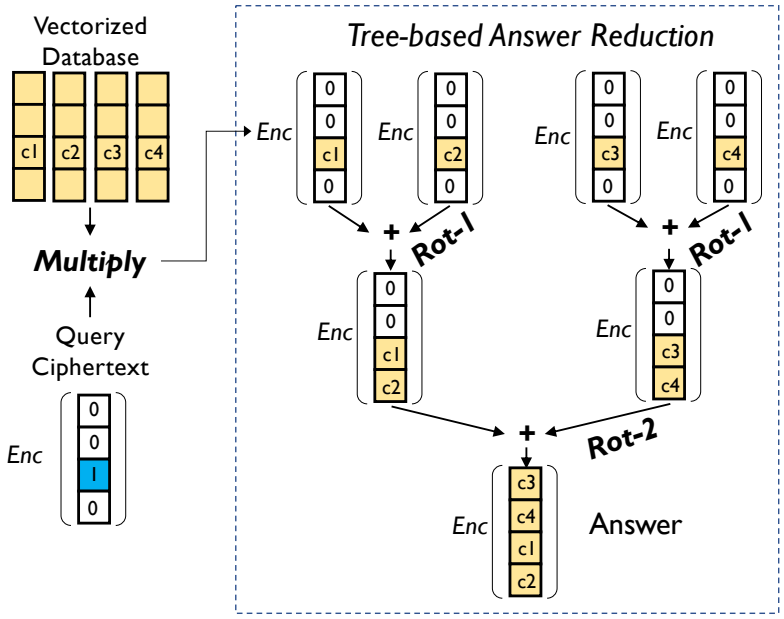


Figure 5.2: The tree-based answer reduction in FastPIR protocol to reduce the answer size using homomorphic rotations and additions.

then aggregates it with the first ciphertext to produce a ciphertext of $[0, 0, c1, c2]$. FastPIR follows this rotation-and-addition recursively to generate the root of the tree, which is a ciphertext of $[c3, c4, c1, c2]$ and contains all the slices of the desired record.

5.1.4 Inefficiencies in FastPIR

Even though FastPIR achieves the best performance among existing PIR protocols, the scalability of FastPIR is still poor due to three inefficiencies:

(a) Large Query: As shown in Fig. 5.1, the query length in FastPIR grows linearly with the number of records. This results in an unacceptably long query for a large database. Considering the message box used in anonymous communication systems [91], the query can be as large as 27GiB for 1B users, which results in a significant query load time.

(b) Large Recursion Stack: The tree-based recursion in FastPIR is not hardware-friendly for a large tree due to the large buffer needed for the recursion stack. Fig. 5.3

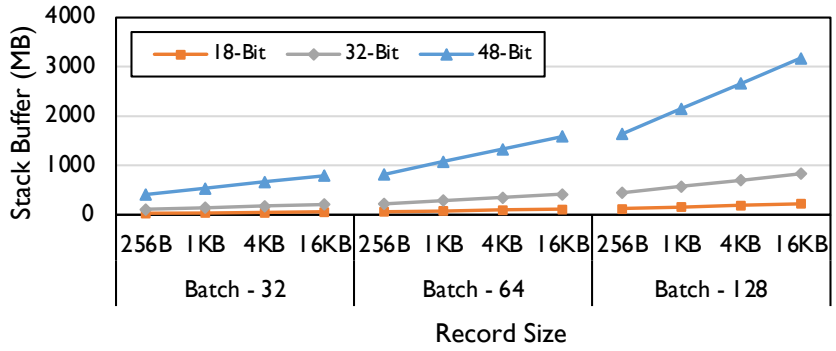


Figure 5.3: The size of stack used for the recursive tree-based rotation in FastPIR. We slice the records in the database into slices of 18/32/48-bits. We also vary the size of the records and the size of the number of queries being processed simultaneously (batch size).

illustrates the growth of stack size with the batch size (the number of PIR queries being processed simultaneously) and the number of database records. The database records are partitioned into slices of the width of 18, 32, and 48 bits for each column. We find that the stack size is significantly enlarged, e.g., over a GiB, for higher values of slice and record size.

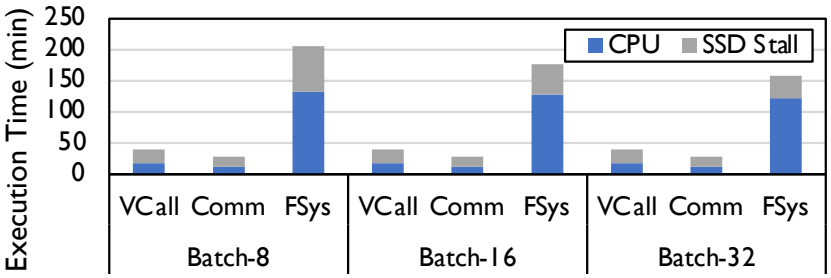


Figure 5.4: The execution time of FastPIR running on three workloads: Vcall (Voice Calling), Comm (Communication), and FSys (File System). The execution time is normalized by the batch size. The breakdown of CPU time and SSD stall is shown.

(c) Long SSD Stall: The PIR processing has to access and manipulate the entire database stored in SSD storage, which causes a severe performance bottleneck. Fig. 5.4 shows the breakdown of FastPIR's execution time by the CPU and SSD. We find that for batch size 8, 16, and 32, an average of 55.4%, 57.3%, and 28.6% of the execution is

stalled on SSD. This indicates that SSD accessing is a major bottleneck in large-scale PIR processing.

5.1.5 Motivation and High-level Ideas

To meet the intensive storage access demands, in-storage processing (ISP) appears to be a promising solution. The ISP technique directly puts the computation logic near or inside the storage device, such that the application benefits from shorter access latency and higher internal bandwidth. Prior work has broadly engaged ISP architecture with various applications, including deep learning [30], recommendations [111], and graph analytics [31]. However, applying in-storage processing to PIR is non-trivial, as naively attaching an FHE accelerator to a storage device results in sub-optimal performance.

First, the ISP technique cannot address the query size issue. In particular, when processing a batch of queries, the query data can be even larger than the database size. Even though the ISP architecture can provide $4 - 8\times$ higher bandwidth than external I/O, it hardly meets the growing throughput demand of large query data.

Second, the heterogeneous computation pattern in PIR requires dedicated hardware and dataflow design. Directly attaching a monolithic FHE accelerator to the SSD device cannot fully leverage the internal parallelism from multiple flash devices, because the accelerator can only gain limited bandwidth from the attached DRAM buffer, which is not enough to satisfy the memory-bound problem.

This chapter describes INSPIRE which leverages protocol and architecture co-design to accelerate IN-Storage Private Information REtrieval. At the protocol level, INSPIRE's key insight is to use the classical idea of recursion [112], but while making a better trade-off between query size and computation overhead. The key trick is to partition the database hierarchically and process smaller queries in each hierarchy. Further, INSPIRE

amortizes the computation overhead with a multi-stage query process. At the lower stage, it uses a block query to process the data from the entire database, which avoids the heavy computation needed for rotations. At the higher stage, it performs FastPIR-like column reductions, but while optimizing the rotation flow to reduce the large memory consumption in FastPIR. As a result, the INSPIRE protocol reduces the 27GiB query size in FastPIR to 3.6MiB for the 288GiB database.

On the hardware side, the key insight of INSPIRE's architecture is to integrate the hierarchical query processing with the micro-architecture hierarchy inside the SSD device, which can fully utilize the internal bandwidth parallelism from multiple storage channels. Specifically, we design INSPIRE as a heterogeneous architecture. As a first step, we design a block collector to perform block-level reduction and equip each flash channel with a block collector. Therefore, the block collectors leverage channel-level parallelism for higher internal bandwidth. Next, we extend the original embedded controller in SSD to support homomorphic computation, such that the results from different channels are sent to this module to perform the more complicated answer aggregation. Through a customized dataflow that processes all data in a streaming manner, the INSPIRE architecture achieves a 22.9× speedup compared with the vanilla CPU baseline.

5.2 INSPIRE Protocol

In this section, we introduce the INSPIRE protocol. We first introduce our design approach, followed by the protocol overview and optimizations.

5.2.1 Design Approach

Although the query in PIR can be extremely large, it contains many encryptions of the same data. As shown in Fig. 5.1, FastPIR encodes every unwanted record as a

zero in the plaintext. However, naively reusing encryptions of these zeros will leak the query information to the server. Our INSPIRE protocol leverages the classical design of recursion [112]. The key idea is to hierarchically partition the database and reduce the query size by sharing the query in each hierarchy. Meanwhile, we keep the ciphertexts within the query independent from each other. Thus, the server cannot tell the difference between them, and the INSPIRE protocol can ensure the security guarantee of PIR.

Specifically, as shown in Fig. 5.5(a) and (b), we partition the database into *columns*, *groups*, and *blocks*. We design two types of queries: block query and group query. The query consists of different ciphertexts, encrypting the location of the desired record in the block/group. The ciphertexts within the query are not sharable, and thus the index information remains private to the server. By sharing the query at different data hierarchies, the server follows a multi-stage reduction process. At the lowest level, the protocol performs block reduction over the entire database, while using multiple ciphertexts to avoid heavy rotation operations. At the middle level, the group reduction aggregates block answers with an identical group query for each column. At the top level, the protocol performs a standard FastPIR-like column reduction to derive the final answer.

Second, we further optimize the rotation-heavy computation pattern during the group reduction and column reduction. Instead of utilizing the tree-based rotation, INSPIRE uses a streaming approach to facilitate the architecture design.

5.2.2 Protocol Overview

The INSPIRE protocol is composed of four functions at the client and server side: `DB_Partition`, `Query_Generate`, `Ans_Generate`, and `Ans_Decrypt`. These functions are described in detail in Alg. 5 and Alg. 6. First, `DB_Partition` is the initialization stage that partitions the database. The partitioning parameters, including block size and group

Algorithm 5: INSPIRE Protocol – Client

```
   /* Generate partitioning parameters                                    */
 1 Function Param_Generate():
   |  // l_B – block length, w_B – block width
   |  // n_B – number of blocks in a group
   |  // l_G – group length, n_G – number of groups in a column
 2 return l_B, w_B, n_B, l_G, n_G

   /* Generate query to fetch k-th record                                 */
 3 Function Query_Generate(k):
 4 |  q_B = Vector < Vector (l_B) > (n_B) ;                  // Block Query
 5 |  q_G = Vector(l_B) ;                                    // Group Query
   |  // Find which block has the k-th record
 6 |  q_B_k = k % l_G;
 7 |  for i = 0 : n_B do
 8 |  |  for j = 0 : l_B do
 9 |  |  |  q_B[i][j] = (q_B_k == i * l_B + j) ? 1 : 0;
10 |  |  end
11 |  |  q_B[i] = VecEncrypt(q_B[i])
12 |  end
   |  // Find which group has the k-th record
13 |  block_idx = k % l_B ;                       // Index in block answer
14 |  rot_offset = n_G − k/l_G; ;                   // Number of rotations
15 |  q_G_k = (block_idx + rot_offset) % l_B;
16 |  for i = 0 : l_B do
17 |  |  q_G[i] = (q_G_k == i) ? 1 : 0;
18 |  end
19 |  q_G = VecEncrypt(q_G)
20 return q_B, q_G

   /* Decrypt the answer returned from server                             */
21 Function Answer_Decrypt(ans):
22 |  msg = VecDecrypt(ans); ;                          // Decrypt answer
   |  /* Reorder the data in the msg vector                              */
23 return msg
```

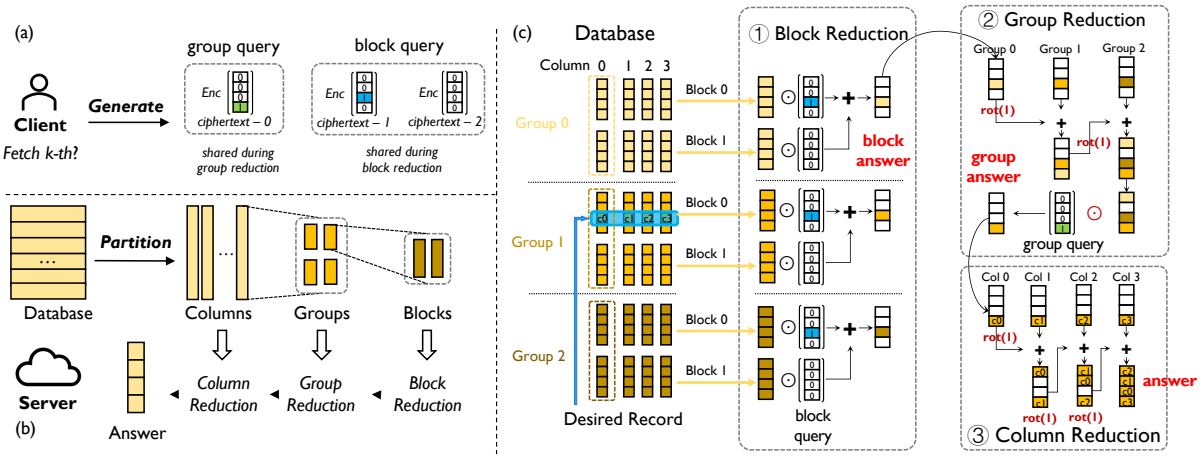


Figure 5.5: An illustration of the INSPIRE protocol. (a) On the client side, instead of generating a long query, the client generates one group query and one block query that consist of multiple ciphertexts for the desired record. (b) The database is hierarchically partitioned into columns, groups, and blocks. The server performs a multi-stage reduction process to derive the answer. (c) A detailed illustration of block reduction, group reduction, and column reduction. The block reduction traverses all the blocks in the database. The group reduction and column reduction shrink the answer with homomorphic rotation and addition.

size, are decided at the client side using the `Param_Generate` function. Second, the client uses `Query_Generate` with the assigned index to generate the query ciphertexts and send them to the server. Third, when the server receives a query, it performs the data retrieval using `Answer_Generate` and sends back the result to the client. Finally, the client uses `Ans_Decrypt` to decrypt and re-arrange the record in the received answer.

`DB_Partition`: INSPIRE adopts a hierarchical database partitioning scheme by arranging the database into blocks, groups, and columns. Specifically, on the server side, the database can be viewed as a 2D matrix with the shape of $N \times n_C$. Each row in the database stands for a record, and each record can be further divided into n_C pieces along the column direction. Suppose we have the ciphertext with length l_B. Then, for each column in the database, every continuous l_B elements are considered as a block. There are N/l_B blocks for each column. Further, several blocks are considered as a group along the row direction. We denote the number of blocks in a group as n_B and the number of

Algorithm 6: INSPIRE Protocol − Server

```
   /* Partition a database with certain params                          */
 1 Function DB_Partition(db, l_B, w_B, n_B):
 2     L, W = db.length, db.width;
 3     n_C = W / w_B;
 4     n_G = L / (l_B × n_B);
       /* Three indices are needed to locate a block:  column id, group id,
          block id                                                      */
 5 return n_G, n_C

   /* Generate answer at the server side                                */
 6 Function Answer_Generate(q_B, q_G):
 7     Initialize ans, ans_block, ans_group;
 8     for c = 0 : n_C do
 9         for g = 0 : n_G do
10             for b = 0 : n_B do
                   // ❶ Block Reduction
11                 db_block = db[(g * n_B + b) l_B : (g * n_B + b + 1) l_B, c];
12                 db_block = Hom_Mul(db_block, q_B[b]);
13                 ans_block = Hom_Add(ans_block, db_block);
14             end
               // ❷ Group Reduction
15             ans_group = Hom_Add(ans_group, ans_block);
16             ans_group = Hom_Rot(ans_group, 1);
17         end
           // ❸ Column Reduction
18         ans_group = Hom_Mul(ans_group, q_G), ans = Hom_Rot(Hom_Add(ans, ans_group),1);
19     end
20 return ans
```

groups in a column as n_G, where $l_B \times n_B \times n_G = N$. As the example shown in Fig. 5.5(c), the database is reshaped as a 24×4 matrix. We use the blue label to indicate the record the client is interested in. Here, 4 elements are encrypted together as a plaintext, and this plaintext is considered as a block. Each column in the database contains 6 blocks. Based on our leveled database partition, each column in the database is divided into 3 groups and each group contains 2 blocks.

`Query_Generate`: The query in INSPIRE also follows the hierarchical scheme and is

composed of a block query and a group query. The block query includes n_B ciphertexts, where one of them encrypts a one-hot vector and others independently encrypt all-zero vectors. The group query includes a single ciphertext that indicates which group holds the desired record. The block query traverses the entire database with relatively simple operations, and INSPIRE streams all the blocks in the database with minimal hardware. The query size of INSPIRE is much smaller than FastPIR because both the block query and group query are shared, unlike FastPIR which has a separate ciphertext for every block in a column.

`Ans_Generate`: During the data retrieval, the server takes the encrypted query from the client and generates an encrypted answer. INSPIRE adopts a three-stage processing method to compute the answer on the server: block reduction, group reduction, and column reduction. We explain the process flow of these stages in detail next.

5.2.3 Multi-Stage Answer Generation

Block Reduction: Each group performs the block reduction with the shared block query. During the reduction, each block in the database performs `Hom_Mul` with the corresponding ciphertext in the block query. Then the resulting ciphertexts within a group are aggregated through `Hom_Add` to get the block answer. As shown in Fig. 5.5(c), each group has two blocks, and these two blocks are multiplied by the two ciphertexts in the block query. Therefore, the shared block query has to traverse the entire database, but the resulting homomorphic operations are relatively simple with only one multiplication and addition for each block.

Group Reduction: Through block reduction, we already derive the desired record in Group 1. But we still have unnecessary data in Group 0 and Group 2. Thus, group reduction aggregates and eliminates this data. As shown in Fig. 5.5(c), the protocol has

three block results shifted by different steps via `Hom_Rot` operations. Then, it adds them together using `Hom_Add`. The group query then multiplies with this aggregated answer, which produces the group answer. The group answer keeps only a piece c_i of the desired record in the i-th column.

Column Reduction: The reduction process in the column reduction phase is very similar to the group reduction. We shift the ciphertexts with different steps, and the final answer is the aggregation of the group answers. In this example, we have the same number of columns as ciphertext length ($l_B = n_C$), and the record C exactly fills the result ciphertext. In case we have more columns than ciphertext length, we can simply use longer ciphertext or multiple ciphertexts.

Optimized Rotation Flow: As shown in Fig. 5.5(c), our rotation flow is different than the tree-based reduction in FastPIR. Instead, we use an answer ciphertext and perform in-place computation. When we need to aggregate the answer ciphertext with the next block/group result, we rotate the answer ciphertext by 1 and directly add the new ciphertext to it. We call this the RNA (rotation and add) scheme. It has two benefits: first, it eliminates the large recursion stack used for the tree traversal, and we only need a small buffer to store the answer ciphertext on-chip. Second, all the rotation operations are performed with $step = 1$. Therefore, we use only one rotation key and avoid a large buffer for storing different keys.

5.2.4 Complexity Analysis

Query Complexity: Our INSPIRE query consists of a block query and a group query. The block query has the size of $n_B \times l_B \times M$, where n_B is the number of blocks per group, l_B is the block length, and M is the size of the polynomial coefficient. The group query has the size of $l_B \times M$. Thus, the total query size is $(n_B + 1) \times l_B \times M$. Compared with

FastPIR whose query size is $N \times M$, we reduce the query size by a factor approximately equal to the number of groups n_G.

Table 5.1: Computation Complexity of different reduction stages in INSPIRE, with comparison to FastPIR

	`Hom_Add`	`Hom_Mul`	`Hom_Rot`
Block Reduction	$n_C n_G (n_B - 1)$	$n_C n_G n_B$	0
Group Reduction	$n_C (n_G - 1)$	n_C	$n_C (n_G - 1)$
Column Reduction	$n_C - 1$	0	$n_C - 1$
INSPIRE	$n_C n_G n_B - 1$	$n_C n_G n_B + n_C$	$n_C n_G - 1$
FastPIR	$n_C n_G n_B - 1$	$n_C n_G n_B$	$n_C - 1$

Computation Complexity: In Tab. 5.1, we show the number of `Hom_Mul`, `Hom_Rot`, and `Hom_Add` operations in different reduction phases. The total operations in FastPIR are also shown as a comparison. We find that our INSPIRE has the same number of homomorphic additions and slightly increases n_C homomorphic multiplications. On the other hand, we increase homomorphic rotations by n_G times. However, although the total rotations are increased, the rotation operation in INSPIRE is cheaper than FastPIR. Importantly, all rotations in INSPIRE are processed in a streaming fashion, and thus we avoid the expensive recursion stack of FastPIR (Fig. 5.3).

5.3 INSPIRE Architecture

Although our INSPIRE protocol significantly reduces the size of the query, we still expect a large amount of accesses to database blocks. In order to overcome the bandwidth bottleneck at the storage I/O, we further design the INSPIRE architecture, an in-storage processing accelerator. In this section, we first present the design overview of INSPIRE architecture. Then, we present the implementation details and dataflow design in the different hardware units.

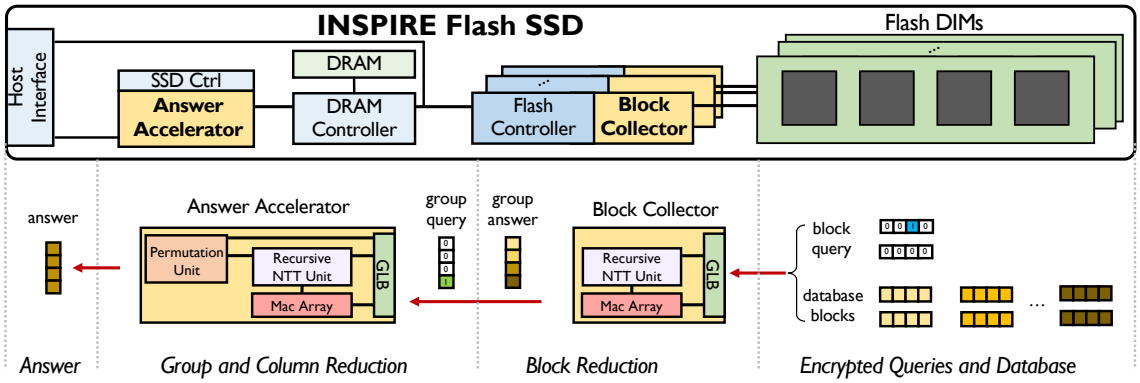


Figure 5.6: Architecture of INSPIRE. Queries and databases are stored on Flash DIMs. INSPIRE adopts a heterogeneous architecture to execute block reduction and group/column reduction on block collectors and answer accelerator respectively. The encrypted answer is finally exported to the host through the host interface.

5.3.1 Architecture Overview

The overall architecture of INSPIRE is shown in Fig. 5.6, which is based on the original flash SSD. During the data retrieving, both the block query and blocks of database records are loaded sequentially from the Flash DIMs. The block reduction is performed at the block collector, leveraging the channel-level parallelism to generate block answers. Further at the answer accelerator, block answers are aggregated together to generate the group answer with the group query. And finally, we use column reduction to derive the final answer and send it back through the host interface. Therefore, INSPIRE adopts a heterogeneous architecture. This subsection details how INSPIRE architecture handles the memory-bounded block reduction and computation-bounded group/column reduction.

The Block Collector is located beside each flash controller. Because block reduction needs to traverse the entire database, we locate it near memory to leverage the internal memory bandwidth. All the groups are interleaved in different flash channels, such that each block collector processes blocks in a group without inter-channel communication. In the Block Collector, the block query and block answer are stored in the

global buffer (GLB), and database blocks are streamed in from the flash DIM. The required homomorphic operations for block reduction can be realized through a number theory transform (NTT) unit and a MAC array. We will detail the NTT units in the following subsections. After aggregating block answers in a group, the Block Collector sends the group answer to the DRAM.

The Answer Accelerator is located beside the SSD controller to execute the computation-bounded group/column reduction stages, which contain complex homomorphic rotation. During the group reduction stage, the block answers from different flash channels are loaded from DRAM to the GLB in Answer Accelerator. Then, the RNA (`Hom_Rot` and `Hom_Add`) based combination is achieved through the permutation unit, NTT unit, and MAC array. After the group answer is acquired from group reduction, all the group answers in the same column are combined into the final answer with RNA, which follows the same computation scheme as the RNA in group reduction.

5.3.2 MAC Array

The MAC array in both the Block Collectors and the Answer Accelerator is a SIMD unit consisting of a group of MACs, and the MAC performs modular addition and multiplication. Different from other applications, the data (such as polynomial coefficients) are bounded by a modulus p, i.e., an integer modulo p. p can be a large prime number or a product of multiple prime numbers [113, 110]. Therefore, modular adder and multiplier are needed when we perform computations such as $(a \cdot b) \bmod p$ and $(a + b) \bmod p$.

We follow the design in F1 [110] for modular arithmetic. In particular, F1 adopted the Montgomery multiplier [114] for fast modular multiplications. It also reduced the total number of stages in the multiplier by choosing an appropriate modulus p.

5.3.3 Recursive NTT Unit

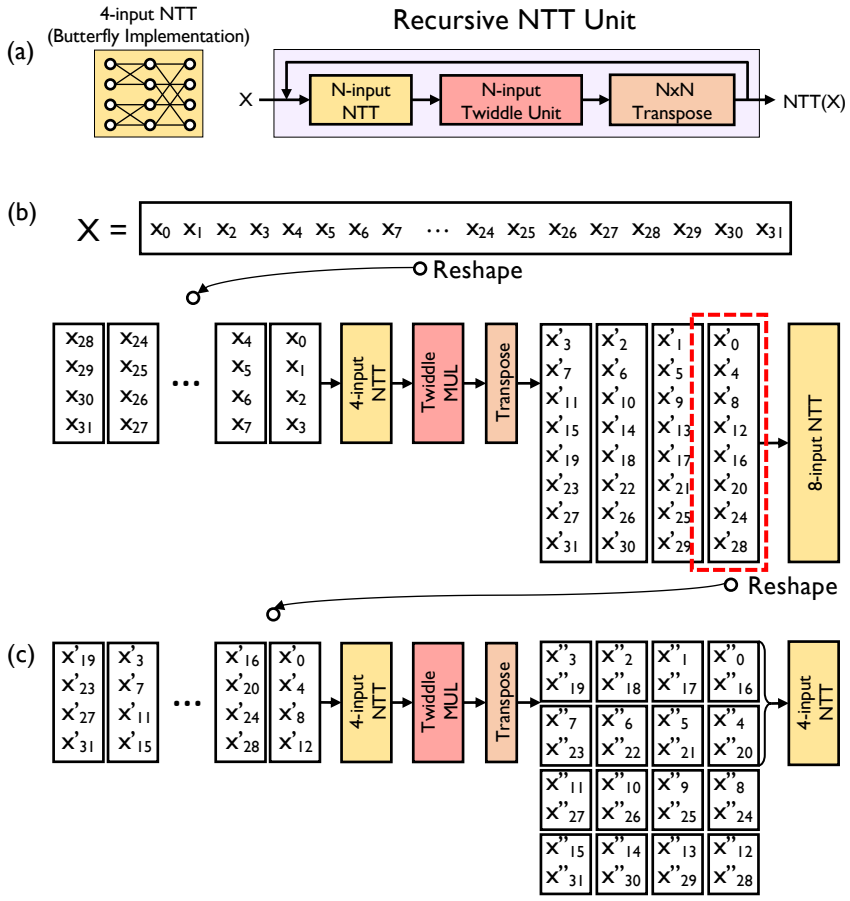


Figure 5.7: (a) The architecture of block collector that implements butterfly computation. (b) The dataflow of breaking down long-sequence NTT to 2D NTT. (c) We further conduct the 2D NTT recursively.

NTT operations are the dominant operations in `Hom_Mul` and `Hom_Rot` [110]. The NTT computation is the same as Discrete Fourier Transform (DFT) but over a polynomial ring. As shown in Fig. 5.7(a), we design the recursive NTT unit to compute the NTT transform of an input vector X with arbitrary length. The recursive NTT unit is composed of a fixed input NTT unit, a twiddle unit, and a transpose unit. Specifically, the fixed NTT unit is implemented as a customized butterfly hardware which takes a length-N input and computes the length-N NTT result. The twiddle unit scales the results from the

NTT unit using MACs. The transpose unit transposes the results through a crossbar. We put register buffers at each level of the butterfly hierarchy such that the NTT unit is pipelined for streaming the input. As shown in Figure 5.6, both the Block Collector and the Answer Accelerator contain an NTT unit, but with different sizes to facilitate unique throughput requirements.

Break down Long-Sequence NTT: Since the ciphertext polynomial is usually quite large, with length from 4K to 16K, it is infeasible to directly implement an NTT unit at this scale. Thus, we need to efficiently map a long polynomial onto smaller NTT units.

INSPIRE recursively adopts a 2D NTT algorithm to achieve such mapping [115]. The key idea of 2D NTT is that we can break down a length-$L = m \times n$ NTT into smaller length NTTs, by performing length-m NTT n times and performing length-n NTT m times. More specifically, the 2D NTT has the following steps: ❶ Reshape the length-L vector into an $m \times n$ matrix. ❷ Perform m-input NTT for every column, and multiply the result with a known parameter (called twiddle factor). ❸ Perform n-input NTT unit for every row. ❹ Finally, reshape the matrix back to length-L vector.

Fig. 5.7(b) shows an example of this algorithm. Suppose we need to do a 32-input NTT on a vector X. First, X is reshaped into a 4×8 matrix. Then, we feed each column into a 4-input NTT unit and have them go through the twiddle unit. Finally, we transpose the matrix and perform 8-input NTT for each column (with length 8).

Recursive Processing: The naive 2D NTT still suffers from inflexibility, as the length of the input ciphertext may vary in different applications. To address this problem, INSPIRE further supports a recursive NTT scheme to perform NTT computation with arbitrary vector length. The key idea is to apply the 2D reshaping recursively for each dimension until the NTT size can fit into our hardware NTT unit. We reshape the length-L NTT as a N^d tensor (instead of a matrix), where N is the length of our hard-

ware butterfly. Thus, we apply the 2D algorithm recursively for a long sequence NTT computation. We show an example of our recursive NTT in Fig. 5.7(c). For the 8-input NTT desired in Fig. 5.7(b), we keep reshaping this vector into a 4×2 matrix. Then we can apply another 2D NTT to compute the result with the same 4-input NTT unit. Note that we can still use the 4-input NTT unit to compute a 2-input NTT, since the data in the butterfly is decouplable. Our NTT hardware can be programmed to compute two 2-input NTTs simultaneously.

5.3.4 Permutation Unit

Permutation Operation: The permutation unit is essentially a switch to reorder the data in the `Hom_Rot` operation. The permutation step in HOM_ROT is to map all the polynomial coefficients to different positions. For example, we can permute the polynomial $2x^3 + 5x^2 + 7x + 1$ into $5x^3 + 7x^2 + x + 2$. A more formal way to describe the permutation is as follows: Suppose the length of the ciphertext vector is N, and we use i and i' to indicate the old and new index of a coefficient before and after permutation. To rotate the plain vector by r steps, the permutation is performed as $i' = (i \times k^r)\ mod\ N$, where k is co-prime with $2N$.

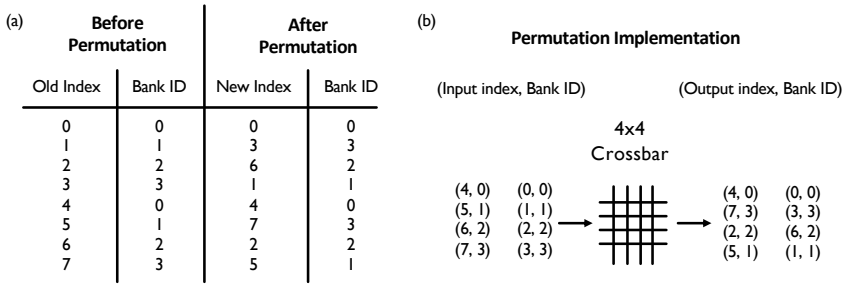


Figure 5.8: Data mapping of the permutation in HOM_ROT. Here, $N = 4096$, $B = 8$, $k = 3$.

It is guaranteed that each coefficient will go to a unique position. Thus, a crossbar switch is enough to route all the data in one cycle (because there is no destination

conflicts). However, since the ciphertext can be as large as 4K-16K (Sec. 5.3.3), it is infeasible to have such a large crossbar.

Key Insight: Our key finding is that such destination conflicts do not exist for every continuous power of 2 coefficients. Fig. 5.8 shows an example of this interesting property. Assume that we have 8 coefficients stored in 4 ($=2^2$) banks. After permutation, the data in position (0,1,2,3,4,5,6,7) now goes to position (0,3,6,1,4,7,2,3). For the 4 coefficients located in bank (0,1,2,3), the new bank ID is (0,3,2,1). There is no bank conflict to relocate these 4 data elements. Therefore, we use a small crossbar with a size of 4 to process a long permutation. To permute a 4096-length polynomial, we can directly permute (relocate) the data 0-3, 4-7, ..., 4092-4095 in a sequential order, where each permutation is done in one cycle.

5.4 Evaluation

In this section, we evaluate the performance of the INSPIRE protocol and architecture. We first introduce our evaluation methodology. Then, we show the performance gain from INSPIRE. Then, we evaluate the scalability and sensitivity of our design. Finally, we present the area and power overhead of INSPIRE.

5.4.1 Methodology

Software Implementation: We implemented the INSPIRE protocol using the Microsoft SEAL library [113]. We choose the BFV encryption scheme [106, 107] and selected the BFV parameters to provide the highest security level according to the Homomorphic Encryption Standard [116]. We stress that our protocol also works with other FHE schemes such as BGV [108] and CKKS [109]. Our implementation uses multiple

Table 5.2: INSPIRE Architecture Configurations

SSD Device			
Read/Program/Erase Latency		75/750/3800 ns	
Channel-Chip-Die-Plane		16-4-2-2	
Blocks/Plane	2048	Pages/Block	512
Channel Width	1B	Channel Rate	1033MT/s
Page Size	8KiB	Capacity	2TiB
Host Interface	PCIe 3.0 ×4	Flash Protocol	NVDDR3
Answer Accelerator (**AA**) and Block Collector (**BC**)			
Tech Node	28nm	Frequency	400MHz
Xbar Switch (AA)	4×4	Operand	32b
NTT input size (AA)	32	NTT input size (BC)	8
Total mMuls (AA)	160	Total mMuls (BC)	24
Total mAdds (AA)	224	Total mAdds (BC)	32
Transpose unit (AA)	32×32	Transpose unit (BC)	8×8
GLB (AA)	2MiB	GLB (BC)	1.25MiB

threads for answer generation using OpenMP, to fully leverage the data-level parallelism.

Hardware Implementation: We implemented the INSPIRE architecture logic in RTL and synthesized it with Design Compiler and 28 nm technology node to derive the hardware parameters, including timing, power, and area. We built a cycle-accurate simulator on top of MQSim [117], an NVMe/SATA SSD simulator, to model the performance of software-hardware co-optimized INSPIRE.

Configurations: We configured the SSD device similar to our CPU baseline, as shown in Table 5.2. The hierarchy in the SSD is organized as channel-chip-die-plane-block-page. With a page size of 8KiB, the total capacity of SSD is 2TiB. The page read and program latency for LSB/CSB/MSB are 75 and 750 ns, respectively. The block erase latency is 3800 ns. Each flash channel is equipped with the NVDDR3 protocol, providing a channel width of 1B and a transfer rate of 1033MT/s. The host communicates with the SSD using PCIe 3.0 ×4, with an ideal bandwidth of 4GiB/s.

For the INSPIRE architecture configurations, we set the input size of NTT units to 32

Table 5.3: Database Workloads

	Record Length	Num. Records	Database Size
Voice Calling (VCall)	96B	2^{32}	384GB
Communication (Comm)	288B	2^{30}	288GB
File System (FSys)	10MB	2^{17}	1.25TB
Synthetic DB 1 (Syn-1)	1KB	2^{29}	512GB
Synthetic DB 2 (Syn-2)	18KB	2^{26}	1.13TB

and 8 for the answer accelerator (AA) and each block collector (BC), respectively. An X input NTT unit is realized through $X(log_2X-1)/2$ modular multiplications and $Xlog_2X$ modular additions (mAdds). The answer accelerator has 160 mMuls and 224 mAdds in total (in the twiddle unit and MAC array). Each block collector involves 24 mMuls and 32 mAdds. The transpose unit sizes for the answer accelerator and block collector are 32×32 and 8×8, respectively. The answer accelerator needs an additional 4×4 Xbar switch to realize the homomorphic rotation. The global buffers for the answer accelerator and block collector are 512KiB and 64KiB, respectively. For homomorphic parameters, we set the element in plain vector to 18bits and the coefficient in the ciphertext to 109bits. After the RNS decomposition [118], each coefficient in ciphertext is composed of 4 32-bit numbers. Moreover, the mMul is realized through optimized Montgomery multiplier [119, 110] which simplifies the complex modular multiplication.

FHE Parameters: We set the polynomial degree as 4096 ($l_B = 4096$ in Algorithm 6). We use the security level of $\lambda = 128$ bits and 109-bit ciphertext coefficient, which follows the same setting as in FastPIR. The plain message to be encrypted is set to 18-bits. The INSPIRE protocol has a multiplication depth of 2 in total.

Workloads: Table 5.3 summarizes the characteristics of the database workloads we evaluate. We run the PIR protocol for three applications: voice calling [85], anonymous communication [91, 102], and file system [101]. The main difference between these workloads is the record type, which determines the record length. We scale the number

of records to evaluate the storage-based database. In addition, we have two synthetic databases, Syn-1 and Syn-2, to enhance the workload diversity.

Baselines: We use FastPIR [85] as our software baseline, since it demonstrates the best performance among existing PIR schemes including XPIR [101] and SealPIR [102]. For a fair comparison, both FastPIR and INSPIRE use the BFV encryption scheme with the same security level. We also optimize and parallelize the source code of FastPIR with OpenMP.

We use both CPU and FHE accelerator as our hardware baseline. The CPU platform is a 64-thread Ryzen Threadripper 3970X @ 2.2GHz (3.7GHz with turbo boost). The storage is a 2TiB Intel 660p NVMe SSD interfaced with PCIe 3.0 ×4. The measured bandwidth of the SSD is 1.8GiB/s. We also include F1 [110], the state-of-the-art FHE accelerator as an additional baseline. We implemented F1 with the ISP architecture, where we attach F1 to the DRAM buffer inside the SSD. The DDR4 DRAM has an ideal bandwidth of 12.8GiB/s.

5.4.2 Performance

In this section, we present key results of INSPIRE, including the overall performance, network bandwidth reduction, noise growth, and query size.

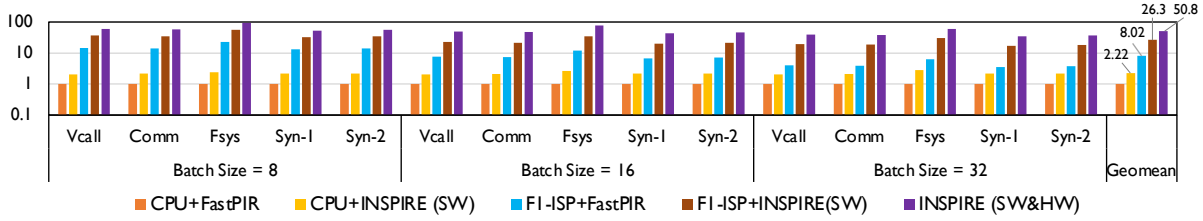


Figure 5.9: The overall performance of INSPIRE compared against FastPIR and F1. We evaluate our protocol and FastPIR on both CPU and F1-ISP platforms. The results are normalized to the FastPIR on CPU baseline. Five workloads and three batch sizes are used.

Overall Performance: Fig. 5.9 shows the overall performance of INSPIRE, along-

side a comparison with FastPIR and F1. The results are shown across three batch sizes: 8, 16, and 32. As an intuitive example, FastPIR spends 28.4min on average to process a Comm query (Fig. 5.4). Our INSPIRE protocol only takes 13.3min for the same query, and our INSPIRE architecture further reduces the time to 36s.

At the software level, the INSPIRE protocol achieves 2.22× performance speedups against FastPIR on CPU, with 3.28× speedup on in-storage F1 (F1-ISP). The performance gain of INSPIRE protocol mainly comes from the reduced memory access for queries. Also, INSPIRE simplifies the dataflow in rotations, which avoids the memory overhead caused by the large recursion stack. Moreover, the INSPIRE protocol demonstrates better performance in the ISP architecture of F1-ISP. The key reason is that the small query and rotation buffer required by INSPIRE are more friendly to accelerator architectures, which usually have limited on-chip resources. For FastPIR, it is much more time-consuming to wait for tiled queries from the host.

At the hardware level, the INSPIRE architecture shows 22.9× speedup against the CPU baseline, with 1.93× speedup compared with F1-ISP. The performance gain of INSPIRE mainly lies in two aspects: first, we leverage much higher aggregated bandwidth to process the memory-bound workloads. While F1-ISP utilizes the bandwidth from the DRAM buffer, the heterogeneous architecture of INSPIRE better leverages the channel-level parallelism via the block collectors in the flash channels. Second, the INSPIRE architecture is tightly coupled with and specialized for the protocol. Different reduction stages are pipelined across block collectors and the query accelerator. Thus, we avoid the sophisticated data mapping and communication in an F1-like architecture.

Network Bandwidth: The network bandwidth directly shows the communication efficiency with compact queries. Fig. 5.10 shows the upload traffic of the INSPIRE protocol, with a comparison to FastPIR. The x-axis shows the number of users scaling from 2^{10} to 2^{20}. We find that INSPIRE reduces the upload bandwidth by 41943×,

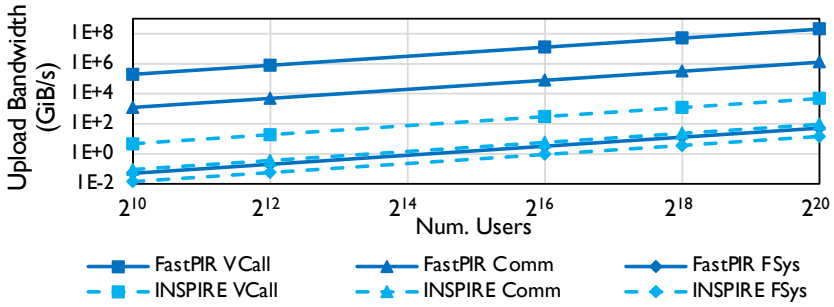


Figure 5.10: The upload traffis of INSPIRE and FastPIR, which depends on the query size and query frequency.

13706×, and 3.56× for VCall, Comm, and FSys, respectively. The significant bandwidth reduction for VCall and Comm is because these two workloads have a very large number of records, and our hierarchical query substantially decreases the total query size. Also, we find that VCall requires much higher (52−333×) upload bandwidth than the other two applications. The reason is that users in VCall have to frequently query the database to fetch the newest message, leading to more simultaneous uploads. Finally, since PIR protocol does not share query data across users, the upload traffic increases linearly as the number of users grow for both FastPIR and INSPIRE.

5.4.3 Sensitivity Study

In this section, we study the scalability of INSPIRE, along with the performance sensitivity to different security levels.

Scalability: We analyze the scalability of FastPIR and INSPIRE with the variants of VCall and Comm workloads. As shown in Fig. 5.11, the number of records in the database scales from 512M to 8B for VCall, and 256M to 4B for Comm. A large batch size of 32 is used, and the absolute execution latency is presented. Compared to FastPIR, we find that the INSPIRE protocol demonstrates 2.08× better performance. The INSPIRE architecture offers an additional performance gain of 49.8×. Further, the scaling of

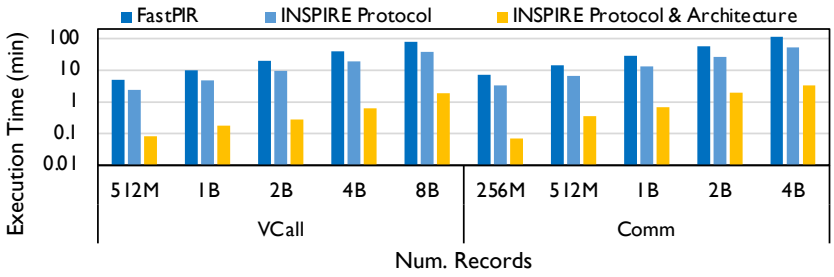


Figure 5.11: The scalability of FastPIR and INSPIRE when the number of records (i.e., the size of database) increases. Two workloads, VCall and Comm, are used for evaluation. The batch size is set to 32.

INSPIRE is approximately linear. This is because PIR applications are memory-bound. When the size of the database grows, the accessed data and the number of compute operations increase accordingly. Also, we observe that FastPIR spends tens of minutes processing a query. In comparison, our INSPIRE architecture considerably reduces the processing time to the second-level. This makes it possible to deploy PIR in real database systems.

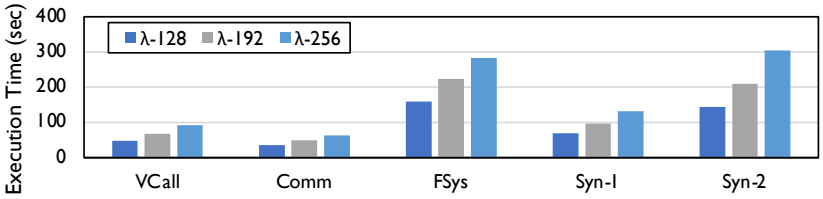


Figure 5.12: The performance of INSPIRE as a function of different security levels, where $\lambda = 256$ denotes the highest security level and $\lambda = 128$ denotes the lowest security level.

Sensitivity to Security Level: We study how different security parameters impact the performance of INSPIRE. As shown in Fig. 5.12, we choose three security levels that are provided by SEAL: $\lambda = 128 - bit$, $\lambda = 192 - bit$, and $\lambda = 256 - bit$. The $\lambda = 256 - bit$ gives the strongest security guarantee [116]. We find that by applying a higher security level of $\lambda = 192 - bit$ and $\lambda = 256 - bit$, the performance will be downgraded by 1.40× and 1.89×, respectively. This is because with larger λ, the coefficient size in the plaintext is

reduced, and thus the message that can be encrypted is smaller. Therefore, the database has to be partitioned at a finer granularity and thus takes more time to process.

5.4.4 Area and Power Overhead

Table 5.4 presents the area and power overhead of the INSPIRE architecture, broken down into each hardware component. We find that computation logics, including the NTT engine and MAC array, take 14.44% of the area and 19.08% of the power. The buffer and routing units, including the transpose unit, GLB, and the crossbar switch occupy 85.56% and 80.92% of the total area and power, respectively. Most of the area and power consumption is taken by the GLB which is used to store the block query in block collector and the temporal answers in the answer accelerator. Note that in the INSPIRE architecture there are 8 block collectors, and the results in Tab. 5.4 accumulate the resource consumption for all block collectors.

Table 5.4: Area and Power Estimation.

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
NTT Engine	2.745	1183.59	**Transpose Unit**	0.122	115.87
mMAC Array	2.954	1393.74	**Global Buffer**	31.592	12385.56
Xbar Switch	0.001	0.40	**Control&Others**	0.055	52.25
Block Collector	33.722	13509.07	**Answer Accelerator**	5.838	2438.29
Total Area 39.56mm^2***; Total Power 15.947***W					

5.5 Conclusion

This chapter follows a software-hardware co-design approach to address the performance bottleneck in existing PIR schemes. We first present the INSPIRE protocol that leverages hierarchical database partitioning and multi-stage answer reduction to reduce

the communication overhead in PIR. Based on the protocol, we present the INSPIRE architecture, an in-storage processing architecture that utilizes the large internal bandwidth and a specialized accelerator to boost the performance of query processing. The INSPIRE protocol achieves 2.22× performance speedup compared to the state-of-the-art FastPIR scheme. Meanwhile, the INSPIRE architecture further brings 22.9× performance speedup over CPU and 1.93× speedup over F1, the state-of-the-art FHE accelerator.

Chapter 6

SIGHT: Enhance the Reliability of In-Memory-Processing Architecture

So far, we have introduced designing NDP-based architecture for various big data applications, from the regular-patterned classification workload to the irregular-patterned graph workload. We also study how to scale the near-memory processing to in-storage processing when the workload size increases. As NDP architecture greatly boosts the performance of existing memory subsystems, this chapter starts to investigate the next-generation memory, Resistive Random Access Memory (RRAM), which is able to fundamentally address the memory wall and power wall issue.

As introduced in Section 2.2, RRAM offers the in-memory-processing (IMP) ability to perform computation within the memory array. However, existing RRAM technology shows severe reliability issues. This can greatly degrade the accuracy of computation. In this chapter, we specifically look into the accuracy impact of deep neural networks with RRAM-based IMP accelerator. We first introduce the overview and background of RRAM's reliability issue and our proposal. Then, we characterize and formulate RRAM's reliability issue into 3 categories. Thus, we present a SynergIstic alGorithm-arcHitecture

fault-Tolerant framework, namely SIGHT, to holistically address the problem. Finally, we evaluate the effectiveness of our design and conclude this chapter.

6.1 Background and Design Overview

Neural network (NN) is now at the core of big data applications, given its excellent performance on various machine learning topics, including image recognition [3], object detection [120], natural language processing [121] and many more NN origins from mimicking the neuron system in humans [122]. The architecture of an NN often consists of artificial neurons and synapses between them. The neurons are organized layer by layer. The neurons in one layer receive the outputs from the previous layer and then propagate their outputs to the next layer.

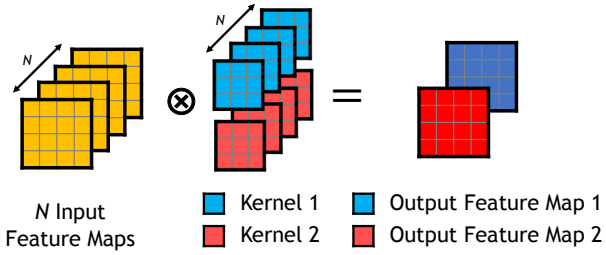


Figure 6.1: An illustration of a CNN layer, which is composed of N (N = 4 in the figure) input feature maps convoluted by 2 kernels.

Convolutional Neural Network (CNN) is an important branch of the NN family that mainly targets computer vision tasks. As shown in Fig. 6.1, the inputs for a convolutional layer are a bunch of 2-D images, which are called feature maps. A group of 3-D convolutional kernels (shown as blue and red) then filter the feature maps with a fixed-size sliding window. Since each kernel generates one output feature map, we finally get 3-D output for the next layer. The computation in the convolutional layer can be expressed in Eq. 6.1:

$$\boldsymbol{d}^{l+1}_{x,y,n^{l+1}} = f(\sum_{n^l=0}^{N^l-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1}\boldsymbol{d}^{l}_{x+r,y+s,n^l} \times \boldsymbol{w}^{l}_{r,s,n^l,n^{l+1}}) \tag{6.1}$$

where $\boldsymbol{d}$ is a 3-D tensor representing the feature map and $\boldsymbol{w}$ is a 4-D tensor representing the convolutional kernels. The superscript l denotes the l-th layer. Therefore, $\boldsymbol{d}^l$ has the shape of $X \times Y \times N^l$ ($4 \times 4 \times 4$ as shown in Fig. 6.1) and represents the input feature map, while $\boldsymbol{d}^{l+1}$ represents the out feature map. $\boldsymbol{w}^l$ has the shape of $R \times S \times N^l \times N^{l+1}$ ($3 \times 3 \times 4 \times 2$ as shown in Fig. 6.1) and represents the convolutional kernels. f is an activation function that aims to add non-linearity into the neural network.

However, the inference of neural networks is typically considered memory-intensive, as a high volume of memory bandwidth is usually required. Thus, RRAM-based PIM accelerators attract considerable research interest in accelerating NN workloads [26, 27]. RRAM is able to perform matrix multiplications in O(1) complexity inside the memory array, which is the key operation in NN models. This reduces half of the data fetching [17, 123] and make RRAM a competitive candidate for next-generation memory.

6.1.1 RRAM Reliability Issues

Previous studies on RRAM characterizations have revealed that current RRAM devices exhibit several reliability issues and non-ideal faults, and we hardly have the RRAM resistance being the exact value expected [19, 124, 125, 126]. Different from using RRAM as a memory device, such faults can lead to severe accuracy loss when using RRAM for computation [127, 128].

In this chapter, we pay special attention to three types of faults that are commonly seen in RRAM devices. ***(1) Non-linear Resistance Distribution***: Multi-level cell (MLC) has been broadly leveraged in RRAM-based accelerators as it significantly increases the data density and saves the design budget. It has been indicated that the

resistance in MLC is not continuously tunable and there may be resistance gaps between different resistance levels [25, 22]. However, prior work simply applied linear quantization to the NN models assuming that an n-bit fix-pointing value can be mapped to an n-bit cell, which actually does not seem to hold for all the RRAM cells. For example, we found that different resistance states in some RRAM models are exponentially increased [126]. The non-linear resistance distribution exposes a challenge that mapping traditional NN to such RRAM may not work and novel quantization algorithms specialized for RRAM-based computing are demanded. ***(2) Static Variation***: It has been widely known that RRAM exhibits serious variations, meaning that the actual value we write into one cell can deviate a lot from the expected one [129, 130, 21]. There are mainly two types of static variations in the current RRAM technology: device-to-device variation and write-to-write variation. The device-to-device variation makes it difficult to generate a unified solution for a single RRAM crossbar array since different cells deviate differently from each other. The write-to-write variation results in the huge overhead in the conventional re-write scheme to address the variation issue, as we may need to re-write multiple times until getting the correct value. ***(3) Dynamic Variation***: As the static variation refers to the variation caused by programming the RRAM cell, the dynamic variation means that the cell resistance keeps changing over time [131, 132, 133]. This issue is particularly concerning because it could get worse in the NN acceleration scenario where those read operations need to charge the RRAM crossbar much more frequently and thus degrade the performance in NN workloads.

6.1.2 Related Work and Our Contributions

Prior work has made efforts on addressing such reliability issues on both hardware and software sides. From the hardware side, Li proposed a verify-after-write approach to

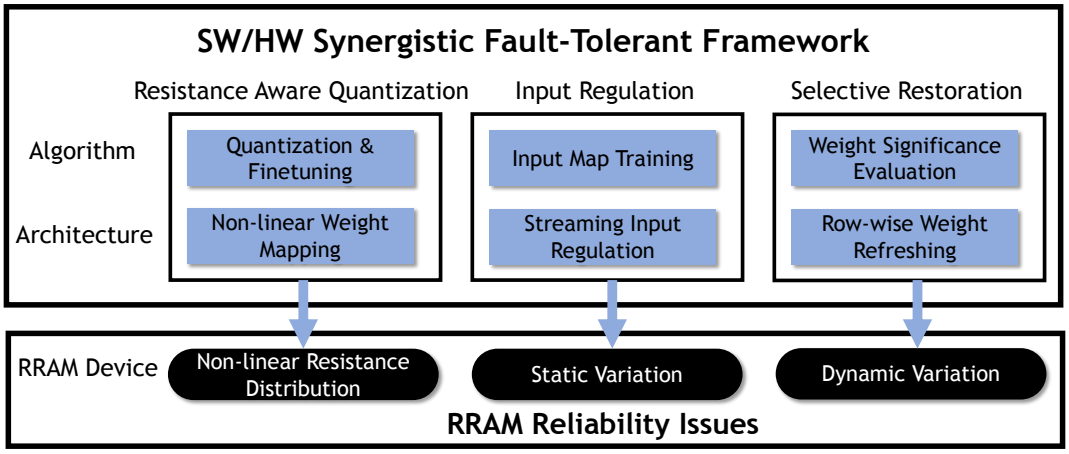


Figure 6.2: An overview of SIGHT, a SW/HW synergistic fault-tolerant framework. We leverage algorithm-architecture co-design to address three reliability issues in RRAM-based computing.

tune the RRAM cell more accurately against the variation [17] while Cheng further improved the energy efficiency of this solution with a RESET-free approach [134]. From the software side, Chen proposed a mapping scheme for NN parameters to avoid important weights being in the variation-affected RRAM cells[127]. NN training techniques have also been explored to enhance the model robustness against variations and non-linear resistance distribution [128, 135]. Lammie1 proposed a variation-aware CNN architecture that is specific for RRAM to solve the variation problem [136]. However, as these studies only touched only one or two reliability issues at a time, there lacks a systematic and unified solution to tackle all three faults.

Therefore, we present SIGHT, a SynergIstic alGorithm-arcHitecture fault-Tolerant framework, to holistically address those problems. As shown in Fig. 6.2, SIGHT leverages algorithm-architecture co-optimizing, which trains and maps NN models with the awareness of RRAM faults and facilitates the processing with dedicated architectural support. We summarize our key contributions as follows:

- We propose a resistance-aware NN quantization algorithm, which forces the weights

in the NN model to follow the resistance distribution as RRAM and tackle the non-linearity issue. Compared with prior work [135], we extend and evaluate our technique to various resistance distributions and demonstrate no accuracy loss.

- We introduce an input regulation method to avoid the accuracy degradation incurred by the static variation. We compensate for the error caused by variation through an integrated input map to regulate input vectors. Compared with prior work [128], the input regulation could resume the accuracy under much larger static variation.

- We propose an RRAM refreshing scheme to resolve the dynamic variation issue. We define the significance of each cell by its weight, and selectively refresh the important RRAM cells at run-time.

- We architect *general* and *low-cost* hardware based on existing RRAM-based accelerator [27] for supporting our fault-tolerant techniques above. The hardware simulation reveals that SIGHT introduces 7.14% performance overhead on average for various NN workloads.

6.2 RRAM faults modeling

In this section, we introduce the fault models are going to address in RRAM-based computing. We analyze the faults under three categories: non-linear resistance distribution, static variation, and dynamic variation.

6.2.1 Non-linear Resistance Distribution

As mentioned, MLC helps us gain more data density and provides considerable savings under the limited hardware budget. Since the parameter $c_{i,j}$ in a matrix should be

linearly mapped into the corresponding RRAM's conductance $g_{i,j}$ according to Eq. (2.2), the conductance/resistance distribution must be exactly the same as the distribution of matrix parameters. However, we find that as the RRAM processing technology has not converged and various types of RRAM based on different materials exist, the distributions of multiple resistance states in MLC are quite diverse.

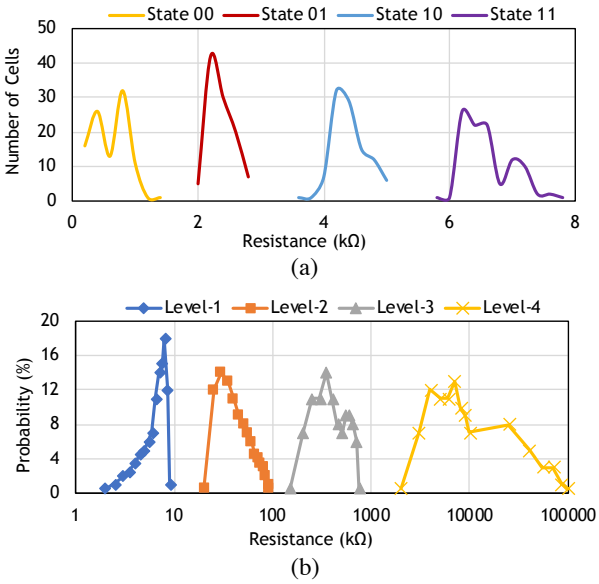


Figure 6.3: The multi-state resistance distribution in MLC for (a) WO_x-based RRAM [125] and (b) HfO_2-based RRAM [25].

Here we show in Fig. 6.3 two cases of resistance distribution. Both of them are 2-bit cells from existing work, with WO_x [125] and HfO_2 [25] based RRAM respectively. To distinguish the non-linear distribution (among all resistance states) and the static variation (for one particular resistance state), we here only consider the mean value for each state. As seen, we hardly expect the resistance distribution to be perfectly linear. For the distribution illustrated in Fig. 6.3(a) (where it shows the number of cells measured in different resistances), four different resistance levels appear to be linear. However, some deviations are making the resistance gaps between them not strictly the same. For the distribution illustrated in Fig. 6.3 (b), we find four resistance levels exponentially

increased under the logarithmic axis, where the highest resistance state is about 1000× larger than the lowest one.

To mathematically analyze the non-linear resistance distribution for further discussion, we propose three fitting functions to model the various resistance distribution:

- **Deviated Linear Model:** For the case shown in Fig. 6.3(a), we apply a deviated linear model where the conductance states are approximately linear and we add random noise δ_k to them, as expressed in Eq. 6.2(a) where k means the k-th conductance state.

- **Exponential Model:** For the case shown in Fig. 6.3(b), we fit the conductance states with an exponentially increased function. As expressed in Eq. 6.2(b), the base β in the exponential function will reflect how the conductance grows.

- **Power Model:** For other cases that can be seen in other work [22, 137, 25] where either the linear or exponential function may not be proper to represent them, we propose a power model that has a moderate growth speed between the linear and exponential model, as shown in Eq. 6.2(c) where the α is the exponent/index.

$$\begin{cases} g_k = Ck + \delta_k & (a) \\ g_k = C\beta^k & (b) \\ g_k = Ck^\alpha & (c) \end{cases} \tag{6.2}$$

To decide the parameters δ, β and α, one can simply apply a minimal square root error (MSE) based fitting algorithm.

6.2.2 Static Variation

Unlike the binary cell that only stores 0 or 1, RRAM as an analog device exhibits serious uncertainty regarding the resistance value, as we observed in Fig. 6.3. Here we call such uncertainty static variation since the variation is fixed after setting or unsetting it. This variation is usually caused by the non-uniformity when forming the filament between two electrodes as it is very difficult to control two filaments to be exactly the same, either for two RRAM cells or for two SET operations. Therefore, there are conventionally two types of static variations: device-to-device variation and write-to-write variation, referring to the resistance difference between two RRAM cells and two SET operations respectively.

When used as a memory device, the static variation seems not troubling because we do not necessarily require the RRAM cell to be precise, as long as two different resistance states are distinguishable. However, it surely becomes a huge threat to the performance when using RRAM for computing, since any change to the operators may lead to incorrect results. Therefore, this RRAM characteristic must be taken into consideration. Here we assume that the higher resistance level suffers from more serious variation, which is indicated in Fig. 6.3 and other previous work [21, 25, 138]. As expressed in Eq. 6.3, we add a stochastic noise to the ideal resistance to model the static variation, which obeys the standard normal distribution. The variance of the distribution is linearly related to the resistance itself with a coefficient λ.

$$r_{actual} = r_{ideal} + \Delta, \Delta \sim \mathcal{N}(0, \lambda r_{ideal}) \tag{6.3}$$

6.2.3 Dynamic Variation

Both the non-linear resistance distribution and static variation are statically fixed after mapping a NN model. On the other hand, researchers also observed the RRAM resistance may drift over time, as the formation of filament is not stable and can be easily affected by the temperature, read current or other environmental factors [139, 140, 141, 142]. This issue is also known as the retention problem describing how long an RRAM device would keep its data. When using RRAM for accelerators, the resistance may drift even worse as the number of reads increases and multiple rows are opened simultaneously.

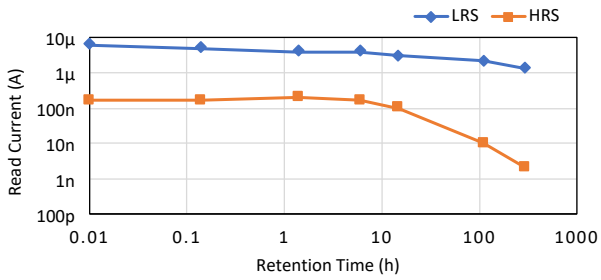


Figure 6.4: The resistance drifting for both HRS and LRS in an HfO_2-based RRAM [133] baking at 200°C. The y-axis represents the read current, and the median value is shown. The x-axis represents the time in logarithmic unit.

Fig. 6.4 presents how two resistance states drift over time for an HfO_2-based RRAM [133]. We find that different from the static variation, the dynamic variation tends to increase the cell resistance as the formed filament tends to be narrowed instead of keeping growing. Therefore, here we use a simplified model and make the assumption that the dynamic variation is single-directional, meaning the resistance keeps increasing over time. Since the dynamic variation also shows stochasticity and Fig. 6.4 presents only median values, we still use noises with standard normal distribution to model the dynamic variation. As expressed in Eq. 6.4, we apply a similar model as static variations but take the absolute value of the noise instead.

$$r_{drifted} = r_{original} + |\Delta|, \Delta \sim \mathcal{N}(0, \lambda r_{ideal}) \tag{6.4}$$

6.3 Fault-tolerant Scheme

In this section, we discuss how to address the three types of RRAM faults mentioned above at the algorithm level.

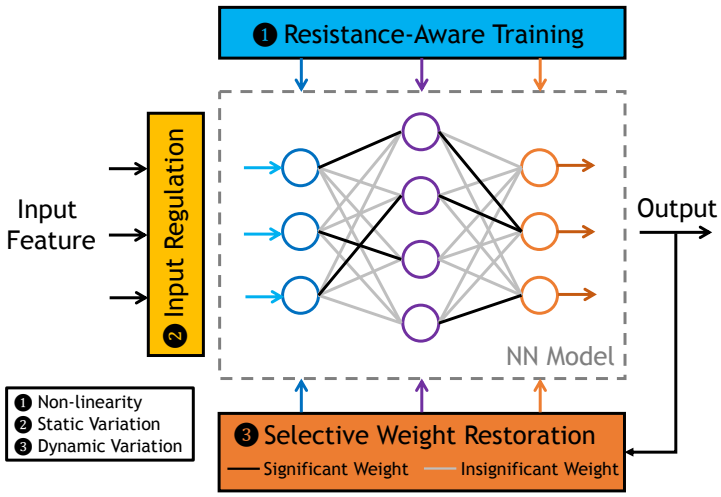


Figure 6.5: The workflow for the proposed fault-tolerant scheme, where three techniques are proposed to address three types of faults respectively.

6.3.1 Overview

Fig. 6.5 shows an overview of our scheme workflow, where we propose three dedicated techniques to tackle the three types of RRAM faults respectively. First, given an RRAM-based NN accelerator, we derive the resistance/conductance distribution according to the specific RRAM technology, which indicates the values we can map to the RRAM crossbar. With this distribution information, we quantize the NN models with our proposed ❶ resistance-aware quantization algorithm. The technique is performed offline and forces

the NN parameters to follow the exact distribution as RRAM. Second, after mapping the NN model to RRAM successfully, we read the static variation of each cell from RRAM crossbars and train an input map offline with our ❷ input regulation technique. The input map regulates the input during inference to compensate for the known variations in RRAM crossbars. Finally, before online execution, we extract significant weights in the NN model under an RRAM-friendly pattern. We determine the significance of weight with respect to a whole RRAM row. After the accelerator starts to execute the inference, we ❸ periodically refresh the resistance of such significant cells in selected rows to resist dynamic variations.

6.3.2 Resistance-Aware Quantization

The non-linear distribution in RRAM devices makes it difficult to map the NN models, because existing quantization algorithms are mostly linear quantization [143, 20, 144]. Some research proposed non-linear quantizing the NN model using {1, 2, 4, 8, ...}, facilitating the NN inference in CMOS-based platforms by switching multiplication to bit operation, since multiplying the input by 4 means shifting left by 2 bits [145, 146]. However, there is very little work offering an RRAM-aware quantization and tackling the non-linear problem in RRAM-based NN accelerators [135].

Therefore, we propose a resistance-aware quantization algorithm to make NN parameters aligned with the RRAM resistance/conductance distribution, which is presented in Algorithm 7 and 8. First, the algorithm receives a positive weight tensor as input since an RRAM crossbar can only represent positive values. So, we need to extract the positive and negative parts of a weight tensor and quantize them separately. Other inputs include the quantization width, and conductance distribution information as discussed in Section 6.2. Second, we initialize a quantized weight tensor and derive a scale factor

Algorithm 7: Resistance-Aware Quantization

Require: Positive weight tensor $\mathcal{W}$; Quantization width n; Conductance list $G = [g_1, g_2, ..., g_{2^n}]$; Distribution Model M; Base β (for exponential) **or** Index α (for power)
Initialize:
  Quantization level $L = 2^n$;
  Decision boundaries $B = [b_0, b_1, b_2, ..., b_L]$, $b_0 = 0$, $b_L = \infty$;
  Quantized weight $\mathcal{W}_{quan} = zero\_tensor$ ($\mathcal{W}$.shape);
  Weight scale $\gamma = \mathcal{W}.\max$ / g_L
if M = Deviated Linear **then** $B = boundary\_decision\_linear(G)$;
else if M = Exponential **then** $B = boundary\_decision\_exp(G)$;
else if M = Power **then** $B = boundary\_decision\_power(G)$;
for $k = 1 : L$ **do**
  $\mathcal{W}_k = ((\mathcal{W}_k > b_{k-1})\ \&\ (\mathcal{W}_k \leq b_k)) \times g_k \times \gamma$;
  $\mathcal{W}_{quan}$ += $\mathcal{W}_k$;
end
Return $\mathcal{W}_{quan}$;

γ between the max value of $\mathcal{W}$ and the highest conductance state. Then, we calculate the decision boundaries that decide what a particular weight value should be quantized to. For the deviated linear model, we follow the traditional quantization approach and take the midpoint (mean value) of two quantization intervals as the decision boundary, as shown in Algorithm 8. But for the exponential model and power model, such a choice does not make too much sense because the distribution of quantization intervals is not uniform. Therefore, we here apply the midpoint of exponents (for the exponential model) and bases (for the power model) as the decision boundary. That is, $\beta^{k+0.5}$ and $(k+0.5)^\alpha$ respectively. Finally, with these decision boundaries, we can decide which quantization interval a weight locates at and combine them together to form the quantized weight tensor, which can be mapped to the RRAM crossbar array favorably.

Note that the quantization algorithm does not work alone, as the model accuracy may degrade after pure quantization. Therefore, we take a finetune process for the quantized

Algorithm 8: Boundary Decision Function

Function `boundary_decision_linear(`G`):`
  for $k = 1 : L - 1$ **do**
    $b_k = \gamma \times (g_k + g_{k+1})/2$
  end
return B

Function `boundary_decision_exp(`β`):`
  for $k = 1 : L - 1$ **do**
    $b_k = \gamma \times \beta^{k+0.5} \times (g_k/\beta^k)$;
  end
return B

Function `boundary_decision_power(`α`):`
  for $k = 1 : L - 1$ **do**
    $b_k = \gamma \times (k + 0.5)^{\alpha} \times (g_k/k^{\alpha})$
  end
return B

NN model as described in [20]. Therefore, we keep quantizing and re-training the model iteratively until the accuracy converges. The quantization algorithm is applied before both the inference and the forward stage in the training.

6.3.3 Input Regulation

After mapping the quantized model to RRAM crossbar arrays, we may still encounter the static variation in RRAM that severely hurts the performance. Previous work on tackling the RRAM variation mostly relies on RRAM cell re-writing [134, 17] or NN parameter re-mapping [127]. However, these methods may cause large hardware overheads due to the write-to-write variation. On the other hand, some work leverages NN training to make the model more robust against variation [128, 147] by injecting stochastic variation into the training process. We will show in the experiment regarding comparison to such technique.

To tackle the static variation, we leverage the opportunity that such variations are

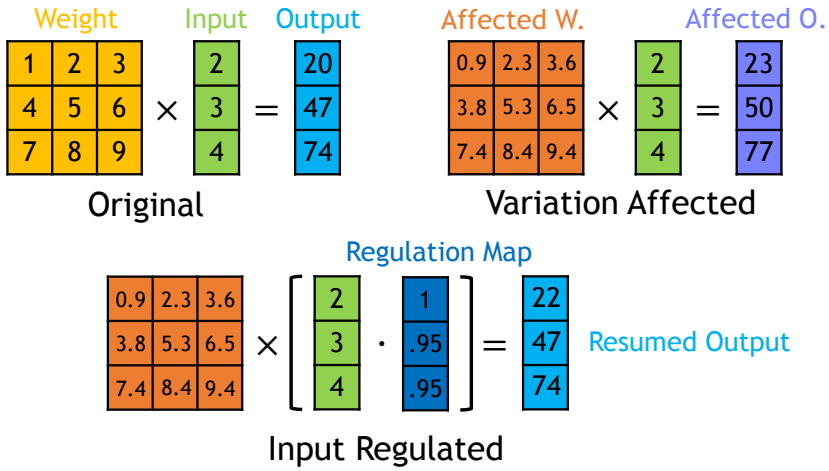


Figure 6.6: An illustration of the input regulation technique. We show on the top that variation-affected weights lead to incorrect output, where all the outputs are rounded to integers. The bottom figure shows how the input regulation map works to resume the output.

fixed and can be known immediately after mapping. Therefore, we propose to regulate the input and compensate for the known variations instead of pursuing a perfectly correct RRAM cell. As the output relies on both weight and input, we slightly adjust the input voltage to keep the final results correct under the variation-affected weight. As shown in Fig. 6.6, we take a matrix-vector multiplication as an example. The original computation is $\mathcal{W}x = y$. After we map the $\mathcal{W}$ onto the RRAM crossbar, we get a variation-affected weight matrix $\mathcal{W}_{noised}$, which would lead to an incorrect output shown as purple. Then, we add an input map $\mathcal{M}$ to the input x accordingly, where element-wise multiplication will be performed over $\mathcal{M}$ and x. This process will scale the input a little to compensate for the certain variation pattern. Finally, we resume the output more accurately to the original one.

So, our goal is to search for an optimal input map $\mathcal{M}$ that best resists against the static variation. This problem can be formulated mathematically to an optimization problem, where we wish to minimize the computation error under the matrix-vector multiplication constraint, as expressed in Eq. 6.5:

$$\begin{aligned} \min_{\mathcal{M}} \quad & Error(y, y_{noised}) = \sqrt{||y - y_{noised}||_2} \\ \text{s.t.} \quad & y = \mathcal{W}x \\ & y_{noised} = \mathcal{W}_{noised} \times (\mathcal{M} \cdot x) \end{aligned} \tag{6.5}$$

However, the input x in the equation above remains unknown. Our opportunity is that since the neural network itself is trained over a specific dataset, for a particular piece of weight, the input may thus follow some specific patterns. For example, the input in the corner of an image may have more white pixels. Therefore, we propose to obtain the $\mathcal{M}$ through the same training procedure as to train the neural network. First, we initialize the $\mathcal{M}$ to an all-1 tensor, meaning that the x remains unchanged after getting multiplied by 1. Second, we can approximate the optimal $\mathcal{M}$ using common training techniques such as stochastic gradient decent. In this step, we fix the weight matrix and update the input map only. Through going over all training images, we finetune the input map iteratively until convergence.

6.3.4 Selective Weight Restoration

The resistance-aware quantization and input regulation help tolerate the static faults which can be detected at the mapping stage. However, the reliability issue also occurs at the inference state when we keep running the RRAM-based accelerator. As we discussed in Section 6.2.3, the resistance of RRAM may drift over time. Previous work mainly focused on the endurance problem [148] to protect RRAM cells from frequent writing, but the retention problem in the NN acceleration scenario has been rarely touched.

To address this problem, we propose a selective weight restoration method which determines the significant weights in an NN model and restores them from dynamic variation. The idea origins from the observation that only a small fraction of weights

show the importance to NN models and slight changes to the insignificant weight will not hurt the accuracy due to the intrinsic robustness of NN [20]. Therefore, we can back up these significant weights and restore them when they suffer from the resistance drifting/dynamic variation.

However, the challenge is that the significant weights are in fact randomly distributed among RRAM crossbars, and thus it would take a long time to re-write all these weights. To overcome the writing overhead, we then introduce an RRAM-friendly weight restoration method, which leverages the RRAM crossbar's nature that it is possible to write one row in the RRAM crossbar simultaneously to reduce the latency [148]. So, after we map the NN model to RRAM, we detect the weight significance in a row-wise manner for each crossbar. First, we sum up the weight within a whole row as expressed in Eq. 6.6, which indicates the significance of this row. Since we already separate the positive and negative weights, we do not need to take the absolute value. Second, we select a number of rows with the largest significance and back up them in the main memory. Every crossbar selects the same number of rows to avoid the restoration imbalance. Finally, we periodically refresh these rows in RRAM crossbars to ensure the accuracy performance.

$$\mathcal{S}_j = \sum_{i=0}^{H} w_{i,j} \tag{6.6}$$

Note that to avoid the contradiction to input regulation that has fixed input maps for the static variation, we need to write the cell precisely during refreshing RRAM crossbars. We consider this overhead very small because we only re-write a small portion in an RRAM crossbar and the refreshing is conducted after a period of time. We will quantitively evaluate the overhead in Section 6.5.

6.4 Architecture Design

In this section, we discuss how we architect the RRAM-based accelerator and equip it with fault-tolerant capability according to the techniques proposed in Section 6.3.

6.4.1 Overview

As RRAM reliability issues broadly exist in various accelerators, our design principle is to make our design (1) *unified*, where we address all the reliability issues in one framework; (2) *general*, meaning it applies to various RRAM-based accelerators; (3) *low-cost*, as negligible performance and hardware overhead would be introduced. For these purposes, we architect add-on hardware in the existing RRAM-based accelerator to support our fault-tolerant scheme.

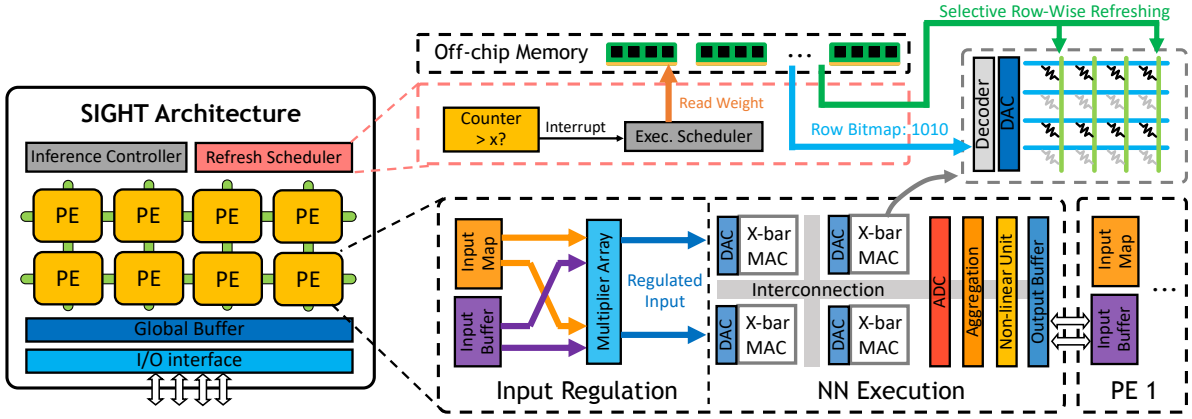


Figure 6.7: The SIGHT architecture overview. We mainly design (a) the input regulation unit for each PE to execute the NN inference and (b) the weight refreshing scheduler to restore the significant weights in RRAM crossbars.

Fig. 6.7 gives an overview of the SIGHT, which is based on the ISAAC-like [27] processing flow. The SIGHT consists of two parts, the NN inference accelerator and fault-tolerant units. To process the NN inference, SIGHT distributes the workload to a number of RRAM-based processing elements (PEs). Each PE tile is mainly composed of the

input/output buffer, RRAM crossbar arrays for matrix multiplication, registers for result aggregating and non-linear units for activation functions. The PEs are organized in a mesh manner, for better reconfiguring with various NN workloads. Beyond the functional modules for NN executions, we design fault-tolerant units to resist the RRAM faults. First, we add a multiplier array beside the RRAM crossbar, which regulates the input voltage to resist the static variation. Second, we design a weight refreshing scheduler. The scheduler issues interrupt signals periodically or on-demand to NN executions, and then refreshes the significant weights to recover the accuracy loss caused by the dynamic variation.

6.4.2 Hardware Design

RRAM-based PE is similar to the tile design in ISAAC. Multiple RRAM crossbar arrays are used to compute the matrix multiplication. The crossbars are interfaced by the digital-to-analog converter (DAC) and analog-to-digital converter (ADC). As the area overhead of ADC is usually considerably larger than DAC [27], we make each crossbar have its own DAC arrays but share the ADC arrays with other crossbars within the PE. We use SRAM to buffer the input and output activations. A PE's input/output buffers are interconnected with adjacent PEs for the convenience of layer propagation. We also use aggregation registers to aggregate the partial sums of (1) split matrix and (2) split precision from different RRAM crossbars. The latter one is for the reconfigurable precision purpose. Since the RRAM, DAC and ADC are fixed-precision and some NN workloads demand higher precision, splitting most significant bits (MSBs) and least significant bits (LSBs) thus becomes a common technique used in RRAM-based computing [26, 27]. Finally, we put a non-linear unit to support activation functions and pooling operations in an NN model.

Input Regulation Unit is to adjust the input voltage according to the fault-tolerant scheme presented in Section 6.3.3. This unit is composed of a multiplier array, which locates right beside RRAM crossbar arrays. We also put an extra buffer within the PE to store the input map $\mathcal{M}$. Then, the multiplier array receives operators from the input buffer and map buffer and sends the regulated data to DACs. This only incurs very little performance overhead of one multiplier cycle, which we will discuss in detail in Section 6.5.

Refreshing Scheduler is a separate piece of control logic that aims to refresh the RRAM crossbars affected by the dynamic variation. The scheduler decides to refresh according to an internal counter, which records the time interval from the last refreshing. Whenever the counter exceeds the pre-set threshold, the scheduler issues an interrupt signal and starts to refresh the RRAM crossbars. As all the backup weights are stored offline, the significant weights will be read from off-chip memory and written to crossbars row-wisely. For each crossbar, we use a bitmap to decide which rows are going to refresh and put a dedicated decoder to decode the bitmap (searching for non-zero positions) and process the rows sequentially.

Interconnection & Controller: The PEs are organized as a 2-D mesh, where one PE can receive the input activations from others or route its output to adjacent PEs. The RRAM crossbars within a PE are connected with a shared bus. Once an NN model is mapped, all the data paths for execution are fixed offline. The execution controller then runs a finite state machine and takes charge of all the pipelines according to the control registers, which determines how the results are routed.

6.4.3 Execution Flow

Mapping: Similar to the ISAAC and other RRAM-based accelerators, SIGHT puts all NN parameters on-chip, which are distributed over all the RC tiles. This assumption holds because the RRAM itself is a memory device and writing RRAM repeatedly will significantly reduce its endurance [148]. Therefore, we first map the model to the accelerator offline, and one layer could be mapped to one or several PEs. After mapping, we read actual weight values from RRAM crossbars that are affected by static variations. Then, we train the input map $\mathcal{M}$ offline and write the map to each regulation buffer. Finally, we program the control registers in the execution controller with respect to the data flow fixed by the mapping.

Execution: After we start the execution controller, the global buffer loads the image from the off-chip memory and sends it to PEs that process layer 0. The input will first be regulated by multiplying with the input map and then sent to the corresponding RRAM crossbars. The computation results are temporarily stored at the aggregation registers, where the partial sums are accumulated. Finally, after going through the non-linear function unit, the output of this layer is then stored in the output buffer and waiting for routing to the next PE.

Refreshing: The RRAM crossbars refresh themselves with a pre-set time interval. When it is time for refreshing, the refreshing scheduler first sends an interrupt to the execution controller and suspends the inference processing. As backup weights are stored offline, the scheduler then issues memory requests to the I/O interface and reads the row-wise significant weight and the bit map. To reduce the overhead, we wish all crossbars to refresh simultaneously. Therefore, we make the weight reading and crossbar refreshing in the pipeline. First, one significant row is read for each crossbar at a time. Second, the crossbar refreshes this row with the address decoded from the bit map. Meanwhile,

the scheduler starts to read the next row to hide the I/O latency. Since we restore a certain percentage of rows for every crossbar, no refreshing imbalance would be introduced consequently.

6.5 Evaluation

In this section, we provide the experiment results of our proposals and analyze the insight from them.

6.5.1 Methodology

We evaluate our proposal from both software and hardware aspects. For the software aspect, we demonstrate the efficacy of the fault-tolerant scheme by the accuracy results from different neural network benchmarks. For the hardware aspect, we evaluate our architecture design by simulations, showing the performance, energy breakdown, and area overhead.

Evaluation Tools: We implement our fault-tolerant schemes with PyTorch, a commonly-used python-based NN framework. We also build up an in-house simulator to evaluate the hardware performance. The RRAM parameters including area, energy, and latency, are derived from NVSim [149], while the SRAM are simulated by CACTI [150]. We also implement other digital components, such as activation units and multipliers, in Verilog and synthesize them with Synopsys Design Compiler to estimate their performance.

Accelerator Configuration: As shown in Table 6.1, the system is configured to 1.2GHz and simulated under 32nm technology. For each PE, we use 32 RRAM crossbar arrays with a size of 128×128. We assume 2-bit precision for each RRAM cell. We use 1-bit DAC and 8-bit ADC as described in [27], where each RRAM crossbar has 128

Table 6.1: The SIGHT Configuration

SIGHT Configuration @ 1.2 GHz, 32 nm Technology		
Component	**Parameters**	**Spec**
RRAM Crossbar	Precision	2-bit
	Size	128*128
	Number/PE	32
Input/Output Buffer	Size	8KB
	# Banks	4
	Data Rate	64B/cycle
Input Map Buffer	Size	4KB
	# Banks	4
	Data Rate	32B/cycle
ADC	Precision	8
	Number/PE	32
DAC	Precision	1
	Number/PE	32*128
Regulation Array	# Multipliers	32
PE	Number	256
Global Buffer	Size	32KB
	# Banks	8
	Data Rate	128B/cycle
I/O	Config.	PCIe 4.0*4
	Bandwidth	16GB/s

DACs and shares one ADC. The regulation array is set to 8-bit precision where it has 32 multipliers. We apply 8KB SRAM for input and output buffer within one PE, and each buffer has 4 banks contributing to a data rate of 64B/cycle. The input map buffer is a 4KB SRAM separately whose data rate is 32B/cycle. We equipped SIGHT with 256 PEs in total, and there is a 32KB global buffer with 128B/cycle data rate. We use four PCIe 4.0 lanes to connect the accelerator and the host, providing 16GB/s off-chip bandwidth in total.

Benchmarks: For better understanding how the model structure and complexity are sensitive to our fault-tolerant scheme, we consider two typical types of CNN, VGG and ResNet, for evaluation, where VGG stands for a plain and straightforward network and ResNet has a residual structure [151, 3]. We also choose models with different

depths, including VGG-11/16 and ResNet-18/34. We apply the public implementations from GitHub as baselines [152, 153]. The dataset we use is CIFAR-10, which consists of 60000 32×32 color images in 10 classes. Among them, 50000 images are used for training and 10000 images are used for testing [154].

6.5.2 Accuracy Results

(1) Resistance-Aware Quantization. We quantize and finetune the four pre-trained models with the three mentioned resistance distributions. For a better sensitivity study, we quantize them into 2, 3, and 4 bits as current RRAM devices are unlikely to have very high precision. We also select different parameters for different distributions and choose the base of power distribution and the index of exponential distribution to be $\sqrt{2}$,2,3.

Table 6.2: The accuracy results of the resistance-aware quantization. Models are quantized into different precision for different types of resistance distribution. For power and exponential distribution, different parameters are chosen.

		ResNet-18			ResNet-34			VGG-11			VGG-16		
Baseline		**94.78**			**94.74**			**92.23**			**93.58**		
Quantization		2-bit	3-bit	4-bit	2-bit	3-bit	4-bit	2-bit	3-bit	4-bit	2-bit	3-bit	4-bit
Linear(δ)	0.10	93.47	94.68	94.82	93.39	94.37	94.65	85.94	91.21	91.86	89.23	93.03	93.38
Power(β)	$\sqrt{2}$	94.36	94.60	94.73	94.40	94.48	94.59	90.63	91.89	92.15	92.80	93.27	93.48
	2	94.38	94.67	94.79	94.14	94.57	94.66	90.64	91.98	92.14	92.72	93.29	93.46
	3	94.31	94.51	94.68	94.19	94.56	94.54	90.35	91.34	91.97	92.43	93.11	93.52
Exp(α)	$\sqrt{2}$	93.40	94.71	94.69	93.36	94.59	94.61	85.18	91.91	91.92	88.09	93.30	93.38
	2	94.66	94.67	94.72	94.27	94.45	94.52	90.86	91.52	91.55	93.04	93.14	93.24
	3	94.45	94.38	94.39	94.14	94.14	94.09	90.15	90.11	90.06	92.81	92.83	92.81

As shown in TABLE 6.2, the accuracy result under full precision can be found in the top line while the accuracy after quantization is below them. We find that: (a) For most cases, we achieve no accuracy loss or insignificant accuracy loss (∼1%), demonstrating the efficacy of our resistance-aware quantization; (b) The overall results for ResNet models are better than VGG ones. We resume the accuracy of ResNet models back to 94% as original for almost all the cases, except for exponential quantization to 2 bits where

the accuracy is 93%. Meanwhile, VGG models suffer more accuracy loss as usually 1-2% degradation is introduced, especially for VGG-11. This tells the ResNet structure is more robust than a VGG-like plain network when applying the non-linear quantization; (c) The exponential distribution with a small index plus low quantization width will degrade the performance noticeably. As shown, when using parameter $\alpha=\sqrt{2}$ and quantizing the model with 2-bit precision, the accuracy of ResNet models drops 1% while the accuracy of VGG-11/VGG-16 decreases to 85.18% and 88.09%. This is because in such cases, the quantized values have a smaller range and thus are not enough to represent the NN models. Also, higher quantization precision generally makes quantized values more representative, which explains that it basically occurs at the low-bit quantization.

(2) Input Regulation. To evaluate the efficacy of input regulation, we inject variations with increasing variances to see how models get affected by static variations, where a larger variance indicates a worse variation. We take the deviated-linear and exponential quantization methods for all the four models and recover the accuracy by regulating the input.

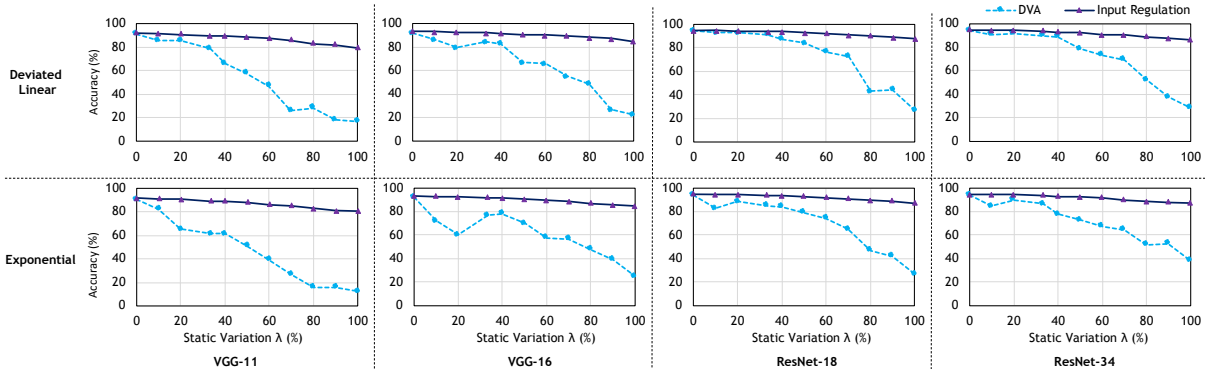


Figure 6.8: The efficacy of input regulation against the static variations, compared with device-variation-aware (DVA) training [128]. Deviated-linear and exponential quantization are shown. The x-axis represents the variance of variations, and the bigger the worse. The y-axis represents the accuracy.

The results are shown in Fig. 6.8, where we compare the accuracy between the device-

variation-aware (DVA) training [128] (shown as dotted lines) and our proposed input regulation (shown as solid lines). We observe that: (a) With the input regulation, we keep the accuracy against dropping from variations. Even when injecting the variation as large as 100%, the model delivers less than 10% accuracy loss most of the time; (b) The input regulation outperforms the DVA with an average accuracy gain of 25.2%. The DVA appears effective in a smaller variation range when $\lambda < 50\%$, but the accuracy degrades significantly after that. Specifically, our input regulation achieves an accuracy gain of 13.0% when $0 < \lambda \leq 50\%$ and an accuracy gain of 43.7% when $50 < \lambda \leq 100\%$, compared with the DVA. (c) The DVA shows stochastic characteristics, as in some cases such as the VGG-16, smaller variation may cause larger accuracy loss. This is because the real variation scenario is unpredictable. Although the model is trained to be more robust, the later injected variation could cause huge accuracy loss. On the opposite, our input regulation is obtained for a specific variation distribution and thus achieves better performance.

(3) Run-Time Weight Restoration. Similar to the static variation, we inject dynamic variations with increasing variances to the model and evaluate the deviated-linearly and exponentially quantized models. We also restore different percentages of weights to see the trade-off between the restoration overhead and model accuracy.

As shown in Fig. 6.9, the dotted line presents how the accuracy drops as we injected larger dynamic variations, and three solid lines with different markers present the recovery of accuracy after the weight restoration. From the results we find that: (a) As a larger dynamic variation leads to lower model accuracy, the ResNet models suffer more from the dynamic variation. For the deviated linear model with an 8% dynamic variation, the accuracy of ResNet-18/34 drops rapidly to 17.47%/10.43%, while the VGG-11/16 only drops to 54.05%/46.86%. This may be caused by the single-directional drifting for dynamic variations, meaning the weight drifting will accumulate layer by layer. Therefore,

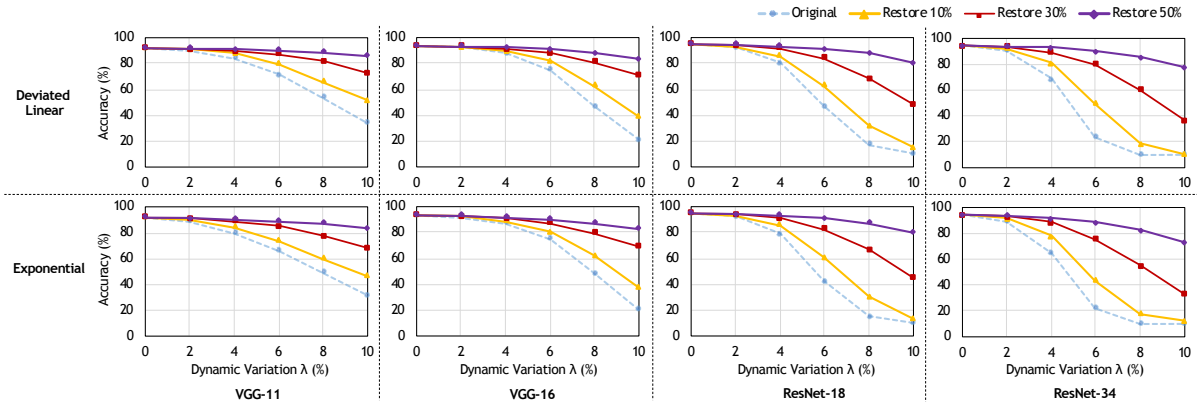


Figure 6.9: The efficacy of selective weight restoration against the dynamic variations. Deviated-linear and exponential quantization are shown. The x-axis represents the variance of variations, and the bigger the worse. The y-axis represents the accuracy.

deeper networks with more complex structures may have worse accuracy. This hypothesis also stands for the same model with different depths, as VGG-11 and ResNet-18 appear to perform better than VGG-16 and ResNet-34; (b) The weight restoration notably improves the accuracy under dynamic variations. For VGG models, we notice that restoring 30% weight recover 30-50% accuracy when injecting 8% dynamic variations, while further restoring 50% weight would almost resume the whole accuracy. Meanwhile, restoring 10% weight seems not that helpful as it only recovers 10% accuracy. Besides, for ResNet models, it seems that we need to restore about half of the weights to fully recover the accuracy. This can also be explained by the variation accumulating in the NN model since ResNet is much deeper than others. (c) As we take a look into the results for deviated-linear and exponential quantization, there is no much difference between them, implying that quantization methods are not affecting the model accuracy as much as the model itself.

6.5.3 Hardware Results

As our fault-tolerant framework is built on top of existing RRAM-based accelerators, we mainly discuss the overhead introduced by SIGHT in the hardware evaluation. We first provide baseline results and present the performance overhead, including inference latency and energy consumption. Then we show the area and power breakdown of SIGHT. Finally, we do a sensitivity study on how the performance of SIGHT is affected by various hardware configurations.

Table 6.3: The baseline results including the latency, energy consumption, and run–time performance for four NN models. The results of one image is shown.

Model	# Active Xbar	Latency (ms)	Energy (nJ)	Performance (TOP/W)
VGG-11	1164	0.18	3,883	27.03
VGG-16	1840	0.35	8,243	24.18
ResNet-18	1392	0.86	15,766	35.23
ResNet-34	2636	1.34	29,531	19.63

(1) Performance. TABLE 6.3 shows the baseline results for the four models where our fault-tolerant framework is not enabled. We present the number of active crossbars, latency, energy consumption and run-time performance. Besides, we set 2-bit weight and 8-bit activation for inference and directly map the model with a batch size equal to 1. We first find that ResNet-34 consumes most crossbar resources and other models are using a comparative number of crossbars. Also, the two ResNet models take a much longer time for execution and consume more energy than VGG models. This is because ResNet models have more convolutional layers that require long repeated computation over them. Finally, the run-time performance of the four models is about 20-30 TOP/W.

We show the results of SIGHT in Fig. 6.10, which are represented by the percentage of overhead compared with the baseline. We set the weight refreshing frequency to 4 images as the weight would be re-loaded every 4 images. First, SIGHT causes a small amount of

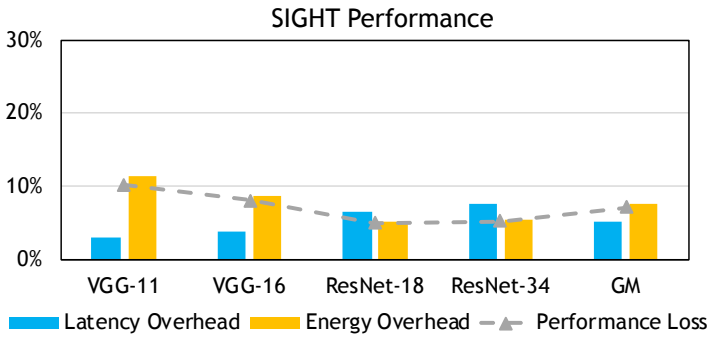


Figure 6.10: The performance of SIGHT. The blue bar shows the execution latency while the yellow bar shows the energy consumption, and the dotted line represents the performance.

overhead which is typically less than 10%. The average latency overhead is 5.23%, which is mainly introduced by the regulation array and weight refreshing. The geometric mean of energy overhead is 7.75%, which is slightly larger than latency overhead due to a large amount of RRAM writing, and the overall performance overhead is 7.14%. Second, from the NN model perspective, we find that VGG models have larger energy overhead but smaller latency overhead than ResNet models. This is because VGG models have three large FC layers, and they require more RRAM crossbars but essentially perform less computation, compared with convolutional layers. Therefore, the energy overhead comes from refreshing FC layers in VGG, which is relatively larger than ResNet. On the other hand, as convolutional layers require more computation, its latency suffers more from the regulation array as the array increases data loading latency, so the ResNet appears to have a larger latency overhead.

(2) Area/Power Analysis. We present the area and power breakdown in Fig. 6.11. As shown, the total area of SIGHT is 53.26 mm^2 and the peak power is 48.96 W. The largest components include the SRAM buffer and ADC, where the input/output buffer takes 24.74 mm^2, the input map buffer takes 11.13 mm^2, and ADC takes 9.83 mm^2. Our add-on components of regulation arrays introduce 3.52 mm^2 area overhead, which is 6.6% of the whole area. The input/output registers for DAC/ADC take 2.94 mm^2

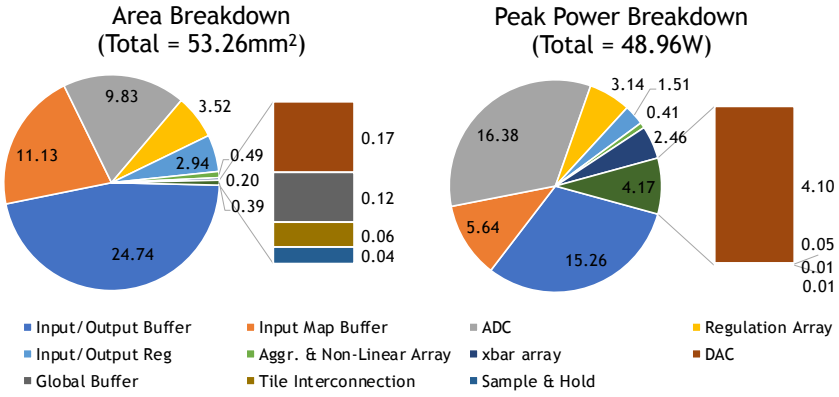


Figure 6.11: The area and power breakdown of SIGHT.

area while other components including RRAM itself, logic unit, DAC and so on occupy a negligible area in the whole architecture.

From the peak power perspective, the SRAM buffer and ADC still consume the most power, where the input/output buffer takes 15.26 W, the input map buffer takes 5.64 W, and ADC takes 16.38 W. The regulation array consumes 3.14 W peak power, which is 6.4% of the whole system. The DAC and RRAM crossbar take another noticeable fraction with 4.10 W and 2.46 W peak power, respectively.

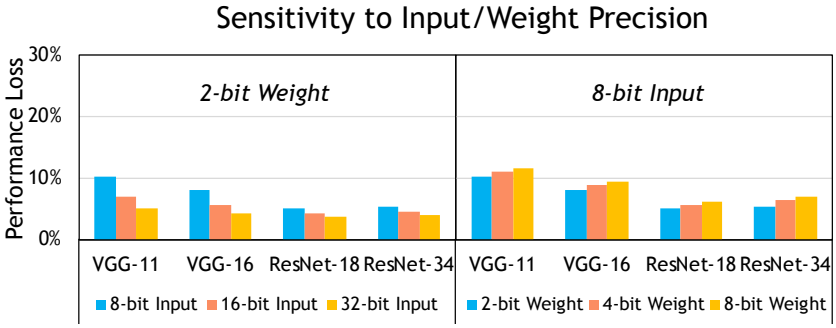


Figure 6.12: The performance sensitivity to input and weight precision. The left part fixes weight to 2-bit while the right part fixes input to 8-bit.

(3) Sensitivity Study. To better understand the trade-off in hardware configuration, we also change the design parameters in SIGHT to see how its performance is sensitive to different architecture settings.

Sensitive to Input/Weight Resolution: As shown in Fig. 6.12, we fix the model to 2-bit weight/8-bit activation, and tune the activation/weight precision respectively to see how the performance overhead changes. First, from the left part where the weight is fixed to 2-bit, we find that with higher activation precision, the performance overhead is actually reducing. This is because when increasing the activation precision, we do not necessarily need more RRAM crossbars but perform computation over the same RRAM crossbar repeatedly. Then, the proportion of energy consumption in weight refreshing is then decreased relatively, so we have less overall performance loss. On the other hand, when fixing the input precision to 8-bit, higher weight precision would increase the performance loss, as observed in the right part of Fig. 6.12. Because more RRAM crossbars are needed in such a case and thus more energy overhead is introduced when refreshing the weight.

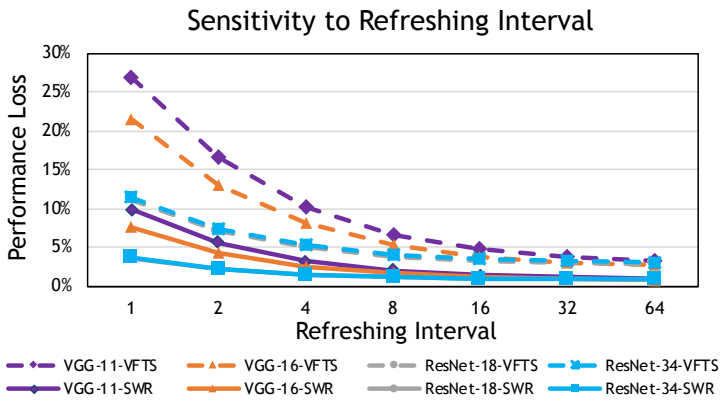


Figure 6.13: The performance sensitivity of selective weight restoration (SWR) to different refreshing intervals from refreshing per 1 image to per 64 images, compared with variation-free tuning scheme (VFTS) [134].

Sensitive to Refreshing Interval: Since various RRAM devices emerge based on different materials, they may exhibit unique retention character and thus require different refreshing frequency. As shown in Fig. 6.13, we present the performance loss of selective weight restoration (SWR, solid lines) when ranging the refreshing interval from every 1 image to every 64 images, with the comparison to variation-free tuning scheme (VFTS,

dotted lines) [134]. Our SWR causes much less performance degradation that is less than 10%, compared with VFTS that could lead to near 30% performance loss when refreshing the weight frequently. Also, the performance loss can be reduced significantly by increasing the refreshing interval, especially when the interval is less than 8. The benefit mainly comes from the reduction of relative energy consumption of weight refreshing. These observations demonstrate that the proposed SWR enjoys more benefits when the RRAM device is more vulnerable to dynamic variations.

6.6 Conclusion

In this chapter, we present SIGHT, a SynergIstic alGorithm-arcHitecture fault-Tolerant framework, to holistically address the reliability issues in RRAM devices. Specifically, we consider three major types of faults for RRAM computing: non-linear resistance distribution, static variation, and dynamic variation. From the algorithm level, we propose a resistance-aware quantization to compel the neural network parameters to follow the exact non-linear resistance distribution as RRAM and introduce an input regulation technique to compensate for RRAM variations. We also propose a selective weight refreshing scheme to address the dynamic variation issue that occurs at run-time. Finally, we propose a *general* and *low-cost* architecture accordingly for supporting our fault-tolerant scheme. Our evaluation demonstrates almost no accuracy loss for our fault-tolerant algorithms, and the SIGHT architecture incurs performance overhead as little as 7.14%.

Chapter 7

Learning the Sparsity for RRAM: Mapping and Pruning Sparse Neural Network for RRAM based Accelerator

Chapter 6 has presented our approach to tolerant the reliability issue, which makes RRAM-based PIM possible for neural network acceleration. In this chapter, we further describe how to map the network model to the RRAM-based PIM accelerator. The challenge of mapping is due to the intrinsic sparsity in the model. This chapter first discusses the background and motivation of mapping sparse neural networks to RRAM. Then, we introduce our reordering-based mapping scheme and RRAM-aware pruning algorithm. Finally, we evaluate our proposal based on the model accuracy and performance gain.

7.1 Background and Motivation

With the in-memory processing ability, RRAM-based computing gets more and more attractive for accelerating neural networks (NNs). RRAM-based NN acceleration leverages the RRAM crossbar array to achieve $O(1)$ computation complexity. The computation is performed by mapping the weight matrix to the crossbar, and then the vector is input by activating all rows.

Current NN often contains enormous parameters. Previous work has found that NN models usually exhibit intrinsic sparsity. This means a large portion of the parameters are redundant and thus can be pruned [20, 155]. Therefore, sparsity becomes a popular way to make the model more efficient. After pruning, the weight in sparse NN contains plenty of zeros (often >70% sparsity), whose computation can be skipped.

However, existing RRAM-based NN accelerators cannot support efficient mapping for sparse NN, and we still need to map the whole dense matrix onto the RRAM crossbar array. As shown in Fig. 2.2, to remain the $O(1)$ computation complexity of the matrix-vector multiplication, we must map the whole matrix into the crossbar according to Eq. 2.1. Even though there could be lots of zeros in the crossbar, their computations are non-skippable.

Overcoming the intrinsic contradiction between dense crossbars and the sparse matrix is very challenging. Researchers have made some exploration to make the mapping of the sparse matrix more efficient. For graph processing, Song tried to use ultra-small ReRAM processing element (PE), like 8×8 or 4×4 crossbar, to traverse the adjacency matrix [156], which, however, will consume lots of energy for large scale NN, because it has to read/write the weights repeatedly. Wang focused on mapping structured sparse NN [155] with block building approach [157], but not looking into common irregular sparse NN, which can prune more redundant parameters. Besides, some work proposed

to train a sparse NN that fits the hardware structures of ReRAM crossbar [158]. It may be a good way but still cannot deal with the mapping problem for existing sparse NN.

In this chapter, we propose a novel mapping scheme for sparse NN, along with a software-level pruning algorithm to efficiently accelerate sparse NN with RRAM-based accelerators. Simulation results show that our mapping scheme with the proposed pruning algorithm achieves 3∼5× energy efficiency and 2.5∼6× speedup, compared with PRIME [26], the well-known RRAM-based NN accelerator.

7.2 Sparse NN Mapping Scheme

To reduce the redundancy in RRAM crossbar and make use of the sparsity of NN, we next introduce our mapping scheme for sparse NN with as fewer crossbar arrays used as possible. We first introduce the insight that motivates our column-exchanging-based mapping algorithm. Then, we explain the mapping procedure in detail and discuss the overhead it causes.

Insight: There are two observations that help us explore the efficient mapping design. First, to achieve higher accuracy in complicated tasks, the development of NN is going deeper and larger-scale. In common-used neural networks, the weight matrix can be very large. For instance, the first full connected (FC) layer of VGG-16, which contains more than 90% parameters of the whole network, has the weight matrix with the size of 25088×4096. Obviously, such a massive matrix cannot be mapped on one single RRAM crossbar and needs to be split into smaller blocks. Meanwhile, in the current tape-out chip of RRAM-based computing systems, the size of fabricated RRAM crossbar is quite small, like 32×32 or 64×64 [159, 160]. The second observation is that for those extremely large layers, although there are massive parameters, the pruning results show that these layers will be really sparse. Again, take the FC layer in VGG-16 for example,

after pruning, the density of it is 4% [20], which means that only 4% elements remain and 96% of the matrix is zero. Further, since we need two crossbars to represent the positive and negative parts of the matrix respectively, the density will be half smaller if half the parameters are positive and the others are negative, which means about 98% of parameters of the matrix become zero.

Key Idea: The observations inspire us that those smaller blocks of the split weight matrix are probably all-zero, or have all-zero columns/rows. If we eliminate such zero parts, we could save considerable resources. However, although there is only one non-zero element in the column/row of the RRAM crossbar, we need to keep this column/row to make the computation of matrix in the correct order. Here we proposed the k-means clustering based column exchanging scheme for mapping. In fact, some work applied the spectrum clustering algorithm to gather the neurons for pruning [158]. But here our goal is to make the non-zero elements more concentrated through column exchanging. Thus we can gain more all-zero rows and save more crossbars.

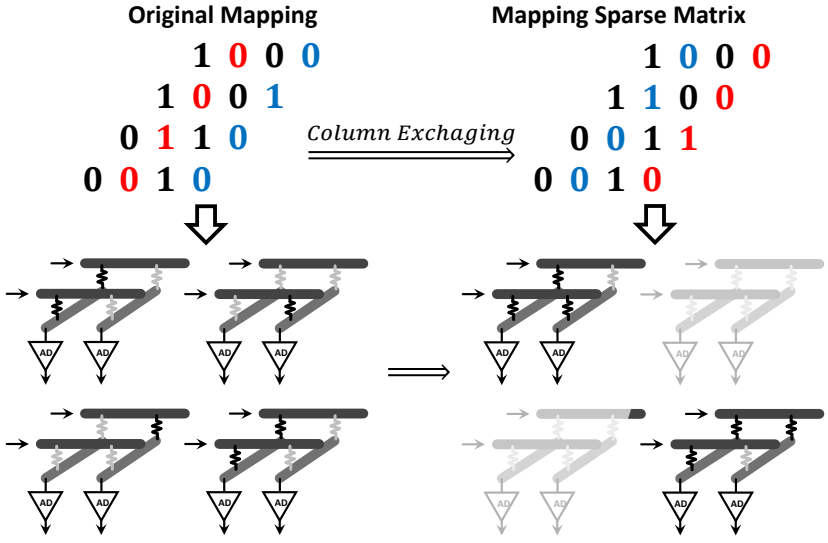


Figure 7.1: An illustration of our column-exchanging-based mapping algorithm. (a) Column exchanging for sparse matrix. (b) The whole mapping scheme

k-means is a wide-used method for cluster analysis in data mining. Given n samples,

k-means algorithm aims to cluster them into k labels, with the minimal within-cluster variance. In our application, we have n columns in the original large matrix and crossbar size is L, so we need to partition the n columns, or say n vectors into n/L labels, with L vectors in each label. k-means algorithm will not assign specific number of samples in each label, so our solution is to take the most concentrated L vectors as clustering results, and then through a recursive procedure we repeat the clustering for remaining vectors. Also, since k-means is sensitive to the initial value, we choose to repeat our algorithm to avoid local optimal results.

Algorithm 9: k-means-base Column Exchanging for Mapping Sparse Maxtrix

Require: Weight matrix W, splitting function f_{split}, RRAM crossbar size L.
Initialize:
  (x, y) = size of W, $ext = y$ mod L; Pad W with ext zero columns;
  sp = sparsity of W, splitting size $(x_{split}, y_{split}) = f_{split}(x, y, sp)$;
  Split W into m small blocks W_i;
  Clustering result $R_i = \{\}, i = 1, 2, ..., m$;
for $i = 0 : m$ **do**
  Unclustered Set $U = \{v\}$; `// v is each column in` W_i
  while $U! = \emptyset$ **do**
    $n = |U|$;
    Cluster U into n subset C_j using k-means with hamming distance;
    for $j = 0 : n$ **do**
      if $|C_j| \geq L$ **then**
        Find the nearest L vectors $\{v\} \subset C_j$;
        Add $\{v\}$ into R_i, and remove them from U;
      end
    end
  end
end
for $\{v\}$ *in* R_i **do**
  Compose the block matrix $W_{i,block}$ with $\{v\}$;
  Eliminate all-zero parts and shrink $W_{i,block}$, then map it into RRAM crossbar;
end

Algorithm 9 shows our mapping algorithm which requires a weights matrix, RRAM crossbar size, and a pre-splitting function f_{split} as input. Clustering an extremely large

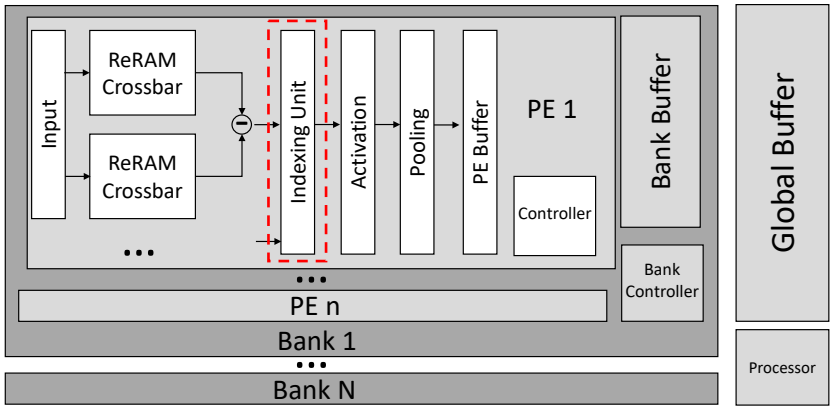


Figure 7.2: The accelerator architecture.

matrix, like 25088×4096, can consume much more time and not achieve the optimal result for smaller pieces. Therefore, we propose to pre-split the matrix before k-means clustering as shown in Lines 3-5 with the pre-splitting function f_{split}. Lines 7-19 describe our key operation for k-means-based column exchanging. We cluster the vectors in the split matrix and pick up L columns into the target set R repeatedly, and through the recursive calls we will shuffle all the columns into specific sets we need. After we exchange the columns according to clustering results, the size of the split matrix will shrink as we eliminate the all-zero parts. Then we can further split the shrunk matrix into crossbar size and finish the mapping procedure as described in line 20-23.

Note that we use a splitting function f_{split} to decide the split size, and here is how it works: We first produce plenty of random matrices to start different splitting strategies and choose the best one by iteration. Through those experienced parameters, we infer the splitting size for a given matrix. So in our experiments, the f_{split} is actually an interpolating function.

Fig. 7.2 shows the architecture with our mapping scheme. The flowchart of our architecture is similar to PRIME but we add extra indexing units. Because we need to reorder the columns and rows of a regular matrix. The index registers are only added for the output vectors because we can schedule the corresponding input voltages after

NN mapping. Note that the index units are needed for RRAM crossbars but not every matrix element. Besides, Due to the pre-splitting procedure, we do not need to index among the huge matrix but inside the split small block.

7.3 Crossbar-Grained Pruning

In the section above, we introduce our proposed sparse mapping scheme based on column exchanging strategy, which will gather non-zero elements together as concentrated as possible in order to eliminate the zero crossbars. However, we could still find that in some crossbars, the utilization of ReRAM cells is much lower, and there may exist only few, or even just one non-zero element(s) in one row. We implement our mapping algorithm on 64×64 ReRAM crossbar for VGG-16. Fig. 7.3 shows the distribution of utilization ratio of those crossbars, compared with original mapping in the dense way. The percentile in x-axis represents the ratio for non-zero cells in the ReRAM crossbar. We can easily find that in more than 20% crossbars, there are only 15% ReRAM cells storing valid parameters and 85% of them are zero.

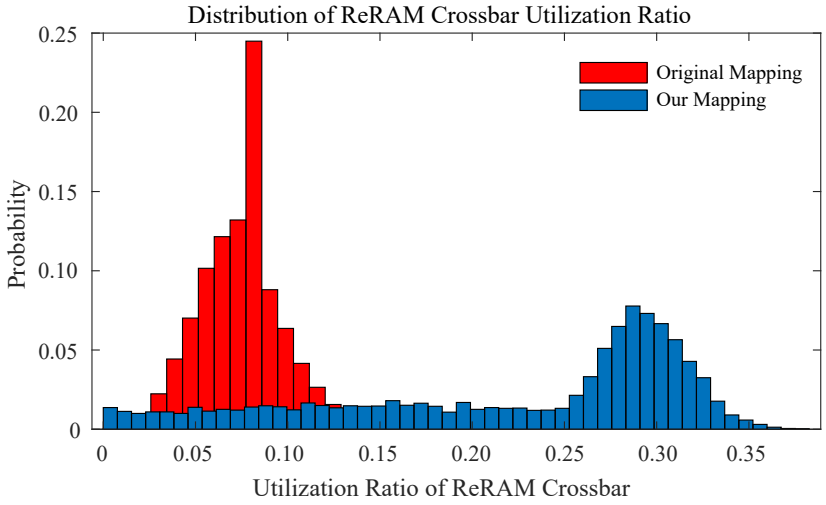


Figure 7.3: The distribution of utilization ratio for ReRAM crossbar, with mapping a sparse VGG-16.

Obviously, if we can further delete those low utilization crossbars, we can achieve much more hardware resource reduction. Here we will introduce our crossbar-grained NN pruning to further compress the sparse NN and save ReRAM crossbars. The key idea is to throw away those ReRAM crossbars with only a few parameters. First, we pre-prune the neural network and implement column exchanging for NN mapping according to Section 7.2. Then we start crossbar-grained pruning for NN, which will remove multiple weights in the crossbar rows. We adopt the pruning criterion in Mao's work [161]: Compute the Salience $S_i = \sum_{w \in G_i} |w|$, i.e. the sum of L1-norm weights, to decide which group of weights should be deleted. Here the pruning grain G_i is the ReRAM crossbar rows. After weights pruning, we finetune the NN to rescue the accuracy, expecting the least accuracy loss. To achieve better pruning results, we can repeat the procedure of "Pruning - Finetuning - Pruning", and get the final NN model step by step.

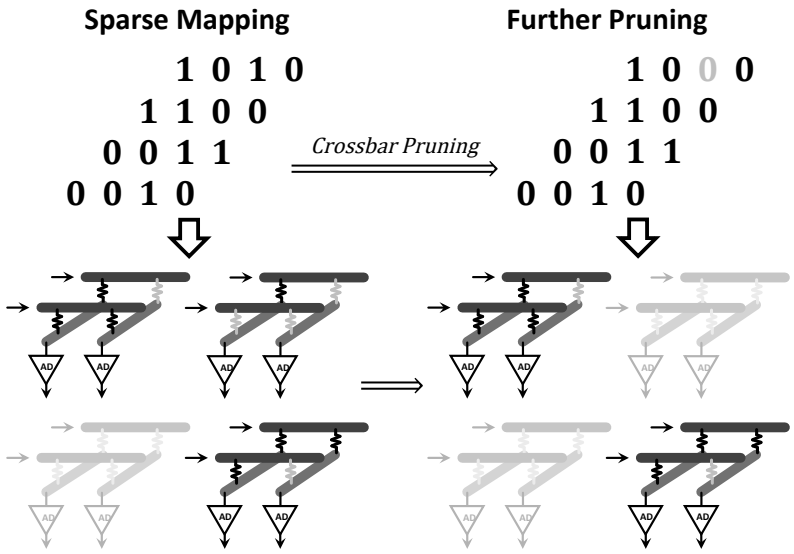


Figure 7.4: ReRAM crossbar-grained pruning for sparse NN.

Finally, mention that our pruning algorithm is conducted on sparse NN, where the model is already compressed. Therefore, further pruning may damage the NN performance on accuracy. In our experiments section, we will discuss the above issue in detail.

7.4 Simulation Results

7.4.1 Simulation Setup

We implement our design, including our sparse NN mapping algorithm along with the proposed precision composing circuits design on PRIME, which means we modify the PRIME design and its data scheduling. We mainly look into energy efficiency and speedup as our evaluation metrics. We assume 2-bit ReRAM cell and simulate the energy cost through NVSim [162]. To evaluate the performance of ReRAM crossbar-grained pruning, we will compare the accuracy loss after pruning for different NNs.

To thoroughly exam our design for different NN structures, we choose to conduct our simulation on CNN (Convolutional Neural Network) and RNN (Recurrent Neural Network). CNN mainly consists of convolutional layers, like ResNet series[3], which only have one fully connected layer, while RNN is a typical fully connected neural network, like LSTM[4]. The benchmarks in our simulation are LeNet-5, AlexNet, VGG-16, ResNet-20, and LSTM-5. The LSTM-5 model we use is a 5-layer bi-directional LSTM RNN, and the length of the hidden unit is 800. The sparsity, which represents the degree of pruning, of each NN can be found in Table 7.4.1.

Table 7.1: The Sparsity of each NN

NN	LeNet-5	AlexNet	VGG-16	ResNet-18	LSTM-5
Sparsity	92%	89%	92.5%	75%	85%

7.4.2 Energy Results with Sparse Mapping

Before we conduct our experiments on different NNs, we look into how the crossbar size affects the system's energy efficiency. We take VGG-16 to implement our experiments and get our energy results through simulation. As shown in Fig. 7.5, the x-axis represents

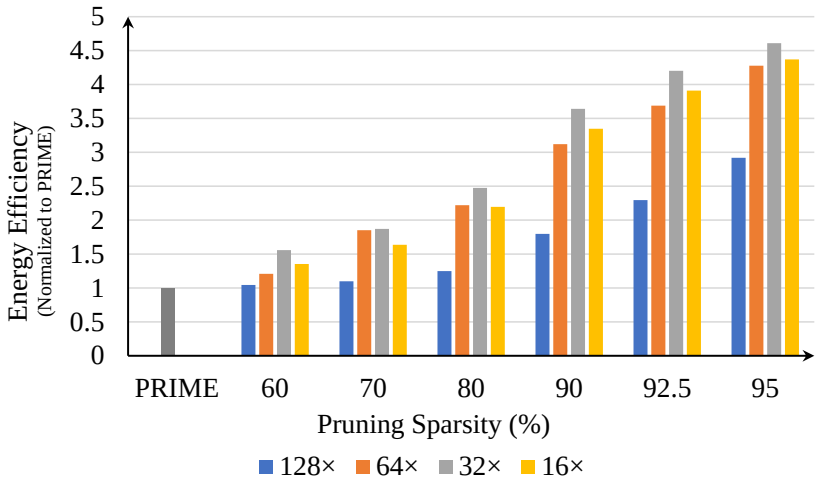


Figure 7.5: The energy results with gradually deeper NN pruning sparsity, and different ReRAM crossbar size.

deeper pruned NNs with different crossbar size, and the y-axis plots the normalized energy efficiency compared with PRIME. We can find that as the ReRAM crossbar gets smaller, we may first save more energy, but the energy cost becomes larger with 16×16 crossbar, which may be caused by the huge amount of interfaces when mapping such a large NN into those really small ReRAM crossbar. Therefore, we will take 32×32 ReRAM crossbar in our next experiment.

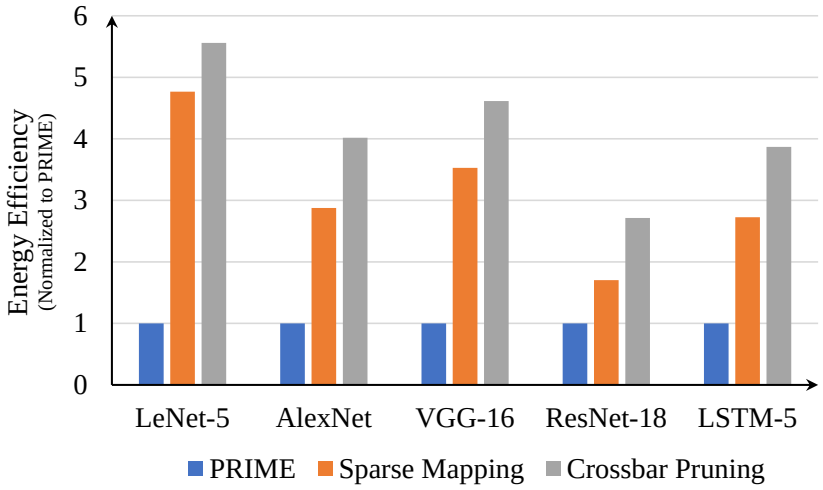


Figure 7.6: The Energy Results with Different NNs.

Fig. 7.6 plots the energy performance, and the results have been normalized to PRIME. As seen, our system for sparse NN achieves $3-5\times$ energy efficiency improvement for popular NNs after our crossbar-grained pruning, which makes the energy efficiency further improved. Through the results, we find that the system performs much better in NN with large layers, like AlexNet, VGG or LSTM, because such layer often occupy too much energy cost and will be quite sparse after pruning. Meanwhile, for the ResNet which mainly contains convolutional layers, it is more difficult to cut down the parameters, but the energy efficient will be improved a lot after our crossbar grained pruning. The system can achieve more energy savings as the NN get sparser, as we learn from Fig. 7.5. Finally, we have to note that the excellent result of LeNet-5 is that mapping such a small NN into 256×256 crossbar will cause huge resources wasted, while our small crossbar show its charm to LeNet-5.

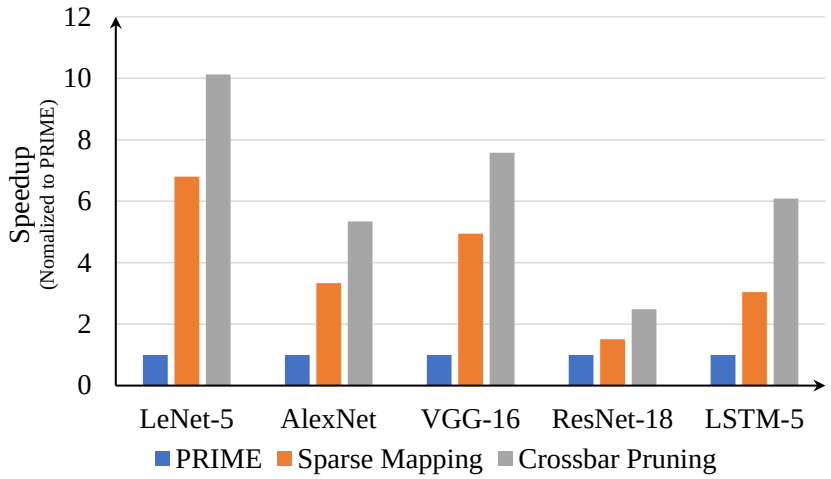


Figure 7.7: The Speedup Results with Different NNs.

The system's speedup is mainly brought by the data re-using in those ReRAM crossbar we saved, and faster time-cycle for smaller crossbar. Fig. 7.7 plots the time consuming results for different NN, where we can learn that our system achieve more than $5\times$ speedup for most NNs. However, The results for ResNet is not so impressing, the rea-

son is the same, for convolutional layers, compressing with pruning and mapping sparse irregular weights matrix is more difficult.

7.4.3 Accuracy Results with Crossbar-Grained Pruning

We conduct our pruning experiments through LeNet-5 on MNIST dataset, VGG-16 and ResNet-18 on CIFAR-10 dataset, and LSTM-5 on 1000h LibriSpeech.

Fig. 7.8 shows the comparison between the parameters we removed and the crossbars we saved, both of which have been normalized to the whole NN model. We find that through pruning a small amount of parameters, we can save plenty of crossbar resources. Table 7.2 presents the accuracy for those NNs, through which we can find that for three CNNs and LSTM, our crossbar-grained pruning achieves almost no accuracy loss (<1%). Besides, note that the performance for accuracy highly depends on the redundancy in NN, and in our experiments, we adjusted the pruning rate gradually for the trade-off between accuracy and hardware efficiency.

Table 7.2: Accuracy Results After Crossbar Pruning

Nerual Networks	LeNet-5	VGG-16	ResNet-18	LSTM-5
Original	**99.23%**	93.64%	**92.37%**	**89.24%**
Normal Pruning	99.13%	93.62%	92.07%	88.49%
Crossbar Pruning	99.15%	**93.72%**	91.78%	88.01%

7.5 Conclusions

In this chapter, we propose a novel sparse NN mapping scheme based on weight columns clustering, to achieve better ReRAM crossbar utilization. Further, we propose crossbar-grained pruning algorithm to reduce the crossbars with low utilization. The simulation results show that compared with those accelerators for dense NN, our mapping

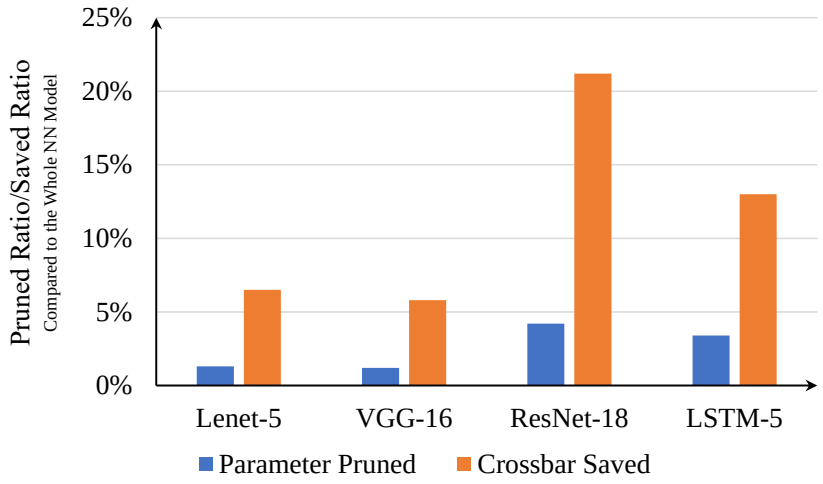


Figure 7.8: The NN parameters we pruned V.S. the crossbar resources we saved, both of which are normalized to the whole NN model.

scheme for sparse NN with proposed pruning algorithm achieves 3∼5× energy efficiency and more than 2.5∼6× speedup. Also, our pruning algorithm shows there is almost no accuracy loss.

Chapter 8

Summary

Memory is the most important component in various computing systems, especially in the big data era. However, due to the slowdown in technology scaling, the performance of memory can become a significant bottleneck to the system. To overcome the limitations in memory subsystems, this dissertation makes efforts from two perspectives: (1) We leverage near-data processing (NDP) to improve the performance of existing memory technology; (2) We design RRAM-based in-memory processing (IMP) architecture as the next-generation memory.

We first explore how to design a near-memory-processing (NMP) architecture to facilitate the deep learning workload, which is at the core of big data applications. for the extreme classification task, which appears in most deep learning workloads. In particular, we investigate the extreme classification that is the essential component of large-scale deep learning systems. However, as classification categories keep scaling in real-world applications, the classifier's parameters could reach several thousands of Gigabytes, way exceeding the on-chip memory capacity. Moreover, naive NMP designs with a limited area and power budget cannot afford the computational complexity of full classification. To tackle the problem, we take the approach of algorithm and architecture

co-designing. We propose a novel screening method to reduce computation and memory consumption by efficiently approximating the classification output and identifying a small portion of key candidates that require accurate results. Then, we design a new extreme-classification-tailored NMP architecture, namely ENMC, to support both screening and candidates-only classification. Overall, our approximate screening method achieves 7.3× speedup over the CPU baseline, and ENMC further improves the performance by 7.4× and demonstrates 2.7× speedup compared with the state-of-the-art NMP baseline.

While extreme classification usually exhibits regular-patterned computation and data access, we also explore how irregular-patterned workloads can benefit from NMP architectures. Thus, we look into the graph processing, which is known to suffer from two memory inefficiencies: the on-chip cache inefficiency caused by dominated expensive random accesses, and the off-chip bandwidth inefficiency caused by the small data granularity. To tackle this problem, we present G-MEM, a customized memory hierarchy design for graph processing applications. First, we propose a coherence-free scratchpad as the on-chip memory, which leverages the power-law characteristic of graphs for frequent-accessed data. Second, we design an elastic-granularity DRAM (EG-DRAM) based on NMP technique, which processes and coalesces multiple fine-grained memory accesses together to maximize the bandwidth efficiency. Putting them together, we see a 2.63× speedup over the traditional CPU, where the NMP architecture brings 1.79× speedup.

Further, we focus on the scalability of NDP. The NMP technique targets workloads with relatively small data footprint that can fit into the main memory. For large-scale workloads that have to stay in storage, we find that NDP can still bring performance gain with in-storage-processing (ISP) architecture. Specifically, we look into the Private Information Retrieval (PIR) protocol that plays a vital role in secure, database-centric applications. However, existing PIR protocols explore a massive working space containing

hundreds of GiBs of query and database data. As a consequence, PIR performance is severely bounded by storage communication, making it far from practical for real-world deployment. We describe a protocol and architecture co-designed solution, INSPIRE. We first design the INSPIRE protocol with a multi-stage filtering mechanism, which achieves a constant PIR query size. For a 1-billion-entry database of size 288GiB, INSPIRE's protocol reduces the query size from 27GiB to 3.6MiB. Further, we propose the INSPIRE hardware, a heterogeneous in-storage architecture, which integrates our protocol across the SSD hierarchy. Together with the INSPIRE protocol, the INSPIRE hardware reduces the query time from 28.4min to 36s, relative to the state-of-the-art FastPIR scheme.

On the other hand, we investigate the opportunity of IMP as the next-generation memory with computation capability. Particularly, emerging Resistive Random Access Memory (RRAM) has shown the great potential of in-memory processing capability in the analog domain. RRAM has attracted considerable research interest in accelerating memory-intensive applications, such as neural networks (NNs). However, the accuracy of RRAM-based NN computing can degrade significantly, due to the intrinsic statistical variations of the resistance of RRAM cells. In this dissertation, we propose SIGHT, a fault-tolerant framework with algorithm and architecture co-design to holistically address this issue. We consider three major types of faults for RRAM computing: non-linear resistance distribution, static variation, and dynamic variation. From the algorithm level, we propose a resistance-aware quantization to compel the NN parameters to follow the non-linear resistance distribution as RRAM, and introduce an input regulation technique to compensate for RRAM variations. We also propose a selective weight refreshing scheme to address the dynamic variation issue that occurs at run-time. From the architecture level, we propose a general and low-cost architecture accordingly for supporting our fault-tolerant scheme.

Finally, as our SIGHT framework addresses the reliability issues in existing RRAM

devices, we still need to map neural networks to the RRAM-based IMP accelerator. However, most RRAM-based accelerators cannot support efficient mapping for sparse NN, and we need to map the whole dense matrix onto RRAM crossbar array to achieve $O(1)$ computation complexity. In this dissertation, we propose a sparse NN mapping scheme based on elements clustering to achieve better RRAM crossbar utilization. Further, we propose a crossbar-grained pruning algorithm to remove the crossbars with low utilization. In our experiments, we discuss how the system performs with different crossbar sizes to choose the optimized design. Our results show that our mapping scheme for sparse NN with the proposed pruning algorithm achieves 3$\sim$5$\times$ energy efficiency and more than 2.5$\sim$6$\times$ speedup, compared with the state-of-the-art accelerator for dense NNs. Also, the accuracy experiments show $<$1% accuracy loss in our pruning method.

We hope this dissertation would help to improve the memory subsystem in modern computation systems and also inspire NDP/IMP research for a broad domain of applications in the future.

Bibliography

[1] M. Insight, "Dna's awesome potential to store the world's data." https://www.micron.com/insight/dnas-awesome-potential-to-store-the-worlds-data. Accessed: 2020-03-10.

[2] X.-W. Chen and X. Lin, *Big data deep learning: challenges and perspectives*, *IEEE access* **2** (2014) 514–525.

[3] K. He *et. al.*, *Deep residual learning for image recognition*, in *CVPR*, pp. 770–778, 2016.

[4] F. A. Gers *et. al.*, *Learning to forget: Continual prediction with LSTM*, .

[5] Z. Li, H. Zhao, Q. Liu, Z. Huang, T. Mei, and E. Chen, *Learning from history and present: Next-item recommendation via discriminatively exploiting user behaviors*, in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1734–1743, 2018.

[6] S. Li, *Memory-centric architectures: Bridging the gap between compute and memory*. University of California, Santa Barbara, 2018.

[7] NVIDIA, "Nvidia tesla k80." https://www.nvidia.com/en-gb/data-center/tesla-k80/, 2014.

[8] NVIDIA, "Nvidia a100 tensor core gpu architecture." https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[9] 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, IEEE, 2013.

[10] C. Garcia, "The real amount of energy a data center uses." https://www.akcp.com/blog/the-real-amount-of-energy-a-data-center-use/, 2022.

[11] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz, *Rethinking dram power modes for energy proportionality*, in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 131–142, IEEE, 2012.

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

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

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

[15] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, *The anatomy of the facebook social graph*, *arXiv preprint arXiv:1111.4503* (2011).

[16] X. Shao, J. Xie, T. Hong, and A. Jost, *System and method for identity-based fraud detection through graph anomaly detection*, July 21, 2009. US Patent 7,562,814.

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

[18] Y. Wu, B. Lee, and H.-S. P. Wong, *Ultra-low power al 2 o 3-based rram with 1μa reset current*, in *Proceedings of 2010 International Symposium on VLSI Technology, System and Application*, pp. 136–137, IEEE, 2010.

[19] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, *Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme*, *IEEE Transactions on Computers* **64** (2014), no. 1 180–190.

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

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

[22] J. Woo, K. Moon, J. Song, M. Kwak, J. Park, and H. Hwang, *Optimized programming scheme enabling linear potentiation in filamentary hfo 2 rram synapse for neuromorphic systems*, *IEEE Transactions on Electron Devices* **63** (2016), no. 12 5064–5067.

[23] L.-E. Yu, S. Kim, M.-K. Ryu, S.-Y. Choi, and Y.-K. Choi, *Structure effects on resistive switching of al/tio$_x$/al devices for rram applications*, *IEEE electron device letters* **29** (2008), no. 4 331–333.

[24] U. Russo, D. Ielmini, C. Cagli, and A. L. Lacaita, *Filament conduction and reset mechanism in nio-based resistive-switching memory (rram) devices*, *IEEE Transactions on Electron Devices* (2009) 186–192.

[25] S.-S. Sheu, M.-F. Chang, K.-F. Lin, C.-W. Wu, Y.-S. Chen, P.-F. Chiu, C.-C. Kuo, Y.-S. Yang, P.-C. Chiang, W.-P. Lin, *et. al.*, *A 4mb embedded slc resistive-ram macro with 7.2 ns read-write random-access time and 160ns mlc-access capability*, in *2011 IEEE International Solid-State Circuits Conference*, pp. 200–202, IEEE, 2011.

[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* **44** (2016), no. 3 27–39.

[27] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, *Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars*, *ACM SIGARCH Computer Architecture News* **44** (2016) 14–26.

[28] L. Xia *et. al.*, *Technological exploration of RRAM crossbar array for matrix-vector multiplication*, *Journal of Computer Science and Technology* (2016) 3–19.

[29] D. Fujiki, S. Mahlke, and R. Das, *In-memory data parallel processor*, *ACM SIGPLAN Notices* **53** (2018), no. 2 1–14.

[30] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, *Behemoth: A flash-centric training accelerator for extreme-scale dnns*, in *USENIX Conference on File and Storage Technologies (FAST)*, pp. 371–385, 2021.

[31] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, *Extrav: boosting graph processing near storage with a coherent accelerator*, *Proceedings of the VLDB Endowment* **10** (2017), no. 12 1706–1717.

[32] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson, *Ssd in-storage computing for search engines*, *IEEE Transactions on Computers* (2016).

[33] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, *arXiv preprint arXiv:1310.4546* (2013).

[34] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, *arXiv preprint arXiv:1409.3215* (2014).

[35] K. Shim, M. Lee, I. Choi, Y. Boo, and W. Sung, *SVD-softmax: Fast softmax approximation on large vocabulary neural networks*, in *Neural Information Processing Systems (NeurIPS)*, pp. 5464–5474, 2017.

[36] L. Song, P. Pan, K. Zhao, H. Yang, Y. Chen, Y. Zhang, Y. Xu, and R. Jin, *Large-Scale Training System for 100-Million Classification at Alibaba*, in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2909–2917, 2020. arXiv:2102.0602.

[37] T. Medini, Q. Huang, Y. Wang, V. Mohan, and A. Shrivastava, *Extreme classification in log memory using count-min sketch: A case study of amazon search with 50m products*, in *Neural Information Processing Systems (NeurIPS)*, 2019. arXiv:1910.1383.

[38] X. Zhang, L. Yang, J. Yan, and D. Lin, *Accelerated training for massive classification via dynamic class selection*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[39] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, *Graph convolutional neural networks for web-scale recommender systems*, in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, 2018.

[40] Y. Prabhu, A. Kag, S. Harsola, R. Agrawal, and M. Varma, *Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising*, in *Proceedings of the 2018 World Wide Web Conference*, pp. 993–1002, 2018.

[41] J. Liu, W. C. Chang, Y. Wu, and Y. Yang, *Deep learning for extreme multi-label text classification*, in *SIGIR 2017 - Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, (New York, NY, USA), pp. 115–124, Association for Computing Machinery, Inc, aug, 2017.

[42] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado,

M. Hughes, and J. Dean, *Google's neural machine translation system: Bridging the gap between human and machine translation*, arXiv:1609.0814.

[43] S. Merity, C. Xiong, J. Bradbury, and R. Socher, *Pointer sentinel mixture models*, *arXiv preprint arXiv:1609.07843* (2016).

[44] 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).

[45] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma, *The extreme classification repository: Multi-label datasets and code*, 2016.

[46] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, *fairseq: A fast, extensible toolkit for sequence modeling*, in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[47] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2017.

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

[49] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, *Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules*, in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 283–295, IEEE, 2015.

[50] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, *Chameleon: Versatile and practical near-dram acceleration architecture for large memory*

systems, in *IEEE/ACM international symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[51] D. Achlioptas, *Database-friendly random projections*, in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 274–281, 2001.

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

[53] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, *Recnmp: Accelerating personalized recommendation with near-memory processing*, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 790–803, 2020.

[54] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035. Curran Associates, Inc., 2019.

[55] S. Merity, N. S. Keskar, and R. Socher, *Regularizing and optimizing lstm language models*, *arXiv preprint arXiv:1708.02182* (2017).

[56] J. McAuley and J. Leskovec, *Hidden factors and hidden topics: understanding rating dimensions with review text*, in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 165–172, 2013.

[57] M. Zhang, X. Liu, W. Wang, J. Gao, and Y. He, *Navigating with graph representations for fast and scalable decoding of neural language models*, in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), p. 6311–6322, Curran Associates Inc., 2018.

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

[59] E. Agirre, A. Barrena, and A. Soroa, *Studying the wikipedia hyperlink graph for relatedness and disambiguation*, *arXiv preprint arXiv:1503.01655* (2015).

[60] J. Pujara, H. Miao, L. Getoor, and W. Cohen, *Knowledge graph identification*, in *International Semantic Web Conference*, pp. 542–557, Springer, 2013.

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

[62] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, *Stinger: High performance data structure for streaming graphs*, in *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5, IEEE, 2012.

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

[64] P. Faldu, J. Diamond, and B. Grot, *Domain-specialized cache management for graph analytics*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 234–248, IEEE, 2020.

[65] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, *Graphicionado: A high-performance and energy-efficient accelerator for graph analytics*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[66] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, *Many-core graph workload analysis*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pp. 22:1–22:11, IEEE Press, 2018.

[67] P. Sakarda, T. Brandt, and H. H. Wu, *Memory manager for heterogeneous memory control*, Aug. 4, 2009. US Patent 7,571,295.

[68] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, *Imp: Indirect memory prefetcher*, in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 178–190, 2015.

[69] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, *Exploiting locality in graph analytics through hardware-accelerated traversal scheduling*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, IEEE, 2018.

[70] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, *The dynamic granularity memory system*, in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 548–560, IEEE, 2012.

[71] D. H. Yoon, M. K. Jeong, and M. Erez, *Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput*, in *Proceedings of the 38th annual international symposium on Computer architecture*, pp. 295–306, 2011.

[72] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, *Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses*, in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 267–280, 2015.

[73] Y. Kwon, Y. Lee, and M. Rhu, *Tensor casting: Co-designing algorithm-architecture for personalized recommendation training*, *arXiv preprint arXiv:2010.13100* (2020).

[74] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, *An evaluation of high-level mechanistic core models*, *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).

[75] J. E. Zolnowsky, C. L. Whittington, and W. M. Keshlear, *Memory management unit*, Sept. 25, 1984. US Patent 4,473,878.

[76] B. Egger, J. Lee, and H. Shin, *Scratchpad memory management for portable systems with a memory management unit*, in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pp. 321–330, ACM, 2006.

[77] J. S. Kimmel, R. A. Alfieri, A. Miles, W. K. McGrath, M. J. McLeod, M. A. O'connell, and G. A. Simpson, *Operating system for a non-uniform memory access multiprocessor system*, Aug. 15, 2000. US Patent 6,105,053.

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

[79] JEDEC, "Jesd79-4 - jedec." `https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram`, 2017.

[80] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, *Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures*, in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, ACM, 2009.

[81] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, *Saga-bench: Software and hardware characterization of streaming graph analytics workloads*, 2020.

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

[83] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, *Saga-bench: Software and hardware characterization of streaming graph analytics workloads*, .

[84] Micron, "3-dimensional stack (3ds) ddr4 sdram." `https://www.micron.com/-/media/client/global/documents/products/data\protect\discretionary{\char\hyphenchar\font}{}{}sheet/dram/ddr4/16gb_32gb_x4_x8_3ds_ddr4_sdram.pdf`, 2019.

[85] I. Ahmad, Y. Yang, D. Agrawal, A. El Abbadi, and T. Gupta, *Addra: Metadata-private voice communication over fully untrusted infrastructure*, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[86] F. Li, *Cloud-native database systems at alibaba: Opportunities and challenges*, *Proc. VLDB Endow.* **12** (Aug., 2019) 2263–2272.

[87] M. Talha, M. Sohail, and H. Hajji, *Analysis of research on amazon aws cloud computing seller data security*, *International Journal of Research in Engineering and Innovation* **4** (2020), no. 3 131–136.

[88] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, *Private information retrieval*, in *IEEE Annual Foundations of Computer Science*, pp. 41–50, 1995.

[89] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, *Pir-tor: Scalable anonymous communication using private information retrieval*, in *USENIX Security Symposium (SEC)*, 2011.

[90] A. H. Kwon, D. Lazar, S. Devadas, and B. Ford, *Riffle: An efficient communication system with strong anonymity*, .

[91] S. Angel and S. Setty, *Unobservable communication over fully untrusted infrastructure*, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 551–569, 2016.

[92] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish, *Scalable and private media consumption with popcorn*, in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 91–107, 2016.

[93] S. Wang, D. Agrawal, and A. El Abbadi, *Generalizing pir for practical private retrieval of public data*, in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 1–16, Springer, 2010.

[94] T. Mayberry, E.-O. Blass, and A. H. Chan, *Efficient private file retrieval by combining oram and pir.*, in *NDSS*, Citeseer, 2014.

[95] M. Green, W. Ladd, and I. Miers, *A protocol for privately reporting ad impressions at scale*, in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1591–1601, 2016.

[96] R. Henry, F. Olumofin, and I. Goldberg, *Practical pir for electronic commerce*, in *ACM SIGSAC Conference on Computer and communications security (CCS)*, pp. 677–690, 2011.

[97] D. Demmler, A. Herzberg, and T. Schneider, *RAID-PIR: Practical multi-server pir*, in *ACM Workshop on Cloud Computing Security*, pp. 45–56, 2014.

[98] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond, *Breaking the o (n/sup 1/(2k-1)/) barrier for information-theoretic private information retrieval*, in *IEEE Symposium on Foundations of Computer Science*, pp. 261–270, IEEE, 2002.

[99] C. Devet, I. Goldberg, and N. Heninger, *Optimally robust private information retrieval*, in *USENIX Security Symposium (SEC)*, pp. 269–283, 2012.

[100] C. Dong and L. Chen, *A fast single server private information retrieval protocol with low communication cost*, in *European symposium on research in computer security*, pp. 380–399, Springer, 2014.

[101] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, *Xpir: Private information retrieval for everyone*, *Proceedings on Privacy Enhancing Technologies* **2016** (2016) 155–174.

[102] S. Angel, H. Chen, K. Laine, and S. Setty, *Pir with compressed queries and amortized query processing*, in *IEEE symposium on security and privacy (SP)*, pp. 962–979, 2018.

[103] Y.-C. Chang, *Single database private information retrieval with logarithmic communication*, in *Australasian Conference on Information Security and Privacy*, pp. 50–61, Springer, 2004.

[104] A. Beimel, Y. Ishai, and T. Malkin, *Reducing the servers computation in private information retrieval: Pir with preprocessing*, in *Annual International Cryptology Conference*, pp. 55–73, Springer, 2000.

[105] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

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

[107] J. Fan and F. Vercauteren, *Somewhat practical fully homomorphic encryption.*, *IACR Cryptol. ePrint Arch.* (2012).

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

[109] 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.

[110] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 238–252, 2021.

[111] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, *Recssd: near data processing for solid state drive based recommendation inference*, in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 717–729, 2021.

[112] J. P. Stern, *A new and efficient all-or-nothing disclosure of secrets protocol*, in *International conference on the theory and application of cryptology and information security*, pp. 357–371, Springer, 1998.

[113] "Microsoft SEAL (release 3.6)." `https://github.com/Microsoft/SEAL`, Nov., 2020. Microsoft Research, Redmond, WA.

[114] P. L. Montgomery, *Modular multiplication without trial division*, *Mathematics of computation* **44** (1985), no. 170 519–521.

[115] C. Mermer, D. Kim, and Y. Kim, *Efficient 2d fft implementation on mediaprocessors*, *Parallel Computing* **29** (2003), no. 6 691–709.

[116] M. R. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. E. Lauter, *et. al.*, *Homomorphic encryption standard.*, *IACR Cryptol. ePrint Arch.* **2019** (2019) 939.

[117] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, *Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices*, in *USENIX Conference on File and Storage Technologies (FAST)*, pp. 49–66, 2018.

[118] H. L. Garner, *The residue number system*, in *Papers presented at the the March 3-5, 1959, western joint computer conference*, pp. 146–153, 1959.

[119] A. C. Mert, E. Öztürk, and E. Savaş, *Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture*, in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 253–260, IEEE, 2019.

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

[121] M. Sundermeyer, R. Schlüter, and H. Ney, *Lstm neural networks for language modeling*, in *Thirteenth annual conference of the international speech communication association*, 2012.

[122] H. D. Beale, H. B. Demuth, and M. Hagan, *Neural network design*, *Pws, Boston* (1996).

[123] X. Hong, D. J. Loy, P. A. Dananjaya, F. Tan, C. Ng, and W. Lew, *Oxide-based rram materials for neuromorphic computing*, *Journal of materials science* **53** (2018), no. 12 8720–8746.

[124] B. Traoré, P. Blaise, E. Vianello, L. Perniola, B. De Salvo, and Y. Nishi, *Hfo 2-based rram: Electrode effects, ti/hfo 2 interface, charge injection, and oxygen (o) defects diffusion through experiment andab initiocalculations*, *IEEE Transactions on Electron Devices* (2015).

[125] W. Chien, Y. Chen, K. Chang, E. Lai, Y. Yao, P. Lin, J. Gong, S. Tsai, S. Hsieh, C. Chen, *et. al.*, *Multi-level operation of fully cmos compatible wox resistive random access memory (rram)*, in *2009 IEEE International Memory Workshop*, pp. 1–2, IEEE, 2009.

[126] S. R. Lee, Y.-B. Kim, M. Chang, K. M. Kim, C. B. Lee, J. H. Hur, G.-S. Park, D. Lee, M.-J. Lee, C. J. Kim, *et. al.*, *Multi-level switching of triple-layered taox rram with excellent reliability for storage class memory*, in *2012 Symposium on VLSI Technology*, IEEE, 2012.

[127] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, *Accelerator-friendly neural-network training: Learning variations and defects in rram crossbar*, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 19–24, IEEE, 2017.

[128] Y. Long, X. She, and S. Mukhopadhyay, *Design of reliable dnn accelerator with un-reliable reram*, in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1769–1774, IEEE, 2019.

[129] X. Guan, S. Yu, and H.-S. P. Wong, *On the switching parameter variation of metal-oxide rram—part i: Physical modeling and simulation methodology*, *IEEE Transactions on electron devices* (2012).

[130] X. Guan, S. Yu, and H. P. Wong, *On the variability of hfox rram: From numerical simulation to compact modeling*, in *Proc. Workshop Compact Models*, pp. 815–820, 2012.

[131] Y. Y. Chen, L. Goux, S. Clima, B. Govoreanu, R. Degraeve, G. S. Kar, A. Fantini, G. Groeseneken, D. J. Wouters, and M. Jurczak, *Endurance/retention trade-off on hfo_2 metal cap 1t1r bipolar rram*, *IEEE Transactions on electron devices* (2013) 1114–1121.

[132] C. Cagli, D. Ielmini, F. Nardi, and A. L. Lacaita, *Evidence for threshold switching in the set process of nio-based rram and physical modeling for set, reset, retention and disturb prediction*, in *2008 IEEE International Electron Devices Meeting*, pp. 1–4, IEEE, 2008.

[133] Y. Y. Chen, M. Komura, R. Degraeve, B. Govoreanu, L. Goux, A. Fantini, N. Raghavan, S. Clima, L. Zhang, A. Belmonte, *et. al.*, *Improvement of data retention in hfo 2/hf 1t1r rram cell under low operating current*, in *IEEE International Electron Devices Meeting*, pp. 10–1, IEEE, 2013.

[134] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, *Time: A training-in-memory architecture for rram-based deep neural networks*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **38** (2018), no. 5 834–847.

[135] J. Lin, L. Xia, Z. Zhu, H. Sun, Y. Cai, H. Gao, M. Cheng, X. Chen, Y. Wang, and H. Yang, *Rescuing memristor-based computing with non-linear resistance levels*, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 407–412, IEEE, 2018.

[136] C. Lammie, O. Krestinskaya, A. James, and M. R. Azghadi, *Variation-aware binarized memristive networks*, in *26th IEEE International Conference on Electronics, Circuits and Systems*, pp. 490–493, 2019.

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

[138] C. Nail, G. Molas, P. Blaise, G. Piccolboni, B. Sklenard, C. Cagli, M. Bernard, A. Roule, M. Azzaz, E. Vianello, *et. al.*, *Understanding rram endurance, retention and window margin trade-off using experimental results and simulations*, in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 4–5, IEEE, 2016.

[139] D. Ielmini, F. Nardi, C. Cagli, and A. L. Lacaita, *Trade-off between data retention and reset in nio rrams*, in *2010 IEEE International Reliability Physics Symposium*, pp. 620–626, IEEE, 2010.

[140] A. Fantini, L. Goux, R. Degraeve, D. Wouters, N. Raghavan, G. Kar, A. Belmonte, Y.-Y. Chen, B. Govoreanu, and M. Jurczak, *Intrinsic switching variability in hfo 2 rram*, in *2013 5th IEEE International Memory Workshop*, pp. 30–33, IEEE, 2013.

[141] S. Ambrogio, S. Balatti, Z. Q. Wang, Y.-S. Chen, H.-Y. Lee, F. T. Chen, and D. Ielmini, *Data retention statistics and modelling in hfo 2 resistive switching memories*, in *2015 IEEE International Reliability Physics Symposium*, pp. MY–7, IEEE, 2015.

[142] B. Gao, J. Kang, H. Zhang, B. Sun, B. Chen, L. Liu, X. Liu, R. Han, Y. Wang, Z. Fang, *et. al.*, *Oxide-based rram: Physical based retention projection*, in *2010 Proceedings of the European Solid State Device Research Conference*, pp. 392–395, IEEE, 2010.

[143] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, *Deep neural network compression with single and multiple level quantization*, in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[144] S. Wu, G. Li, F. Chen, and L. Shi, *Training and inference with integers in deep neural networks*, *arXiv preprint arXiv:1802.04680* (2018).

[145] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, *Lognet: Energy-efficient neural networks using logarithmic computation*, in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5900–5904, IEEE, 2017.

[146] D. Miyashita, E. H. Lee, and B. Murmann, *Convolutional neural networks using logarithmic data representation*, *arXiv preprint arXiv:1603.01025* (2016).

[147] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, *Rxnn: A framework for evaluating deep neural networks on resistive crossbars*, *IEEE TCAD* (2020).

[148] Y. Cai, Y. Lin, L. Xia, X. Chen, S. Han, Y. Wang, and H. Yang, *Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification*, in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

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

[150] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, *Cacti 5.1*, tech. rep., Technical Report HPL-2008-20, HP Labs, 2008.

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

[152] C.-Y. Fu, *pytorch-vgg-cifar10*, 2019.

[153] K. Jordan, *PyTorch-ResNet-CIFAR10*, 2018.

[154] A. Krizhevsky, G. Hinton, *et. al.*, *Learning multiple layers of features from tiny images*, .

[155] W. Wen *et. al.*, *Learning structured sparsity in deep neural networks*, pp. 2074–2082, 2016.

[156] L. Song *et. al.*, *GraphR: Accelerating graph processing using reram*, in *HPCA*, pp. 531–543, IEEE, 2018.

[157] P. Wang *et. al.*, *SNrram: An efficient sparse neural network computation architecture based on resistive random-access memory*, in *DAC*, 2018.

[158] A. Ankit *et. al.*, *Trannsformer: Neural network transformation for memristive crossbar based neuromorphic system design*, in *ICCAD*, 2017.

[159] F. Su *et. al.*, *A 462 GOPS/J RRAM-based nonvolatile intelligent processor for energy harvesting ioe system featuring nonvolatile logics and processing-in-memory*, in *VLSI*, pp. T260–T261, 2017.

[160] W.-H. Chen *et. al.*, *A 65nm 1mb nonvolatile computing-in-memory ReRAM macro with sub-16ns multiply-and-accumulate for binary DNN AI edge processors*, in *ISSCC*, pp. 494–496, 2018.

[161] H. Mao *et. al.*, *Exploring the granularity of sparsity in convolutional neural networks*, *IEEE CVPRW* **17** (2017).

[162] X. Dong *et. al.*, *NVSIM: A circuit-level performance, energy, and area model for emerging nonvolatile memory*, *IEEE TCAD* **31** (2012), no. 7 994–1007.