IN-MEMORY COMPUTING WITH CMOS AND EMERGING MEMORY TECHNOLOGIES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Shubham Jain

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

To my family for their unconditional love and support, and for always inspiring me to be better.

ACKNOWLEDGMENTS

First and foremost, I want to express my deepest gratitude to my advisor Prof. Anand Raghunathan for his invaluable teachings, guidance, and support throughout my PhD. Undoubtedly, over the years, his insightful suggestions, work ethics, and inspiration to pursue challenging problems have shaped me as a researcher. His immense energy for research and constant drive for problem-solving, deeply motivated me and enabled me in overcoming numerous challenges throughout my PhD. His strong emphasis on intuitive communication, polished research presentations, and high-quality writing helped me hone my non-technical skills. I thank him for being an outstanding mentor, critic, and advisor to me through the course of my PhD.

I want to thank Prof. Kaushik Roy, Prof. Shreyas Sen, Prof. Vijay Raghunathan, and Prof. Sumeet Gupta for their help, advice, and guidance throughout my PhD. Their insightful thoughts and suggestions helped me considerably in refining my research proposals and ideas. I also thank them for extremely fruitful discussions and brainstorming sessions, and for their availability and willingness to share their thoughts.

Next, I will like to thank the alumni and current members of Integrated Systems Laboratory – Swagath Venkataramani, Ashish Ranjan, Sanchari Sen, Younghoon Kim, Jacob Stevens, Vinod Ganesan, Shrihari Sridharan, Sourjya Roy, Reena Elangovan, Sarada Krithivasan, Manik Singhal and Abinand Nallathambi, for their help and friendship during my PhD. I truly enjoyed our conversations over a broad range of topics and in the process learned so much from each one of them. I will also like to thank my collaborators Sandeep Thirumula, Abhronil Sengupta, Leland Chang, Vijayalakshmi Srinivasan, and Jungwook Choi with whom I was able to explore a wide range of topics that enabled me to gain a more comprehensive research perspective.

Lastly, I want to thank my parents, Tulika and Pradeep, for their unconditional love, support, motivation, and countless sacrifices. They are the true pillars of my aspirations. I want to thank my wife, Radhika, for her immense love, patience, and sacrifices. To my brother Ankit, my sister Devansi, my sister-in-law Bhawna, and my nephew Advit, I thank them for believing in me and being there for me. I solely dedicate the following pages of my dissertation to all of them for their love and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Jain, Shubham PhD, Purdue University, December 2019. In-Memory Computing with CMOS and Emerging Memory Technologies. Major Professor: Anand Raghunathan.

Modern computing workloads such as machine learning and data analytics perform simple computations on large amounts of data. Traditional von Neumann computing systems, which consist of separate processor and memory subsystems, are inefficient in realizing modern computing workloads due to frequent data transfers between these subsystems that incur significant time and energy costs. In-memory computing embeds computational capabilities within the memory subsystem to alleviate the fundamental processor-memory bottleneck, thereby achieving substantial system-level performance and energy benefits. In this dissertation, we explore a new generation of in-memory computing architectures that are enabled by emerging memory technologies and new CMOS-based memory cells. The proposed designs realize Boolean and non-Boolean computations natively within memory arrays.

For Boolean computing, we leverage the unique characteristics of emerging memories that allow multiple word lines within an array to be simultaneously enabled, opening up the possibility of directly sensing functions of the values stored in multiple rows using single access. We propose Spin-Transfer Torque Compute-in-Memory (STT-CiM), a design for in-memory computing with modifications to peripheral circuits that leverage this principle to perform logic, arithmetic, and complex vector operations. We address the challenge of reliable in-memory computing under process variations utilizing error detecting and correcting codes to control errors during CiM operations. We demonstrate how STT-CiM can be integrated within a general-purpose computing system and propose architectural enhancements to processor instruction sets and on-chip buses for in-memory computing.

For non-Boolean computing, we explore crossbar arrays of resistive memory elements, which are known to compactly and efficiently realize a key primitive operation involved in machine learning algorithms, i.e., vector-matrix multiplication. We highlight a key challenge involved in this approach - the actual function computed by a resistive crossbar can deviate substantially from the desired vector-matrix multiplication operation due to a range of device and circuit level non-idealities. It is essential to evaluate the impact of the errors introduced by these non-idealities at the application level. There has been no study of the impact of non-idealities on the accuracy of large-scale workloads (*e.g.*, Deep Neural Networks [DNNs] with millions of neurons and billions of synaptic connections), in part because existing device and circuit models are too slow to use in application-level evaluation. We propose a Fast Crossbar Model (FCM) to accurately capture the errors arising due to crossbar non-idealities while being four-to-five orders of magnitude faster than circuit simulation. We also develop RxNN, a software framework to evaluate DNN inference on resistive crossbar systems. Using RxNN, we evaluate a suite of large-scale DNNs developed for the ImageNet Challenge (ILSVRC). Our evaluations reveal that the errors due to resistive crossbar non-idealities can degrade the overall accuracy of DNNs considerably, motivating the need for compensation techniques. Subsequently, we propose CxDNN, a hardware-software methodology that enables the realization of large-scale DNNs on crossbar systems with minimal degradation in accuracy by compensating for errors due to non-idealities. CxDNN comprises of (i) an optimized mapping technique to convert floating-point weights and activations to crossbar conductances and input voltages, (ii) a fast re-training method to recover accuracy loss due to this conversion, and (iii) low-overhead compensation hardware to mitigate dynamic and hardware-instance-specific errors. Unlike previous efforts that are limited to small networks and require the training and deployment of hardware-instance-specific models, CxDNN presents a scalable compensation methodology that can address large DNNs (*e.g.*, ResNet-50 on ImageNet), and enables a common model to be trained and deployed on many devices.

For non-Boolean computing, we also propose TiM-DNN, a programmable hardware accelerator that is specifically designed to execute ternary DNNs. TiM-DNN supports various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) ternary systems. TiM-DNN is an in-memory accelerator designed using TiM tiles — specialized memory arrays that perform massively parallel signed vector-matrix multiplications on ternary values per access. TiM tiles are in turn composed of Ternary Processing Cells (TPCs), new CMOS-based memory cells that function as both ternary storage units and signed scalar multiplication units. We evaluate an implementation of TiM-DNN in 32nm technology using an architectural simulator calibrated with SPICE simulation and RTL synthesis. TiM-DNN achieves a peak performance of 114 TOPs/s, consumes 0.9W power, and occupies 1.96mm^2 chip area, representing a 300X improvement in TOPS/W compared to a state-of-the-art NVIDIA Tesla V100 GPU . In comparison to popular quantized DNN accelerators, TiM-DNN achieves 55.2X-240X and 160X-291X improvement in TOPS/W and TOPS/mm^2, respectively.

In summary, the dissertation proposes new in-memory computing architectures as well as addresses the need for scalable modeling frameworks and compensation techniques for resistive crossbar based in-memory computing fabrics. Our evaluations show that in-memory computing architectures are promising for realizing modern machine learning and data analytics workloads, and can attain orders-of-magnitude improvement in system-level energy and performance over traditional von Neumann computing systems.

1. INTRODUCTION

In recent years, we have witnessed a surge in modern computing workloads such as data analytics, machine learning, bioinformatics, and graphics. In particular, machine learning workloads such as Deep Neural Networks (DNNs) have gained tremendous popularity due to their ability to achieve super-human accuracy in many cognitive tasks involving video, image, speech, and natural language processing. Consequently, they are being used in several real-world products and services for speech recognition (Apple Siri, Google Assistant, Amazon Alexa), image analysis (Google+ image search, Facebook DeepFace), natural language processing (Google Translate, Facebook DeepText), search engines, recommendation systems, and more [1, 2]. These workloads differ from traditional workloads in their computing characteristics, *i.e.*, they perform simple computations on large amounts of data leading to frequent data transfers and memory accesses. Von Neumann Computing systems with the separate processor and memory subsystems, such as multi-cores and GPUs, have been the mainstay of computing platforms for the past several decades. However, their performance and energy efficiency on data-intensive workloads are limited by the enormous amount of time and energy spent in frequent data transfers between these subsystems. Von Neumann systems also suffer from the known processor-memory data transfer bottlenecks [3], wherein the memory subsystem is much slower as compared to the processor.

The closer integration of compute and memory is a promising approach to alleviate the processor-memory bottleneck and reduce the frequent memory accesses enabling substantial improvement in system performance and energy. This idea has been explored at various levels of the storage hierarchy leading to numerous academic research efforts [4–15] as well as a few commercial solutions [16, 17]. Prior efforts that have explored the closer integration of logic and memory are variedly referred to in the literature as logic-in-memory, computing-in-memory, and processing-in-memory. These efforts may be classified into two categories – moving logic closer to memory, or *near-memory computing* [4–17], and performing computations within memory structures, or *in-memory computing* [18–30]. In-memory computing reduces the number of memory accesses and the amount of data transferred between processor and memory, and exploits the wider internal bandwidth available within memory systems to achieve computing performance and efficiency beyond traditional Von Neumann systems. In this dissertation, we focus on in-memory computing using CMOS and emerging memory technologies to efficiently execute modern computing workloads.

Emerging memories technologies such as spintronic memories, Resistive RAM (ReRAM), and Phase Change Memory (PCM) are promising candidates for future memories due to several desirable attributes such as non-volatility, high density, and near-zero leakage. In particular, spintronic memories such as Spin Transfer Torque Magnetic RAM (STT-MRAM) have garnered significant interest with various prototype demonstrations and early commercial offerings [31–33]. In this dissertation, we explore three exemplary approaches to in-memory computing with emerging memories and a new CMOS-based memory cell that realize Boolean and non-Boolean computations within the memory array. In the first approach, we propose Spin-Transfer Torque Compute-in-Memory (STT-CiM), a design for in-memory computing with STT-MRAM. We exploit the resistive nature of STT-MRAM to realize a range of bit-wise, arithmetic, and vector compute-in-memory operations. We propose architectural enhancements to the processor instruction set and on-chip bus to integrate STT-CiM in larger systems. We also address the challenge of reliable in-memory computing by extending ECC schemes to detect and correct errors during CiM operations.

In the second approach, we explore crossbars with resistive memory elements to realize vector-matrix multiplications, *i.e.*, the primitive compute kernel in machine learning workloads. We highlight a key challenge with resistive crossbars, *i.e.*, the actual computed functions deviate from the desired vector-matrix multiplication operations due to numerous device and circuit-level non-idealities. The cumulative effect of all these non-idealities manifests as errors in the vector-matrix multiplications leading to accuracy degradation in DNNs. We address the need for a fast and accurate simulation framework to enable functional evaluation of large-scale DNNs on resistive crossbars. We propose a Fast Crossbar Model (FCM) to accurately capture the errors arising due to crossbar non-idealities while being four-to-five orders of magnitude faster than circuit simulation. We also develop RxNN, a software framework to evaluate DNN inference on resistive crossbar systems. Our evaluations reveal that the degradation in accuracy due to non-idealities can be significant for large-scale DNNs, motivating a need for cross-layer mitigation and compensation schemes to enable use of resistive crossbar based fabrics. Subsequently, we focus on error mitigation techniques for crossbars systems and propose CxDNN, a hardware-software methodology that enables the realization of large-scale DNNs on crossbar systems with minimal degradation in accuracy by compensating for errors due to non-idealities. CxDNN comprises of (i) an optimized mapping technique to convert floating-point weights and activations to crossbar conductances and input voltages, (ii) a fast re-training method to recover accuracy loss due to this conversion, and (iii) low-overhead compensation hardware to mitigate dynamic and hardware-instance-specific errors. CxDNN presents a scalable compensation methodology that can address large DNNs (e.g., ResNet-50 on ImageNet), and enables a common model to be trained and deployed on many devices.

Finally, in the third approach; we also explore in-memory computing using new CMOS-based memory cells. While emerging memories show considerable promise as future memories due to much higher density and lower leakage, CMOS-based memories are currently (and in near-future may continue to be) the backbone of memory design in computing platforms. We focus on realizing vector-matrix multiplication operations with signed ternary values and propose TiM-DNN, a programmable hardware accelerator that is specifically designed to execute state-of-the-art ternary DNNs. TiM-DNN supports various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) ternary systems, enabling it to execute a broad range of ternary DNNs.

In the following sections, we first present the limitations of traditional Von Neumann computing systems and motivate emerging memory technologies. Next, we provide a brief description of the thesis contribution and the thesis outline.

1.1 The Processor-Memory Bottleneck

In traditional Von Neumann computing systems, the processing and memory units are separate subsystems connected via a system bus or Network-on-Chip (NoC) as shown in Figure 1.1(a). These systems have been the backbone for computing platforms for the past several decades and will continue to be so. However, they are not very efficient in realizing several modern computing workloads such as data analytics and machine learning that perform simple computations on large amounts of data. Realizing these workloads on Von Neumann systems lead to frequent data transfers between the processor and memory subsystem [Figure 1.1(a)], thereby consuming a significant fraction of the overall system energy and time. We can consider the energy and time spent during data transfers undesirable as no useful computations are performed.

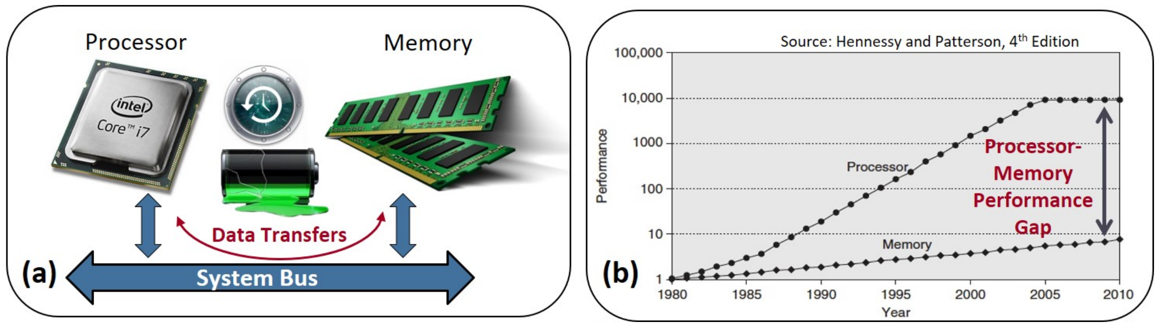


Fig. 1.1.: Von Neumann computing Systems: (a) Separate processor and memory subsystems lead to significant data transfers, (b) Processor-memory bottleneck [3].

Processor-memory gap, *i.e.*, the performance gap between the processor and memory, as shown in Figure 1.1(b) is another source of inefficiency in Von Neumann systems. Modern workloads lead to enormous amounts of slow memory access and perform very simple and little computations on the fetched data, thereby leading to processor-memory bottlenecks that degrade the overall system performance.

1.2 Emerging Memories

The growth in data processed and the increase in the number of cores place high demands on the memory systems of modern computing platforms. Consequently, a growing fraction of transistors, area, and power are utilized towards memories. CMOS memories (SRAM and embedded DRAM) have been the mainstays of memory design for the past several decades. However, recent technology scaling challenges in CMOS memories, along with an increased demand for memory capacity and performance, have fueled an active interest in alternative memory technologies.

Several alternative non-volatile memory (NVM) technologies such as spintronic memory, Phase Change Memory (PCM), and Resistive RAM (ReRAM) have emerged as potential candidates for future memories due to several highly desirable characteristics, *viz.*, high density, ultra-low leakage energy, and non-volatility. Figure 1.2 shows the comparison of different emerging and CMOS memory technologies using access speed, memory capacity, and endurance. ReRAM and PCM offer memory capacity in G-bit range with an access speed of 100-1000ns, thereby limiting them to be an alternative for only off-chip memories. In contrast, spintronic memories such as STT-MRAM and p-STT-MRAM (shown in Figure 1.2) offer both high access speeds (1 ns to 10 ns) and capacity (1 M-bits to 10 G-bits), enabling them to be used as both on-chip and off-chip memories. Moreover, unlike PCM and ReRAM, spintronic memories have practically unlimited endurance adding to their bid for being the next generation memory technology.

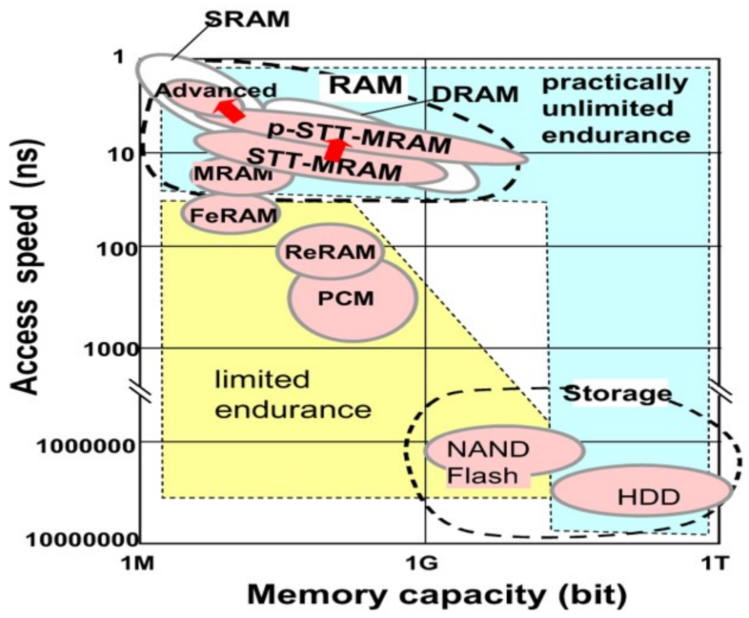


Fig. 1.2.: Comparison of different memory technologies [34].

1.3 Thesis contributions

In this section, we detail the contributions of this dissertation wherein we explore three exemplary approaches to in-memory computing, viz., boolean computing with Spin-Transfer Torque Magnetic RAM, non-boolean computing with resistive crossbars, and non-boolean computing with novel CMOS memory cells.

1.3.1 Boolean Computing with Spin-Transfer Torque Magnetic RAM

Spintronic memories use electron spin as the state variable to represent logic, as opposed to SRAMs and DRAMs that use charge. There are two spin states - Up and Down - enabling 1-bit of storage. Spin Transfer Torque Magnetic RAM (STT-MRAM) is currently the most popular spintronic memory with commercial chips [31–33] already available in the market. In recent years, there have been several research efforts to boost the efficiency of STT-MRAM at the device, circuit, and architectural levels [35–61]. In this dissertation, we explore in-memory computing with STT-MRAM motivated by the observation that the movement of data from bit-cells in the memory to the processor and back (across the bit-lines, memory interface, and system interconnect) is a major performance and energy bottleneck in computing systems. By exploiting the ability to simultaneously enable multiple wordlines within a memory array, we enhance STT-MRAM arrays to perform a range of arithmetic, logic and vector operations. We propose circuit and architectural techniques for reliable in-memory computing under process variations and for enabling the proposed design to be used in a programmable processor-based system. We also address the question of how should STT-CiM be integrated into a general-purpose computing system. To this end, we propose architectural enhancements to processor instruction sets and on-chip buses to enable utilization of STT-CiM as a scratchpad memory. Finally, we present data mapping techniques to increase the effectiveness of STT-CiM. We evaluate STT-CiM using a device-to-architecture modeling framework and integrate cycle-accurate models of STT-CiM with a commercial processor and on-chip bus (Nios II and Avalon from Intel). Our system-level evaluation shows that STT-CiM provides system-level performance improvements of 3.93x on average (upto 10.4x), and concurrently reduces memory system energy by 3.83x on average (upto 12.4x).

1.3.2 Non-Boolean Computing with resistive crossbars

Resistive crossbar based in-memory computing systems have garnered significant interest for realizing DNNs due to their ability to perform the underlying computational kernel, *viz.*, vector-matrix multiplications, efficiently. They may be designed using any of the above mentioned emerging resistive memory devices, including ReRAM, PCM, and Spintronics [62–65]. These devices offer the prospect of highly compact and energy-efficient DNN implementations due to their intrinsic properties. Several research efforts have explored the design of crossbar based systems at the device, circuit, architecture, and algorithmic levels [28–30, 63, 66–86].

A key challenge with resistive crossbars is that the computed function (input voltages multiplied with the weights stored as conductances to obtain output currents) is only an approximation of the desired vector-matrix multiplication. In practice, resistive crossbars suffer from various device and circuit level non-idealities, *viz.*, driver resistance, sensing resistance, sneak paths, interconnect parasitics, process and programming variations, and non-linearities in the peripheral circuits such as ADCs and DACs, which lead to errors in the computed vector-matrix multiplications. These errors can degrade the overall application-level accuracy of a DNN realized on a resistive crossbar system. Although DNNs are resilient to some inaccuracy in their computations [87–90], this resilience is not unlimited. Therefore, it is necessary to evaluate the impact of non-idealities present in imperfect computational fabrics such as resistive crossbars.

In this dissertation, we first evaluate the impact of non-idealities in resistive crossbars to characterize errors due to these non-idealities. We show that the errors show significant data-dependence, *i.e.*, they depend on the weight programmed into, and input applied to, the crossbar, and hardware-instance dependence (due to variations), motivating the need for a detailed crossbar model. Next, we propose a Fast Crossbar Model (FCM) that can accurately capture the impact of crossbar non-idealities. FCM abstracts non-idealities using simple linear algebra operations to achieve orders-of-magnitude faster simulation compared to SPICE. We realize FCM using the well-known BLAS (Basic Linear Algebra Subprograms) library and develop RxNN, a software framework to evaluate DNNs realized on resistive crossbar systems. We use RxNN (which is based on the popular Caffe [91] deep learning framework) to evaluate several large-scale DNNs with millions of neurons and billions of synaptic connections. Our evaluation shows that the accuracy degradation due to crossbar non-idealities can be significant, motivating a need for cross layer error mitigation and compensation schemes.

To address the accuracy degradation due to crossbar non-idealities, we also propose CxDNN, a hardware-software methodology to realize high accuracy DNNs on resistive crossbar systems. CxDNN is composed of a conversion algorithm, a fast re-training method, and a low-overhead hardware compensation scheme. The conversion algorithm takes any given floating-point DNN and converts its weights to conductances and activation values to voltages by determining appropriate scaling factors. Subsequently, CxDNN uses a fast re-training method to recover accuracy loss due to the conversion with very few iterations. We note that our re-training method is hardware instance independent and therefore needs to be performed only once overall (and not once per instance). Finally, CxDNN mitigates data-dependent and hardware-instance-specific errors using low-overhead error compensation circuits that post-process the crossbar outputs. We evaluate CxDNN using 6 state-of-the-art DNNs on the ImageNet dataset. Our results show that CxDNN improves top-1 accuracy by 16%-49% and obtains classification accuracies comparable to a software fixed-point implementation, thereby alleviating non-idealities as an impediment to the use of resistive crossbar based in-memory computing fabrics.

1.3.3 Non-Boolean Computing with CMOS-based memory cells

The use of lower precision to perform computations has emerged as a popular technique to enable complex Deep Neural Networks (DNNs) to be realized on energy-constrained platforms. In the quest for lower precision, studies to date have shown that ternary DNNs, which represent weights and activations by signed ternary values, represent a promising sweet spot, and achieve accuracy close to full-precision networks on complex tasks such as language modeling and image classification. We propose TiM-DNN, a programmable hardware accelerator that is specifically designed to execute state-of-the-art ternary DNNs. TiM-DNN supports various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) ternary systems. TiM-DNN is an in-memory accelerator designed using TiM tiles — specialized memory arrays that perform massively parallel signed vector-matrix multiplications on ternary values per access. TiM tiles are in turn composed of Ternary Processing Cells (TPCs), new bit-cells that function as both ternary storage units and signed scalar multiplication units. We evaluate an implementation of TiM-DNN in 32nm technology using an architectural simulator calibrated with SPICE simulations and RTL synthesis. TiM-DNN achieves a peak performance of 114 TOPs/s, consumes 0.9W power, and occupies 1.96mm^2 chip area, representing a 300X and 388X improvement in TOPS/W and TOPS/mm^2, respectively, compared to a state-of-the-art NVIDIA Tesla V100 GPU. In comparison to popular DNN accelerators, TiM-DNN achieves 55.2X-240X and 160X-291X improvement in TOPS/W and TOPS/mm^2, respectively. We compare TiM-DNN with a well-optimized near-memory accelerator for ternary DNNs across a suite of state-of-the-art DNN benchmarks including both deep convolutional and recurrent neural networks, demonstrating 3.9x-4.7x improvement in system-level energy and 3.2x-4.2x speedup.

1.4 Thesis outline

We have organized the rest of the thesis as follows. Chapter 2 details the related efforts in the area of near- and in-memory computing and also presents the thesis contributions in their context. Chapter 3 provides the necessary background on the spintronic devices, resistive crossbar systems, and deep neural networks (DNNs). Chapter 4 proposes in-memory computing with STT-MRAM. Chapter 5 presents a fast, accurate, and scalable simulation framework to evaluate and re-train large-scale DNNs on resistive crossbar systems. Chapter 6 details hardware-software compensation schemes to mitigate the impact of crossbar non-idealities on DNNs realized using resistive crossbars. Chapter 7 proposes a new CMOS-based in-memory computing design for accelerating ternary DNNs. Finally, chapter 8 concludes the thesis.

2. RELATED WORK

The idea of closer integration of logic and memory has been explored at various levels to alleviate the fundamental processor-memory bottlenecks. There have been numerous research efforts that explore the possibility of near- and in-memory computing using the traditional CMOS memory technologies such as SRAM and DRAM, as well as emerging memory technologies, *viz.*, ReRAM, PCM, and Spintronic. These efforts are variedly referred to in the literature as logic-in-memory, computing-in-memory, and processing-in-memory. They can be broadly classified into two categories, as shown in Figure 2.1. We limit the scope of our discussion to approaches that improve the efficiency of active computation. For example, we do not discuss the embedding of non-volatile memory elements into a logic circuit [92–95] in order to enable the system to shut-down and wakeup efficiently for improved power management. In this chapter, we discuss the research efforts directed towards processing-in-memory using both CMOS and emerging memory technologies.

2.1 Near-memory Computing

Near-memory computing refers to bringing logic or processing units closer to memory. Notwithstanding the closer integration, processing units still remain distinct from memory arrays. Near-memory computing has been explored at various levels of the memory hierarchy [4–8, 10–17]. Intelligent RAM (IRAM) [4] is an early example, which integrated a processor and DRAM in the same chip to improve the bandwidth between them. Embedding simple processing units within each page of the main memory [5] and within secondary storage [6] enables computations to be performed near memory. Application-specific examples of near-memory computation are: (i) memory that can generate interpolated values, enabling the evaluation of

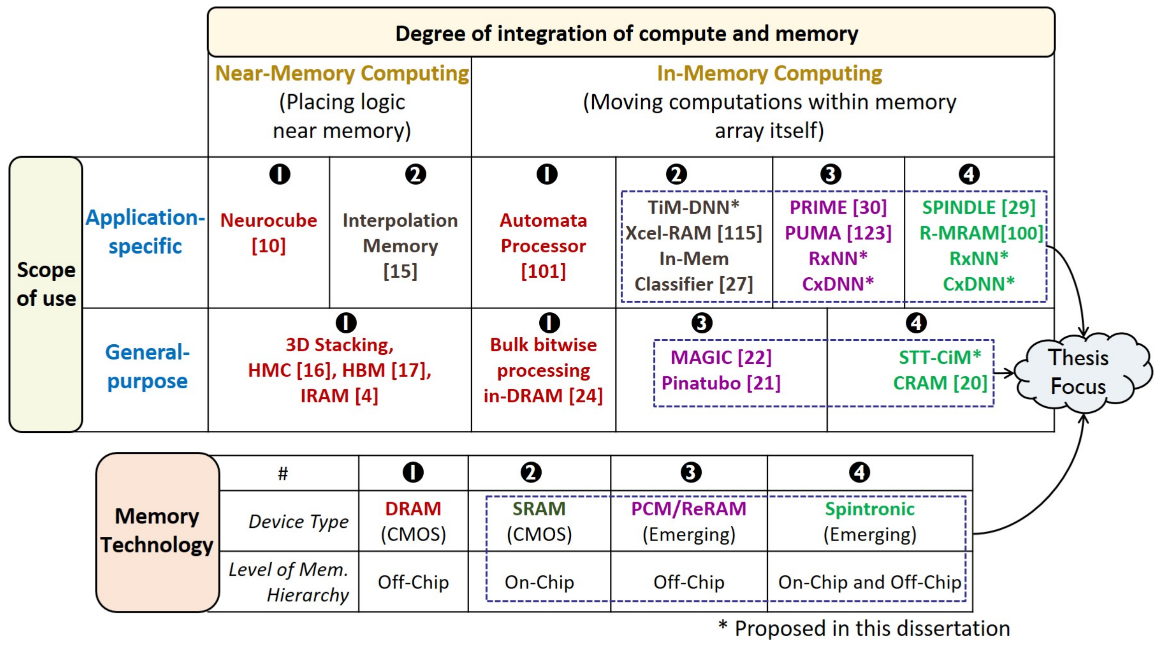

Degree of integration of compute and memory
Near-Memory Computing
(Placing logic near memory)
In-Memory Computing
(Moving computations within memory array itself)
Scope of use
Application-specific
❶ Neurocube [10]
❷ Interpolation Memory [15]
❶ Automata Processor [101]
❷ TiM-DNN* Xcel-RAM [115] In-Mem Classifier [27]
❸ PRIME [30] PUMA [123] RxNN* CxDNN*
❹ SPINDLE [29] R-MRAM[100] RxNN* CxDNN*
General-purpose
❶ 3D Stacking, HMC [16], HBM [17], IRAM [4]
❶ Bulk bitwise processing in-DRAM [24]
❸ MAGIC [22] Pinatubo [21]
❹ STT-CiM* CRAM [20]
Thesis Focus
Memory Technology
#
Device Type
Level of Mem. Hierarchy
❶ DRAM (CMOS) Off-Chip
❷ SRAM (CMOS) On-Chip
❸ PCM/ReRAM (Emerging) Off-Chip
❹ Spintronic (Emerging) On-Chip and Off-Chip
* Proposed in this dissertation

Fig. 2.1.: Taxonomy of computing near/in memory

complex mathematical functions [15], and (ii) Deep Neural Network (DNN) accelerators [10, 96, 97]. Near-memory computing has gained significant interest in recent years, with industry efforts like Hybrid Memory Cube (HMC) [16] and High Bandwidth Memory (HBM) [17].

2.2 In-memory Computing

In-memory computing [18–21, 24–27, 98] integrates logic operations into memory arrays, fundamentally blurring the distinction between processing and memory. In comparison to near-memory computing, a key advantage of in-memory computing is that it enables a much higher level of parallelism by enabling and computing on multiple memory rows simultaneously, thereby achieving superior performance and energy efficiency. We can classify previous proposals for in-memory computing based on whether they target application-specific or general-purpose computations, and based on the underlying memory technology that they consider.

2.2.1 General-purpose in-memory computing

General-purpose examples of in-memory computing include evaluation of bitwise logic operations using emerging memories [21–23, 26, 99] and DRAM [24]. CRAM [20] that introduces an extra transistor (2T-1R) to enable evaluation of complex functions within the array is another example of general-purpose in-memory computing. The key challenge of general-purpose in-memory computing is to realize it without impacting the desirability of the resulting design as a standard memory (*i.e.*, density or efficiency of standard read and write operations). Due to these constraints, general-purpose in-memory computing designs are typically limited to performing a small number of simple operations.

2.2.2 Application-specific in-memory computing

Application-specific examples of in-memory computing include vector-matrix multiplication [27–30] and sum-of-absolute difference [19] computation. Ternary content-addressable memory [18], ROM-embedded RAM [100], AC-DIMM [25] and Micron's automata processor [101] can also be viewed as examples of in-memory computing that target specific operations such as pattern matching or evaluation of transcendental functions. Next, we discuss examples of in-memory computing targeting Deep Neural Network (DNN) applications in more detail.

2.2.3 In-memory computing for Deep Neural Networks (DNNs)

Deep neural networks (DNNs) have greatly advanced the state-of-the-art in machine learning tasks involving video, image, speech and natural language processing, and are currently used in several real-world products and services [1]. The high computation and memory requirements of DNNs often pose challenges to the computing platforms on which they are executed. Currently, GPUs and specialized accelerators like Google's TPU and Microsoft's Brainwave are the mainstays for DNN execution. However, the continuing increase in DNN complexity and amount of data processed has fueled an active interest in new hardware fabrics that can deliver further improvements in efficiency, and thereby enable the virtuous cycle of advances in deep learning algorithms and hardware to continue.

In recent years, several research efforts have focused on improving the energy efficiency and performance of DNNs at various levels of design abstraction [87–89,102–108]. In this subsection, we limit our discussion to efforts on in-memory computing for DNNs [28, 30, 109–117]. Table 2.1 classifies prior in-memory computing efforts based on the memory technology [CMOS and Non-Volatile Memory (NVMs)] and the targeted precision. One group of efforts [28, 30, 109, 110] has focused on in-memory DNN accelerators using emerging Non-Volatile memory (NVM) technology such as PCM and ReRAM. Although NVMs promise density and low leakage relative to CMOS, they still face several open challenges such as errors due to the device and circuit-level non-idealities, which we address in this dissertation. The other group of efforts [111–118] has focused on in-memory DNN accelerators using the mainstay CMOS technology.

,

2.2.4 In-memory computing for DNN applications using CMOS-based memory cells

Efforts on SRAM-based in-memory accelerators can be classified into those that target binary [111–115] and high-precision [116–118] DNNs. Accelerators targeting binary DNNs [111–115] can execute massively parallel vector-matrix multiplication per array access. However, the restriction to binary networks is a significant limitation as binary networks (known to date) incur a large drop in accuracy. Efforts [116–

Table 2.1.: Examples of In-memory Computing for DNN applications

		In-memory Computing for DNN applications	
Weight Precision		CMOS	NVM
Binary		In-Mem Classifier [111], Conv-RAM [112], R Lui. et al. [113], XNOR-SRAM [114], Xcel-RAM[115]	Binary-RRAM[110], XNOR-RRAM[109], **RxNN, CxDNN** [This dissertation]
Ternary	Unweighted (-1,0,1)	**TiM-DNN** [This dissertation]	-
	Weighted (-a,0,b)	**TiM-DNN** [This dissertation]	-
> 2 bits		Neural Cache [116], A Jaiswal et al. [117], Compute-mem[118]	RENO [28], PRIME[30], PUMA [123] **RxNN, CxDNN** [This dissertation]

118] that target higher precision (4-8 bit) DNNs require multiple execution steps (array accesses) to realize signed dot-product operations, wherein both weights and activations are signed numbers. For example, Neural cache [116] computes bitwise Boolean operations in-memory but uses bit-serial near-memory arithmetic to realize multiplications, requiring several array accesses per multiplication operation (and many more to realize dot-products).

2.2.5 In-memory computing for DNN applications using emerging memories

Resistive crossbars have witnessed significant research interest in recent years due to their ability to efficiently realize vector-matrix multiplications, *i.e.*, the primitive machine learning kernel [119–122]. In this subsection, we focus on prior works that target DNNs on resistive crossbar systems. These efforts can be broadly classified into specialized hardware accelerators [28–30, 66–68, 123], non-ideality mitigation schemes [69–78], and design tools for resistive crossbar systems [79–81, 124].

Specialized hardware accelerators. Resistive crossbar based specialized hardware systems have been proposed for accelerating DNN inference [28–30, 68, 125] and training [66, 67] operations. These efforts focus on the evaluation of the proposed architecture using performance, energy, and area as their metrics. They either do not explicitly consider non-idealities or model only the limited-precision aspect of non-idealities.

Non-ideality mitigation schemes. We now discuss prior efforts that attempt to mitigate the effects of crossbar non-idealities. These efforts can be grouped into (i) methods that use (re-)training to compensate for errors [69, 71, 73, 75, 126–128], (ii) mapping to reduce computational errors [70, 74], (iii) closed-loop re-programming methods to overcome imperfect programming and drift in synaptic conductances [72, 76,78], and (iv) ECC schemes to reduce errors due to a specific device non-ideality [129]. The focus of all these efforts is to evaluate and mitigate errors due to crossbar non-idealities. However, they are restricted to simple networks and small datasets since they lack a scalable simulation framework. Many of these efforts also lack a detailed crossbar model as they consider only a subset of crossbar non-idealities.

Training-based compensation [69,71,73,75,126] using software models of the crossbar is the most popular approach to compensating for non-idealities. However, re-training can only mitigate accuracy degradation to a limited extent. Further, it ties the network to the specific hardware instance and therefore is not scalable. For example, to create re-trained models for crossbar-based inference accelerators deployed in millions of IoT devices will correspondingly require millions of training runs. Another limitation of these efforts [69, 71, 73, 75, 126] is that they correct errors due to only a subset of crossbar non-idealities. For example, [126] can only mitigate errors due to driver resistance, source resistance, and chip-to-chip variations. It ignores other prominent non-idealities such as wire resistance, sneak paths, non-linear ADCs and DACs, and intra-chip process variations (each crossbar or synaptic device within a chip is uniquely impacted by process-variations). Finally, all (re-)training based prior efforts [69–73, 75, 76, 78, 126] are limited to small scale networks and data sets.

ECC scheme using AN codes have also been proposed to correct errors in resistive crossbars [129]. However, AN codes [130] can correct errors only in linear functions realized using digital systems, wherein errors are discrete (at the bit-level) and can be quantified using the digital number representation (multiple bits). Although vector-matrix multiplications realized using digital systems are linear, the functions executed in resistive crossbars are not. For example, vector-matrix multiplications in resistive crossbars consist of several non-linear operations (viz., non-linear DACs and ADCs functions). The errors in resistive crossbars due to the cumulative effect of all non-idealities are analog and non-linear in nature. Therefore, ECC schemes cannot effectively mitigate errors in analog systems (experimental results shown in Section 6.3.5). Further, apart from assuming crossbar operations to be linear, another limitation of the prior effort [129] is that it only models errors due to a specific device non-ideality (random telegraph noise) and ignores circuit-level non-idealities, viz., wire resistances, sensing resistances, sneak paths, driver resistances, and non-linear ADCs and DACs.

Design tools. To aid design space exploration, prior efforts [79–81, 124] have proposed circuit-level macro models to evaluate crossbar systems. These efforts include (i) MNSIM [79], a simulation platform to evaluate inference accelerators designed using resistive crossbars, (ii) NeuroSim [124], a framework to evaluate crossbars systems designed for on-chip training, (iii) technological exploration tool to optimize resistive crossbar design space [80], and (iv) AutoNCS [81], a tool to optimize the utilization and efficiency of a resistive crossbar system. The primary focus of all these tools has been the performance, energy, and area evaluation of resistive crossbar systems to facilitate design space exploration. These tools also have simplistic accuracy/error models that are reasonable for design space exploration, but inadequate for evaluating application-level accuracy of DNNs.

2.3 Thesis contributions

The primary contributions of this dissertation are different or complementary to the prior efforts in the following aspects:

Boolean Computing with Spin-Transfer Torque Magnetic RAM. Our work differs from application-specific in- and near-memory computing designs as we focus on embedding a broader set of operations (arithmetic, logic and vector operations) within memory. Our work also differs from efforts [21–23] that realize bitwise logic operations in several important aspects. First, we focus on in-memory computing for spintronic memory, which involves fundamentally different prospects and design challenges. For example, the proposed operations are not destructive to the contents stored in the accessed bit-cells (unlike [24]). On the other hand, the much lower ratio of on to off resistance in spintronic memory leads to lower sensing margins. Second, we use a different sensing and reference generation circuitry, which enables us to natively realize a wider variety of operations. For example, the proposed design requires only one array access (unlike two in the case of [21]) to perform bit-wise XOR operations. Third, our design goes beyond bitwise logic operations and realizes arithmetic as well as complex vector operations. Fourth, we propose architectural extensions (bus and ISA extensions) and data mapping techniques to enable in-memory computing within a general-purpose processor system. Finally, we address a key challenge associated with in-memory computing, *viz.*, reliable operation under process variations. In contrast to the effort that uses 2T-1R STT-MRAM bit-cells [20], our proposal enables in-memory computation within a standard STT-MRAM array with no changes to the bit-cells. Moreover, unlike [20, 26] we do not restrict ourselves to device and circuit level considerations and also address the architectural challenges of in-memory computing.

Non-Boolean Computing: Framework for Evaluating Deep Neural Networks on Resistive Crossbars. Our work complements the prior efforts on hardware accelerators [28–30, 66–68] and non-ideality mitigation schemes [69–78]. We focus on the functional evaluation of large-scale DNNs, including winners from previous ImageNet challenges [131]. We address the need for a fast, scalable, and accurate framework for resistive crossbar systems to enable evaluation and re-training of large-scale DNNs. Moreover, unlike simple error estimation model [79, 80], we focus on a more accurate model that captures the cumulative effects of all non-idealities including data-dependent errors, *i.e.*, the errors that depend on the weights programmed into, and inputs applied to the crossbar.

Non-Boolean Computing: Hardware-Software Compensation methods for Deep Neural Networks on Resistive Crossbar Systems. CxDNN complements and improves upon previous efforts by proposing a hardware-software methodology that scales to large DNNs, avoids hardware-instance-specific (re-)training, can correct errors due to a wider range of crossbar non-idealities (including both device and circuit-level non-idealities), and uses low-overhead compensation circuits to better alleviate dynamic and hardware instance specific errors. Other prior efforts [72,76,78] using closed-loop programming may be combined with CxDNN to specifically address programming errors.

Non-Boolean Computing with CMOS: Ternary in Memory accelerator for Deep Neural Networks. In contrast to previous proposals, TiM-DNN is the first specialized and programmable in-memory accelerator for ternary DNNs that supports various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) ternary systems. TiM-DNN utilizes a new CMOS-based bit-cell (i.e., TPC) and enables multiple memory rows simultaneously to realize massively parallel in-memory signed vector-matrix multiplications with ternary values per memory access, enabling efficient realization of ternary DNNs. As illustrated in our experimental evaluation, TiM-DNN achieves 3.9x-4.7x improvement in system-level energy and 3.2x-4.2x speedup over a well-optimized near-memory accelerator. In comparison to the near-memory ternary accelerator [97], it achieves 55.2X improvement in TOPS/W.

3. BACKGROUND

In this chapter, we provide the necessary background information on the emerging memory technologies, the resistive crossbar system, and DNNs.

3.1 STT-MRAM

This section presents the STT-MRAM bit-cell structure and describe the basic read and write operations.

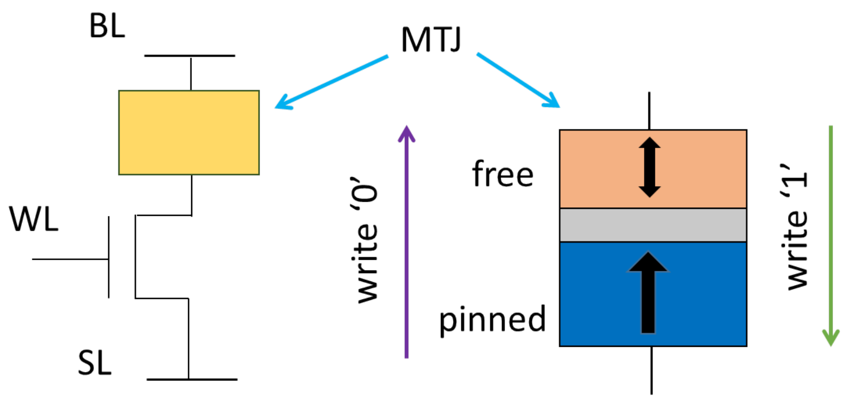


Fig. 3.1.: STT-MRAM bit-cell

An STT-MRAM bit-cell consists of an access transistor and a magnetic tunnel junction (MTJ), as shown in Figure 3.1. An MTJ consists of a pinned layer that has a fixed magnetic orientation and a free layer whose magnetic orientation can be switched. The magnetic layers are separated by a tunneling oxide. The relative magnetic orientation of the free and pinned layers determines the resistance offered by the MTJ (the resistance for the parallel configuration, R_P, is lower than the anti-parallel resistance, R_{AP}). The two resistance states encode a bit (we assume that parallel represents the logic "1", and anti-parallel represents the logic "0"). A read operation is performed by applying a bias (V_{read}) between the bitline (BL) and the source line (SL) and enabling the wordline (WL). The resultant current flowing through the bit-cell (I_P or I_{AP}) is compared against a global reference to determine the logic state stored in the bit-cell.

A write is performed by passing a current greater than the critical switching current of the MTJ through the bit-cell. The logic value written, is dependent on the direction of the write current, as shown in Figure 3.1. The write operation in STT-MRAM is stochastic in nature, and the duration and magnitude of the write current determine the write failure rate. Apart from write failures, STT-MRAMs are also subject to read decision failures, where the value stored in a bit-cell is incorrectly sensed due to process variations, and read disturb failures where a read operation inadvertently ends up writing into the bit-cell. These failures are addressed through a range of techniques including device and circuit optimization, manufacturing test and self-repair, and error correcting codes [43, 44, 46, 47]. Apart from write/read failures, memories may also have failures due to thermal noise, which causes stochastic flipping in the bit-cells. However, such failures are negligible in STT-MRAM due to the high energy barrier between the two resistance states.

3.2 Spintronic synaptic device

Next, we provide a brief overview of the spintronic synaptic device [64] used in our experiments.

Spintronic devices [64] have emerged as promising candidates for designing synaptic devices. They have several key advantages over 2-terminal memristive devices [62], *viz.*, low-voltage write operations, high endurance, immunity to drift (*i.e.*, change in synaptic conductance during the read operations [62]), and no read-write conflicts due to separate read and write paths. Figure 3.2 shows a spintronic device [64] that can be used as a building block for resistive crossbar arrays. It is a 3-terminal device consisting of a Magnetic Tunneling Junction (MTJ) and an underlying Heavy Metal (HM)

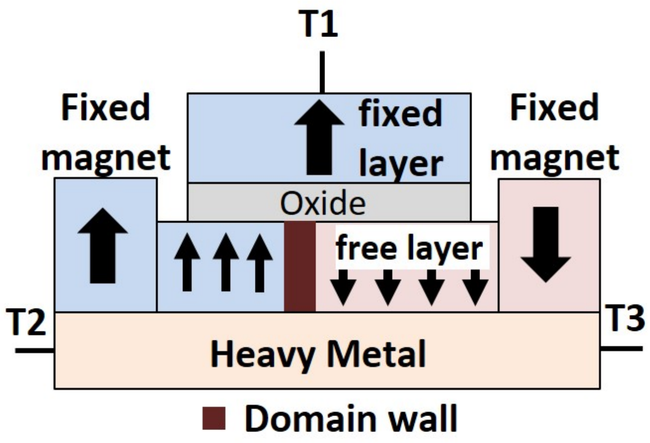


Fig. 3.2.: Spintronic synaptic device

layer. An MTJ is composed of a fixed Ferro-Magnetic (FM) layer, a tunneling oxide, and a free FM layer. A write operation is performed by applying a voltage at the two ends of the HM layer, *i.e.*, between terminals T_2 and T_3. The resulting current flowing through the HM exerts a spin orbit torque on the free layer that causes domain wall motion. The position of the domain wall in the free layer determines the conductance of the MTJ that lies between G_{MIN} (when the domain wall is to the far right) and G_{MAX} (when the domain wall is to the far left). Moreover, the number of unique locations at which the domain wall can reside determines the precision of the device. A read operation in this device is performed by applying a voltage across terminals T_1 and T_3, and the resultant current is proportional to the MTJ's conductance. In summary, the device shown in Figure 3.2 operates as a programmable resistor wherein the resistance is programmed through the HM layer and sensed via the MTJ.

3.3 Resistive crossbar system

In this section, we provide a brief background on resistive crossbar based accelerator and vector-matrix multiplications executed using resistive crossbar arrays.

Figure 3.3 (left) depicts a representative resistive crossbar based accelerator, *viz.*, Resistive crossbar processing unit (XPU), connected to a host processor and main memory via a system bus. The host processor off-loads execution of compute kernels having vector-matrix multiplications (*e.g.*, convolution and fully-connected layers in DNNs) to the XPU, which in-turn realizes them using multiple Crossbar processing tiles (XPTs) and a scheduler. XPTs are composed of Resistive Crossbar Arrays (RCAs) that store weights and compute vector-matrix multiplications, local memories in the form of input and output buffers to store activations, a local bus, a controller to orchestrate various operations, and special function units (SFUs) to execute other DNN operations such as ReLU, max pooling, etc. Next, we describe RCAs, the fundamental compute units of resistive crossbar systems, in detail.

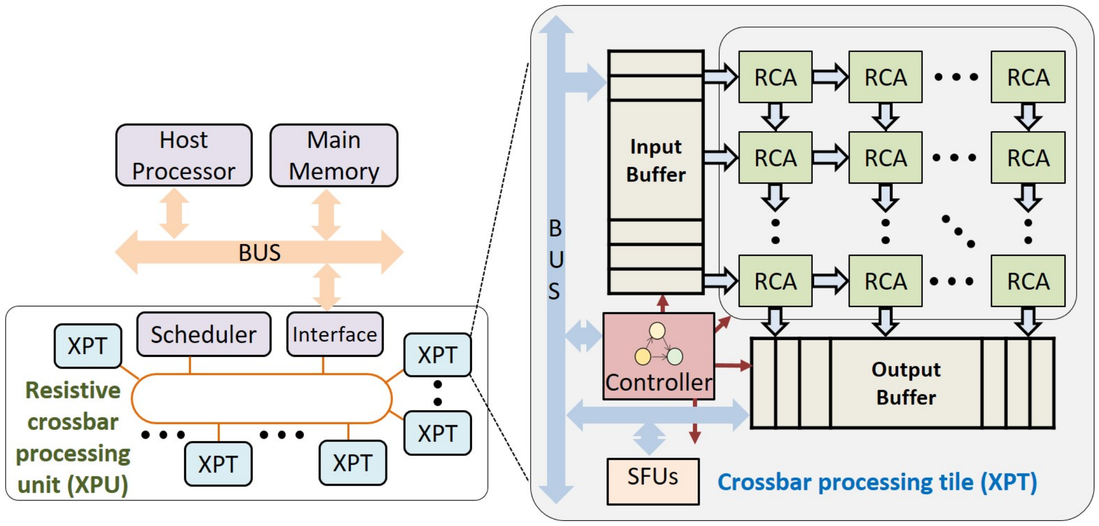


Fig. 3.3.: Architecture for resistive crossbar system

As shown in Figure 3.4, a resistive crossbar array comprises of a 2D array of synaptic elements, Analog-to-Digital Converters (ADCs), Digital-to-Analog Converters (DACs), registers, a decoder, a column MUX, a reduce unit and write circuits. The synaptic elements are programmable resistors that store DNN weights as conductances. The number of available conductance levels determines the precision of the synaptic element. They can be realized using emerging NVMs technologies, *viz.*,

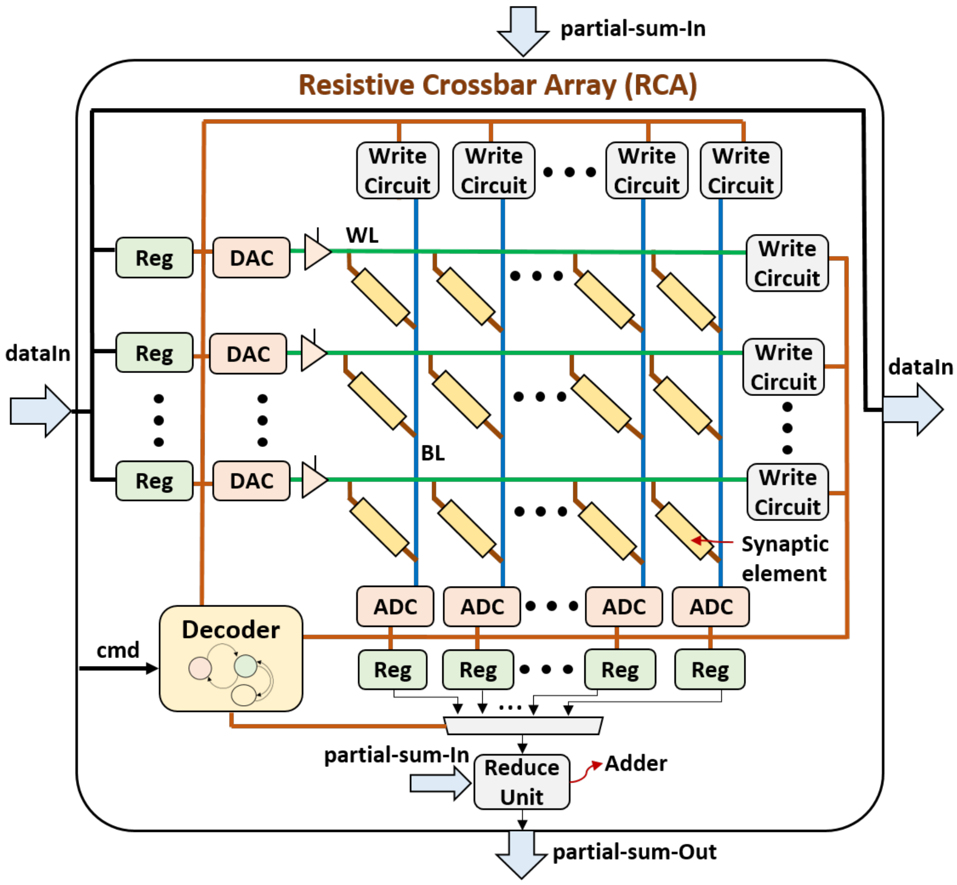


Fig. 3.4.: Resistive crossbar array

PCM, ReRAM, and spintronics [64,120,121]. RCA supports two main operations: (i) vector-matrix multiplication and (ii) programming. A vector-matrix multiplication is performed by driving all wordlines (WL) to analog voltages using DACs and sensing the resultant currents flowing through the columns (BL) using ADCs. The vector-matrix multiplication ideally realized in an MxN RCA is shown in Equation 3.1, where Vin_{ideal} represents a 1xM vector of input voltages, G_{ideal} is an MxN matrix comprising of the programmed conductances, and $Iout_{ideal}$ denotes a 1xN vector of output currents. In contrast, the programming operations, *i.e.*, write operations on synaptic elements, are performed row-wise, wherein, the write circuitry applies the necessary currents and sets them to the desired conductance.

$$\mathbf{Iout_{ideal}} = \mathbf{Vin_{ideal}} * \mathbf{G_{ideal}} \tag{3.1}$$

An XPT realizes vector-matrix multiplications involving a large weight matrix by partitioning it into fragments and mapping them to multiple RCAs. The partitions can have dependencies, *i.e.*, they can either share inputs or outputs (produce partial sums for the same output). We map partitions sharing the inputs (outputs) to same rows (columns). RCAs also have additional logic units (*e.g.*, reduce unit) to handle partial sums and to support input sharing across rows.

3.4 Deep Neural Network (DNN)

Finally, we present a very brief overview of Deep Neural Networks (DNNs).
DNNs are multi-layered neural networks comprising of the basic compute units called artificial neurons. The layers are predominantly of Convolutional (*Conv*) and Fully-Connected (*FC*) types, and these layers are associated with a set of weights. The computations in CONV and FC layers can be expressed as block 2D-convolution (matrix-matrix multiplication) and vector-matrix multiplication, respectively. Further, CONV and FC layers consume 90%-95% of the overall computation time and energy in DNNs. DNNs operate in 2 key phases: (i) Training and (ii) Inference. Training is done on a labeled dataset, wherein the weights of the DNN are iteratively refined using the Stochastic Gradient Descent (SGD) algorithm. In contrast, the inference is performed using a pre-trained model to classify unseen inputs. In this thesis, we focus on improving the energy efficiency of DNN inference using in-memory computing.

4. BOOLEAN COMPUTING WITH SPIN-TRANSFER TORQUE MAGNETIC RAM

In this chapter, we propose our first approach to in-memory computing with emerging memory technologies. As discussed in chapter 1, in-memory computing is motivated by the observation that the movement of data from bit-cells in the memory to the processor and back is a major performance and energy bottleneck in computing systems. In-memory computing is a promising approach that reduces the number of memory accesses and the amount of data transferred between processor and memory. It also exploits the wider internal bandwidth available within memory systems to enable substantial system-level performance and energy benefits for modern computing workloads.

Our proposal is based on the observation that by enabling multiple wordlines simultaneously [1] and sensing the effective resistance of each bit-line, it is possible to directly compute logic functions of the values stored in the bit-cells. Based on this insight, we propose STT-CiM, a design for in-memory computing with STT-MRAM that can perform a range of arithmetic, logic, and vector operations. In STT-CiM, the core data array is the same as standard STT-MRAM; hence, memory density and the efficiency of read and write operations are maintained. Reliable sensing under the limited tunneling magneto-resistance (TMR) of STT-MRAM bit-cells is known to be a challenge [43–47, 60], and we show that challenge this is further aggravated for in-memory computations. In order to enhance the robustness of STT-CiM under process variations, we extend error correction codes (ECC) to errors that occur during in-memory computations. To evaluate the benefits of STT-CiM, we utilize it as a scratchpad in the memory hierarchy of the Intel Nios II [132] processor. We propose

[1]Note that this is much easier in STT-MRAM than in CMOS memories, due to the resistive nature of the bit-cells.

enhancements to the on-chip bus and extend the instruction set of the processor to support compute-in-memory operations and expose them to software. We also present suitable data mapping techniques to maximize the benefits of STT-CiM.

We note that earlier efforts (*e.g.*, [21]) have proposed enabling multiple wordlines to perform computations within Non-Volatile Memories (NVMs). Although our work shares this principle, we differ from previous work in several key aspects: (i) we address reliable in-memory computing under process variations, (ii) we go beyond bitwise logic operations to also perform arithmetic and vector operations, which are commonly present in modern computing workloads, and (iii) we propose architectural enhancements (bus and ISA extensions), and data mapping techniques to enable in-memory computation in the context of on-chip scratchpad memories.

In summary, the key contributions of this work are as follows:

- We explore compute-in-memory with spintronic memories as an approach to improving system performance and energy.
- We propose STT-CiM, an enhanced STT-MRAM array that can perform a range of arithmetic, logic and vector compute-in-memory operations without modifying either the bit-cells or the core data array.
- We address a key challenge in STT-CiM, *i.e.* reliably performing in-memory operations under process variation, by demonstrating suitable error correction mechanisms.
- We propose extensions to the instruction set and on-chip bus to integrate STT-CiM into a programmable processor system and demonstrate the viability of these extensions using Intel's Nios II processor and Avalon on-chip bus.
- We evaluate the performance and energy benefits of STT-CiM, achieving average improvements of 3.83x (upto 12.4x) and 3.93x (upto 10.4x) in the total memory energy and system performance, respectively.

The rest of the chapter is organized as follows. Section 4.1 describes the STT-CiM design and how it supports in-memory computation. Section 4.2 outlines architectural enhancements for STT-CiM. Section 4.3 describes the experimental methodology and experimental results are presented in section 4.4. Section 4.5 concludes the chapter.

4.1 STT-MRAM based Compute-in-Memory (STT-CiM)

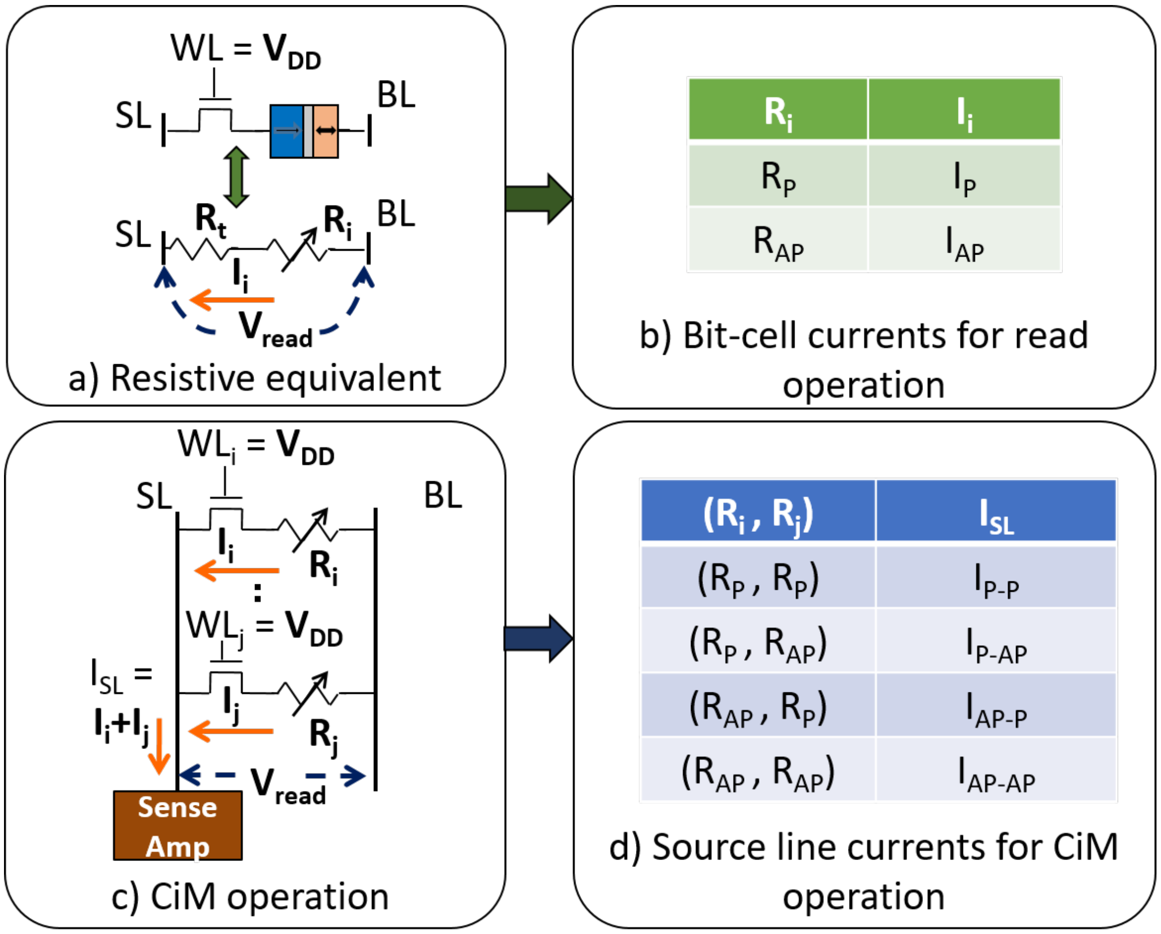
In this section, we describe STT-MRAM based Compute-in-Memory (STT-CiM), a design for in-memory computing using standard STT-MRAM arrays.

4.1.1 STT-CiM overview

The key idea behind STT-CiM is to enable multiple wordlines simultaneously in an STT-MRAM array, leading to multiple bit-cells being connected to each bitline. With enhancements that we propose to the sensing and reference generation circuitry, we can directly compute logic functions of the enabled words. Note that such an operation is feasible in STT-MRAMs since the bit-cells are resistive, and since the write currents are typically much higher than read currents. In contrast, enabling multiple wordlines in SRAM can lead to short-circuit paths through the memory array, leading to loss of data stored in the bit-cells.

Figure 4.1 explains the principle of operation of STT-CiM. First, consider the resistive equivalent circuit of a single STT-MRAM bit-cell shown in Figure 4.1(a). R_t represents the on-resistance of the access transistor and R_i the resistance of the MTJ. When a voltage V_{read} is applied between the bitline (BL) and the source line (SL), the net current I_i flowing through the bit-cell can take two possible values depending on the MTJ configuration, as shown in Figure 4.1(b). A read operation involves using a sensing mechanism to distinguish between these two current values.

Figure 4.1(c) demonstrates a Compute-in-Memory (CiM) operation, where two wordlines (WL_i and WL_j) are enabled, and a voltage bias (V_{read}) is applied to the bitline. The resultant current flowing through the SL (denoted I_{SL}) is a summation of



R_i	I_i
R_P	I_P
R_{AP}	I_{AP}

(R_i, R_j)	I_{SL}
(R_P, R_P)	I_{P-P}
(R_P, R_{AP})	I_{P-AP}
(R_{AP}, R_P)	I_{AP-P}
(R_{AP}, R_{AP})	I_{AP-AP}

Fig. 4.1.: STT-CiM: Principle of operation

the currents flowing through each of the bit-cells (I_i and I_j), which in turn depends on the logic states stored in these bit-cells. The possible values of I_{SL} are shown in Figure 4.1(d). We propose enhanced sensing mechanisms to distinguish between these values and thereby compute logic functions of the values stored in the enabled bit-cells. We discuss the details of these operations in turn below.

Bitwise OR (NOR). In order to realize logic OR and NOR operations, we use the sensing scheme shown in Figure 4.2(a), where I_{SL} is connected to the positive input of the sense amplifier and a reference current I_{ref-or} is fed to its negative input. We choose I_{ref-or} to be between I_{AP-AP} and I_{AP-P}, as shown in Figure 4.2(c). As a result, among the possible values of I_{SL} [Figure 4.1(d)], only $I_{SL} = I_{AP-AP}$ is less than

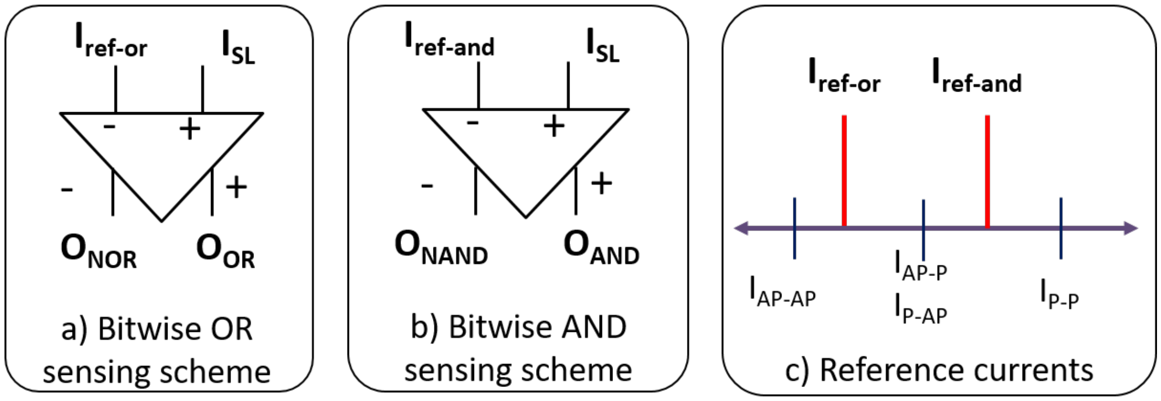


Fig. 4.2.: STT-CiM sensing schemes

I_{ref-or}. Consequently, only the case where both bit-cells are in the AP configuration, *i.e.*, both store "0", leads to an output of logic "0" ("1") at the positive (negative) output of the sense amplifier, while all other cases lead to logic "1" ("0"). Thus, the positive and negative outputs of the sense amplifier evaluate the logic OR and NOR of the values stored in the enabled bit-cells.

Bitwise AND (NAND). A bitwise AND (NAND) operation is realized at the positive (negative) terminal of the sense amplifier by using the sensing scheme shown in Figure 4.2(b). Note that in this scheme, a different reference current ($I_{ref-and}$) is fed to the sense amplifier.

Bitwise XOR. A bitwise XOR operation is realized when the two sensing schemes shown in Figure 4.2 are used in tandem, and O_{AND} and O_{NOR} are fed to a CMOS NOR gate. In other words, $O_{XOR} = O_{AND}$ NOR O_{NOR}.

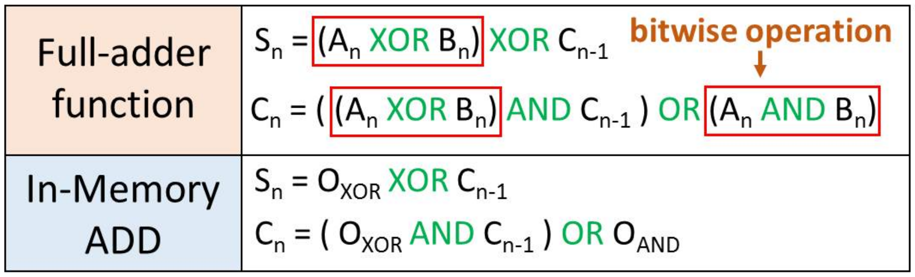
Table 4.1 summarizes the logic operations achieved using the two sensing schemes discussed above. Note that, all the logic operations described above are symmetric in nature, and hence it is not necessary to distinguish between the cases where the two bit-cells connected to a bitline store "10" and "01".

ADD Operation. An ADD operation is realized by leveraging the ability to concurrently perform multiple bitwise logical operations, as illustrated in Figure 4.3.

Table 4.1.: Possible outputs of various sensing schemes in STT-CiM

I_{SL}	O_{OR}	O_{NOR}	O_{AND}	O_{NAND}	O_{XOR}
I_{AP-AP}	0	1	0	1	0
I_{AP-P}	1	0	0	1	1
I_{P-AP}	1	0	0	1	1
I_{P-P}	1	0	1	0	0

Suppose A_n and B_n (the n-th bits of two words, A and B) are stored in two different bit-cells of the same column within an STT-CiM array. Suppose that we wish to compute the full-adder logic function (n-th stage of an adder that adds words A and B). As shown in Figure 4.3, S_n (the sum) and C_n (the carry out) can be computed using A_n XOR B_n and A_n AND B_n, in addition to C_{n-1} (carry input from the previous stage). Figure 4.3 also expresses the ADD operation in terms of the outputs of bitwise operations, O_{AND} and O_{XOR}. Three additional logic gates are required to enable this computation. Note that the sensing schemes discussed enable us to perform the bitwise XOR and AND operations simultaneously, thereby performing an ADD operation with a *single array access*.



Full-adder function	$S_n = \boxed{(A_n \text{ XOR } B_n)} \text{ XOR } C_{n-1}$ $C_n = (\boxed{(A_n \text{ XOR } B_n)} \text{ AND } C_{n-1}) \text{ OR } \boxed{(A_n \text{ AND } B_n)}$
In-Memory ADD	$S_n = O_{XOR} \text{ XOR } C_{n-1}$ $C_n = (O_{XOR} \text{ AND } C_{n-1}) \text{ OR } O_{AND}$

bitwise operation

Fig. 4.3.: In-Memory ADD operation using STT-CiM

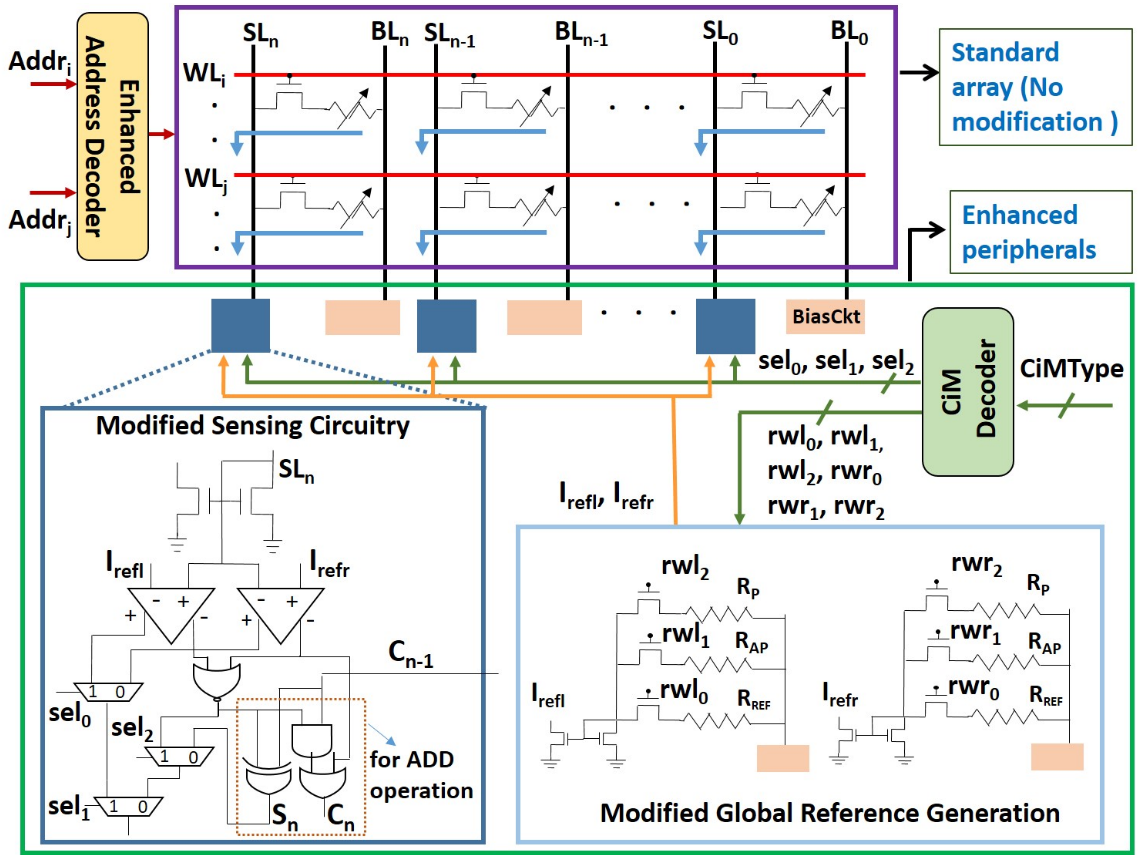


Fig. 4.4.: STT-CiM array structure

4.1.2 STT-CiM array

In this section, we present the array-level design of STT-CiM using the circuit-level techniques described above. As shown in Figure 4.4, the proposed STT-CiM memory array takes an additional input CiMType that indicates the type of compute-in-memory operation that needs to be performed for every memory access. The CiM decoder interprets this input and generates appropriate control signals to perform the desired logic operation. In order to enable compute-in-memory operations, the read peripheral circuits present in each column (sensing circuit and global reference generation circuit in Figure 4.4) are enhanced, while the core data array remains the same as in standard STT-MRAM. The address (row) decoder needs to enable multiple wordlines for CiM operations. Specifically, we utilize two address decoders, with each decoding the corresponding input address. The corresponding outputs of the decoders are OR-ed and connected to each wordline. This configuration allows any of the two decoders to activate random WL locations. While the row decoder overhead is roughly doubled, it represents a small fraction of total area and power for configurations involving large arrays (1.8% in our evaluation). The write peripheral circuits are unchanged, as write operations are identical to standard STT-MRAM. We next describe enhancements to sensing and reference generation circuits to enable CiM operations.

Sensing circuitry. Figure 4.4 shows the sensing circuit enhanced to support all the logic operations discussed in Section 4.1.1. It consists of two sense amplifiers, a CMOS NOR gate, three multiplexers and three additional logic gates for the ADD operation. We note that the area and power overheads associated with these enhancements are minimal since the sensing circuit constitutes a small fraction of the total memory area/power. As shown in the figure, the reference currents (I_{refl}, I_{refr}) produced by the global reference generation circuit are fed to the two sense amplifiers in order to realize the sensing schemes discussed in Section 4.1.1. The three MUX control signals (sel_0, sel_1, sel_2) are generated by the CiM decoder to select the desired compute-in-memory operation.

Reference generation. Figure 4.4 illustrates the modified reference generation circuit used to produce the additional reference currents necessary for the proposed sensing schemes. It includes two reference stacks, one for each of the two sense amplifiers in the sensing circuit. Each stack consists of three bit-cells programmed to offer resistances R_P, R_{AP} and R_{REF}, respectively. R_{REF} [2] represents the fixed resistance reference MTJ used in a standard STT-MRAM to perform read operations. The CiM decoder generates control signals (rwl_0, rwl_1, ... rwr_1, rwr_2) that enable a subset of these bit-cells in the reference stacks, which in turn produces the desired reference currents. Table 4.2 presents the values of these control signals so as to achieve the required reference currents.

[2] $R_{AP} > R_{REF} > R_P$

Table 4.2.: STT-CIM operations control signals

Operation	rwl_0	rwl_1	rwl_2	rwr_0	rwr_1	rwr_2	sel_0	sel_1	sel_2
READ	1	0	0	0	0	0	1	1	x
NOT	0	0	0	1	0	0	0	1	x
AND	1	0	1	0	0	0	1	1	x
OR	1	1	0	0	0	0	1	1	x
NAND	0	0	0	1	0	1	0	1	x
NOR	0	0	0	1	1	0	0	1	x
XOR	1	1	0	1	0	1	0	0	1
ADD	1	1	0	1	0	1	0	0	0

The STT-CiM array can perform both regular memory operations and a range of CiM operations. The normal read operation is performed by enabling a single wordline and setting sel_0, sel_1, and rwl_0 to logic '1'. On the other hand, a CiM operation is performed by enabling two wordlines and setting CiMType to the appropriate value, which results in computing the desired function of the enabled words. The control signal values for a read operation as well as CiM operations are shown in Table 4.2.

4.1.3 CiM operation under process-variations

The STT-CiM array suffers from the same failure mechanisms (read disturb failures, read decision failures and write failures) that are observed in standard STT-MRAM. In this section, we compare the failure rates in STT-CiM and standard STT-MRAM. Normal read/write operations in STT-CiM have the same failure rate as in a standard STT-MRAM, since the read/write mechanisms are identical. However, CiM operations differ in their failure rates, since the currents that flow through each bit-cell differ when enabling two wordlines simultaneously. In order to analyze the read disturb and read decision failures under process variations for CiM operations, we performed a Monte-carlo circuit-level simulation on 1 million samples considering variations in MTJ oxide thickness ($\sigma/\mu = 2$ %), transistor V_T ($\sigma/\mu = 5\%$), and MTJ cross sectional area ($\sigma/\mu = 5\%$) [43]. Figure 4.5 shows the probability density distribution of the possible currents obtained during read and CiM operations on these 1 million samples.

CiM disturb failures. As shown in Figure 4.5, the overall current flowing through the source line is slightly higher in case of a CiM operation as compared to a normal read. However this increased current is divided between the two parallel paths, and consequently the net read current flowing through each bit-cell (MTJ) is reduced. Hence, the read disturb failure rate is lower for CiM operations than for normal read operations.

CiM decision failures. The net current flowing through the source line (I_{SL}) in case of a CiM operation can have 3 possible values, *i.e.*, I_{P-P}, I_{AP-P} (I_{P-AP}), I_{AP-AP}. A read decision failure occurs during a CiM operation when the current I_{P-P} is interpreted as I_{AP-P} (or vice versa), or when I_{AP-AP} is inferred as I_{AP-P} (or vice versa). In contrast to normal reads, CiM operations have two read margins — one between I_{P-P} and I_{AP-P} and another between I_{AP-P} and I_{AP-AP} [Figure 4.5(b)]. Our simulation results show that the read margins for CiM operations are lower as compared to normal reads, therefore they are more prone to decision failures. Moreover, the read margins in CiM operations are unequal [3]. Thus, we have more failures arising due to the read margin between I_{P-P} and I_{AP-P}.

ECC for STT-CiM. In order to mitigate these failures in STT-MRAM, various ECC schemes have been previously explored [43–45]. We show that ECC techniques that provide single error correction and double error detection (SECDED) and double error correction and triple error detection (DECTED) can be used to address the decision failures in CiM operations as well. This is feasible because the codeword properties for most ECC codes are retained for a CiM XOR operation. Figure 4.6 shows the

[3]Although resistances may be equally separated, the currents are not since they depend inversely on resistance.

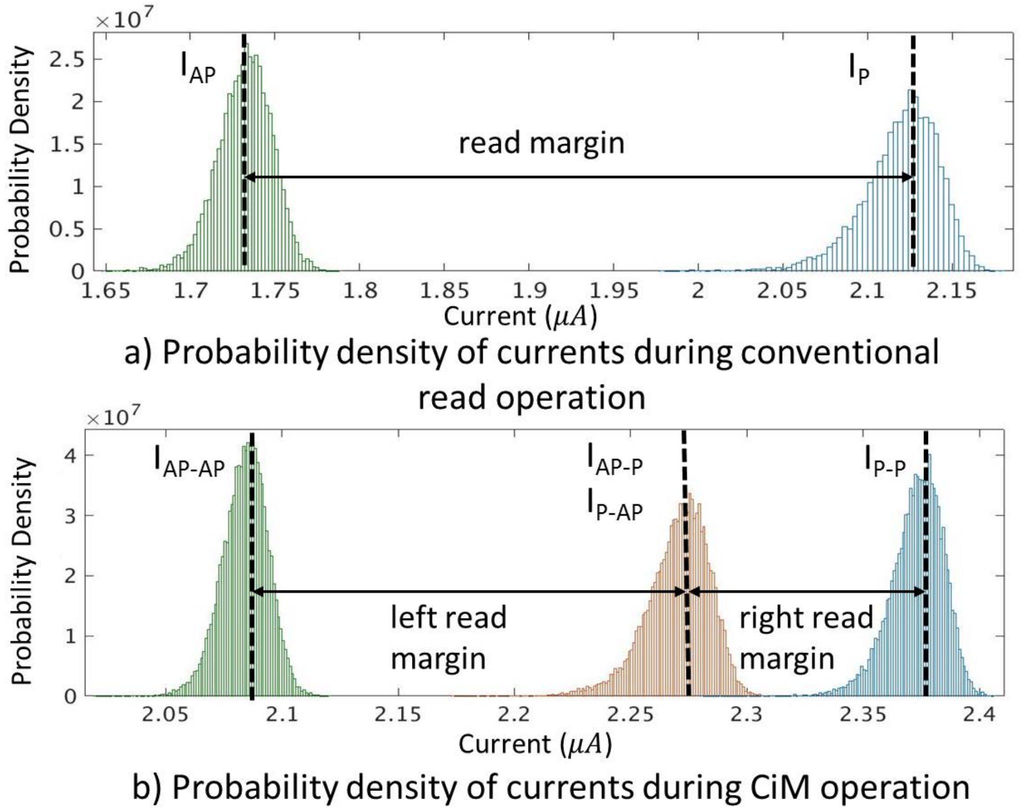


Fig. 4.5.: Probability density distribution of I_{SL} under process variations during read and CiM operations in STT-CiM

codeword retention property of a CiM XOR operation using a simple Hamming code. As shown in the figure, $word_1$ and $word_2$ are augmented with ECC bits (p_1, p_2, p_3) and stored in memory as $InMemW_1$ and $InMemW_2$ respectively. A CiM XOR operation performed on these stored words ($InMemW_1$, $InMemW_2$) results in the ECC codeword for $word_1$ XOR $word_2$, therefore the codewords are preserved for CiM XORs. We leverage this codeword retention property of CiM XORs to detect and correct errors in *all CiM operations*. This is enabled by the fact that STT-CiM always computes bitwise XOR (CiM XOR) irrespective of the CiM operation that is being performed.

We demonstrate the proposed error detection and correction mechanism for CiM operations in Figure 4.7. Let us assume that data bit d_1 suffers from a decision failure during CiM operations, as shown in the figure. As a result, the combination of logic 1 and 1 in the two bit-cells (I_{P-P}) is inferred as logic 1 and 0 (I_{AP-P}), leading to erroneous CiM outputs. An error detection logic operating on the CiM XOR output (Figure 4.7) detects an error in the d_1 data bit. This error can be corrected directly for a CiM XOR operation by simply flipping the erroneous bit. For other CiM operations, we perform two conventional reads on words $InMemW_1$ and $InMemW_2$, and correct the erroneous bits by recomputing them using an error detection and correction unit (discussed in section 4.2). Note that such corrections lead to overheads, as we need to access memory array 3 times (compared to 2 times in STT-MRAM). However, our variation analysis shows that error corrections on CiM operations are infrequent, leading to overall improvements.

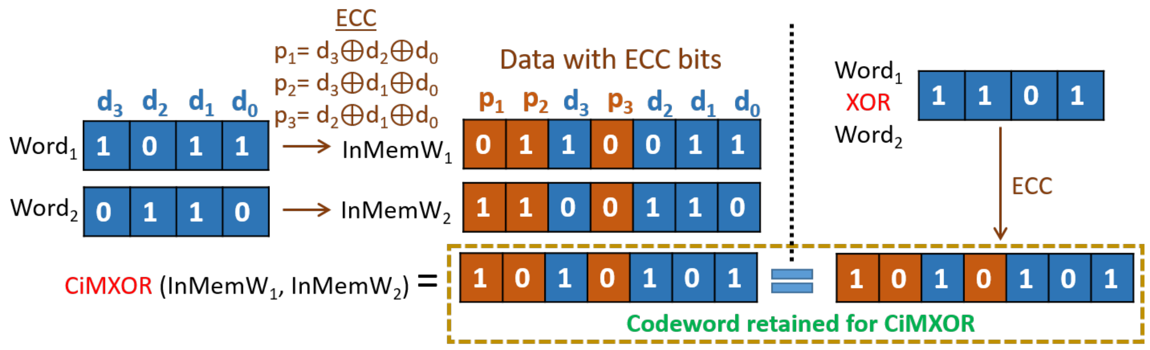

ECC
p1= d3⊕d2⊕d0
p2= d3⊕d1⊕d0
p3= d2⊕d1⊕d0
Data with ECC bits
d3 d2 d1 d0
Word1 1 0 1 1 → InMemW1 0 1 1 0 0 1 1
Word2 0 1 1 0 → InMemW2 1 1 0 0 1 1 0
p1 p2 d3 p3 d2 d1 d0
Word1 XOR Word2 1 1 0 1
ECC
CiMXOR (InMemW1, InMemW2) = 1 0 1 0 1 0 1 = 1 0 1 0 1 0 1
Codeword retained for CiMXOR

Fig. 4.6.: Codeword retention property of CIM XOR

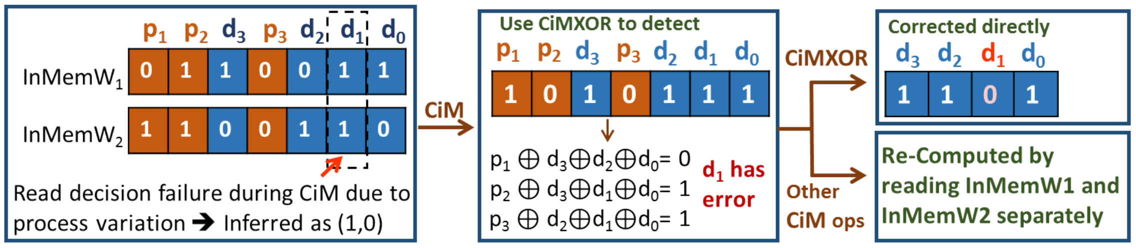


Fig. 4.7.: Error detection and correction for CiM operations in STT-CiM

ECC design methodology. We use the methodology employed in [43] to determine ECC requirements for both the baseline STT-MRAM and the proposed STT-CiM design. The approach uses circuit level simulations to determine the bit-level error probability, which is then used to estimate the array level yield. Moreover, the ECC scheme is selected based on the target yield requirement. Our simulation shows that 1 bit failure probability of normal reads and CiM operations are 4.2x10^{-8} and 6x10^{-5} respectively. With these obtained bit-level failure rates and assuming a target yield of 99%, the ECC requirement for 1MB STT-MRAM is single error correction and double error detection (SECDED), whereas for 1MB STT-CiM is three error correction and four error detection (3EC4ED). Note that the overheads of the ECC schemes [43, 133] are fully considered and reflected in our experimental results. Moreover, our simulation shows that the probability of CiM operations having errors is at most 0.1, *i.e.*, no more than 1 in 10 CiM operations will have an error. Errors on all CiM operatons are detected by using 3EC4ED code on the XOR output. Detected errors are directly corrected for CiM XORs using the 3EC4ED code, and by reverting to near-memory computation for other CiM operations.

Apart from ECC schemes, STT-CiM can also leverage various reliability improvement techniques proposed for STT-MRAMS [58–61]. Further, recent efforts [58, 59] that increase the Tunneling Magneto Resistance (TMR) of the MTJ and improve sensing margins will reduce read failure in CiM operations as well. These techniques can be used along with ECC to cost-effectively mitigate failures in STT-CiM operations.

4.2 STT-CiM Architecture

In order to evaluate the application-level benefits of STT-CiM, we integrate it as a scratchpad memory within the memory hierarchy of a programmable processor [132]. This section describes architectural enhancements for STT-CiM and hardware/software optimizations to increase its efficiency.

4.2.1 Optimizations for STT-CiM

In order to further the efficiency improvements obtained by STT-CiM, we propose additional optimizations.

Vector CiM operations. Modern computing workloads exhibit significant data parallelism. To further enhance the efficiency of STT-CiM for data-parallel computations, we introduce Vector Compute-in-Memory (VCiM) operations. The key idea behind VCiM operations is to perform CiM operations on all the elements of a vector concurrently. Figure 4.8 shows how the internal memory bandwidth (32xN bits) can be significantly larger than the limited I/O bandwidth (32 bits) visible to the processor. We exploit the memory's internal bandwidth to perform vector operations (N words wide) within STT-CiM.

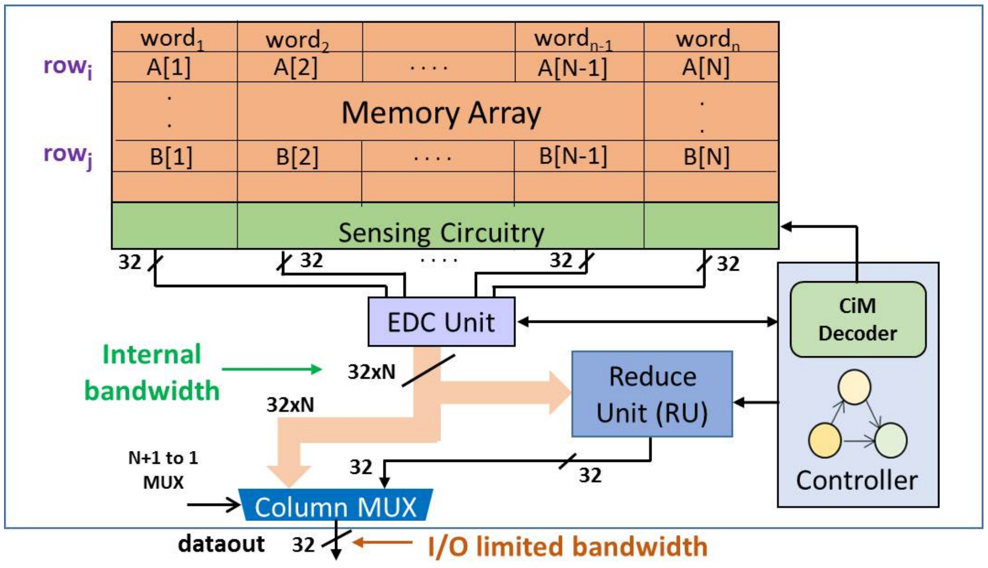


Fig. 4.8.: STT-CiM supporting In-memory vector operation

Note that the data resulting from a vector operation may also be a vector, and hence transferring it back to the processor is subject to the limited I/O bandwidth. To address this issue, we observe that vector operatons are often followed by reduction operations. For example, a vector dot-product involves element-wise multiplication of two vectors followed by a summation (reduction) of the resulting vector of products to produce a scalar value. Based on this observation, we introduce a Reduce Unit (RU) before the column multiplexer, as shown in Figure 4.8. The RU takes an array of data elements as inputs and reduces it to a single data element. The RU can support various reduction operations such as summation, Euclidean distance, L1 and L2 norm, zero-comparision, *etc.*, of which two are described in Table 4.3. Consider the computation of $\sum_{i=1}^{N} A[i] + B[i]$, where arrays A and B are stored in rows i and j respectively (shown in Figure 4.8). To compute the desired function using a VCiM operation, we activate rows i and j simultaneously, and configure the sensing circuitry to perform an ADD operation and the RU to perform accumulation of the resulting output. Note that the summation would require 2N memory accesses in a conventional memory. With scalar CiM operations, it would require N memory accesses. With the proposed VCiM operations, only a single memory access is required.

The overheads of the RU depend on two factors: (i) the number of different reduction operations supported, and (ii) the maximum vector length allowed (can be between 2 to N words). To limit the overheads, we restrict our design to vector lengths of 4 and 8.

Table 4.3.: Examples of reduction operations

Type	Function RuOut = f (IN_1, IN_2,...,IN_N)
Summation	RuOut=IN_1 + IN_2+ .. + IN_N
Zero-Compare	RuOut[k] = (IN_k == 0) ? 0 : 1

Error Detection and Correction. To enable correction of erroneous bits for CiM operations, we introduce an Error Detection and Correction (EDC) unit that implements the 3EC4ED ECC scheme. The EDC unit checks for errors using the CiM XOR output (recall that the XOR is evaluated along with all CiM operations) and signals the controller (shown in Figure 4.8) upon detection of erroneous computations. Upon receiving this error detection signal, the controller performs the required corrective actions.

4.2.2 Architectural Extensions for STT-CiM

To integrate STT-CiM in a programmable processor based system, we propose the following architectural enhancements.

ISA extension. We extend the ISA of a programmable processor to support CiM operations. To this end, we introduce a set of new instructions in the ISA (CiMXOR, CiMNOT,CiMAND, CiMADD ...) that are used to invoke the different types of operations that can be performed in the STT-CiM array. In a load instruction, the requested address is sent to the memory, and the memory returns the data stored at the addressed location. However, in the case of a CiM instruction, the processor is required to provide addresses of two memory locations instead of a single one, and the memory operates on the two data values to return the final output.

$$\begin{aligned} &\text{Format: } \textbf{Opcode} \text{ Reg1 Reg2 Reg3} \\ &\text{Example: } \textbf{CiMXOR } R_{ADDR1}\ R_{ADDR2}\ R_{DEST} \end{aligned} \tag{4.1}$$

Equation 4.1 shows the format of a CiM instruction with an example. As shown, both the addresses required to perform CiMXOR operations are provided through registers. The format is similar to a regular arithmetic instruction that accesses two register values, performs the computation, and stores the result back in a register.

Program transformation. To exploit the proposed CiM instructions at the application-level, an assembly-level program transformation is performed, wherein specific sequences of instructions in the compiled program are mapped to suitable CiM instructions in the ISA. Figure 4.9 shows an example transformation where two load instructions followed by an XOR instruction are mapped to a single CiMXOR instruction.

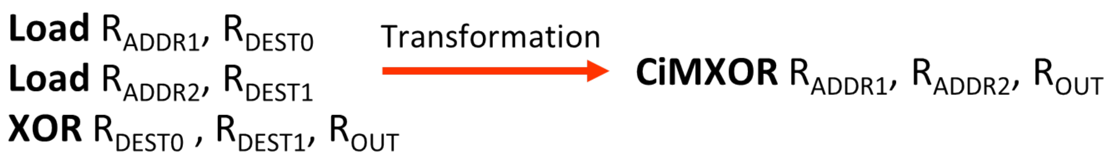

Load R_{ADDR1}, R_{DEST0}
Load R_{ADDR2}, R_{DEST1}
XOR R_{DEST0} , R_{DEST1}, R_{OUT}
Transformation
CiMXOR R_{ADDR1}, R_{ADDR2}, R_{OUT}

Fig. 4.9.: Program transformation for CiMXOR

Bus and interface support. In a programmable processor based system, the processor and the memory communicate via a system bus or on-chip network [4]. This makes it essential to analyze the impact of CiM operations on the bus and the corresponding bus interface. As discussed above, a CiM operation is similar to a load instruction with the key difference that it sends two addresses to the memory. Conventional system buses only allow sending a single address onto the bus via the address channel. In order to send the second address for CiM operations, we utilize the unused writedata channel of the system bus, which is unutilized during a CiM operation. Besides the two addresses, the processor also sends the type of CiM operation (CIMType) that needs to be performed. Note that it may be possible to overlay the CIMType signal onto the existing bus control signals; however, such optimizations strongly depend on the specifics of the bus protocol being used. In our design, we assume that 3 control bits are added to the bus to carry CIMType, and account for the resulting overheads in our experiments.

4.2.3 Data Mapping

In order to perform a CiM instruction, the locations of its operands in memory must satisfy certain constraints. Let us consider a memory organization consisting of several banks where each bank is an array that contains rows and columns. In this case, a CiM operation can be performed on two data elements only if they satisfy

[4]While we consider the case of a shared bus for illustration, the same enhancements can be applied to more complex interconnect networks.

three key criteria: (i) they are stored in the same bank, (ii) they are mapped to different rows, and (iii) they are stored in the same set of columns.

Consequently, a suitable data placement technique is required that maximizes the use of CiM operations. We observe that the target applications for STT-CiM have well-defined computation patterns, facilitating such a data placement. Figure 4.10 shows three general computation patterns. We next discuss these compute patterns and the corresponding data placement techniques.

Type I. This pattern, shown in the top row of Figure 4.10, involves element-to-element operations (OPs) between two arrays, *e.g.*, A and B. In order to effectively utilize STT-CiM for this compute pattern, we utilize the *array alignment* technique (shown in Figure 4.10(a)) that ensures alignment of elements A[i] and B[i] of arrays A and B for any value of i. This enables the conversion of operation A[i] OP B[i] into a CiM operation. An extension to this technique is the *row-interleaved placement* shown in Figure 4.10(b). This technique is applicable to larger data structures that do not fully reside in same memory bank. It ensures that the corresponding elements, *i.e.*, A[i] and B[i], are mapped to the same bank for any value of i, and satisfy the alignment criteria for a CiM operation.

Type II. This pattern, shown in the middle row of Figure 4.10, involves a nested loop in which the inner loop iteration consists of a single element of array A being operated with several elements of array B. For this one-to-many compute pattern, we introduce a *spare row* technique for data alignment. In this technique, a spare row is reserved in each memory bank to store copies of an element of A. As shown in Figure 4.10(c), in the k^{th} iteration of the outer for-loop, a special write operation is used to fill the spare rows in all banks with A[k]. This results in each element of array B becoming aligned with a copy of A[k], thereby allowing CiM operations to be performed on them. Note that the special write operation introduces energy and performance overheads, but this overhead is amortized over all inner loop iterations, and is observed to be quite insignificant in our evaluations.

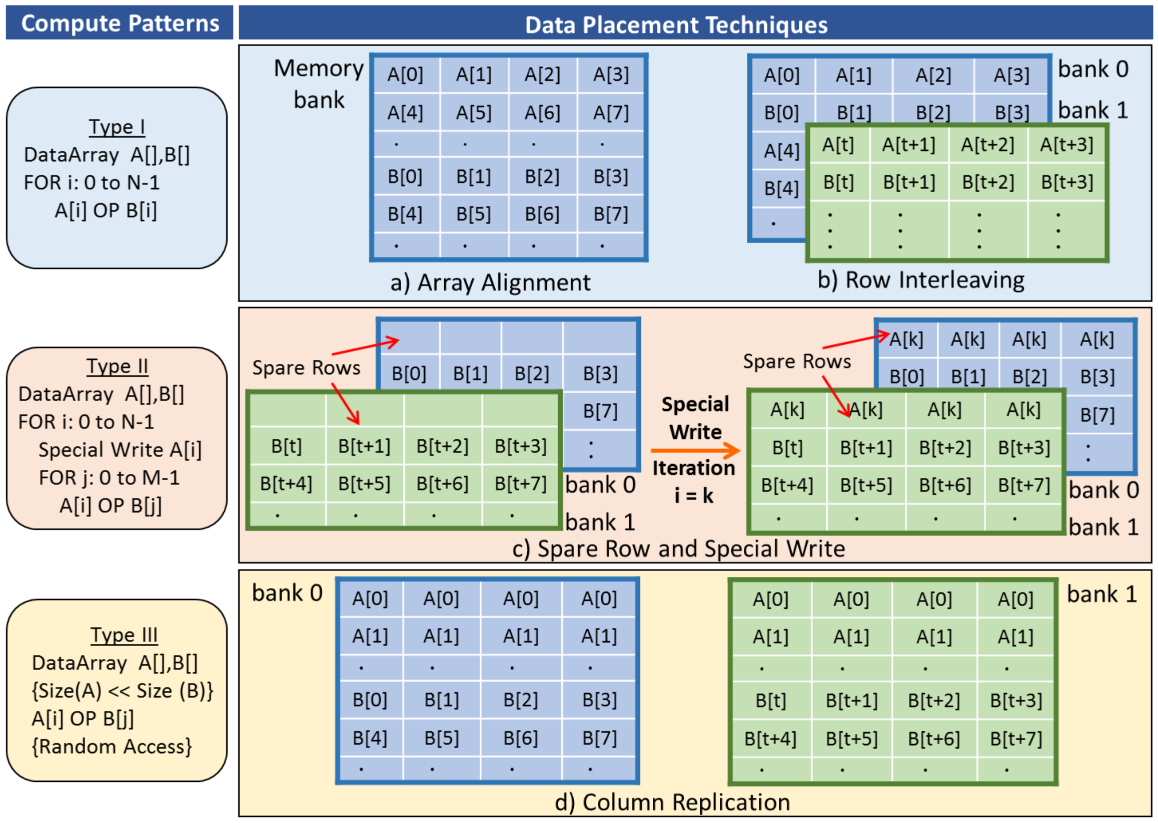


Fig. 4.10.: Data mapping for various computation patterns

Type III. In this pattern, shown in the bottom row of Figure 4.10, operations are performed on an element drawn from a small array A and an element from a much larger array B. The elements are selected arbitrarily, *i.e.*, without any predictable pattern. For example, consider when a small sequence of characters needs to be searched within a much larger input string. For this pattern, we propose a *column replication* technique to enable CiM operations, as shown in Figure 4.10(d). In this technique, a single element of the small array A is replicated across columns to fill an entire row. This ensures that each element of A is aligned with every element of B, enabling a CiM operation to be utilized. Note that the initial overhead due to data replication is very small, as it pales in comparison to the number of memory accesses to the larger array.

4.3 Experimental Methodology

In this section, we discuss the device-to-architecture simulation framework (Figure 4.11) and application benchmarks used to evaluate the performance and energy benefits of STT-CiM at the array-level and system-level.

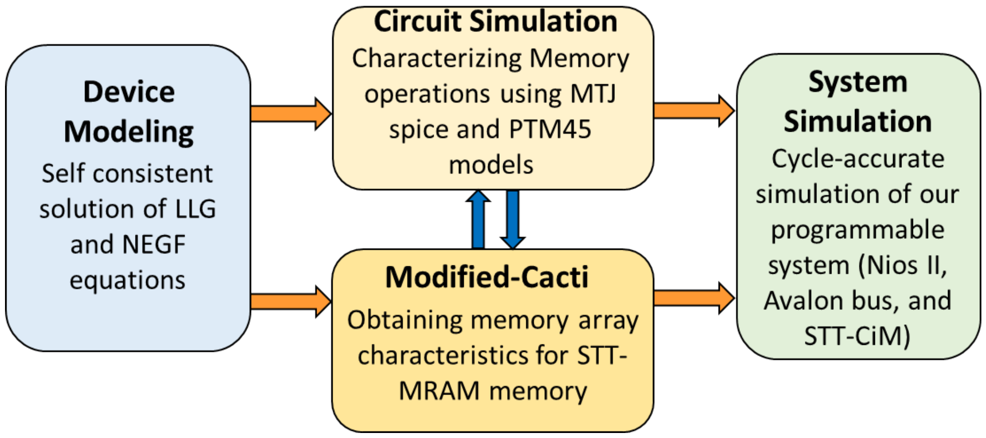

Device Modeling
Self consistent solution of LLG and NEGF equations
Circuit Simulation
Characterizing Memory operations using MTJ spice and PTM45 models
Modified-Cacti
Obtaining memory array characteristics for STT-MRAM memory
System Simulation
Cycle-accurate simulation of our programmable system (Nios II, Avalon bus, and STT-CiM)

Fig. 4.11.: STT-CiM device-to-architecture evaluation framework

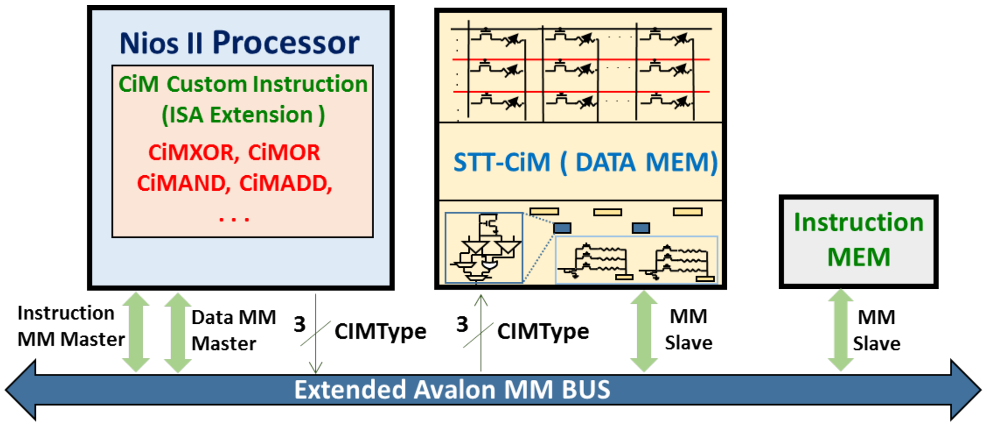


Fig. 4.12.: System level integration of STT-CiM

Device/Circuit modeling. We first characterize the bit-cells using SPICE-compatible MTJ models that are based on self-consistent solution of Landau-Lifshitz-Gilbert (LLG) magnetization dynamics and Non-Equilibrium-Green's Function (NEGF) elec-

Table 4.4.: Device parameters for evaluating STT-CiM

Material System	Ta/CoFeB/MgO
MTJ Type	PMA
Saturation Magnetization(M_S)	1.58T
Damping Factor, (α)	0.028
Polarization	0.62
Interface Anisotropy	1.3mJ/m^2
MTJ Dimension	40nm x 40nm x 1.32nm
Oxide Thickness (t_{ox})	1.1nm
Energy Barrier	65KT
T	300K
RA Product	18ohm-μm^2
TMR	124%
CMOS Technology	45nm Bulk CMOS
Assumed Variation (σ/μ)	t_{ox} = 2%, MTJ Area = 5% transistor V_T=5%

tron transport [134]. Table 4.4 shows the MTJ device parameters [135] used in our experiments. Using 45nm bulk CMOS technology and the MTJ models, the memory array along with the associated peripherals and extracted parasitics was simulated in SPICE for read, write and CiM operations to obtain array-level timing and energy characteristics. The obtained characteristics were then used as technology parameters in a modified version of CACTI [136] that is capable of estimating system-level properties for a spin-based memory. The variation analysis to compute failure rates was performed considering variations in MTJ oxide thickness ($\sigma/\mu = 2$ %), transistor V_T ($\sigma/\mu = 5\%$), and MTJ cross sectional area ($\sigma/\mu = 5\%$).

System level simulation. We evaluated STT-CiM as a 1MB scratchpad for an Intel Nios II processor [132]. Figure 4.12 shows the integration of STT-CiM in the memory hierarchy of the programmable processor. In order to expose the STT-CiM operations to software, we extended the Nios II processor's instruction set with custom instructions. The Avalon on-chip bus was also extended to support CiM operations. Cycle-accurate RTL simulation was used to obtain the execution time and the memory access traces for various benchmarks. These traces along with the energy results obtained through the modified CACTI tool were used to estimate the total memory energy.

Table 4.5.: Benchmark applications for evaluating STT-CiM

Algorithm	**Application**
Aho-Corasick (AHC)	String matching
Knuth-Morris-Pratt (KMP)	String matching
Edit Distance (EDIST)	Text processing, Computational biology
Bit-Blit (BLIT)	Low-level graphics
Longest Common Subsequence (LCS)	Data compression, Bio-informatics
Rivest Cipher 4 (RC4)	Cryptography
K-means clustering (IMGSEG)	Image segmentation
GLVQ	Eye detection
K-means clustering (KMEANS)	Digit recognition
K-nearest neighbors (OCR)	Optical character recognition
Multi Layer Perceptron (MLP)	Protein structure classification
Support Vector Machines (SVM)	Text classification

Benchmark applications. We evaluate STT-CiM on a suite of twelve algorithms drawn from various applications (Table 4.5).

4.4 Results

In this section, we first present an array-level analysis of STT-CiM and then quantify its benefits through system-level energy and performance evaluation.

4.4.1 Array-level analysis

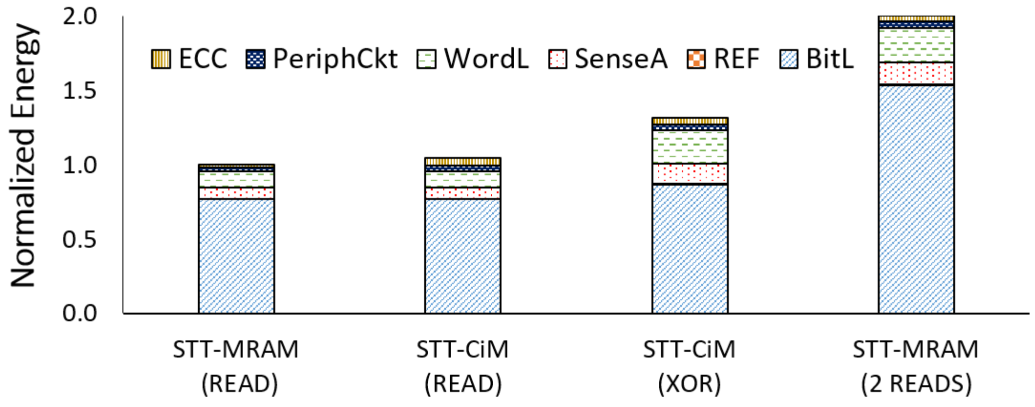


Fig. 4.13.: Array-level energy evaluation of STT-CiM

Energy. The second and third bars in Figure 4.13 show the energy consumed by a standard read operation and a representative CiM operation (CiMXOR) in a 1MB STT-CiM array. Each bar shows the energy breakdown into the major components, *i.e.*, peripheral circuitry (PeriphCkt), wordline (WordL), bitline (BitL), reference generation circuitry (REF), sense amplifier (SenseA), and error correction circuitry (ECC). For comparison, we provide the read energy for an STT-MRAM array of the same capacity (first bar in Figure 4.13) and all energy numbers are normalized to this value. A normal read operation in STT-CiM incurs an energy overhead of about 4.4%, which arises primarily due to the extra peripheral circuits (PeriphCkt) and stronger ECC. STT-CiM uses a 3EC4ED ECC scheme (as compared to SECDED in the baseline STT-MRAM), which accounts for about 3% of the 4.4% energy overhead. The CiMXOR operation consumes higher energy than a standard read operation mainly due to the charging of multiple wordlines and a slightly higher source line current. However, since a CiM operation replaces *two* normal read operations, we also present the energy required for two reads in a standard STT-MRAM (last bar in Figure 4.13). Note that an array-level comparison greatly understates the benefits of STT-CiM, since it does not consider the system-level impact of reduced data transfers between the processor and memory (system-level evaluation is presented in the next subsection). Nevertheless, it is worth noting that even at the array level, STT-CiM consumes 34.2% less energy than STT-MRAM. The benefits mainly arise from a lower bitline dynamic energy (BitL), since only a single access to the memory array is required for STT-CiM.

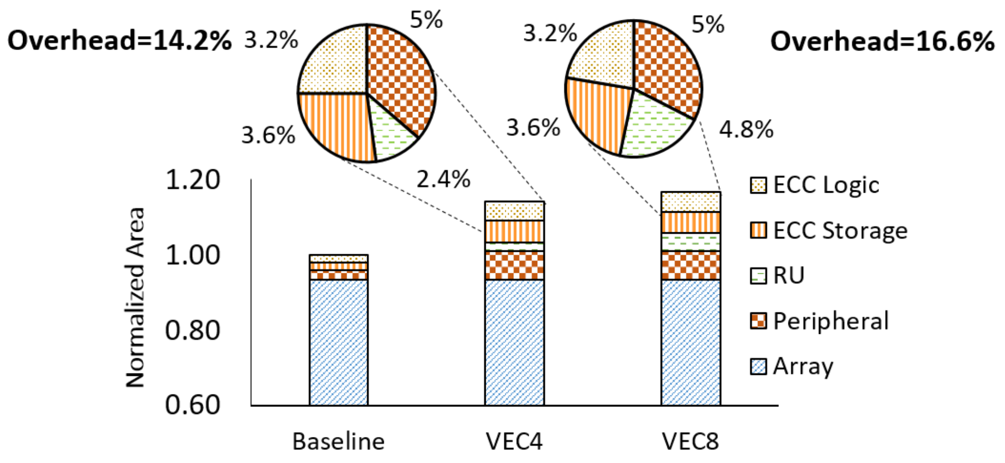


Fig. 4.14.: Array-level area evaluation of STT-CiM

Area and access time. Figure 4.14 shows the area breakdown for two STT-CiM designs that support vector operations of length 4 (VEC4) and 8 (VEC8). As compared to the STT-MRAM baseline, the area overheads for VEC4 and VEC8 are 14.2% and 16.6%, respectively. As shown in Figure 4.14, Peripheral circuits, ECC storage and ECC Logic are the causes of area overheads (5%, 3.6% and 3.2% respectively). Peripheral circuits include the enhanced address decoder (1.8%), sense amplifier (0.9%), and reduce unit (2.3%). Note that the total area is still dominated by the core array, which remains unchanged. Finally, the access time overhead for STT-CiM was found to be only ~0.8%, because the wordline and bitline delays dominate the total memory access latency.

4.4.2 Application-level memory energy

We next present the system level memory energy benefits of using STT-CiM in the programmable processor based system described in Figure 4.12. We evaluated the total memory energy consumed by STT-CiM across the application benchmarks, and compared it with a baseline design that uses standard STT-MRAM. Figure 4.15 shows the breakdown of different energy components, *viz.* Read, Write and CiM, that contribute to the overall memory energy in both the STT-MRAM and proposed STT-CiM designs. In addition, it also shows the energy overheads due to near memory corrections (NMCorrections) on failing CiM operations. The total memory energy for an application is normalized to the memory energy consumed by the baseline design. For the proposed STT-CiM design, we evaluated a version without vector operations (STT-CiM), and two versions with vector lengths of 4 and 8 (STT-CiM+VEC4 and STT-CiM+VEC8, respectively). Across all benchmarks, we observe 1.26x, 2.77x and 3.83x average improvement in energy for STT-CiM, STT-CiM+VEC4 and STT-CiM+VEC8, respectively.

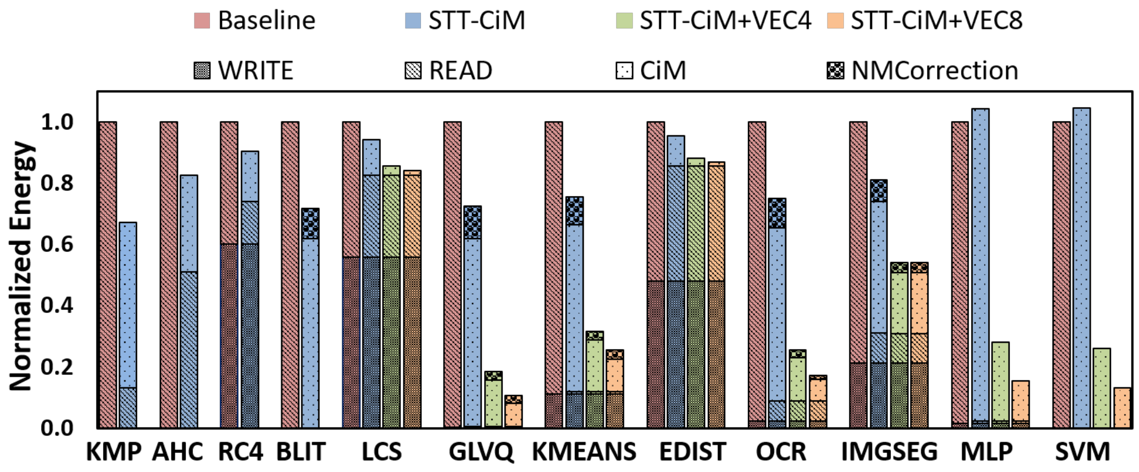


Fig. 4.15.: Improvement in application-level memory energy using STT-CiM

To provide further insights into the energy benefits, Figure 4.16 presents a breakdown for memory accesses made by each application into 3 categories — writes, reads that cannot be converted into CiM operations (CiM non-convertible reads or CNC-Reads), and CiM convertible reads (CC-Reads). We see that applications where CC-Reads dominate the total memory accesses (KMP, BLIT, GLVQ, KMEANS, OCR, IMGSEG, MLP, SVM in Figure 4.16) experience higher energy benefits from STT-CiM (Figure 4.15). Among these applications, those that benefit from vectorization achieve the highest savings (GLVQ, KMEANS, OCR, MLP, SVM). Applications with relatively fewer CC-Reads or more frequent writes (AHC, LCS, RC4, EDIST) exhibit relatively lower energy savings. CNC-Reads and writes are not benefited by STT-CiM, and writes in particular consume significantly (∼3x) higher energy than reads. The energy overheads due to additional writes incurred for data alignment in Type II and Type III compute patterns was observed to be 0.8% and 0.3%, respectively.

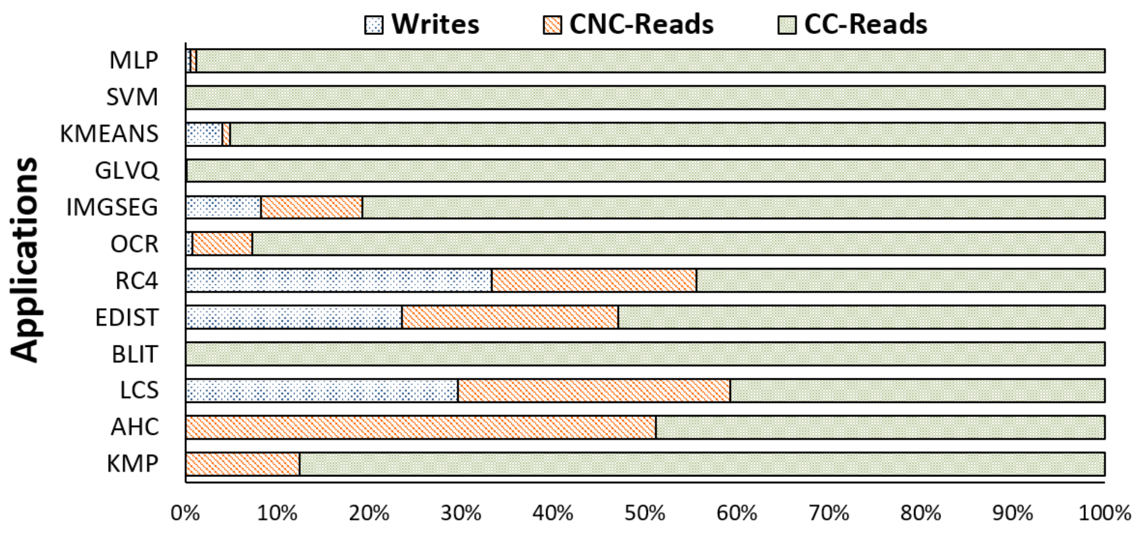


Fig. 4.16.: Memory access breakdown of applications used in evaluation of STT-CiM

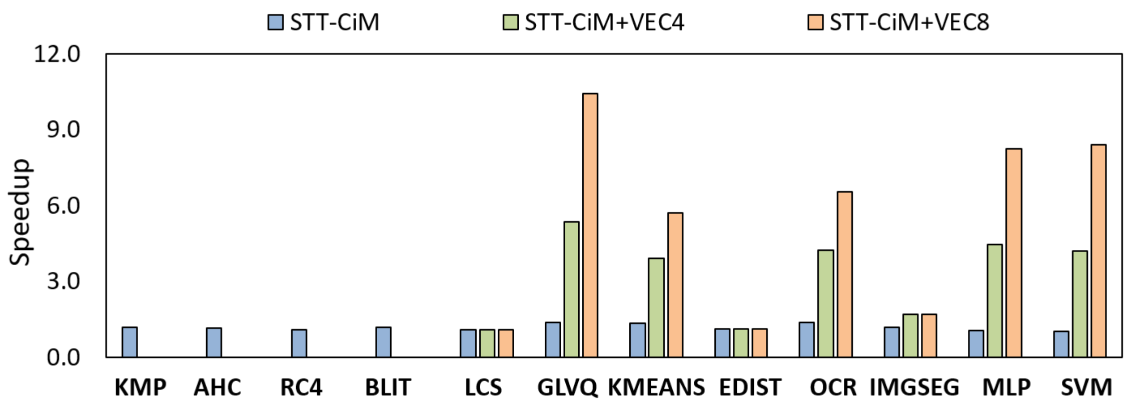


Fig. 4.17.: Improvement in application-level system performance using STT-CiM

4.4.3 System-level performance

Figure 4.17 shows the speedup for the Nios II processor system integrated with STT-CiM across various applications. The speedup shown in the figure is with respect to the baseline design, *i.e.*, the processor system integrated with a standard STT-MRAM based memory. As discussed in Section 4.2.2, CiM lowers the total number of memory accesses as well as the number of instructions executed, which leads to performance benefits at the system level. Overall, for STT-CiM without vector operations, we observe performance benefits ranging from 1.07X to 1.36X. With vector operations, the average speedup increased to 3.25x and 3.93x for vector lengths of 4 and 8, respectively. Comparing Figures 4.16 and 4.17, we see that the factors that indicate higher energy savings for an application (large fraction of memory accesses are CC-Reads, opportunities for vectorization exist) are also predictive of higher performance improvements.

In order to demonstrate the performance sensitivity to memory latency, we vary the memory latency and evaluate the execution time for each application. Figure 4.18 shows the results of this sensitivity analysis. On the Y-axis, we have the speedup of STT-CiM over STT-MRAM, and on the X-axis the memory latency. We observe that STT-CiM yields higher performance benefits at higher memory latency. This is attributed to the fact that the reduced number of memory accesses for STT-CiM has a larger impact on system performance. On an average, we achieve 1.13x speedup for a memory latency of 1 cycle, and 1.26x speedup for a memory latency of 16 cycles, thereby illustrating the effectiveness of the proposed approach.

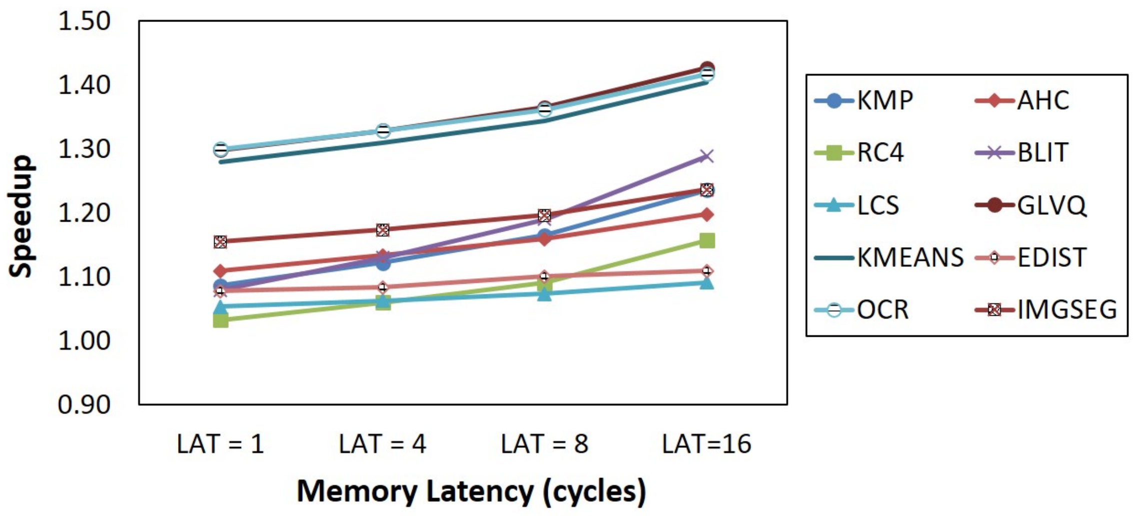


Fig. 4.18.: STT-CiM performance sensitivity to memory latency

4.5 Summary

STT-MRAM is a promising candidate for future on-chip memories. In this work, we proposed STT-CiM, an enhanced STT-MRAM that can perform a range of arithmetic, logic and vector compute-in-memory operations. We addressed a key challenge associated with these in-memory operations, *i.e.* reliable computation under process variations. We utilized the proposed design (STT-CiM) as a scratchpad in the memory hierarchy of a programmable processor, and introduced ISA extensions and on-chip bus enhancements to support in-memory computations. We proposed architectural optimizations and data mapping techniques to enhance the efficiency of

STT-CiM. A device-to-architecture simulation framework was used to evaluate the benefits of STT-CiM. Our experiments indicate that STT-CiM achieves substantial improvements in energy and performance, and shows considerable promise in alleviating the processor-memory gap.

5. FRAMEWORK FOR EVALUATING DEEP NEURAL NETWORKS ON RESISTIVE CROSSBARS

Deep neural networks are the state-of-the-art in many machine learning tasks, as discussed in chapter 1. However, the large and rapidly growing computation requirements of DNNs pose severe challenges to performance and energy efficiency in the systems on which they are deployed. Resistive crossbar based systems are promising due to their ability to compactly and efficiently realize vector-matrix multiplications, the underlying computational kernels in DNNs. However, in practice, the functionality of resistive crossbar deviate considerably from the ideal abstraction due to the device and circuit-level non-idealities, as discussed in chapter 1. Therefore, it is essential to evaluate the impact of non-idealities in resistive crossbars on the overall application-level accuracy of the realized workloads.

Most previous efforts on resistive crossbar based DNN implementations either do not consider non-idealities or model non-idealities in a very limited manner (*e.g.*, as limited precision). Moreover, they focus their analysis on simple networks and datasets (*e.g.*, MNIST). Thus, they leave open the question of how non-idealities impact the accuracy of *large-scale neural networks* realized on resistive crossbars. State-of-the-art DNNs often contain tens to hundreds of layers, millions of neurons and billions of synaptic connections. Evaluating such networks requires a fast and scalable, yet accurate simulation framework for resistive crossbars that can be integrated into state-of-the-art DNN software frameworks. Unfortunately, such a framework is currently unavailable. Device and circuit simulation (SPICE) models of resistive crossbars are accurate but extremely slow and infeasible for large-scale network evaluation. Architectural models of resistive crossbars [79,80,124] target design space exploration and use highly simplified error models that are reasonable for their context, but inadequate for evaluating application-level accuracy of DNNs. For example, these models do not consider error dependence on the crossbar inputs, programmed conductances, and the crossbar column performing the computation. In this work, we address the need for a fast and accurate simulation framework to enable functional evaluation of large-scale DNNs on resistive crossbars.

We first study the impact of crossbar non-idealities to characterize errors in the realized vector-matrix multiplications. We observe the errors to show significant data dependence and hardware-instance dependence (due to variations), motivating the need for a detailed crossbar model. Next, we propose a Fast Crossbar Model (FCM) that can accurately capture the impact of crossbar non-idealities. FCM abstracts non-idealities using simple linear algebra operations to achieve orders-of-magnitude faster simulation compared to SPICE. We realize FCM using the well-known BLAS (Basic Linear Algebra Subprograms) library and develop RxNN, a software framework to evaluate DNNs realized on resistive crossbar systems. We use RxNN (which is based on the popular Caffe [91] deep learning framework) to evaluate six large-scale DNNs for classifying the ImageNet [131] dataset and three simple networks for classifying CIFAR-10 and MNIST datasets. Our evaluation reveals that crossbar non-idealities do not impact accuracy significantly for such small/simple networks. However, our results on large networks tell a very different story, wherein non-idealities result in significant accuracy loss for large-scale DNNs on resistive crossbar systems, motivating the need for further research in cross-layer mitigation and compensation techniques. One of the key contributions of this chapter is to also draw attention to the challenges of crossbar non-idealities that become significant as we move from the small networks (LeNet and ConvNet) to larger DNNs (ResNet, etc.).

In summary, the key contributions of this work are:

- We study the cumulative effect of all crossbar non-idealities to characterize errors in the realized vector-matrix multiplications. We find the errors to show significant data and hardware-instance dependence that should be considered for accurately modeling vector-matrix multiplications in crossbars.

- We propose FCM, a fast and accurate functional crossbar model to capture the effects of crossbar non-idealities.
- We develop RxNN a software framework that can evaluate large-scale DNNs on resistive crossbar systems and help re-train to compensate for the effects of non-idealities.
- We evaluate the application-level accuracy of 6 state-of-the-art DNNs, *viz.* ResNet-50, VGG-16, GoogleNet, AlexNet, OverFeat, and NiN on a resistive crossbar based system. Our evaluation reveals that the degradation in accuracy due to non-idealities can be significant (9.6%-32%) for large-scale DNNs. This degradation can be partially alleviated by re-training, but calls for further research in compensation techniques.

The rest of the chapter is organized as follows. Section 5.1 discusses crossbar non-idealities and demonstrates their impact on vector-matrix multiplications realized using crossbars. Section 5.2 describes the proposed FCM models. Section 5.3 presents the RxNN software framework. Section 5.4 details the experimental methodology. Experimental results are presented in section 5.5. Section 5.6 concludes the chapter.

5.1 Crossbar Non-Idealities

In this section, we analyze non-idealities in resistive crossbars and examine their impact on vector-matrix multiplications.

5.1.1 Crossbar Non-idealities

To illustrate the device and circuit level non-idealities in resistive crossbars, we present the equivalent resistive circuit for the crossbar array and the peripherals (DAC and ADC) in Figure 5.1. The key sources of non-idealities are wire resistances of the crossbar interconnects, sensing resistances of the circuits that sense the output currents, driver resistances of the circuits that drive the crossbar rows, sneak paths, variance in synaptic conductance due to process variations, device-level noise, and imperfect programming, and non-ideal DACs. While we consider all these non-idealities in subsequent sections, we select the non-idealities due to DACs and sneak paths for a more detailed treatment below, in order to illustrate the complexity of error modeling.

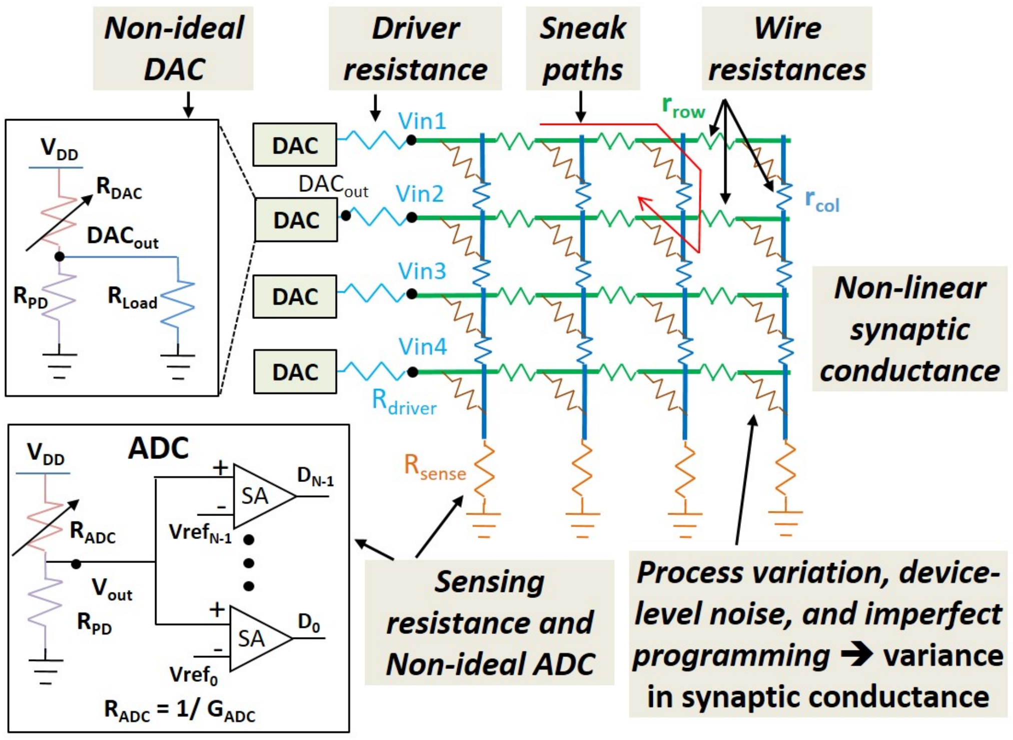


Fig. 5.1.: Crossbar non-idealities overview

Non-ideal DAC. Figure 5.1 shows the equivalent circuit for a DAC that is represented using a resistive divider circuit with an input determined resistance (R_{DAC}) and a fixed resistance (R_{PD}). An applied digital input determines the value of R_{DAC} and subsequently decides the DAC's output voltage (DAC_{out}). Note that, DAC_{out} also depends on the effective resistive load (R_{Load}), leading to deviations from the ideal value. R_{Load} is a function of the synaptic conductances within the crossbar array and therefore varies with the crossbar state (the values of all synaptic conductances). The equation in Figure 5.2(a) shows the error incurred due to DAC non-idealities which is a function of both applied inputs (R_{DAC}) and synaptic conductances (R_{Load}). Figure 5.2(a) also illustrates errors' dependence on the applied inputs. The plot shows the outputs of the non-ideal DAC (Non-ideal DAC_{OUT}) for two load resistances 3.2kΩ and 32kΩ, respectively. As evident, the voltage plots of the two R_{Load} differ with the applied inputs.

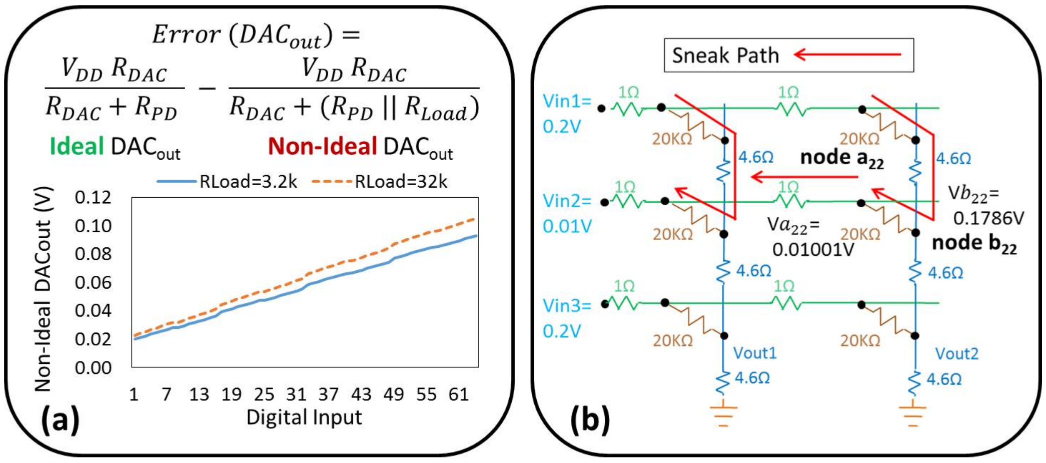


Fig. 5.2.: Example of non-idealities in resistive crossbar: (a) Non-ideal DAC, (b) Sneak paths

Sneak paths during vector-matrix multiplication. Ideally, currents in resistive crossbars would be expected to flow from left to right along the rows and from top to bottom through the columns. However, due to the non-idealities described above (specifically, wire resistances), internal node voltages within the crossbar may vary, resulting in additional current paths, which we refer to as sneak paths. Figure 5.2(b) illustrates sneaks paths during vector-matrix multiplications for a 3x2 crossbar array. We consider a crossbar state with all synaptic devices programmed to 20KΩ, and the applied input voltages at the rows are 0.2V, 0.01V and 0.2V, respectively. For this crossbar state, we observe that the direction of current between nodes a_{22} and b_{22} is flipped, *i.e.*, the current flows from b_{22} towards the input (Vin2), instead of the expected direction. Sneak paths are a function of both the crossbar state and the applied inputs, and therefore further contribute to the overall dynamism in errors due to non-idealities.

5.1.2 Errors due to Non-Idealities

Next, we study the impact of non-idealities on the computational accuracy of the vector-matrix multiplication realized using resistive crossbars. To this end, we compare the outputs of vector-matrix multiplications obtained from HSPICE simulations of non-ideal crossbar arrays with the ideal computations (Equation 3.1) and analyze the errors' sensitivity to various parameters.

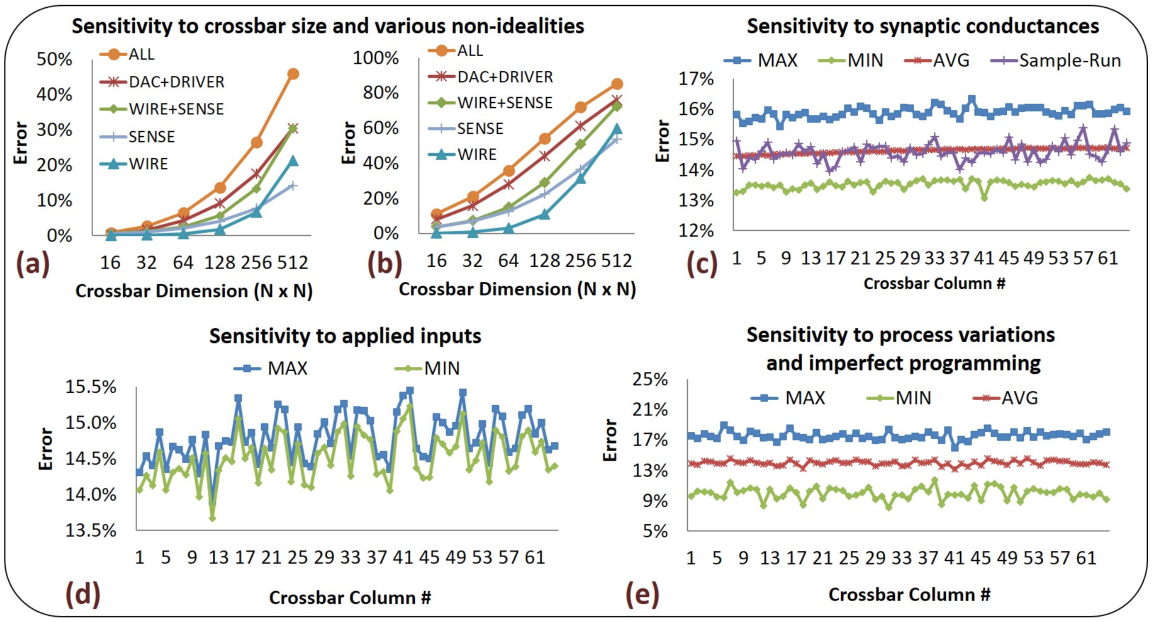


Fig. 5.3.: Crossbar non-idealities: (a)-(b) Sensitivity to crossbar dimension with all synaptic conductances programmed to G_{MIN} and G_{MAX}, respectively, (c)-(d) Sensitivity to synaptic conductances and applied inputs, (e) Sensitivity to process variation and imperfect programming

Sensitivity to crossbar size. We first examine how the errors incurred due to the individual non-idealities (WIRE, SENSE), combinations of non-idealities (DAC+DRIVER, WIRE+SENSE), and due to the cumulative effect of all non-idealities (ALL) vary with the crossbar dimension. Figures 5.3(a) and 5.3(b) show the errors incurred during the vector-matrix multiplication realized using crossbars, with all synaptic conductances programmed to G_{MIN} and G_{MAX}, respectively. In both graphs, the Y-axis represents the error in the last (N^{th}) column of an NxN crossbar, and the X-axis represents the crossbar dimension (N). In both cases, we observe that the overall errors due to all non-idealities (ALL), as well as due to individual non-idealities, increase with the crossbar dimension. This is expected because: (i) the overall wire resistances increase with crossbar array size, (ii) the sensing resistance contribution to the overall bitline resistance increases, and (iii) the DAC non-ideality increases due to a decrease in the effective load resistance [1]. Further, we also observe that for smaller crossbars, the non-ideality due to DAC is predominant, whereas, for larger crossbars, the wire and sensing resistance effect becomes equally significant.

Sensitivity to crossbar state. Next, we characterize errors' dependence on the crossbar state, *i.e.*, the conductances of all synaptic devices. To this end, we fix the inputs to a 64x64 crossbar array and vary the conductances of the synaptic devices to obtain different crossbar states. Figure 5.3(c) shows the maximum (MAX), minimum (MIN), and average (AVG) errors across columns of the crossbar over 1000 random crossbar states. We observe that the errors show significant dynamism across these states. In Figure 5.3(c), we also plot the errors for a sample crossbar state (Sample-Run) to demonstrate the irregular pattern shown by them across crossbar columns. Moreover, this irregular pattern deviates notably from the patterns observed for MAX, MIN, and AVG errors.

Sensitivity to crossbar inputs. To analyze the errors' dependence on the applied inputs, we fixed the conductances of all synaptic devices and varied the inputs. Figure 5.3(d) shows the variations in errors across inputs. We observe that the variance across inputs (MAX and MIN) for a particular column is noticeable, but small in

[1]Higher crossbar dimensions have more columns leading to increase in parallel paths, consequently lowering the effective load resistance

comparison to the variance across crossbar states. However, errors' variance across columns is significant.

Sensitivity to crossbar columns. Figures 5.3(c-d) depicts how errors vary across crossbar columns. While there is a slight trend of increase in error as we go from the first to the last column, it is not always the last column that incurs the maximum error. Rather any column can incur the maximum error depending on the crossbar states and the applied inputs.

Sensitivity to process variation and imperfect programming. Finally, we also evaluate the impact of variations by performing Monte-Carlo simulation on a sample set of 10,000 crossbar states obtained by considering variations in synaptic conductances ($\sigma/\mu = 10\%$) [137]. Figure 5.3(e) shows the maximum, minimum, and average error observed on a 64x64 crossbar array across these samples. The variations in synaptic conductances can occur due to two prominent reasons: (i) Process variations and (ii) Imperfect programming, *i.e.*, errors during write operations.

In summary, the non-idealities in resistive crossbars can have a significant impact on the computations that they perform, and that the errors due to non-idealities are highly dependent on various factors, including the conductances, applied inputs, crossbar column and hardware-instance performing the computation. In order to accurately capture the impact of non-idealities on application-level accuracy, a crossbar model should consider these factors.

5.2 Crossbar Modeling

In this section, we present a Fast Crossbar Model (FCM) that accurately captures the impact of non-idealities on the vector-matrix multiplications realized using resistive crossbars.

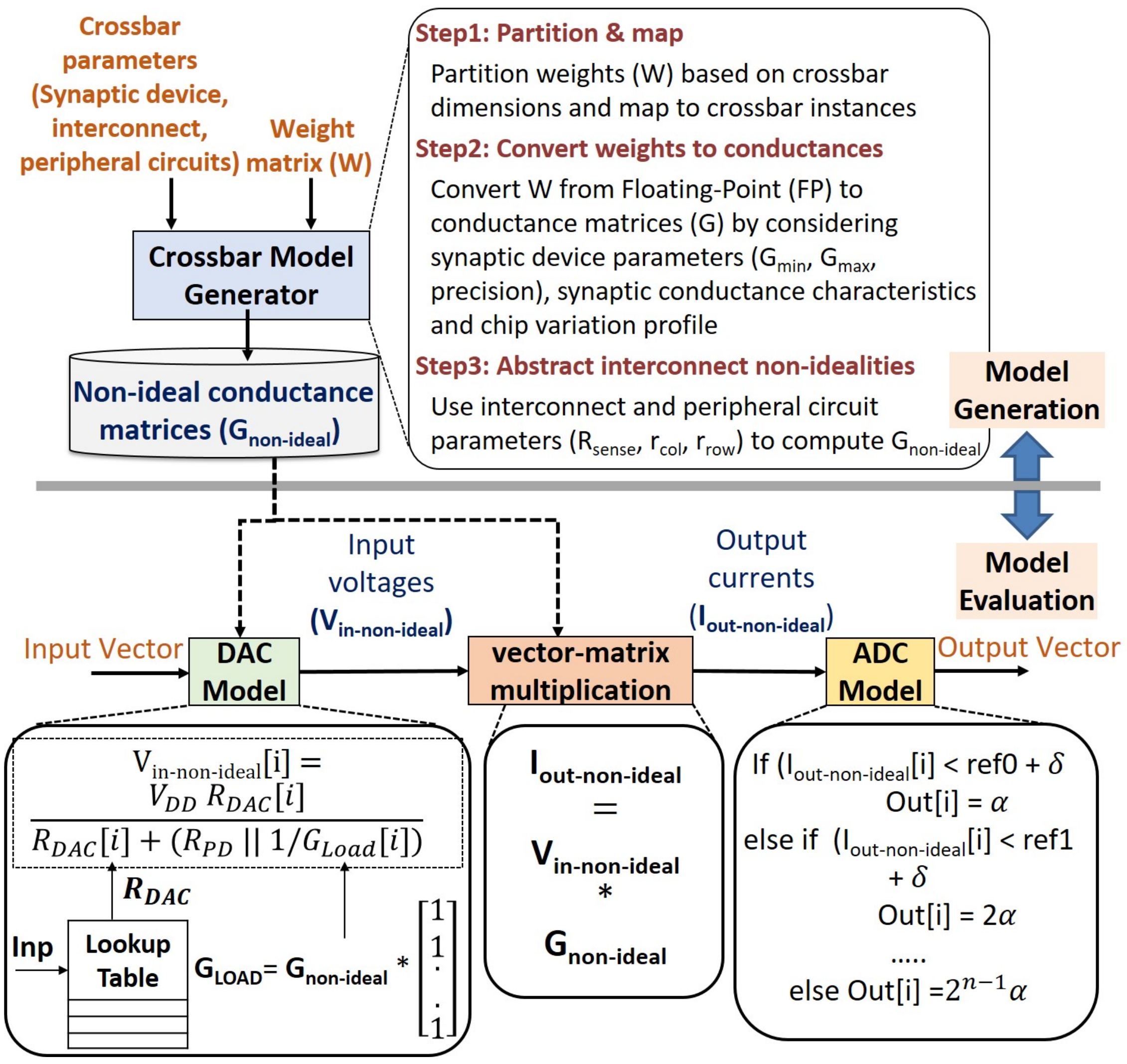

Crossbar parameters (Synaptic device, interconnect, peripheral circuits)
Weight matrix (W)
Crossbar Model Generator
Non-ideal conductance matrices ($G_{non\text{-}ideal}$)
Step1: Partition & map
Partition weights (W) based on crossbar dimensions and map to crossbar instances
Step2: Convert weights to conductances
Convert W from Floating-Point (FP) to conductance matrices (G) by considering synaptic device parameters (G_{min}, G_{max}, precision), synaptic conductance characteristics and chip variation profile
Step3: Abstract interconnect non-idealities
Use interconnect and peripheral circuit parameters (R_{sense}, r_{col}, r_{row}) to compute $G_{non\text{-}ideal}$
Model Generation
Model Evaluation
Input voltages ($V_{in\text{-}non\text{-}ideal}$)
Output currents ($I_{out\text{-}non\text{-}ideal}$)
Input Vector
DAC Model
vector-matrix multiplication
ADC Model
Output Vector
$V_{in\text{-}non\text{-}ideal}[i] = \frac{V_{DD} R_{DAC}[i]}{R_{DAC}[i] + (R_{PD} \| 1/G_{Load}[i])}$
R_{DAC}
Inp
Lookup Table
$G_{LOAD} = G_{non\text{-}ideal} * [1\ 1\ \cdot\ \cdot\ 1]^T$
$I_{out\text{-}non\text{-}ideal} = V_{in\text{-}non\text{-}ideal} * G_{non\text{-}ideal}$
If ($I_{out\text{-}non\text{-}ideal}[i] < ref0 + \delta$
Out[i] = α
else if ($I_{out\text{-}non\text{-}ideal}[i] < ref1 + \delta$
Out[i] = 2α
.....
else Out[i] = $2^{n-1}\alpha$

Fig. 5.4.: FCM: Overview

5.2.1 FCM Overview

Figure 5.4 overviews the proposed fast crossbar model that consists of two phases: (i) Model generation and (ii) Model evaluation. Model generation is a design time phase that is performed only once for a DNN, whereas model evaluation is a runtime phase that is invoked to evaluate each inference operation using the DNN. The key idea behind FCM is to first abstract non-idealities using the crossbar model generator to transform a weight matrix (W) into a non-ideal conductance matrix ($G_{non-ideal}$).

Subsequently, using the generated $G_{non-ideal}$ matrix and non-linear peripheral (ADC and DAC) models, FCM emulates the non-ideal vector matrix multiplications on resistive crossbars. We discuss these steps of FCM in detail below.

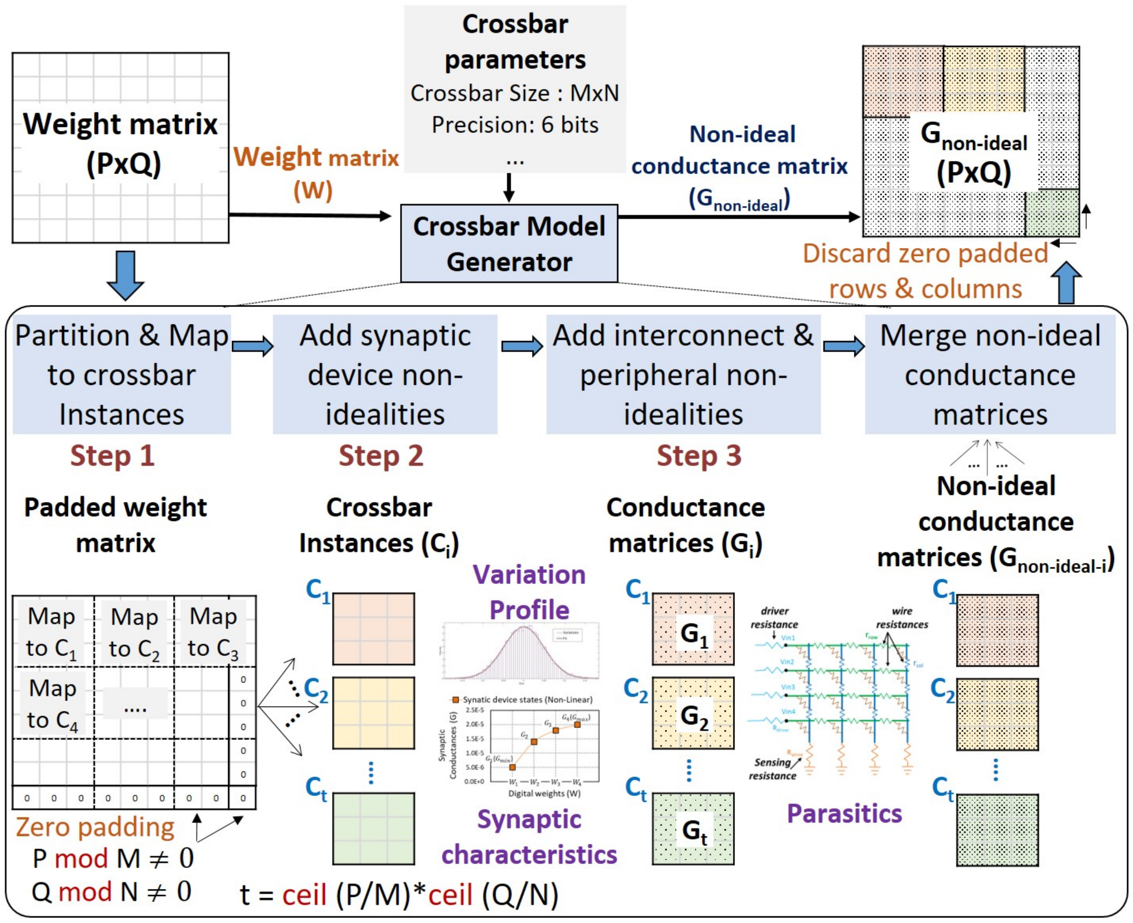


Fig. 5.5.: Crossbar model generator: Details

Crossbar model generator. FCM uses a crossbar model generator to abstract the hardware instance, the synaptic device and the interconnect specific crossbar non-idealities arising due to the process variation, the non-linear synaptic conductance characteristics, the sensing resistance, and the wire resistances. The model generator takes crossbar parameters and a weight matrix (W) as inputs and generates a non-ideal conductance matrix ($G_{non-ideal}$) as the output. Using a three-step transformation mechanism (listed in Figure 5.4), it converts W to $G_{non-ideal}$ based

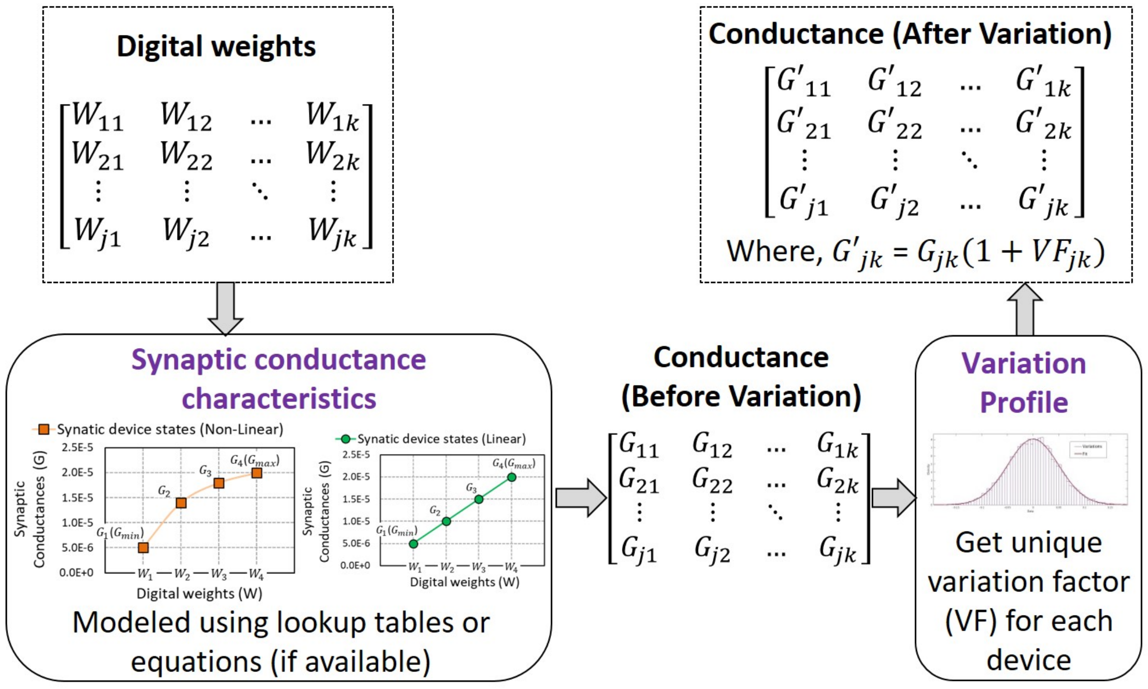

Digital weights
$\begin{bmatrix} W_{11} & W_{12} & \dots & W_{1k} \\ W_{21} & W_{22} & \dots & W_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ W_{j1} & W_{j2} & \dots & W_{jk} \end{bmatrix}$
Synaptic conductance characteristics
Synatic device states (Non-Linear)
Synatic device states (Linear)
Synaptic Conductances (G)
Digital weights (W)
Modeled using lookup tables or equations (if available)
Conductance (Before Variation)
$\begin{bmatrix} G_{11} & G_{12} & \dots & G_{1k} \\ G_{21} & G_{22} & \dots & G_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ G_{j1} & G_{j2} & \dots & G_{jk} \end{bmatrix}$
Variation Profile
Get unique variation factor (VF) for each device
Conductance (After Variation)
$\begin{bmatrix} G'_{11} & G'_{12} & \dots & G'_{1k} \\ G'_{21} & G'_{22} & \dots & G'_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ G'_{j1} & G'_{j2} & \dots & G'_{jk} \end{bmatrix}$
Where, $G'_{jk} = G_{jk}(1 + VF_{jk})$

Fig. 5.6.: Abstracting synaptic device non-idealities

on crossbar parameters including synaptic device characteristics (G_{min}, G_{max}, precision), interconnect (R_{Sense}, r_{row}, r_{col}), and circuit (crossbar size) parameters, and the chip variation profile. Figure 7.10 illustrates the model generation process using an example, where we consider the mapping of a PxQ weight matrix to crossbars of size MxN. In step 1, the model generator slices the matrix W into fragments and maps them to multiple crossbar instances. The fragment size is same as the crossbar dimension and to achieve this for all fragments the corners of the matrix W are zero padded (if required). Note that, we need t=**ceil**(P/M)***ceil**(Q/N) crossbar instances, 'P **mod** M' zero-padded rows, and 'Q **mod** N' zero-padded columns. Next, in step 2, weights are converted from floating-point (FP) values to conductances (G) considering device conductance characteristics, parameters (G_{min}, G_{max}, precision), and the variation profile. We support synaptic devices with both linear [64, 138] and non-linear [73, 120] conductance characteristics, either using equations (if available) or lookup tables. We sample the variation profile to obtain a unique variation factor

(VF) for each synaptic element within and across crossbar instances. At the end of step 2, we obtain a conductance matrix (G_i) for each crossbar instance. Figure 5.6 details the step 2 of the crossbar model generator. Finally, in step 3, the generator abstracts interconnect non-idealities (R_{sense}, r_{col}, r_{row}) and transforms the conductance matrices (G_i) to the corresponding non-ideal conductance matrices ($G_{non-ideal-i}$). Subsequently, these non-ideal conductance matrices ($G_{non-ideal-i}$) are merged to obtain one $G_{non-ideal}$ matrix. The transformation of G_i to $G_{non-ideal-i}$ is exact, and we provide the mathematical proof in Section 5.2.2.

Peripheral (ADC & DAC) models. Figure 5.4 details the ADC and DAC models used by FCM to incorporate ADC and DAC non-idealities. The DAC model is composed of a resistive divider circuit with a digital input (Inp) dependent resistance (R_{DAC}) and a fixed resistance (R_{PD}). The resistive divider is connected to a variable effective load conductance (G_{Load}) whose value is dependent on the crossbar state (synaptic conductances). FCM uses the equation shown in Figure 5.4 to compute the non-ideal input voltages ($V_{in-non-ideal}$). In the shown equation, R_{DAC} is determined using the digital inputs (Inp), and G_{Load} is computed using the $G_{non-ideal}$ matrix. We note that $V_{in-non-ideal}$ captures the data-dependence of the errors arising due to non-ideal DAC as R_{DAC} and G_{Load} are dependent on the applied inputs and the crossbar state, respectively. Using matrices $G_{non-ideal}$ and $V_{in-non-ideal}$, FCM computes the non-ideal vector-matrix multiplication realized in crossbars to obtain non-ideal output currents ($I_{out-non-ideal}$). The ADC model shown in Figure 5.4 is then used to convert the $I_{out-non-ideal}$ to digital outputs (Out). Note that, FCM utilizes lookup tables to model non-linear functions.

FCM abstracts both device and circuit non-idealities into fast models that achieve several orders-of-magnitude speed-up over SPICE, without compromising on the modeling accuracy (which is within 0.28% of HSPICE). We note that achieving such simulation speed is not possible without some abstraction, and FCM provides a good tradeoff between fidelity vs. simulation speed (detailed in section 5.5.1). FCM realizes algebraic operations using well-optimized BLAS (Basic Linear Algebra Subprograms) routines to further improve the simulation speed.

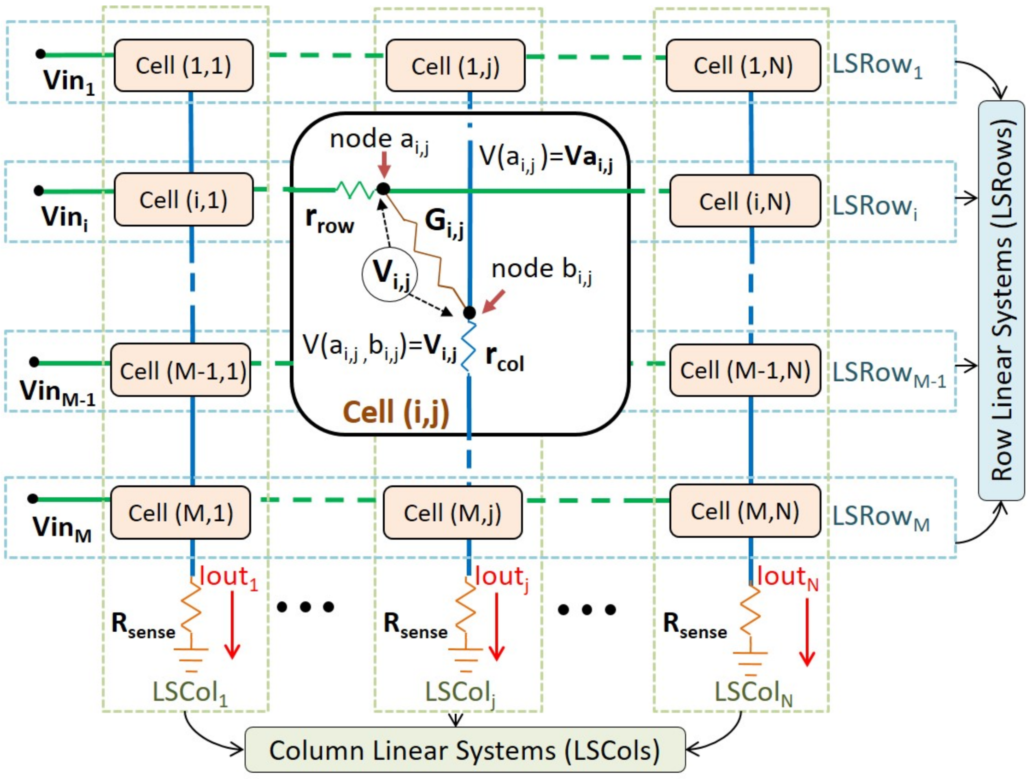


Fig. 5.7.: Equivalent resistance circuit of MxN crossbar array

5.2.2 Abstraction of interconnect non-idealities

In this section, we provide the mathematical formulation for the abstraction of interconnect non-idealities (Step 3 of crossbar model generation). We recall that in this step the generator abstracts interconnect non-idealities (R_{sense}, r_{col}, r_{row}) and transform the conductance matrix (G_i) associated with the i^{th} crossbar instance to the corresponding non-ideal conductance matrix ($G_{non-ideal-i}$). We achieve this transformation by leveraging circuit laws (Kirchhoff's loop laws and Ohm's law) and linear algebraic operations (direct sum, row switching, vector concatenation, row reduction, etc.).

We now explain the formulation using Figure 5.7 that shows the equivalent resistive circuit of an MxN crossbar array. Vin_i represents the input voltage at the i^{th} row of the crossbar, $Va_{i,j}$ denotes the voltage at the node $a_{i,j}$, and $V_{i,j}$ is the voltage difference between the node $a_{i,j}$ and the node $b_{i,j}$. $G_{i,j}$ is the conductance of the synaptic device at the i^{th} row and the j^{th} column. R_{sense}, r_{row}, and r_{col} depict the sensing and distributed wire resistances, respectively, and $Iout_j$ indicates the output current of the j^{th} column. In figure 5.7, we refer vertical and horizontal slices of the crossbar array as Column Linear Systems (LSCols) and Row Linear Systems (LSRows), respectively. To demonstrate the formulation of $G_{non-ideal}$, we employ 6 major steps involving Equations 5.1 to 5.13. We next describe these steps in turn below.

Step 1: Formulate column linear systems. We first formulate column linear systems ($LSCol_1$ to $LSCol_N$) using each vertical slice of the crossbar, shown in Figure 5.7. Let us consider the j^{th} vertical slice corresponding to the $LSCol_j$ system (Equations 5.1 to 5.3). Using Kirchhoff's Current Law (KCL) at all nodes $b_{i,j}$ present in the j^{th} column, we obtain Equations 5.1 and 5.2. Equations 5.1 and 5.2 are then combined to obtain the linear system in Equation 5.3. In the case of an MxN crossbar, we have N such linear systems ($LSCol_1$ to $LSCol_N$).

$$\mathbf{A_j} * \mathbf{Vcol_j} = \mathbf{VAcol_j} - \mathbf{J} * Iout_j \tag{5.1}$$

$$Iout_j = \sum_{x=1}^{x=M} G_{x,j} V_{x,j} \tag{5.2}$$

$$(\mathbf{A_j} + \mathbf{J} * \mathbf{K_j})\mathbf{Vcol_j} = \mathbf{VAcol_j} \tag{5.3}$$

where,

$$\mathbf{A_j} = \begin{bmatrix} -1 & G_{2,j}r_{col} & 2*G_{3,j}r_{col} & \dots & (M-1)*G_{M,j}r_{col} \\ 0 & -1 & G_{3,j}r_{col} & \dots & (M-2)*G_{M,j}r_{col} \\ 0 & 0 & -1 & \dots & (M-3)*G_{M,j}r_{col} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & -1 \end{bmatrix},$$

$$\mathbf{Vcol_j} = \begin{bmatrix} V_{1,j} \\ V_{2,j} \\ \vdots \\ V_{M,j} \end{bmatrix}, \mathbf{VAcol_j} = \begin{bmatrix} Va_{1,j} \\ Va_{2,j} \\ \vdots \\ Va_{M,j} \end{bmatrix}, \mathbf{J} = \begin{bmatrix} R_{sense} + (M-1)*r_{col} \\ R_{sense} + (M-2)*r_{col} \\ \vdots \\ R_{sense} \end{bmatrix}$$

$$\mathbf{K_j} = \begin{bmatrix} G_{1,j} & G_{2,j} & \dots & G_{M,j} \end{bmatrix}$$

Step 2: Merge column linear systems. Next, the column linear systems ($LSCol_1$ to $LSCol_N$) are merged to form a larger Column Linear System (merged-LSCol) as shown in Equation 5.4 and 5.5. We achieve this by using the *direct sum* ($\oplus$) matrix operation on matrices ($\mathbf{A_j}$ + $\mathbf{J}$*$\mathbf{K_j}$) and $\mathbf{K_j}$ to obtain block matrices **COLmat** and **Gmat**, respectively. In Equation 5.4, **CVcol** and **CVAcol** are vectors formed by concatenating $\mathbf{Vcol_j}$ and $\mathbf{VAcol_j}$ vectors, respectively. Note that, the vectors $\mathbf{Vcol_j}$ and $\mathbf{VAcol_j}$ are obtained in Step 1 (Equations 5.1 and 5.3). Further, $\mathbf{Iout_{non\text{-}ideal}}$ in Equation 5.5 is a vector representing the output currents.

$$\mathbf{COLmat} * \mathbf{CVcol} = \mathbf{CVAcol} \tag{5.4}$$

$$\mathbf{Gmat} * \mathbf{CVcol} = (\mathbf{Iout_{non\text{-}ideal}})^T \tag{5.5}$$

where,

$$\mathbf{COLmat} = \bigoplus_{j\in 1,2,..,N}(\mathbf{A_j} + \mathbf{J} * \mathbf{K_j})$$

$$\mathbf{Gmat} = \bigoplus_{j\in 1,2,..,N}(\mathbf{K_j})$$

$$\mathbf{Iout_{non\text{-}ideal}} = \begin{bmatrix} Iout_1 & Iout_2 & \dots & Iout_N \end{bmatrix}$$

$$\mathbf{CVcol} = \begin{bmatrix} \mathbf{Vcol_1^T} & \mathbf{Vcol_2^T} & \dots & \mathbf{Vcol_N^T} \end{bmatrix}^T$$

$$\mathbf{CVAcol} = \begin{bmatrix} \mathbf{VAcol_1^T} & \mathbf{VAcol_2^T} & \dots & \mathbf{VAcol_N^T} \end{bmatrix}^T$$

Step 3: Formulate row linear systems. Similar to Step 1, the row linear systems ($LSrow_1$ to $LSrow_M$) are formulated considering horizontal slices of the crossbar. We use KCL at nodes $a_{i,j}$ present in the i^{th} horizontal slice to obtain Equation 5.6 which represents the $LSrow_j$ system. In case of an MxN crossbar, we have M such row linear systems ($LSrow_1$ to $LSrow_M$).

$$\mathbf{B_i} * \mathbf{Vrow_i} = \mathbf{VrowIN_i} - \mathbf{VARow_i} \tag{5.6}$$

where,

$$\mathbf{B_i} = \begin{bmatrix} G_{i,1} & G_{i,2} & G_{i,3} & \dots & G_{i,N} \\ G_{i,1} & G_{i,2}*2 & G_{i,3}*2 & \dots & G_{i,N}*2 \\ G_{i,1} & G_{i,2}*2 & G_{i,3}*3 & \dots & G_{i,N}*3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ G_{i,1} & G_{i,2}*2 & G_{i,3}*3 & \dots & G_{i,N}*N \end{bmatrix} * r_{row},$$

$$\mathbf{Vrow_i} = \begin{bmatrix} V_{i,1} \\ V_{i,2} \\ \vdots \\ V_{i,N} \end{bmatrix}, \mathbf{VArow_i} = \begin{bmatrix} Va_{i,1} \\ Va_{i,2} \\ \vdots \\ Va_{i,N} \end{bmatrix}, \mathbf{VrowIN_i} = \begin{bmatrix} Vin_i \\ Vin_i \\ \vdots \\ Vin_i \end{bmatrix}$$

Step 4: Merge row linear systems. Next, the row linear systems obtained in Step 3 are merged to obtain a larger Row Linear System (merged-LSrow) as shown in Equation 5.7. **ROWmat** is a block matrix obtained by performing the *direct sum* ($\oplus$) matrix operation on the matrix ($\mathbf{B_i}$). Moreover, **CVrowIN**, **CVrow**, and **CVArow**

are vectors formed by concatenating vectors (obtained in Step 3) $\mathbf{VrowIN_i}$, $\mathbf{VArow_i}$, and $\mathbf{Vrow_i}$, respectively.

$$\mathbf{CVrowIN} - \mathbf{ROWmat} * \mathbf{CVrow} = \mathbf{CVArow} \tag{5.7}$$

where,

$$\mathbf{ROWmat} = \bigoplus_{i \in 1,2,..,M} (\mathbf{B_i})$$
$$\mathbf{CVrow} = \begin{bmatrix} \mathbf{Vrow_1}^T & \mathbf{Vrow_2}^T & \dots & \mathbf{Vrow_N}^T \end{bmatrix}^T$$
$$\mathbf{CVArow} = \begin{bmatrix} \mathbf{VArow_1}^T & \mathbf{VArow_2}^T & \dots & \mathbf{VArow_N}^T \end{bmatrix}^T$$
$$\mathbf{CVrowIN} = \begin{bmatrix} \mathbf{VrowIN_1}^T & \mathbf{VrowIN_2}^T & \dots & \mathbf{VrowIN_N}^T \end{bmatrix}^T$$

Step 5: Eliminate internal variables. Next, the vectors **CVAcol** and **CVcol** comprising of internal variables $Va_{i,j}$ and $V_{i,j}$, respectively, are eliminated. In order to eliminate these variables, we use the merged-LScol and merged-LSrow systems obtained in Step 2 and 4, respectively. However, the merged-LScol and merged-LSrow equations cannot be used directly due to the mismatch in their Right-Hand Sides (RHS) ($\mathbf{CVAcol} \neq \mathbf{CVArow}$). We resolve this mismatch by performing elementary row operations on Equation 5.7 to obtain Equation 5.8. Note that, the **CVrowINA** vector and the **ROWmatA** matrix are obtained by performing row switching, *i.e.*, an elementary row operation, on the **CVrowIN** vector and the **ROWmat** matrix, respectively. Next, the **CVAcol** vector is eliminated using Equations 5.4 and 5.8 to obtain Equation 5.9. Subsequently, the **CVcol** vector is eliminated using Equations 5.5 and 5.9 to yield Equation 5.10. Note that, Equation 5.11 details the **NETmat** matrix introduced in Equation 5.10.

$$\mathbf{CVrowINA} - \mathbf{ROWmatA} * \mathbf{CVcol} = \mathbf{CVAcol} \tag{5.8}$$

$$(\mathbf{COLmat} + \mathbf{ROWmatA}) * \mathbf{CVcol} = \mathbf{CVrowINA} \tag{5.9}$$

$$(\mathbf{Iout_{non\text{-}ideal}})^T = \mathbf{NETmat} * \mathbf{CVrowINA} \tag{5.10}$$

$$\mathbf{NETmat} = \mathbf{Gmat} * (\mathbf{COLmat} + \mathbf{ROWmatA})^{-1} \tag{5.11}$$

Step 6: Reduce matrix dimension. Finally, we reduce the size of matrices **NETmat** and **CVrowINA** by leveraging a key property of the **CVrowINA** vector, *i.e.*, it contains repeated elements. Recall that, the **CVrowIN** vector is formed by concatenating the $\mathbf{VrowIN_i}$ vectors (Step 4), and the **CVrowINA** vector is obtained by performing row switching operations on the **CVrowIN** vector. Since the $\mathbf{VrowIN_i}$ vector (Step 3) has repeated elements, consequently, the vectors **CVrowIN** and **CVrowINA** also have repeated elements. Exploiting this property, the columns of the **NETmat** matrix that are to be multiplied by same elements in **CVrowINA** can be summed using elementary column operations to yield a compressed **NETmatC** matrix (shown in Equation 5.12). Moreover, removing redundancies in vector **CVrowINA** leads to the ${\mathbf{Vin_{non\text{-}ideal}}}^T$ vector. Further, Equation 5.12 can be rewritten as Equation 5.13 to obtain the $\mathbf{G_{non\text{-}ideal}}$ matrix. Note that, $\mathbf{G_{non\text{-}ideal}}$ is a function of ($\mathbf{G}$, R_{sense}, r_{col}, and r_{row}), and therefore can be constructed using the intermediate matrices **COLmat**, **ROWmat**, and **Gmat**.

$$(\mathbf{Iout_{non\text{-}ideal}})^T = \mathbf{NETmatC} * (\mathbf{Vin_{non\text{-}ideal}})^T \tag{5.12}$$

$$\mathbf{Iout_{non\text{-}ideal}} = \mathbf{Vin_{non\text{-}ideal}} * \mathbf{G_{non\text{-}ideal}} \tag{5.13}$$

where,

$$\mathbf{Vin_{non\text{-}ideal}} = \begin{bmatrix} Vin_1 & Vin_2 & \dots & Vin_M \end{bmatrix}$$

$$\mathbf{G_{non\text{-}ideal}} = \mathbf{NETmatC}^T = f(\mathbf{G}, R_{sense}, r_{row}, R_{col})$$

5.3 RxNN Framework

In this section, we present a software framework RxNN that enables evaluation of large-scale DNNs on resistive crossbar systems. RxNN is a functional simulator obtained by modifying a popular deep learning framework, i.e., Caffe [91], to mimic non-ideal vector-matrix multiplications realized on resistive crossbars. Caffe models the convolution and fully-connected layers of DNNs as matrix-matrix and vector-matrix multiplications. RxNN maps these matrix-matrix and vector-matrix multiplications to a resistive crossbar system and evaluates application-level accuracy of DNN inference operations. It takes the network description, resistive crossbar system description, crossbar parameters, and a trained model as inputs, and computes the DNN accuracy using the embedded FCM models. RxNN's primary objective is to evaluate the application-level accuracy of DNNs, however, it is also capable of generating execution traces to enable performance and energy estimation. RxNN is also capable of re-training DNN to improve their inference accuracy.

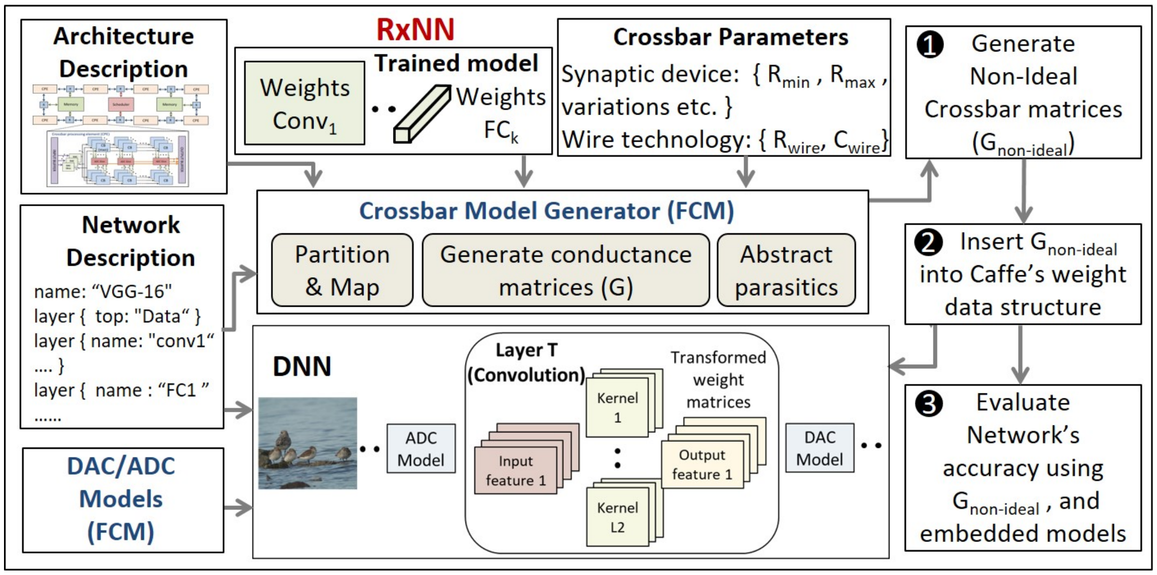

RxNN
Architecture Description
Trained model
Weights Conv1
Weights FCk
Crossbar Parameters
Synaptic device: { Rmin , Rmax , variations etc. }
Wire technology: { Rwire, Cwire}
1 Generate Non-Ideal Crossbar matrices (Gnon-ideal)
Crossbar Model Generator (FCM)
Partition & Map
Generate conductance matrices (G)
Abstract parasitics
Network Description
name: "VGG-16"
layer { top: "Data" }
layer { name: "conv1"
.... }
layer { name : "FC1 "
......
2 Insert Gnon-ideal into Caffe's weight data structure
DNN
Layer T (Convolution)
Transformed weight matrices
Kernel 1
Kernel L2
Input feature 1
Output feature 1
ADC Model
DAC Model
DAC/ADC Models (FCM)
3 Evaluate Network's accuracy using Gnon-ideal , and embedded models

Fig. 5.8.: RxNN Overview

Figure 5.8 depicts the RxNN flow that consists of 3 steps. In step ❶, RxNN maps the neural network to the specified target architecture. The weights are read from the trained Caffe model and virtually programmed into the crossbar array instances. Subsequently, the conductance matrices (G) corresponding to each resistive crossbar instance are generated, which are then transformed into the non-ideal conductance matrices ($G_{non-ideal}$) by abstracting crossbar non-idealities. Next, in step ❷, the $G_{non-ideal}$ matrices associated with each convolution and fully-connected DNN layers are incorporated back into the Caffe's original weight data structure. RxNN transparently utilizes Caffe's underlying data structures and optimized BLAS libraries, which is key to its performance and scalability. We note that steps ❶-❷ are performed only once for a given DNN and architecture template. Thereafter, in step ❸, RxNN evaluates the DNN for the given set of test inputs using embedded $G_{non-ideal}$ matrices and peripheral (ADC and DAC) models. During network evaluation, the DAC/ADC models are invoked as pre- and post-processing steps on the inputs/outputs of each convolutional and fully-connected layer.

Next, we describe re-training with RxNN to improve inference accuracy of DNN on resistive crossbar systems. The major challenges that arise during DNN re-training for crossbar systems are: (i) the data-structures (inputs, outputs, weights) should abide by the range and resolution constraints at all times, and (ii) errors and gradients computed during back-propagation should be appropriately scaled to ensure network convergence[2]. RxNN meets these constraints by using a crossbar abstracted forward pass and a floating-point based backward pass. It appropriately converts and scales the data-structures between abstractions to ensure that the network re-trains with minimal impact on the overall training speed, which is extremely critical in the context of large-scale DNNs.

5.4 Experimental Methodology

In this section, we provide the experimental setup for evaluating FCM and the software framework RxNN.

Device/Circuit simulation. We use an in-house device model of the synaptic element [64] that is based on the solution of Landau-Lifshitz-Gilbert (LLG) magneti-

[2]Stochastic-gradient descent solver assumes the forward and backward passes to be contiguous and differentiable. However, crossbar abstraction of vector-matrix multiplication does not ensure these conditions.

zation dynamics and Non-Equilibrium-Green's Function (NEGF) electron transport. Circuit-level simulations are performed in HSPICE using the 45nm bulk CMOS technology and the synaptic device model. Our simulations use the ADC and DAC circuits proposed in [27, 139]. The interconnect parasitics (r_{row}, r_{col}) are extracted using the device and crossbar array layouts. Figure 5.9 shows these layouts that are performed using the design rules specified in [140]. The Table 5.1 details the device, technology [141], and variation parameters [137] assumed in our experiments. We also characterize a resistive crossbar array to compute energy at the crossbar-level which is used as a technology parameter in RxNN to estimate system-level energy consumption.

Table 5.1.: Device and Technology parameters for evaluating RxNN

Components	**Values**

Synaptic Device	Precision = 6bits, Length = 1280 nm
	Width = 40nm, t_{ox} = 2.12 nm
	R_{min} = 200KΩ, R_{max} = 1.4MΩ

Wire Technology	Resistance = 3.3 Ω/μm, Capacitance = 0.2 fF/μm
Assumed Variation (σ/μ)	Synaptic Conductance = 5%
CMOS Technology	45nm Bulk CMOS
ADC/DAC	Precision = 6 bits
Extracted Parasitic	$r_{row} = 1\Omega$, $c_{row} = 0.05$fF $r_{col} = 4.6\Omega$, $c_{col} = 0.3$fF

Application-Level simulation. We evaluated the application-level accuracy and energy of several popular DNNs on the resistive crossbar system using RxNN. Table 5.2 details our benchmark DNNs using the number of convolution and fully-connected layers, the targeted data-set, and the number of synaptic connections and neurons. We also present the relative model size to highlight the difference between these benchmark DNNs. To evaluate the system-level energy of DNNs, we use the resistive crossbar system architecture presented in Section 3.3.

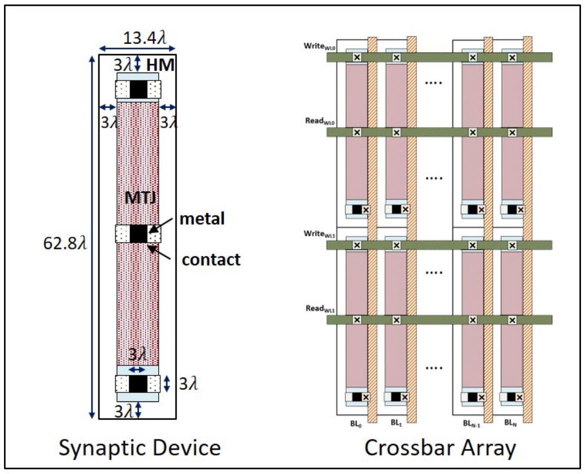


Fig. 5.9.: Layout of synaptic device and crossbar array

Table 5.2.: Benchmark DNN Applications

Data-Set	Network	#Conv Layers	#FC Layers	#Synapses (in billions)	#Neurons (in millions)	Relative Model Size
MNIST	LeNet	2	2	0.0005	0.02	1
CIFAR-10	ConvNet	3	2	0.01	0.05	20
	NiN	9	0	0.3	0.6	600
ImageNet	AlexNet	5	3	0.5	0.5	1000
	NiN	12	0	1.1	1.7	2200
	OverFeat	5	3	2.6	1.9	5200
	VGG-16	13	3	15.5	13.6	31000
	GoogleNet	59	5	1.6	3.2	3200
	ResNet-50	53	1	5.1	13.8	10200

5.5 Results

We now present the experimental results to demonstrate the modeling accuracy and speedups achieved by FCM over circuit simulation. We also evaluate the application-level accuracy of large-scale DNNs on non-ideal resistive crossbar systems using RxNN.

5.5.1 FCM: Crossbar-level Evaluation

Modeling Accuracy. Figure 5.10 shows the errors in vector-matrix multiplications realized using a 64x64 non-ideal crossbar. We compute errors using three different crossbar models, *viz.*, HSPICE, FCM, and MNSIM. HSPICE is our baseline model, whereas MNSIM [79] represents a simple error model. The X-axis represents the crossbar column, and the Y-axis depicts the error incurred during the non-ideal vector-matrix multiplication. We observe that the simple error model (MNSIM) deviates considerably from the HSPICE model. This is expected, as it does not consider errors' dependence on several factors including applied inputs, the crossbar state, and the crossbar columns. In contrast, the FCM model considers these dynamic factors and therefore able to closely match the errors observed in the HSPICE model. The maximum deviation between the errors estimated by MNSIM and the actual errors computed using HSPICE is about 3.51%. In the case of FCM, the maximum deviation is found to be significantly (0.28%) smaller.

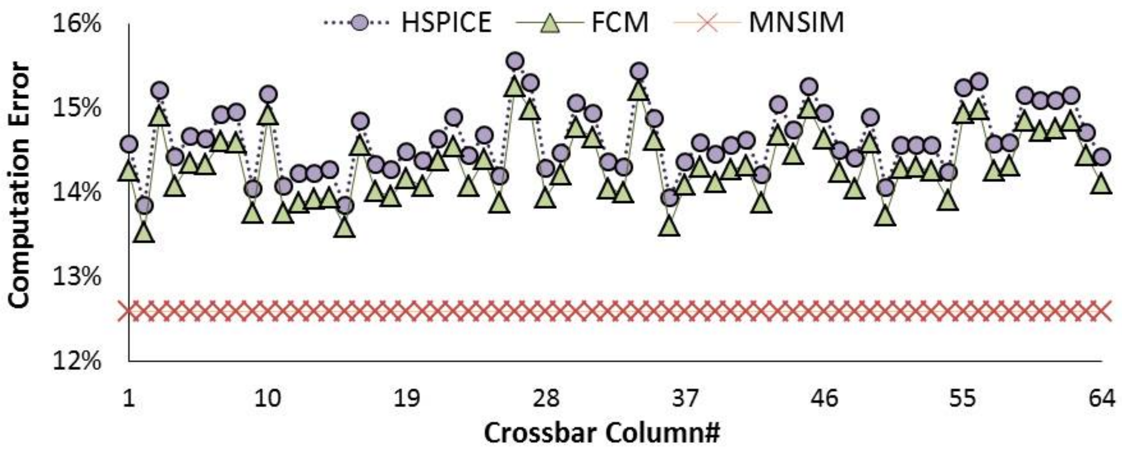


Fig. 5.10.: Computation Errors observed in crossbar for various crossbar models

Speedup. To evaluate the speedup of FCM over HSPICE, we measure the execution time of FCM and HSPICE for various crossbar sizes. Figure 5.11 details the

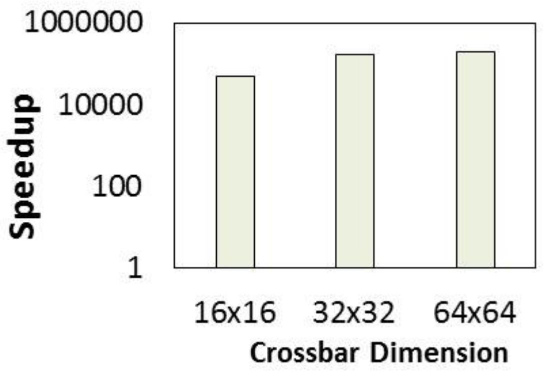


Fig. 5.11.: FCM speedup over HPSICE

speedup achieved using FCM over HSPICE. We observe a speedup of about 5 orders in magnitude across crossbar with different sizes. Moreover, as expected, the speedup increases for larger crossbar arrays.

Model generation overhead. Recall that FCM's crossbar model generator transforms the weights matrix (W) to a non-ideal conductance matrix ($G_{non-ideal}$) which incur a one-time overhead. In our evaluation, we found the modeling overhead to be 0.038, 1.2, and 61 seconds for 16x16, 32x32, and 64x64 crossbar array, respectively. While considerable for larger crossbars, these one-time overheads are amortized over a large number of inputs evaluated by the DNN model.

5.5.2 RxNN: Application-Level Evaluation

Next, we evaluate the accuracy degradation due to crossbar non-idealities at the application-level for the benchmark DNNs using RxNN. We implement three different resistive crossbar systems designed using crossbars of size 16x16 (Cross16), 32x32 (Cross32), and 64x64 (Cross64). Figure 5.12(a) shows the accuracy degradation for these designs with respect to our baseline, *i.e.*, an ideal crossbar with no device and circuit-level non-idealities. We first compare the accuracy degradation of the Cross64 design across DNNs. We observe that for simple networks (LeNet and ConvNet) the accuracy degradation due to non-idealities is quite small. For example, LeNet and

ConvNet networks suffer accuracy degradation of 0.05% and 2.2%, respectively. In contrast, the accuracy loss due to non-idealities is considerable for large-scale DNNs. For instance, VGG-16, OverFeat, and Resnet-50 networks incur accuracy losses of 25.6%, 27.8%, and 32%, respectively. We observe similar accuracy degradation trend across simple and large-scale DNNs for Cross16 and Cross32 designs as well.

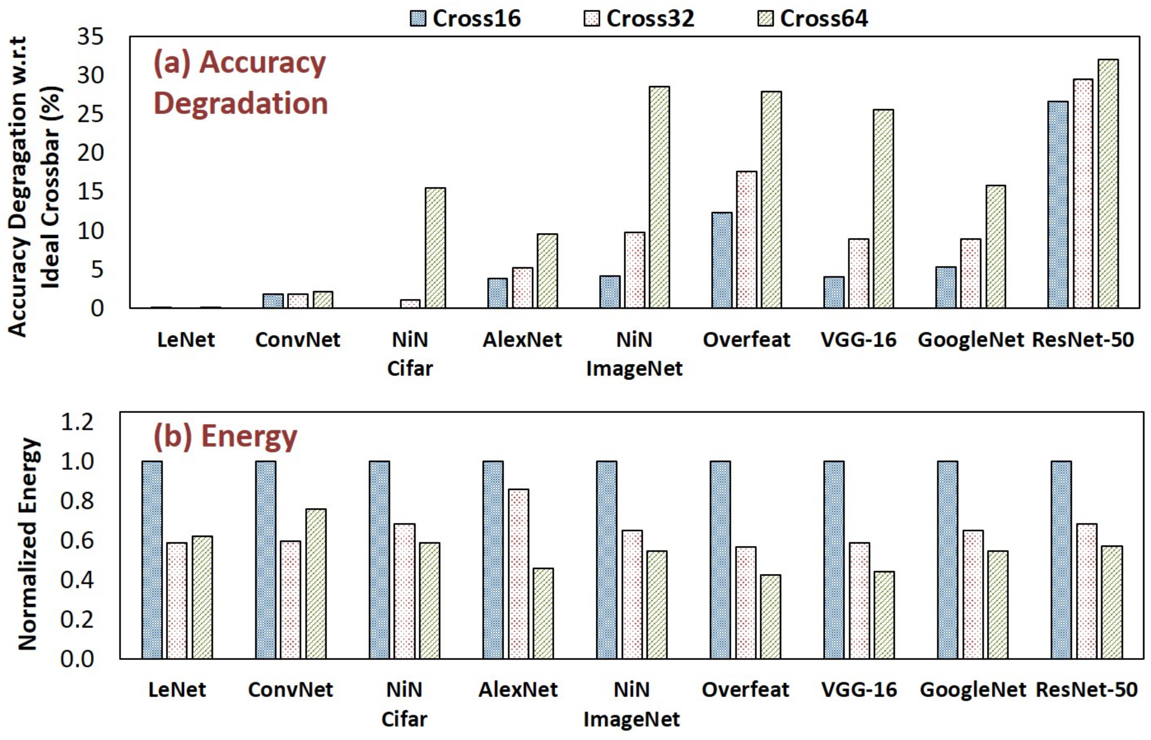


Fig. 5.12.: Application-Level evaluation using RxNN

Next, we compare the accuracy degradation across resistive crossbar system designs (Cross16, Cross32, and Cross64). As evident from Figure 5.12(a), the accuracy degradation for the Cross16 design is much lesser than the Cross32 and Cross64 designs. This trend is expected as the impact of non-idealities is lower for smaller crossbar arrays (Section 5.1.2). However, the Cross16 design consumes higher energy compared to Cross32 and Cross64 designs. Since the major components of the energy consumed in resistive crossbar systems are peripherals (ADC and DAC), therefore, larger crossbar arrays that amortize the energy cost of ADCs and DACs over more number of columns and rows have superior energy efficiency. Figure 5.12(b) depicts the normalized energy consumption per image for the Cross16, Cross32, and Cross64 designs. Note that, in LeNet and ConvNet networks the energy of the Cross64 design is higher than the Cross32 design. This is because the crossbars in the Cross64 design are under-utilized in case of these simple networks. Therefore, Cross64 suffers from energy overheads due to redundant computations performed in the unmapped columns.

To further illustrate the energy consumption of DNNs on resistive crossbar systems, we present the energy breakdown of three networks, *viz.*, VGG-16, GoogleNet, and AlexNet realized on the Cross64 design. Figure 5.13 shows the energy breakdown of these networks considering – read energy for inputs (CMOS-Mem-Read), write energy for outputs (CMOS-Mem-Write), and computation energy for vector-matrix multiplications (Cross-Computation). We observe that the major energy component is the vector-matrix multiplications (Cross-Computation) which is dominated by the ADCs and DACs.

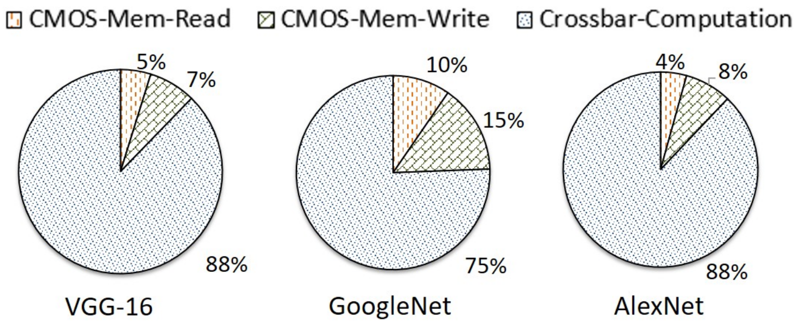


Fig. 5.13.: Energy Breakdown for Cross64 implementation

We also evaluated the slowdown of RxNN with respect to Caffe, which amounts to 2.5X and 2.75X for inference and re-training, respectively, across our benchmark applications. We believe this is a reasonable overhead given the highly optimized nature of Caffe, and the fact that much like Caffe, RxNN can also leverage multi-cores, GPUs, and clusters for increased processing throughput.

In summary, there exists a fundamental trade-off between the application-level accuracy and the system energy which needs to be examined, in order to determine the architectures for future resistive crossbar systems. RxNN intends to drive these decisions by providing a software platform that can precisely evaluate crossbar architectures executing large-scale DNNs.

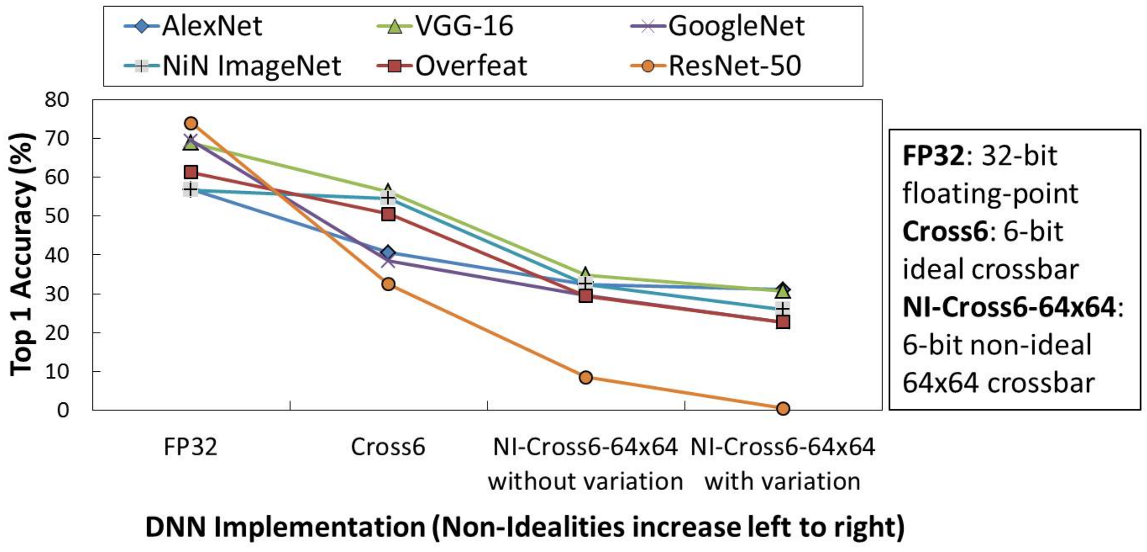


Fig. 5.14.: Accuracy's sensitivity to non-idealities

5.5.3 Sensitivity of accuracy to non-idealities

To further illustrate the impact of non-idealities on the application-level accuracy, we present a sensitivity analysis in Figure 5.14. We plot the accuracies of 6 large-scale networks, *viz.*, AlexNet, VGG-16, GoogleNet, NiN, Overfeat, and ResNet-50 for implementations differing in their degree of non-idealities. The implementations that we use are: (i) floating-point implementation realized on an x86 CPU architecture (FP32), (ii) 6-bit ideal crossbar design (Cross6) without any crossbar non-idealities, and (iii) 6-bit non-ideal crossbar based designs with and without variations (NI-Cross6-64x64). Note, FP32 is a CMOS-based digital implementation that does not use crossbars. As shown in Figure 5.14, the accuracy drops from left to right as more non-idealities are incorporated. We observe two significant accuracy drops, one between FP32 and Cross6 implementations, and other between Cross6 and NI-Cross6-64x64 implementations. The degradation between FP32 and Cross6 is due to the limited precision of the synaptic devices, ADCs, and DACs. This drop in accuracy can be reduced by re-training as discussed in Chapter 6. In contrast, the drop in accuracy from Cross6 to NI-Cross6-64x64 is due to the device and circuit-level non-idealities.

5.5.4 Impact of non-idealities: Insight

To provide further insights into the impact of non-idealities at the application-level, Figure 5.15 compares the output features obtained from an ideal resistive crossbar system and a non-ideal resistive crossbar system, respectively, for two convolution layers (Conv1 and Conv3) of the ConvNet network executing on the CIFAR-10 dataset. Some of the significant distortions in features are identified in the figure using circles. We observe that the impact of non-idealities increases noticeably as we go deeper into the network (Conv3 layer outputs show increased artefacts compared to Conv1 layer outputs in Figure 5.15). This is consistent with the observation from Figure 5.12(a) that deeper DNNs show greater degradation in accuracy due to crossbar non-idealities.

In summary, our results underscore the utility of RxNN in evaluating and re-training large-scale DNNs on resistive crossbar architectures. They also motivate the need for further research into techniques to mitigate and compensate for the effects of crossbar non-idealities in the context of large-scale DNNs.

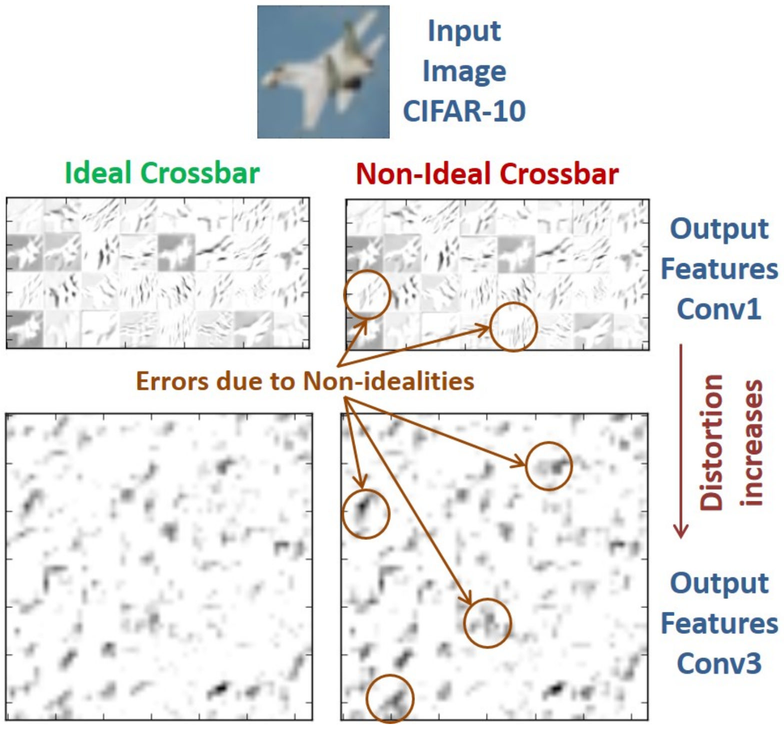


Fig. 5.15.: Visual demonstration of errors using ConvNet

5.6 Summary

Resistive crossbar systems are a promising solution to the energy efficient realization of DNNs. In this work, we evaluate the impact of various device and circuit non-idealities that are present in crossbars on the overall accuracy of large-scale DNNs. We propose FCM, *i.e.*, a fast, scalable, and accurate functional crossbar model to evaluate vector-matrix multiplications realized on resistive crossbars. We present RxNN, a software simulation framework to enable evaluation and re-training of large-scale DNNs on resistive crossbar systems. Our evaluations show that the errors due to non-idealities can degrade the overall accuracy of large scale DNNs considerably, therefore necessitating a need for error correction and compensation schemes.

6. HARDWARE-SOFTWARE COMPENSATION METHODS FOR DEEP NEURAL NETWORKS ON RESISTIVE CROSSBAR SYSTEMS

As mentioned in Chapter 5, a key challenge with resistive crossbars is that the computed function (input voltages multiplied with the weights stored as conductances to obtain output currents) is only an approximation of the desired vector-matrix multiplications. In reality, the computed outputs can deviate significantly from the desired value due to limited precision, as well as numerous device and circuit-level non-idealities, *viz.*, interconnect resistance, sensing and driver resistances, noise in synaptic devices, ADC and DAC non-linearity, sneak paths, imperfect write operations, and process variations [69–73,75,76,78,79,124,126,142]. The cumulative effect of all these non-idealities manifests as errors in the vector-matrix multiplication that depend on data (programmed weights and applied inputs), how the computation is mapped to the crossbar (since errors vary across crossbar columns) and the hardware instance performing the computation (due to variations). These errors in vector-matrix multiplications can accumulate over numerous crossbar operations within a DNN layer and propagate through the different network layers, eventually impacting the accuracy of the DNN. The overall impact of non-idealities on DNN accuracy strongly depends on the scale of the network. The Figure 5.14 in Section 5.5.3 illustrates the accuracy degradation due to limited precision and device and circuit-level non-idealities, for simple and complex DNNs. We can observe that the accuracy degradation to be minimal for simple networks (LeNet and ConvNet) with smaller and fewer layers. However, the accuracy degradation can be substantial (19.8%-57.8%) for large-scale DNNs such as VGG-16, GoogleNet and ResNet-50. *Therefore, it is essential to address the challenges posed by crossbar non-idealities to enable adoption of resistive crossbar based systems.*

Prior efforts that have addressed non-idealities in resistive crossbars are limited by the scale of networks that they consider (*there is no reported work on ImageNet DNNs such as AlexNet, VGG-16, GoogLeNet and ResNet*), and only mitigate a subset of crossbar non-idealities [69–73, 75, 76, 78]. Re-training DNNs using software models of the crossbar is a popular approach to compensating for non-idealities [69, 71, 73, 75, 126–128]. However, re-training can only mitigate accuracy degradation to a limited extent. Moreover, a key limitation of re-training based compensation is that each hardware instance requires separate model re-training and the trained network is tied to the specific hardware instance. Breaking the train-once-deploy-anywhere tenet of current DNN applications raises significant challenges. For example, consider a potential deployment of resistive-crossbar-based inference accelerators in a million edge devices. Deployment of a new DNN on these devices would require characterizing the non-idealities of each instance, communicating these non-idealities to a cloud-based infrastructure, invoking a million runs of re-training in the cloud, and transfering a unique model back to each device, thereby greatly raising deployment complexity and computational requirements. Compensation mechanisms without re-training have only been explored in the limited context of small-scale auto-associative neural networks to mitigate specific non-idealities such as imperfect programming and drift in synaptic conductances [72, 76]. Therefore, there is a need for a design methodology that: (i) can effectively compensate for errors due to a wide range of crossbar non-idealities, (ii) does not use hardware instance specific training, (iii) incurs low hardware overhead, and (iv) is scalable to large DNNs.

We propose CxDNN, a hardware-software methodology to realize high accuracy DNNs on resistive crossbar systems. CxDNN is composed of a conversion algorithm, a fast re-training method, and a low-overhead hardware compensation scheme. The conversion algorithm takes any given floating-point DNN and converts its weights to conductances and activation values to voltages by determining appropriate scaling factors. Subsequently, CxDNN uses a fast re-training method to recover accuracy loss due to the conversion with very few iterations[1]. We note that our re-training method is hardware instance independent and therefore needs to be performed only once overall (and not once per instance). Finally, CxDNN mitigates data-dependent and hardware-instance-specific errors using low-overhead error compensation circuits that post-process the crossbar outputs. We evaluate CxDNN using 6 state-of-the-art DNNs on the ImageNet dataset. Our results show that CxDNN improves top-1 accuracy by 16%-49% and obtains classification accuracies comparable to a software fixed-point implementation, thereby alleviating non-idealities as an impediment to the use of resistive crossbar based neural fabrics.

We organize the rest of the chapter as follows. Section 6.1 details the proposed CxDNN compensation methodology. We outline the experimental methodology in section 6.2. The experimental results are presented in section 6.3, and section 6.4 concludes the chapter.

6.1 CxDNN: Compensation methods for DNNs on resistive crossbars

In this section, we present CxDNN, a hardware-software methodology to realize DNNs with high accuracy on *resistive crossbar based accelerators for DNN inference.* We first motivate the approach adopted by CxDNN and subsequently detail the software and hardware compensation methods.

6.1.1 CxDNN: Approach and Overview

Figure 6.1 contrasts the design approach adopted by CxDNN with prior efforts by highlighting the difference in their usage models. We consider a scenario in which XPUs are deployed on millions of mobile and IoT devices to accelerate DNN inference. As shown in Figure 6.1(a), prior schemes that rely on (re-)training to mitigate hardware-instance specific errors will have to characterize non-idealities for each in-

[1]Training ResNet-50 on the ImageNet dataset requires ~365000 iterations with a batch size of 50. In comparison, CxDNN requires <500 iterations

stance and communicate these characteristics to a cloud-based infrastructure. Subsequently, a run of re-training is invoked for each device, and a unique model is transferred back to each device. Therefore, these schemes are not scalable. In contrast, CxDNN preserves the train-once-deploy-anywhere tenet of current DNN implementations (shown in Figure 6.1(b)). It designs DNNs that are re-trained once and deployed on many XPU instances. Further, the XPU instances (XPU^+) are equipped with low-overhead compensation hardware to mitigate instance-specific errors.

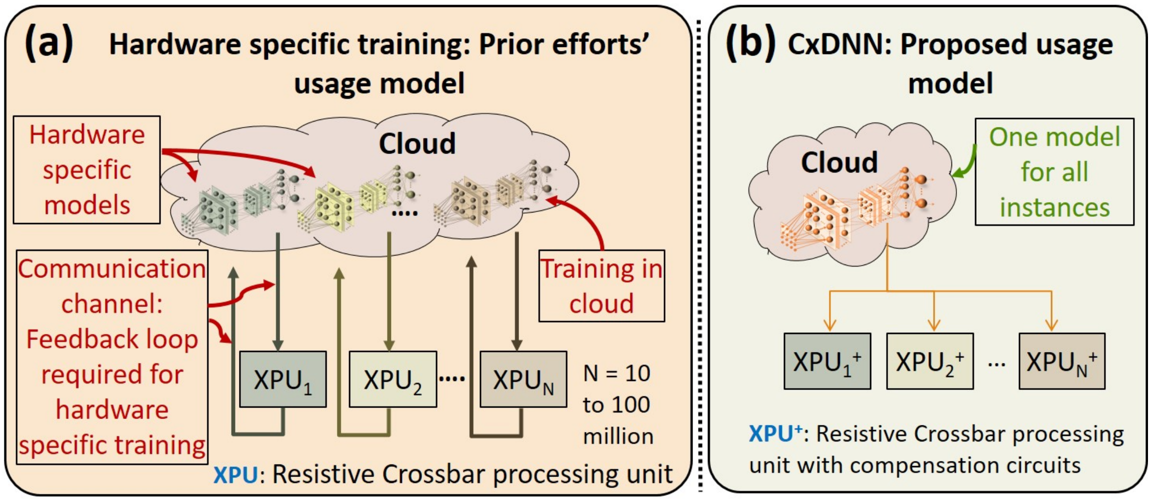

(a) Hardware specific training: Prior efforts' usage model
Hardware specific models
Cloud
Communication channel: Feedback loop required for hardware specific training
Training in cloud
XPU1
XPU2
XPUN
N = 10 to 100 million
XPU: Resistive Crossbar processing unit
(b) CxDNN: Proposed usage model
Cloud
One model for all instances
XPU1+
XPU2+
XPUN+
XPU+: Resistive Crossbar processing unit with compensation circuits

Fig. 6.1.: CxDNN usage model

Figure 6.2 outlines the CxDNN compensation flow, which consists of a quantization and conversion algorithm, hardware-independent re-training, and hardware compensation. CxDNN takes a readily available floating-point (FP32) DNN as an input. It first quantizes the weights and activations of the FP32 network to match the precisions of the synaptic devices and ADCs/DACs, resulting in a fixed-point (FxP) DNN. Subsequently, it transforms the weights into conductances and activations to voltages, to obtain a DNN for an "ideal" crossbar (with no device and circuit non-idealities other than limited precision), denoted DNN_{XI}. CxDNN then uses a fast hardware-instance-independent re-training method to recover accuracy lost during the conversion process. Next, CxDNN maps the DNN_{XI} to a resistive crossbar system that suffers from crossbar non-idealities (denoted DNN_{XNI}), which results in further degradation in accuracy. CxDNN overcomes this accuracy drop using a hardware compensation scheme. Note that, DNN_{XI} can also be obtained using end-to-end DNN training, *i.e.*, used for designing FP32 DNNs. However, training is computationally very expensive, requiring exa-ops of compute and hence days-to-weeks to complete. In contrast, CxDNN designs DNN_{XI} by converting FP32 models and utilizing a fast re-training method to recover accuracy loss during conversion with very few iterations[2]. We next describe CxDNN's software and hardware methods in detail.

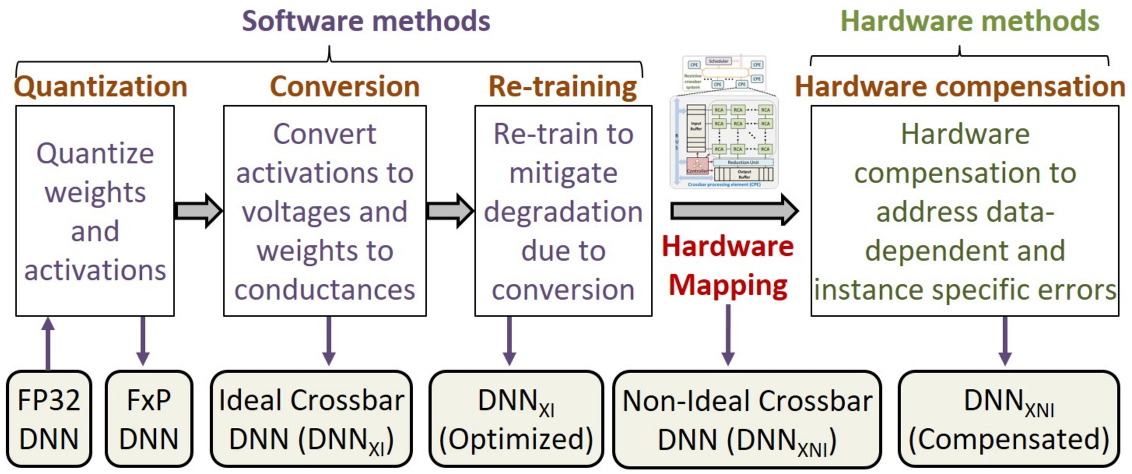

Software methods
Hardware methods
Quantization
Conversion
Re-training
Hardware compensation
Quantize weights and activations
Convert activations to voltages and weights to conductances
Re-train to mitigate degradation due to conversion
Hardware Mapping
Hardware compensation to address data-dependent and instance specific errors
FP32 DNN
FxP DNN
Ideal Crossbar DNN (DNN_XI)
DNN_XI (Optimized)
Non-Ideal Crossbar DNN (DNN_XNI)
DNN_XNI (Compensated)

Fig. 6.2.: CxDNN Overview

6.1.2 Conversion algorithm

CxDNN uses a two-step conversion algorithm to convert a floating-point DNN (DNN_{FP}) to a DNN for ideal crossbar (DNN_{XI}). Algorithm 1 details the pseudo code for the conversion algorithm. It takes as inputs DNN_{FP}, precision constraints for weights and activations (pW, pA), conductance (g_{max}, g_{min}) and voltage (v_{max},

[2]Training ResNet-50 on the ImageNet dataset requires ~365K iterations with a batch size of 50. In comparison, our re-training requires <500 iterations

v_{min}) ranges, and a validation dataset ($ValData$), and produces DNN_{XI} as its output. First, CxDNN quantizes DNN_{FP} to get an optimized fixed-point DNN (DNN_{FxP}) (lines 2). Subsequently, it converts DNN_{FxP} to DNN_{XI} by transforming weights to conductances (lines 3-10). To avoid hardware overhead incurred by variable precision, we assume the same precision for all layers. However, Algorithm 1 can be easily extended to support variable precision across DNN layers and data-structures (weights and activations).

Algorithm 1 DNN Conversion for an ideal crossbar

```
Input: $DNN_{FP}$: floating-point DNN, [pW,pA]: weights and activation precision,
    ValData: Validation Dataset, [$g_{max}$, $g_{min}$]: Conductance range, [$v_{max}$, $v_{min}$]: volt-
    age range
Output: $DNN_{XI}$: DNN converted for ideal crossbar
 1: Begin
 2: $DNN_{FxP}$ = convertToFxP ($DNN_{FP}$, ValData, pW, pA) // $DNN_{FxP}$: Fixed-
    Point DNN
 3: [$\alpha_g$, $\beta_g$] = get-Conversion-Params ($g_{max}$, $g_{min}$, pW)
 4: $DNN_{XI} = DNN_{FxP}$
 5: for Each layer k: 1 to N do
 6:     $DNN_{XI}$= convertToConductances ($DNN_{XI}$,$\alpha_g$, $\beta_g$, k)
 7:     [$WD_k$, $AD_k$]=get_magnitude_distribution ($DNN_{XI}$, ValData,k)
 8:     [$SF_{in}$,$SF_{out}$] = getOptScalingFactor ($DNN_{XI}$, ValData, k, $v_{max}$, $v_{min}$, pA,
        $WD_k$, $AD_k$)
 9:     $DNN_{XI}$ = applyScalingFactor ($DNN_{XI}$, $SF_{in}$, $SF_{out}$, k)
10: end for
11: Return $DNN_{XI}$
```

To convert DNN_{FP} to DNN_{FxP} (line 2), several algorithms have been proposed in recent years [143–146]. We adopt the quantization method proposed in [143], wherein we explore the conversion design space (trade-off between range and resolution) to identify the best FxP format for each DNN layer and data-structure. Next, we convert DNN_{FxP} to DNN_{XI} using the weight-to-conductance conversion parameters (α_g, β_g) and scaling factors (SF_{in}, SF_{out}) that are used for converting digital inputs to voltages and currents to digital outputs, respectively. Figure 6.3 depicts our DNN_{FxP} to DNN_{XI} conversion process. We develop a statistics-driven approach to identify the best scaling factors (SF_{in}, SF_{out}) for each DNN layer. For each layer in the network, we derive weight and activation distributions (line 7) by simulating DNN_{XI} with the validation dataset and determine a list of possible scaling factors. We systematically explore these choices and identify the scaling factors that yield the highest accuracy (line 8). We convert each layer separately starting with the first DNN layer, and once all layers are converted DNN_{XI} is returned (line 11).

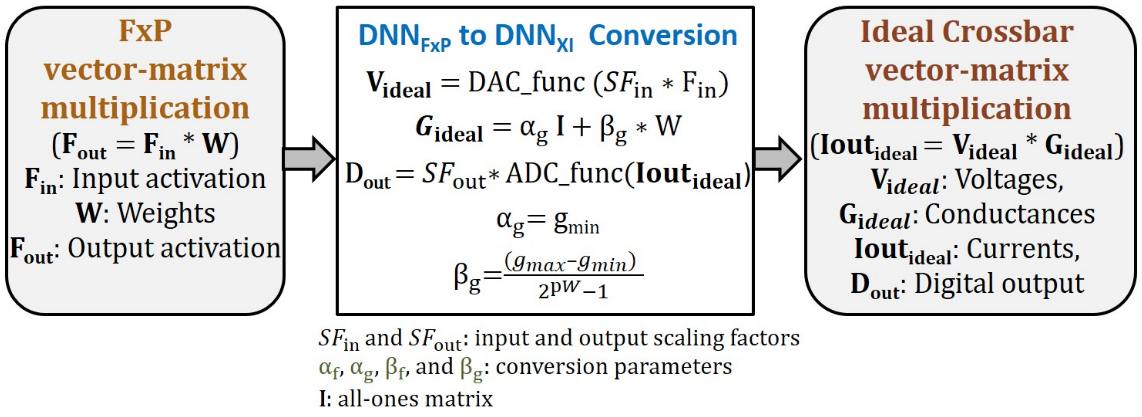

FxP vector-matrix multiplication
$(F_{out} = F_{in} * W)$
F_{in}: Input activation
W: Weights
F_{out}: Output activation
DNN_{FxP} to DNN_{XI} Conversion
$V_{ideal} = \text{DAC\_func}(SF_{in} * F_{in})$
$G_{ideal} = \alpha_g I + \beta_g * W$
$D_{out} = SF_{out} * \text{ADC\_func}(Iout_{ideal})$
$\alpha_g = g_{min}$
$\beta_g = \frac{(g_{max} - g_{min})}{2^{pW} - 1}$
Ideal Crossbar vector-matrix multiplication
$(Iout_{ideal} = V_{ideal} * G_{ideal})$
V_{ideal}: Voltages,
G_{ideal}: Conductances
$Iout_{ideal}$: Currents,
D_{out}: Digital output
SF_{in} and SF_{out}: input and output scaling factors
α_f, α_g, β_f, and β_g: conversion parameters
I: all-ones matrix

Fig. 6.3.: FxP to ideal crossbar conversion function

6.1.3 Re-training method

The conversion of DNN_{FxP} to DNN_{XI} is not perfect and leads to accuracy loss due to two major factors: (i) non-linear ADC and DAC functions introduce distortions in the computed outputs, and (ii) the representation of numerical '0' with a non-zero (g_{min}) conductance for weights and a non-zero voltage (v_{min}) for inputs. For example, multiplications with zero values produce '0' in FxP but yield a non-zero value in ideal crossbar based implementation. Further, the magnitude of the non-zero value is not constant, rather data-dependent. Therefore, the output (D_{out}) computed using ideal crossbar differs from the output (F_{out}) calculated using FxP. Figure 6.4 shows the impact of the incurred data distortion during conversion by highlighting the differences in the distribution of the layer outputs for 6-bit FxP (DNN_{FxP6}) and ideal crossbar (DNN_{XI-6}) implementations of the VGG-16 network. We observe that for the same primary input (shown of the left), the histograms of the output activations get distorted for DNN_{XI-6} design in comparison to DNN_{FxP6}. Moreover, the distortions propagate through DNN layers and increase as we go deeper into the network, eventually impacting DNN accuracy. For example, the output distribution of CONV1-2 layer shows higher distortion than the CONV1-1 layer outputs.

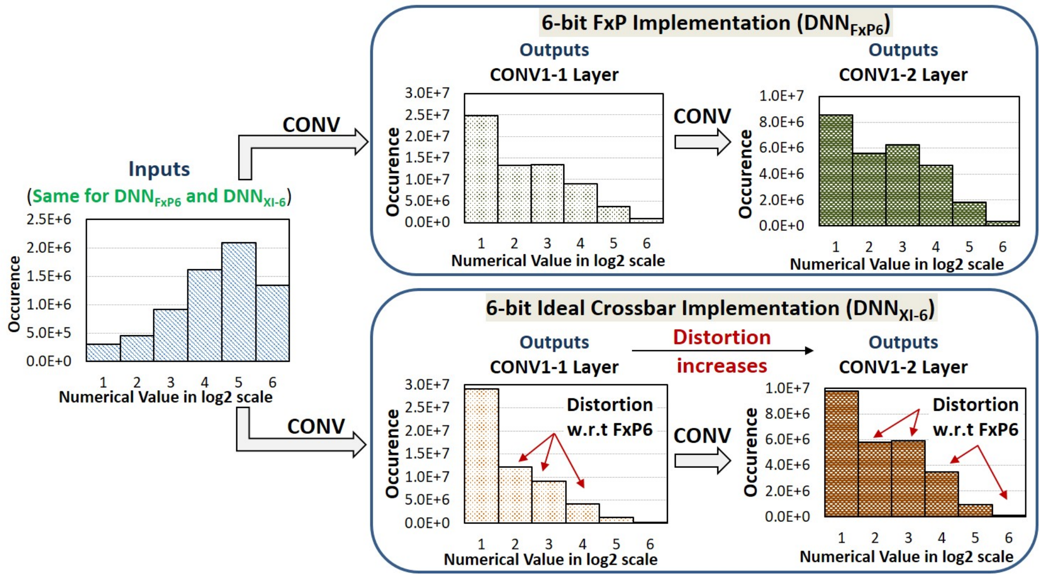


Fig. 6.4.: Activation distortion during conversion

To recover the accuracy lost during conversion, CxDNN uses a hardware-instance-independent re-training method. DNN training consists of three major steps - forward propagation, backward propagation and gradient computation followed by weight update. CxDNN uses a software model of the resistive crossbar during the forward propagation step, reflecting the fact that we are modeling inference on a resistive crossbar based hardware fabric, and a floating point (FP32) model for the backward propagation, gradient computation, and weight update steps.

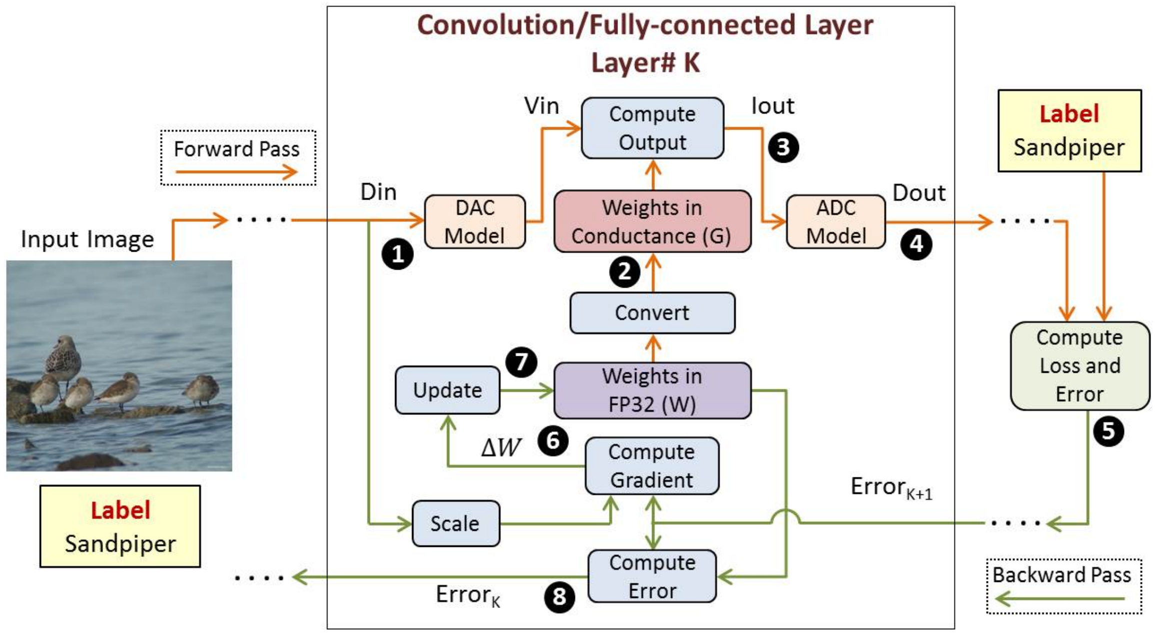


Fig. 6.5.: Re-training method for DNNs on crossbar system

Figure 6.5 presents the CxDNN re-training process that uses a conductance (G) weight matrix during the forward pass and FP32 weight matrices during the backward pass. The DAC and ADC models apply the scaling factors and convert digital inputs (Din) to input voltages (Vin) and crossbar output currents (Iout) to digital output activations (Dout). We synchronize both weight matrices after each mini-batch iteration by converting the updated FP32 weights to conductances. Figure 6.5 illustrates the main steps in the forward and backward passes (steps 1-4 and steps 5-8, respectively). First, we convert inputs (Din) and weights to voltages (Vin) and conductances (G) (Step 1-2). Next, we perform the crossbar based vector-matrix multiplication to obtain output currents (Iout) (Step 3). The ADC model converts the output currents to digital outputs (Dout) (Step 4). We perform Steps 1-4 for each convolution and fully-connected DNN layer. On completion of the forward pass, the loss and error functions are computed using the final layer outputs and a label (Step 5). Next, we execute backward pass, gradient computation, and weight update functions for each DNN layer. Figure 6.5 illustrates these operations for the K^{th} DNN layer, wherein the gradients (ΔW) (Step 6) and output errors ($Error_K$) (Step 8) are computed using the errors from layer K+1 ($Error_{K+1}$). Next, the computed error ($Error_K$) propagates in the backward direction to the previous layer (K-1), and the FP32 DNN model is updated using the computed gradient (ΔW) (Step 7). We re-train DNN_{XI} by repeating Steps 1-8 for a small set of images to compensate the accuracy drop during conversion.

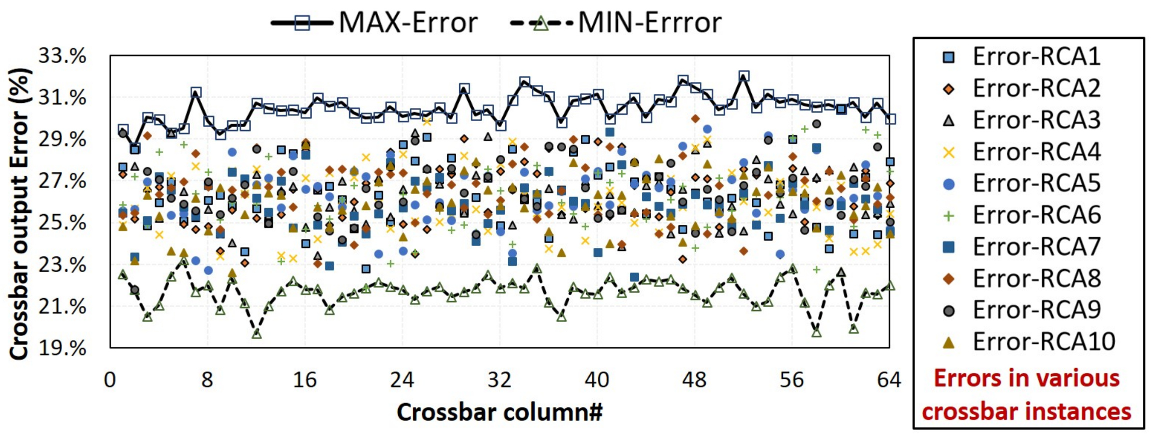


Fig. 6.6.: Error characteristics across crossbar instances and columns

6.1.4 Hardware compensation

As mentioned in Section 6.1.1, DNNs executed on crossbar based systems will suffer from accuracy degradation due to data-dependent and hardware-instance-specific non-idealities. CxDNN employs a hardware compensation method to overcome this degradation. Our hardware compensation is motivated by a key observation that errors in vector-matrix multiplications realized using resistive crossbars result from the cumulative effect of all non-idealities. It is not possible to isolate the effect of individual non-idealities on the executed vector-matrix multiplication as all non-idealities kick in simultaneously. Therefore, we propose error compensation at the crossbar level, where the outputs of the realized vector-matrix multiplications are compensated using Compensation Factors (CFs). CxDNN uses a separate compensation factor for each crossbar column of each crossbar instance, which allows many degrees of freedom in tuning the compensation process. The use of per-column compensation factors is motivated by the error characteristics in Figure 6.6 that show a significant variance across both crossbar instances and columns within an instance.

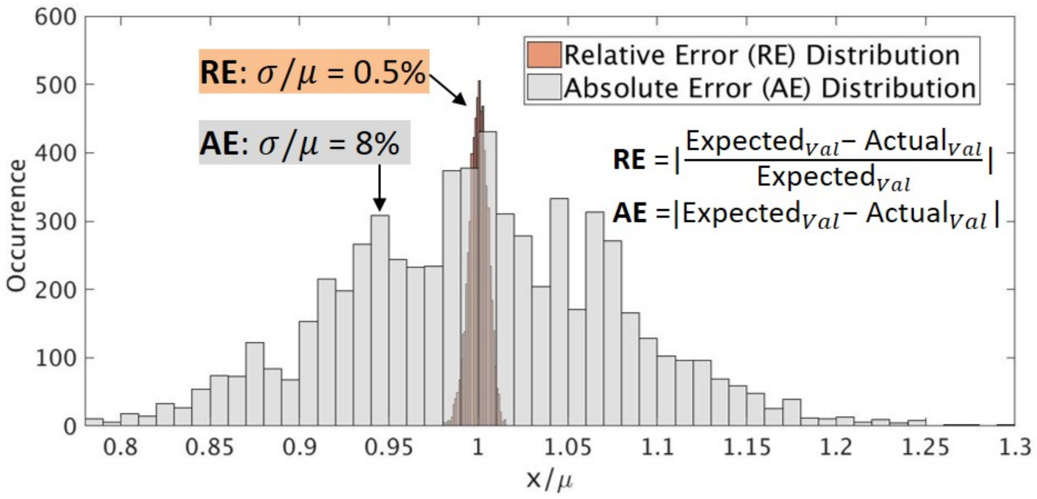


Fig. 6.7.: Histogram of absolute and relative errors at crossbar columns

Next, we determine the type of compensation factor that should be used to mitigate errors due to crossbar non-idealities based upon insights from the error characteristics at the crossbar-level. To that end, we compute errors at each crossbar column by considering all crossbar non-idealities and plot the error histogram. We examine two error metrics - relative error (RE) and absolute error (AE). Figure 6.7 shows the histogram of RE and AE normalized with their respective mean (μ), across these input vectors. We observe that the absolute errors vary substantially across different input vectors as they have large variance ($\sigma/\mu = 8\%$) with a spread of 0.78μ to 1.3μ. In contrast, the relative errors show significantly lower spread and have very small variance ($\sigma/\mu = 0.5\%$). Based on this observation, we can conclude that additive compensation factors where a constant offset is added to correct the errors at crossbar columns will not work as the absolute errors show significant input dependence. Further, it also suggests that we can effectively compensate errors by selecting multiplicative compensation factors (CFs) computed using the mean relative error (RE_{mean}). For example, we can compute the compensation factor (CF_i) for the i^{th} crossbar column using RE_{mean} as shown in Equation 6.1, and correct errors using CF_i as a multiplicative factor (shown in Equation 6.2).

$$CF_i = 1/(1 - RE_{mean}) \tag{6.1}$$

$$RE_{mean} = \frac{\sum_{j=1}^{k} \frac{|Expected_{val-j} - Actual_{val-j}|}{|Expected_{val-j}|}}{k}$$

$$Compensated_{val-i} = Actual_{val-i} * CF_i \tag{6.2}$$

CxDNN's hardware compensation method consists of two phases - a calibration phase to determine CFs and a runtime phase to mitigate errors using these CFs. The host processor initiates the calibration phase after the DNN_{XI} is mapped and programmed to the resistive crossbar system. The calibration phase consists of three main steps. In step 1, the host sends predetermined inputs (Inp) to each RCA in the resistive crossbar system. In step 2, RCAs perform vector-matrix multiplications using these calibration inputs (Inp) and stored weights to obtain the actual (erroneous) output vectors (Out-act) and subsequently send them to the host. In step 3, the host processor computes compensation factors using the actual outputs (Out-act) and the ideal outputs (Out-idl). The host processor could either store or generate the calibration inputs (Inp) and expected outputs (Out-idl). We generate the calibration inputs using a characterization dataset. We note that our method does not restrict the nature or size of the characterization dataset – it can be as large as required, subject to the characterization time being acceptable. In our experiments, we found that a small subset of the validation set was sufficient to compute compensation factors that led to the accuracy improvements reported in the thesis. Figure 6.8 depicts the CF computation process for an RCA using actual and ideal outputs (Out-idl). As shown in the figure, we compute a vector of CFs, one for each column. The computed CFs are then sent back to the respective crossbars to be stored locally in the compensation logic. While all RCAs execute vector-matrix multiplications in parallel, the host processor computes the compensation factor for each RCA sequentially. Therefore, the overall calibration time is dominated by the host processor. Further, the calibration time depends on the total number of RCAs utilized by the DNN. In our evaluation, the overall calibration time was 0.2 sec to 1.3 sec across our benchmark DNNs for a single core Intel-Atom C2350 as our host processor. We invoke the calibration phase whenever a new DNN is mapped to the XPU system, and subsequently execute millions of inference operations during the runtime phase. We can also periodically invoke the calibration phase to cope with aging issues and drift in synaptic conductances. Aging and synaptic drifts are very slow phenomena, hence the calibration phase will still need to be performed rarely (*e.g.*, once every 10K-100K inference operations). The execution time required for calibration is negligible when amortized over 10K-100K inference operations. We also note that the computed CFs are not exact but approximate factors that compensate the errors effectively with limited hardware overhead, as discussed next.

Figure 6.9 shows an RCA with the compensation logic, which consists of registers for storing CFs, a column mux, and a multiplier. Using the compensation logic, we mitigate errors in vector-matrix multiplications at runtime to obtain compensated outputs given by Out-comp = Out-act * CF. Further, we hide the latency of the compensation logic using a two-stage pipeline, where we pipeline crossbar evaluation and error compensation operations. We note that the slowest stage, which determines the pipeline throughput, is still the crossbar evaluation stage [125]. In our evaluation, the latency of the crossbar and the compensation logic are $\sim$60ns and $\sim$0.78ns, respectively. We require $\sim$49ns to compensate 64 (all) columns of an RCA using the compensation logic. Thus, the compensation logic adds no throughput penalty.

Further, we observe the compensation logic area and power overheads to be 7.3% and 4%, respectively, at the RCA level, and 5.5% and 3.5%, respectively, at the system level. We consider these overheads to be reasonable considering the accuracy improvements they enable. The major contributor to the overhead are the additional registers required for storing the CFs.

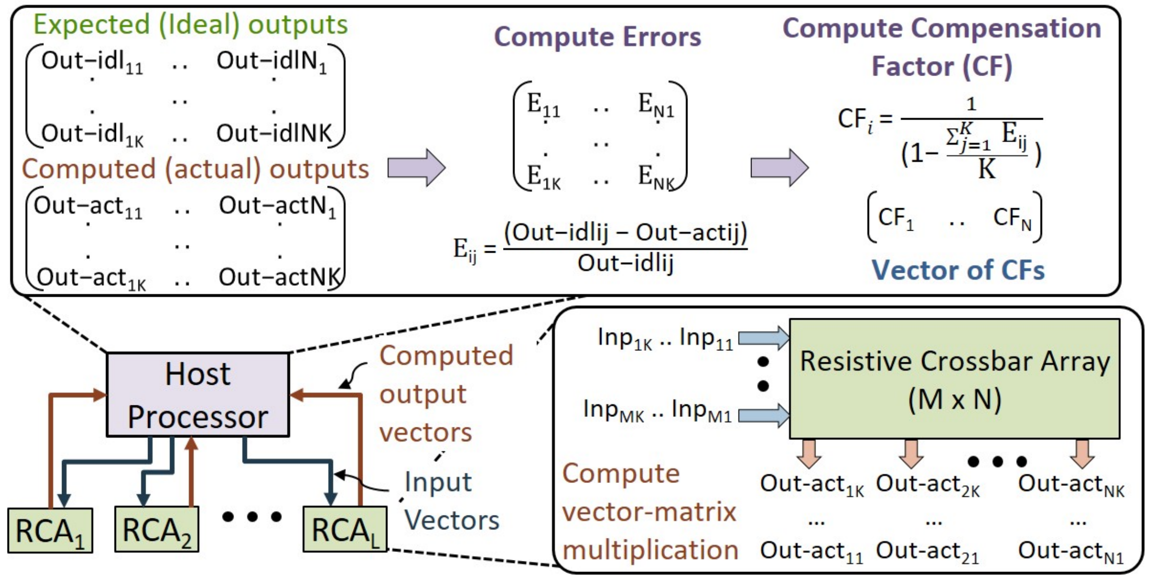

Expected (Ideal) outputs
Computed (actual) outputs
Compute Errors
Compute Compensation Factor (CF)
Vector of CFs
Host Processor
Computed output vectors
Input Vectors
RCA$_1$ RCA$_2$ RCA$_L$
Resistive Crossbar Array (M x N)
Compute vector-matrix multiplication

Fig. 6.8.: CxDNN Hardware compensation: Calibration phase

We note that CxDNN offers multiple design choices that differ based on which non-idealities are compensated in software and which are compensated in hardware. In our implementation, we compensated for limited precision and conversion of weights to conductances and activation values to voltages/currents in software (through re-training), leaving the remaining non-idealities to be compensated in hardware. Alternative design choices are possible where additional non-idealities (e.g., sneak paths) are included during (re-)training to obtain a compensated DNN model. However, as motivated in Section 6.1.1, we cannot model instance-specific errors (*e.g.*, chip-to-chip variations) during re-training as it ties the trained network to the specific hardware instance. This breaks the train-once-deploy-anywhere tenet of current DNN applications. Therefore, the overhead of hardware compensation to mitigate instance-specific errors must be incurred irrespective of the degree of non-idealities considered during re-training.

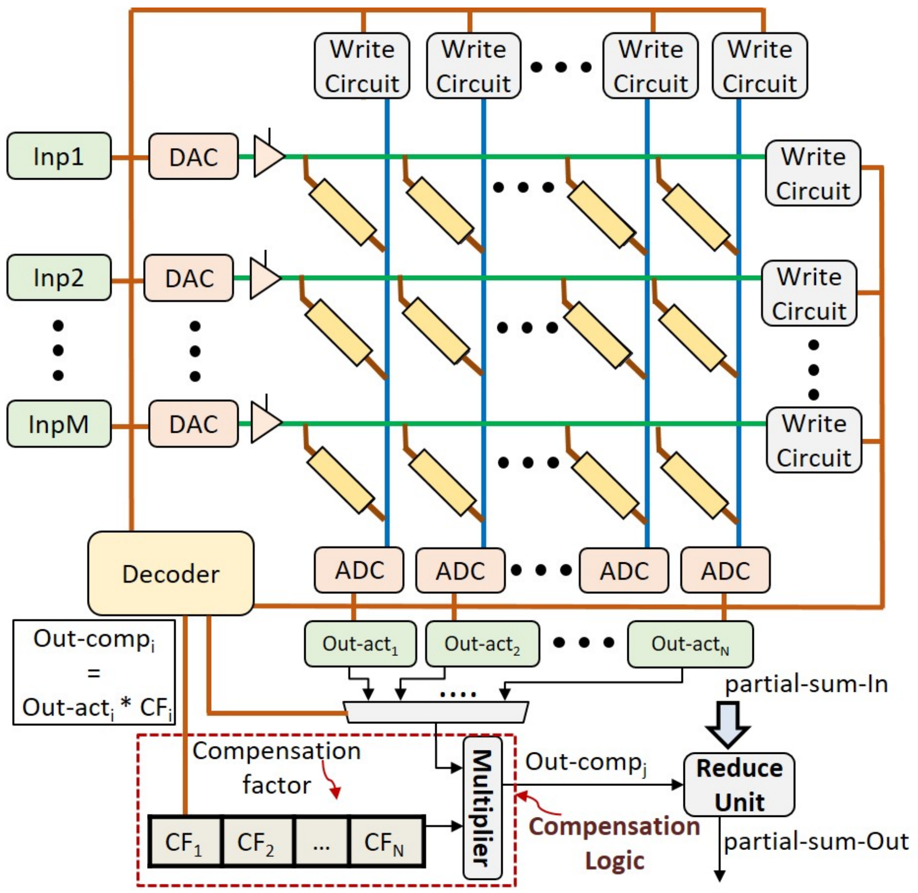


Fig. 6.9.: Hardware compensation logic of CxDNN

6.2 Experimental Methodology

In this section, we present the experimental framework used for evaluating CxDNN.

Accuracy evaluation. To evaluate the application-level accuracy of DNNs on resistive crossbar systems, we use RxNN framework proposed in Chapter 5. The framework is enhanced with the CxDNN design methodology. It takes a resistive crossbar system description as an input and maps convolution (Conv) and fully-connected (FC)

DNN layers to this system. Like Caffe, it models Conv and FC layers as matrix-matrix and vector-matrix multiplications. It functionally evaluates DNNs by mimicking the vector-matrix multiplications realized in resistive crossbars to obtain application-level accuracy. We consider several non-idealities, *viz.*, wire resistances, driver and sensing resistances, non-ideal DACs and ADCs, sneak paths, imperfect write operations, device-level noise (e.g., thermal noise) and process variations in our crossbar model. We provide the modeling details in Table 6.1. We use an effective Gaussian noise function to model errors due to non-ideal synaptic devices. Table 5.2 details the benchmark DNNs that we use to evaluate CxDNN. To demonstrate the scalability of CxDNN, we focus on large-scale DNNs, *viz.*, AlexNet, Network-in-Network (NiN), OverFeat, VGG-16, GoogleNet, and ResNet-50.

Table 6.1.: Crossbar non-idealities modeled during evaluation of CxDNN

Type	**Non-idealities modeled**
Device-level non-idealities	Process variations (Gaussian), device-level noises (Gaussian), imperfect writes (Gaussian)
Circuit-level non-idealities	Wire resistances (r_{row} = 1Ω, r_{col}= 4.6Ω), driver resistances (1.5KΩ), sensing resistances (500Ω), sneak path (data-dependent), non-linear ADCs and DACs (lookup tables)

Device, circuit and system modeling. Table 6.2 details the synaptic device [64], circuit parameters, interconnect technology [141], and variation technology parameters [137] used in our experiments. It also provides the RCA, the host processor, and the compensation logic configurations. We limit the synaptic device precision to 6-bits considering the maximum precision that can be achieved by practical synaptic devices [147]. We perform the device simulation to abstract the circuit-level parameters such as conductance range (Rmin to Rmax) and the number of conductive states (Precision) for a nominal device (without any non-idealities). Next, we characterize the impact of process variations in physical parameters (*e.g.*, synaptic device dimensions, Tox, *etc.*) and intrinsic device-level noise (*e.g.*, thermal noise) that results in variations in the synaptic conductance. Subsequently, we model all device-level non-idealities using a Gaussian distribution in the synaptic conductance (the abstracted circuit-level parameter). We consider the variation in the synaptic conductance to be $\sigma/\mu = 5\%$ to 15% based on variation data reported in [137]. We show the efficacy of CxDNN on this range of variation values $[(\sigma/\mu) = 5\%$ to 15%]. We perform layouts of the synaptic device and the crossbar array (using design rules [140]) to extract inter-connect parasitics (detailed in Table 6.2). Our simulations utilize the ADC and DAC circuits proposed in [27, 148]. We evaluate RCA energy consumption and latency using SPICE simulations. Finally, we synthesize the digital logic units using Synopsys Design Compiler and a commercial 45nm technology to estimate their energy, delay, and area at the gate-level. We assume a spatial architecture with sufficient RCAs to map the largest DNN (i.e, VGG-16). Operations other than vector-matrix multiplications (max-pooling, normalization, etc.) are performed on the Special Function Units (SFUs).

Table 6.2.: Device, circuit, and system parameters used for evaluating CxDNN

Components	**Values**

Synaptic Device	Precision = 6 bits, R_{min} = 200 KΩ , R_{max} = 1.4 MΩ
	Length = 1280nm, Width = 40nm, Tox = 2.12nm

Wire Technology	Resistance = 3.3Ω/um, Capacitance = 0.2fF/um
Variation	Synaptic conductance $(\sigma/\mu) = 5\%$ to 15%
CMOS Technology	45nm Bulk CMOS
Resistive Crossbar Array (RCA)	64 DACs (6 bits each), 16 shared ADCs (10 bits each), 64x64 synaptic elements, 64 input registers (6 bits), 64 output registers (10 bit), 1 Adder (16 bits)
Compensation logic	64 registers (6-bits), 1 multiplier (10 bit)
Extracted Parasitic	$r_{row} = 1\Omega$, $c_{row} = 0.05$fF, $r_{col} = 4.6\Omega$, $c_{col} = 0.3$fF

6.3 Results

We now present results to show the effectiveness of CxDNN in enabling high accuracy large-scale DNNs on resistive crossbar based systems.

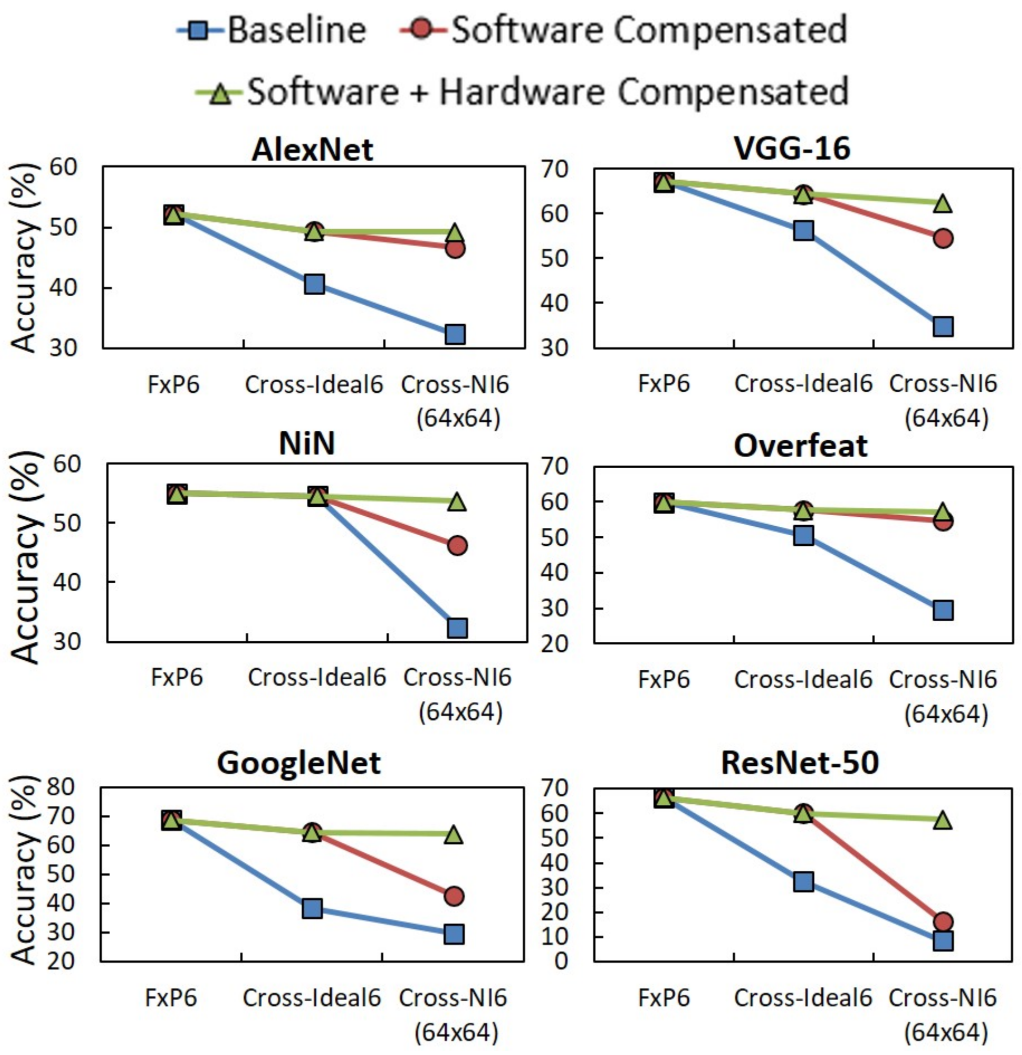


Fig. 6.10.: Application level accuracy benefits

6.3.1 Application-level compensation

We first show CxDNN's compensation ability at the application-level. To this end, we compare DNNs designed using three different compensation techniques. We use (i) a baseline DNN obtained using only the CxDNN conversion algorithm, (ii) a software compensated DNN obtained using the CxDNN conversion algorithm and re-training mechanism, and (iii) a software and hardware compensated DNN obtained using both hardware and software compensation techniques of CxDNN. Figure 6.10 shows the classification accuracy of these DNNs realized on two different resistive crossbar systems - an "ideal" system with 6-bit precision but no device and circuit non-idealities (Cross-ideal6) and a non-ideal crossbar system (Cross-NI6) designed using 64x64 resistive crossbar arrays. In Figure 6.10, FxP6 is a software 6-bit fixed-point implementation. Across all benchmark applications, we observe that DNNs compensated using both software and hardware techniques are considerably superior to the other two versions (baseline and Software Compensated). The baseline DNN suffers distortion due to conversion, and both the baseline and the software compensated DNNs suffer from device and circuit-level non-idealities, leading to poor application-level accuracy on real resistive crossbar systems (Cross-NI6). Overall, CxDNN achieves 16.9%-49% improvement in accuracy relative to the baseline. We observe the effect of crossbar non-idealities to be more prominent on deeper networks with >50 layers (*e.g.*, ResNet and GoogleNet), as the accuracy degradation for the software compensated design (with no compensation for crossbar non-idealities) on Cross-NI6 is substantial. In contrast, we find wider DNNs (e.g., AlexNet and VGG-16) with fewer (9 to 16) layers to show more resilience to crossbar noise, as the software compensation is effective in restoring application-level accuracy. We observed an interesting correlation between the resilience of DNNs to crossbar non-idealities and their resilience to other traditional optimizations, such as pruning [149, 150]. For example, efforts on DNN pruning [149, 150] show that AlexNet and VGG-16 networks can be pruned by 90-95%, but parameters in ResNets can only be pruned by about 50-60%. Correspondingly, we observe that AlexNet and VGG-16 networks are more resilient (robust) than ResNet-50 to crossbar non-idealities.

Next, we provide insights on how errors in vector-matrix multiplication at the crossbar level interact over various crossbar operations and propagate through the network. To that end, we visualize the heat map of output activations obtained from various layers of the GoogleNet DNN as it processes an oxcart image from the ImageNet dataset. Figure 6.11 illustrates the heat map of the first layer, some middle layers, and the last layer of the GoogleNet DNN implemented on an ideal crossbar system (Cross-ideal), and a non-ideal crossbar system (Cross-NI) with and without hardware compensation. We observe that the initial layers of the uncompensated CrossNI and the Cross-ideal cases are very similar. However, the later layers are noticeably different. This is because the errors accumulate as we go deeper into the network, leading to significant and visible distortions in output features of the later layers. This observation also concurs well with the result shown in Figure 6.10, where the impact of crossbar non-idealities on deeper networks is found to be more prominent. Further, we observe the output feature-maps of the Cross-ideal and Compensated Cross-NI cases to be very similar, thereby demonstrating the effectiveness of CxDNN in mitigating errors due to crossbar non-idealities.

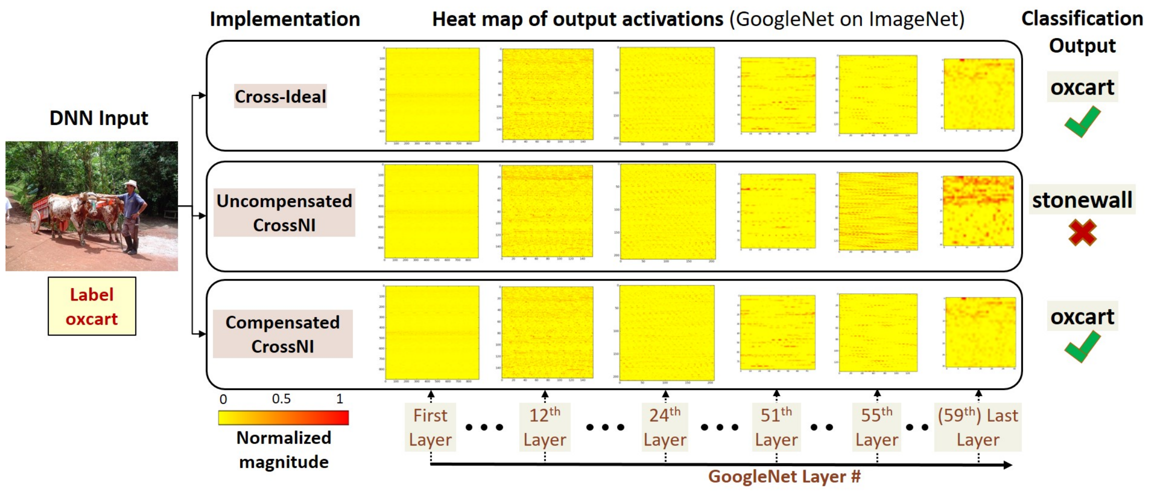


Fig. 6.11.: Visual demonstration of HW error compensation

6.3.2 Compensating hardware-instance-specific errors

We now focus on hardware-instance-specific errors due to process variations and evaluate CxDNN's efficacy in mitigating them. We assume random variations (σ/μ=5%) in the synaptic elements of a resistive crossbar system as the source of non-ideality. Figure 6.12 shows the accuracy of an ideal crossbar system (Cross-Ideal), and a non-ideal crossbar system (Cross-NI) with and without hardware compensation. We observe that hardware compensation recovers the drop in DNN accuracy due to process variation very effectively. On average, we observe an accuracy improvement of 25.83% over the Cross-NI Uncompensated case. Moreover, Hardware compensation achieves accuracy within 1.8%-2.8% of the Cross-ideal implementation.

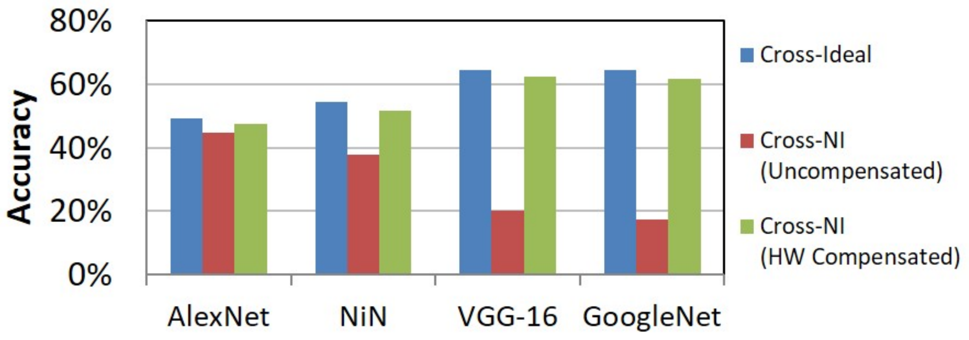


Fig. 6.12.: HW compensation for instance specific errors

We also study CxDNN's effectiveness with increasing process variations on the GoogleNet and NiN networks. Figure 6.13 shows the accuracy of the compensated and the uncompensated Cross-NI designs for various degrees of random variations (σ/μ) in the synaptic devices. We observe that the HW compensation effectively mitigates errors in resistive crossbar systems with high process variations (σ/μ=15%), and achieves considerable accuracy improvement over the uncompensated design irrespective of the degree of variations. As mentioned in section 6.1.4, CxDNN's HW compensation is a low-overhead approximate compensation scheme that does not correct errors exactly. Figure 6.13 suggests that the CxDNN can achieve accuracy close to Cross-ideal case for σ/μ upto 10%, with some degradation for larger values.

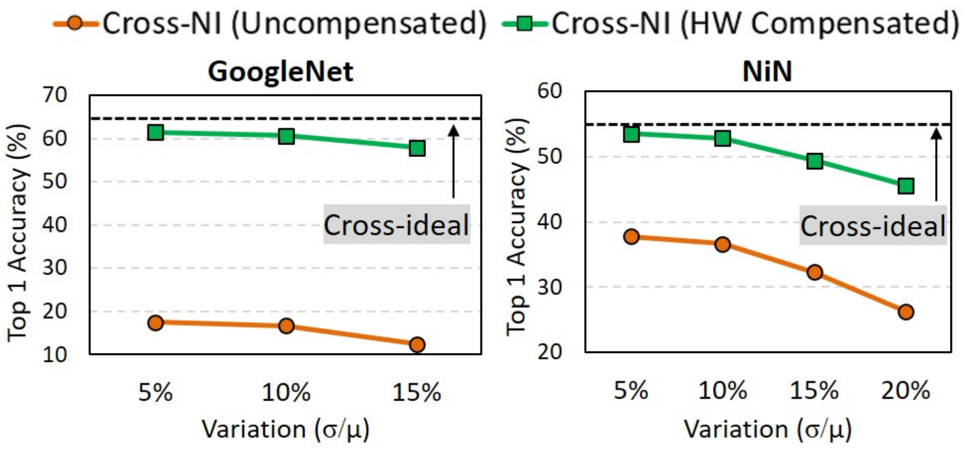


Fig. 6.13.: HW compensation with increasing process variations

6.3.3 Re-training efficiency

Next, we demonstrate the effectiveness of the proposed hardware-instance-independent re-training mechanism. Figure 6.14 shows the accuracy improvement with re-training iterations for various DNNs. As shown, we achieve an improvement of 7%-27.4% for a minimal amount of re-training (150 iterations). In summary, our re-training method is able to bridge the accuracy gap between the fixed-point and the ideal crossbar (Cross-Ideal) DNN implementations effectively.

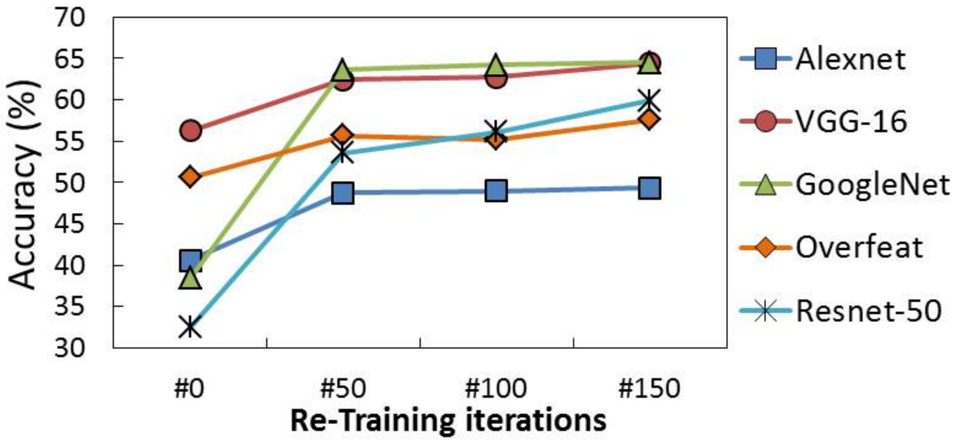


Fig. 6.14.: Efficiency of the re-training method

6.3.4 Hardware compensation at the array-level

We now show the effectiveness of the hardware compensation method in mitigating errors at the crossbar array level. Figure 6.15 shows the observed errors for the Uncompensated and Compensated RCA instances with respect to ideal crossbar across 1000 input vectors. We plot the average error at each column of a 64x64 crossbar. We also show the error spread (maximum and minimum error) for each crossbar column across the 1000 vectors. We use four different crossbar instances with 0%, 5%, 10% and 15% random variations in synaptic conductances. As evident, the observed errors for each crossbar column in the compensated design are drastically smaller than the uncompensated design, underscoring the effectiveness of CxDNN's hardware compensation scheme in mitigating errors at the crossbar array level.

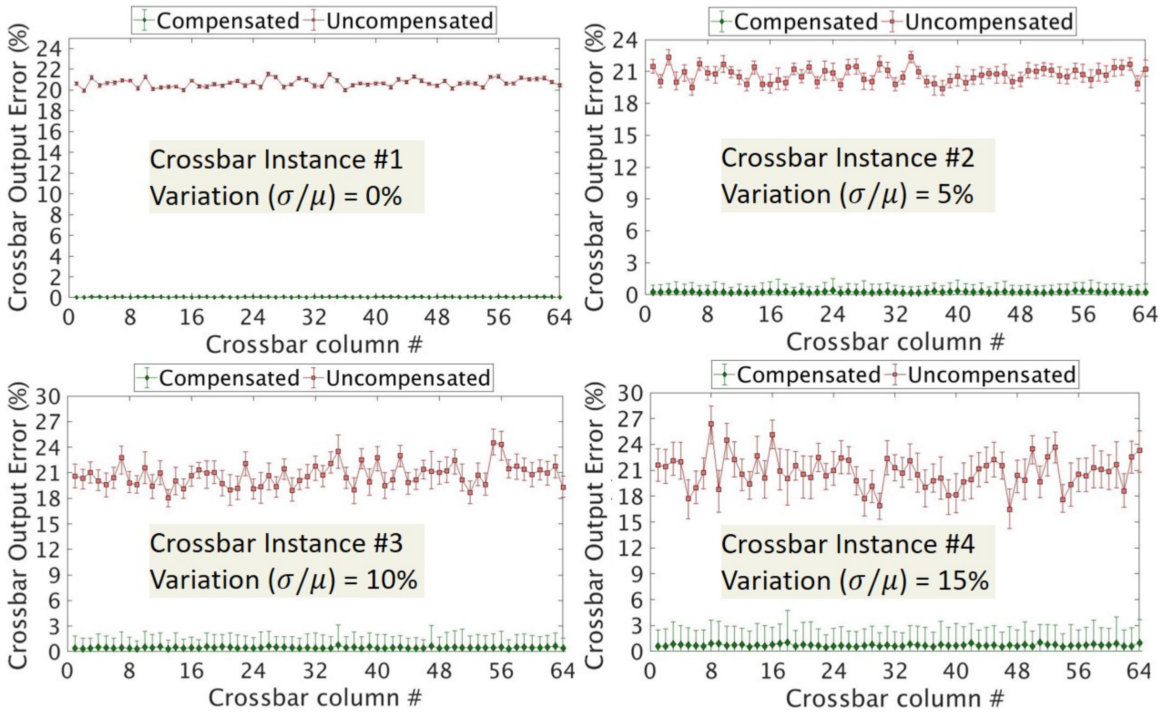


Fig. 6.15.: Hardware compensation in action

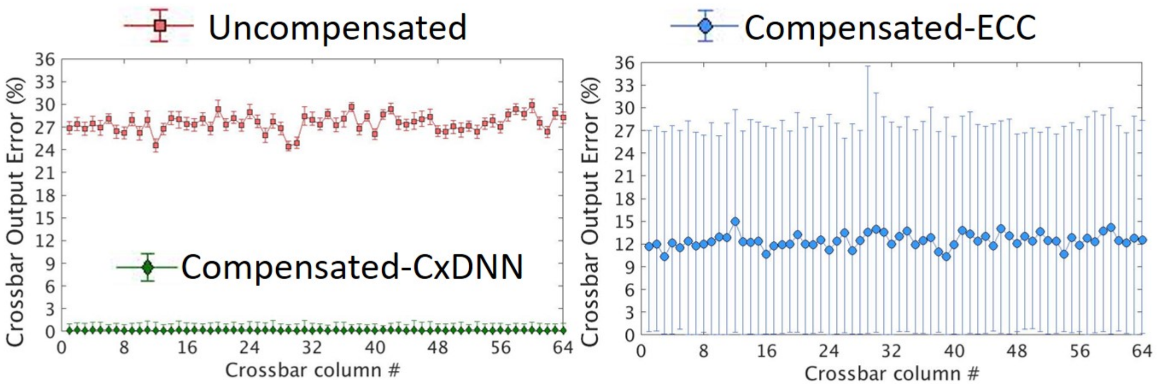


Fig. 6.16.: Comparison of CxDNN and ECC scheme based compensation [129]

6.3.5 Comparison with ECC scheme

Next, we compare the proposed hardware compensation with the AN code based ECC compensation scheme [129] at the crossbar-level. Figure 6.16 shows the computed relative errors across crossbar columns of an RCA instance for three designs - (i) an uncompensated design, (ii) the CxDNN compensated design, and (iii) the ECC compensated design. In the plots, the markers (orange square, green diamond, and blue circle) represent the mean relative error for each crossbar column across 1000 input vectors, and the error bars show the spread. We implemented the AN code based ECC scheme with N=19 which is equivalent to a 5-bit Hamming code and can correct up to 19 syndromes. We note that for 6-bit weights and inputs, the overhead due to the ECC scheme requiring five additional bits will be substantial. However, we found the N=19 code to provide the best result, and thereby use it for the comparison. For instance, N>19 and N<19 both had higher mean error due to overcompensation and inferior compensation, respectively. In Figure 6.16, we observe that the ECC scheme reduces the mean error for each crossbar column. However, it cannot detect and correct a significant number of errors and wrongly compensates a few. In contrast, CxDNN is quite effective in compensating errors at the crossbar level. Further, as mentioned in Chapter 2, AN codes [130] are for correcting errors in linear functions realized using digital systems, where the errors are discrete (at bit-level) and can be quantified using the digital number representation (multiple bits). However, the functions realized in resistive crossbars are analog and non-linear due to several non-linear functions (e.g., ADCs and DACs). Therefore, ECC schemes cannot effectively mitigate errors in analog systems.

6.3.6 Area and Power breakdown

Finally, we discuss the area and the power breakdown of various components in a Resistive Crossbar Array (RCA) including the compensation logic. Figure 6.17 shows the major components, *viz.*, ADC, DAC, input and output registers, and compensation logic (Comp-Logic). The Comp-Logic includes registers for storing CFs, a mux, and a multiplier. ADCs and DACs dominate the area and power consumption in the RCA. The fraction of total area and power consumed by them is 79% and 90%, respectively. The area and power overhead due to the Comp-Logic with respect to RCA is about 7.3% and 4%, respectively. However, the system-level area and power overhead are lower, *i.e.*, 3.5% and 5.5%, respectively, as the resistive crossbar processing unit (XPU) consists of other non-RCA logic units such as input and output buffers, controllers, and special function units (SFUs). These non-RCA logic units occupy 24% of the total XPU area and consume 13% of the total XPU power. The major contributors to the overhead are the additional registers required for storing compensation factors (CFs). Therefore, more efficient methods for storing CF can effectively reduce the overall overhead. For example, the registers for storing CFs can be replaced by denser memories like SRAM, and CFs can also be provided as inputs along with the activations and the partial sums. The overhead can also be improved by sharing CFs across crossbar columns. In this dissertation, we did not optimize the overheads further as we believe them to be reasonable, considering the benefits of CxDNN and its ability in mitigating a key challenge to the use of resistive crossbar based neural fabrics.

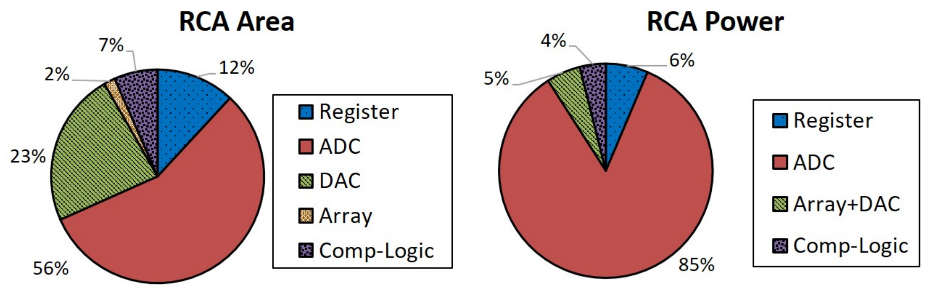


Fig. 6.17.: Area and power breakdown of a crossbar array with compensation logic

6.4 Summary

Resistive crossbar systems are a promising solution to the energy efficient realization of DNNs. However, they suffer from numerous device and circuit-level non-idealities leading to degradation in application-level accuracy of large-scale DNNs. In this work, we present CxDNN, a hardware-software methodology comprising of a conversion algorithm, a hardware-instance-independent re-training mechanism, and a low-overhead hardware compensation scheme to design high accuracy DNNs on resistive crossbars. CxDNN preserves the train-once-deploy-anywhere tenet of current DNN implementations, wherein DNNs are re-trained once to obtain a common model that can be deployed on many resistive crossbar systems (XPUs). Further, CxDNN utilizes low-overhead hardware to mitigate instance-specific errors. Our evaluations reveal that CxDNN can largely restore the accuracy loss due to non-idealities by achieving 16.9%-49% improvements over a range of large-scale DNN benchmarks. We also find that CxDNN to be very effective in compensating hardware-instance specific errors as it can achieve accuracy within 1.8%-2.8% of the crossbar implementation with no non-idealities. The system level area and power overheads of the compensation hardware are minimal, *i.e.*, 5.5% and 3.5%, respectively. Finally, we conclude that CxDNN presents a promising approach to compensate for errors due to crossbar non-idealities and addresses a key bottleneck to the adoption of resistive crossbar based neural fabrics.

7. TERNARY IN-MEMORY ACCELERATOR FOR DEEP NEURAL NETWORKS

The use of lower precision to represent the weights and activations in DNNs is a promising technique for realizing DNN inference (evaluation of pre-trained DNN models) on energy-constrained platforms [96, 104, 105, 145, 151–157]. Reduced bit-precision can lower all facets of energy consumption including computation, memory and interconnects. Current commercial hardware [158,159], includes widespread support for 8-bit and 4-bit fixed point DNN inference, and recent research has continued the push towards even lower precision [96,104,105,145,151–155].

Recent studies [96, 104, 105, 143, 145, 151–155] suggest that among low-precision networks, ternary DNNs represent a promising sweet-spot as they enable low-power inference with high application-level accuracy. This is illustrated in Figure 7.1, which shows the reported accuracies of various state-of-the-art binary [104, 145, 151], ternary [96,105,152–155], and full-precision (FP32) networks on complex image classification (ImageNet) and language modeling (PTB [160]) tasks. We observe that the accuracy degradation of binary DNNs over the FP32 networks can be considerable (5-13% on image classification, 150-180 PPW [Perplexity Per Word] on language modeling). In contrast, ternary DNNs achieve accuracy significantly better than binary networks (and much closer to FP32 networks). Motivated by these results, we focus on the design of a programmable accelerator for realizing various state-of-the-art ternary DNNs.

Ternary networks greatly simplify the multiply-and-accumulate (MAC) operation that constitutes 95-99% of total DNN computations. Consequently, the amount of energy and time spent on DNN computations can be drastically improved by using lower-precision processing elements (the complexity of a MAC operation has a super-linear relationship with precision). However, when classical accelerator architectures

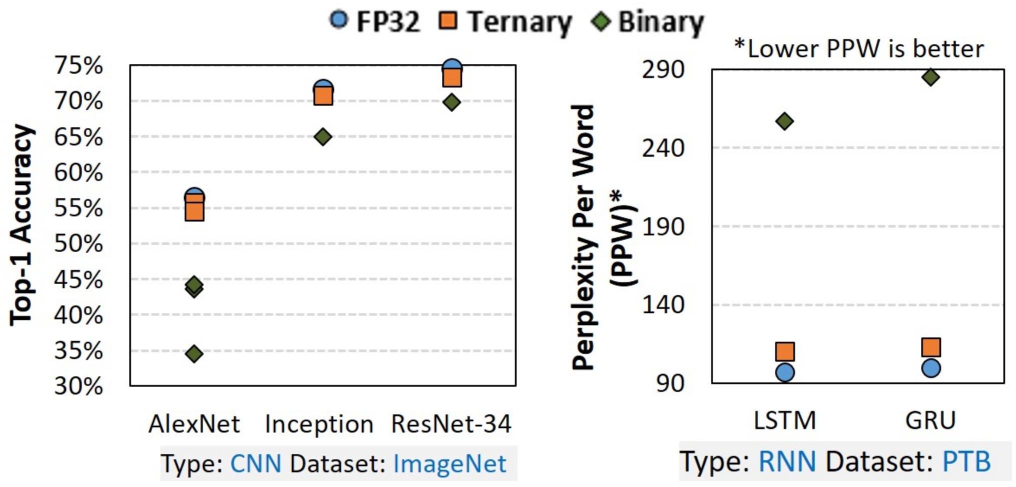


Fig. 7.1.: Accuracy comparison of binary, ternary, and full-precision (FP32) DNNs [96, 104, 105, 145, 151–155]

(*e.g.*, TPU and GPU) are adopted to realize ternary DNNs, on-chip memory due to sequential (row-by-row) reads and leakage in un-accessed rows becomes the energy and performance bottleneck. This chapter explores in-memory computing in the specific context of ternary DNNs and demonstrates that it leads to significant improvements in performance and energy efficiency.

While several efforts have explored in-memory accelerators in recent years, TiM-DNN differs in significant ways and is the first to apply in-memory computing (massively parallel vector-matrix multiplications within the memory array itself) to ternary DNNs using a new CMOS-based bit-cell. Many in-memory accelerators use non-volatile memory (NVM) technologies such as PCM and ReRAM [28, 30, 109, 110] to realize in-memory dot product operations. While NVMs promise much higher density and lower leakage than CMOS memories, they are still an emerging technology with open challenges such as large-scale manufacturing yield, limited endurance, high write energy, and errors due to device and circuit-level non-idealities (detailed in chapters 5 and 6). Other efforts have explored SRAM-based in-memory accelerators for binary networks [111–115] and SRAM-based near-memory accelerators for ternary networks [96, 97]. However, the restriction to binary networks is a significant limitation as binary networks known to date incur a large drop in accuracy as highlighted in Figure 7.1, and the near-memory accelerators are bottlenecked by the on-chip memory where only one memory row is enabled per access. Extended SRAMs that perform bitwise binary operations in-memory may be augmented with near-memory logic to perform higher precision computations in a bit-serial manner [116]. However, such an approach requires multiple sequential execution steps (array accesses) to realize even one multiplication operation (and many more to realize dot-products), limiting efficiency. In contrast, we propose TiM-DNN, a programmable in-memory accelerator that can realize massively parallel *signed ternary vector-matrix multiplications* per array access. TiM-DNN supports various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) systems, enabling it to execute a broad range of state-of-the-art ternary DNNs. This is motivated by recent efforts [152] that show weighted ternary systems can achieve improved accuracies.

The building block of TiM-DNN is a new memory cell, Ternary Processing Cell (TPC), which functions as both a ternary storage unit and a scalar ternary multiplication unit. Using TPCs, we design TiM tiles which are specialized memory arrays to execute signed ternary dot-product operations. TiM-DNN comprises of a plurality of TiM tiles arranged into banks, wherein all tiles compute signed vector-matrix multiplications in parallel.

In summary, the key contributions of this chapter are:

- We present TiM-DNN, a programmable in-memory accelerator supporting various ternary representations including unweighted (-1,0,1), symmetric weighted (-a,0,a), and asymmetric weighted (-a,0,b) systems for realizing a broad range of ternary DNNs.

- We propose a Ternary Processing Cell (TPC) that functions as both ternary storage and a ternary scalar multiplications unit and a TiM tile that is a specialized memory array to realize signed vector-matrix multiplication operations with ternary values.

- We develop an architectural simulator for evaluating TiM-DNN, with array-level timing and energy models obtained from circuit-level simulations. We evaluate an implementation of TiM-DNN in 32nm CMOS using a suite of 5 popular DNNs designed for image classification and language modeling tasks. A 32-tile instance of TiM-DNN achieves a peak performance of 114 TOPs/s, consumes 0.9W power, and occupies $1.96mm^2$ chip area, representing a 300X improvement in TOPS/W compared to a state-of-the-art NVIDIA Tesla V100 GPU [158]. In comparison to low-precision accelerators [97, 116], TiM-DNN achieves 55.2X-240X improvement in TOPS/W. TiM-DNN also obtains 3.9x-4.7x improvement in system energy and 3.2x-4.2x improvement in performance over a well-optimized near-memory accelerator for ternary DNNs.

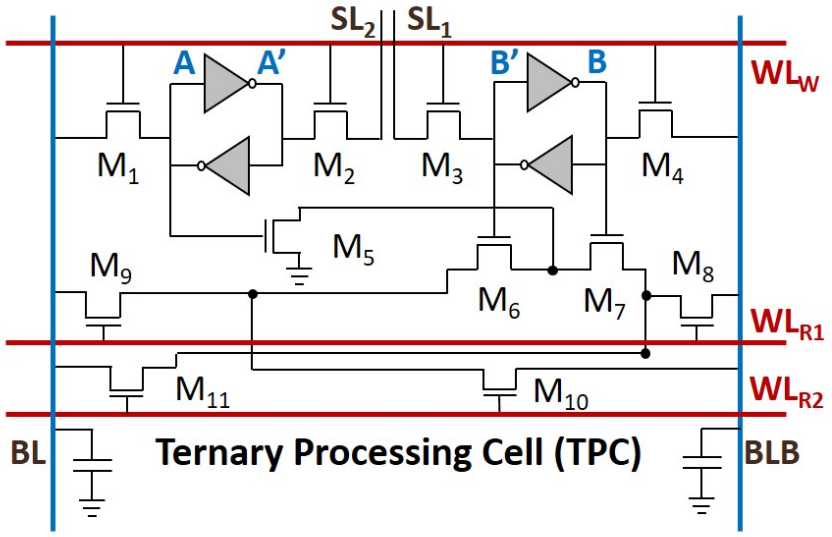
We organize the rest of the chapter as follows. Section 7.1 presents the proposed TiM-DNN architecture. We detail the experimental methodology for evaluating TiM-DNN in section 7.2. Section 7.3 contains our experimental results, and section 7.4 concludes the chapter.

7.1 TiM-DNN architecture

In this section, we present the proposed TiM-DNN accelerator along with its building blocks, *i.e.*, Ternary Processing Cells and TiM tiles.

7.1.1 Ternary Processing Cell (TPC)

To enable in-memory signed multiplication with ternary values, we present a new Ternary Processing Cell (TPC) that operates as both a ternary storage unit and a ternary scalar multiplication unit. Figure 7.2 shows the proposed TPC circuit, which consists of two cross-coupled inverters for storing two bits (‘A’ and ‘B’), a write wordline (WL_W), two source lines (SL_1 and SL_2), two read wordlines (WL_{R1} and WL_{R2}) and two bitlines (BL and BLB). A TPC supports two operations - write and scalar ternary multiplication. A write operation is performed by enabling WL_W and driving the source-lines and the bitlines to either V_{DD} or 0 depending on the data. We can write both bits simultaneously, with 'A' written using BL and SL_2 and 'B' written using BLB and SL_1. Using both bits 'A' and 'B' a ternary value (-1,0,1) is inferred based on the storage encoding shown in Figure 7.2 (Table on the left). For example, when A=0 the TPC stores W=0. When A=1 and B=0 (B=1) the TPC stores W=1 (W=-1).



Storage Encoding

A	B	Stored Weight (W)
0	X	0
1	0	1
1	1	-1

Input Encoding

Input (I)	WL_{R1}	WL_{R2}
0	0	0
1	1	0
-1	0	1

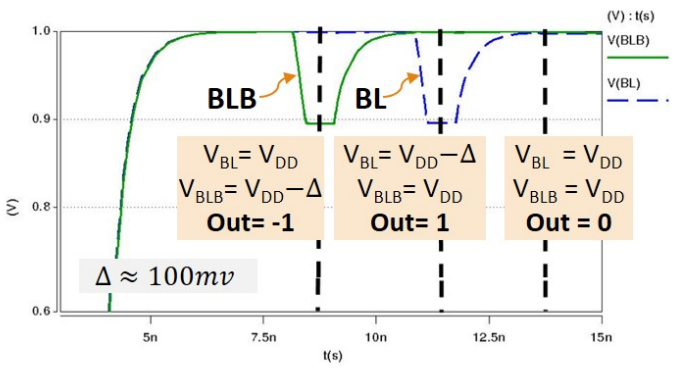
Fig. 7.2.: Ternary Processing Cell (TPC) circuit and encoding scheme

A scalar multiplication in a TPC is performed between a ternary input and the stored weight to obtain a ternary output. The bitlines are precharged to V_{DD}, and subsequently, the ternary inputs are applied to the read wordlines (WL_{R1} and WL_{R2}) based on the input encoding scheme shown in Figure 7.2 (Table on the right). The final bitline voltages (V_{BL} and V_{BLB}) depend on both the input (I) and the stored weight (W). The table in Figure 7.3 details the possible outcomes of the scalar ternary multiplication (W*I) with the final bitline voltages and the inferred ternary output

(Out). For example, when W=0 or I=0, the bitlines remain at V_{DD} and the output is inferred as 0 (W*I=0). When W=I=±1, BL discharges by a certain voltage, denoted by Δ, and BLB remains at V_{DD}. This is inferred as Out=1. In contrast, when W=-I=±1, BLB discharges by Δ and BL remains at V_{DD} producing Out=-1. The final bitline voltages are converted to a ternary output using single-ended sensing at BL and BLB. Figure 7.3 depicts the output encoding scheme and the results of SPICE simulation of the scalar multiplication operation with various possible final bitline voltages. Note that, the TPC design uses separate read and write paths to avoid read disturb failures during in-memory multiplications.

Output Encoding

V_{BL}	V_{BLB}	Digital Output
V_{DD}	V_{DD}	0
$V_{DD}-\Delta$	V_{DD}	1
V_{DD}	$V_{DD}-\Delta$	-1



		Possible outcome		
Weight (W)	Input (I)	V_{BL}	V_{BLB}	Inferred Output (Out = W * I)
0	0	V_{DD}	V_{DD}	0
1	0	V_{DD}	V_{DD}	0
-1	0	V_{DD}	V_{DD}	0
0	1	V_{DD}	V_{DD}	0
1	1	$V_{DD}-\Delta$	V_{DD}	1
-1	1	V_{DD}	$V_{DD}-\Delta$	-1
0	-1	V_{DD}	V_{DD}	0
1	-1	V_{DD}	$V_{DD}-\Delta$	-1
-1	-1	$V_{DD}-\Delta$	V_{DD}	1

Fig. 7.3.: Scalar multiplication using a TPC

7.1.2 Dot-product computation using TPCs

Next, we extend the idea of realizing a scalar multiplication using the TPC to a dot-product computation. Figure 7.4(a) illustrates the mapping of a dot-product operation ($\sum_{i=1}^{L} Inp[i] * W[i]$) to a column of TPCs with shared bitlines. To compute, first the bitlines are precharged to V_{DD}, and then the inputs (Inp) are applied to all TPCs simultaneously. The bitlines (BL and BLB) function as an analog accumulator, wherein the final bitline voltages (V_{BL} and V_{BLB}) represent the sum of the individual TPC outputs. For example, if ‘n/L’ and ‘k/L’ TPCs have output 1 and -1, respectively, the final bitline voltages are $V_{BL} = V_{DD} - n\Delta$ and $V_{BLB} = V_{DD} - k\Delta$. The bitline voltages are converted using Analog-to-Digital converters (ADCs) to yield digital values ‘n’ and ‘k’. For the unweighted encoding where the ternary weights are encoded as (-1,0,1), the final dot-product is ‘n-k’. Figure 7.4(b) shows the sensing circuit required to realize dot-products with unweighted (-1,0,1) ternary system.

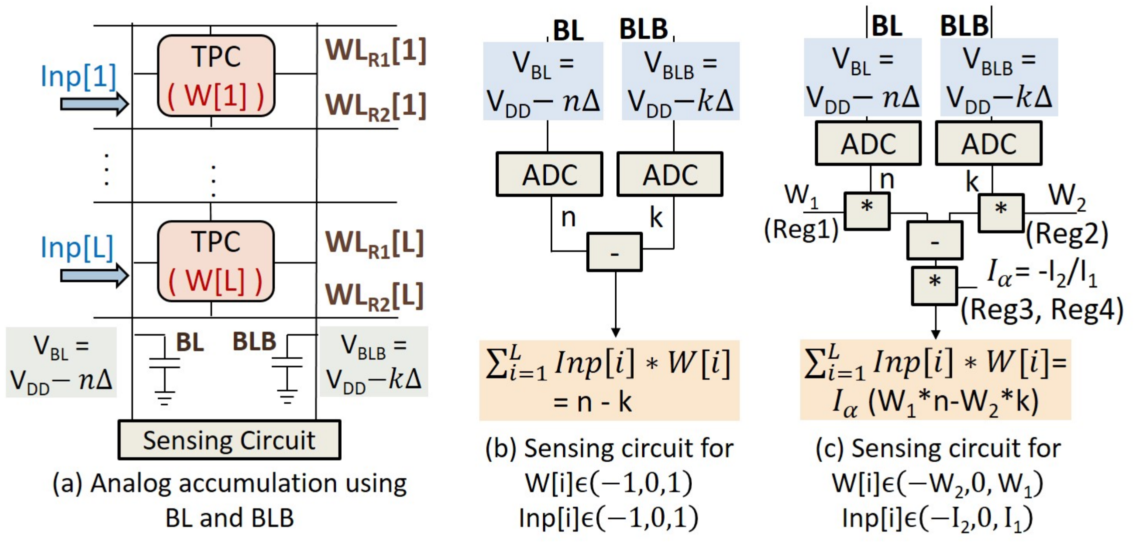

Inp[1]
TPC
(W[1])
$WL_{R1}[1]$
$WL_{R2}[1]$
Inp[L]
TPC
(W[L])
$WL_{R1}[L]$
$WL_{R2}[L]$
$V_{BL} = V_{DD} - n\Delta$
BL
BLB
$V_{BLB} = V_{DD} - k\Delta$
Sensing Circuit
(a) Analog accumulation using BL and BLB
BL
BLB
$V_{BL} = V_{DD} - n\Delta$
$V_{BLB} = V_{DD} - k\Delta$
ADC
ADC
n
k
-
$\sum_{i=1}^{L} Inp[i] * W[i] = n - k$
(b) Sensing circuit for $W[i] \epsilon (-1,0,1)$ $Inp[i] \epsilon (-1,0,1)$
BL
BLB
$V_{BL} = V_{DD} - n\Delta$
$V_{BLB} = V_{DD} - k\Delta$
ADC
ADC
n
k
W_1 (Reg1)
W_2 (Reg2)
*
*
-
*
$I_\alpha = -I_2/I_1$ (Reg3, Reg4)
$\sum_{i=1}^{L} Inp[i] * W[i] = I_\alpha (W_1 * n - W_2 * k)$
(c) Sensing circuit for $W[i] \epsilon (-W_2, 0, W_1)$ $Inp[i] \epsilon (-I_2, 0, I_1)$

Fig. 7.4.: Dot-product computation using TPCs: (a) Analog accumulation using BL and BLB, (b) sensing circuit for unweighted (-1,0,1) ternary system, (c) Sensing circuit for asymmetric weighted (-a,0,b) ternary system

We can also realize dot-products with a more general ternary encoding represented by asymmetric weighted (-a,0,b) values. Support for a more general ternary encoding is motivated by recent efforts [152] that show weighted ternary systems can achieve improved accuracies. Figure 7.4(c) shows the sensing circuit that enables dot-product with asymmetric ternary weights $(-W_2, 0, W_1)$ and inputs $(-I_2, 0, I_1)$. As shown, the ADC outputs are scaled by the corresponding weights (W_1 and W_2), and subsequently, an input scaling factor (I_α) is applied to yield '$I_\alpha(W_1$*n-W_2*k)'. In contrast to dot-products with unweighted values, we require two execution steps to realize dot-products with the asymmetric ternary system, wherein each step computes a partial dot-product (pOut). Figure 7.5 details these two steps using an example. In step 1, we choose I_α=I_1, and apply I_1 and I_2 as '1' and '0', respectively, resulting in a partial output (pOut) given by $pOut_1 = I_1(W_1$*n-W_2*k). In step 2, we choose I_α=-I_2, and apply I_1 and I_2 as '0' and '1', respectively, to yield $pOut_2$ = -$I_2(W_1$*n-W_2*k). The final dot-product is given by '$pOut_1$+$pOut_2$'.

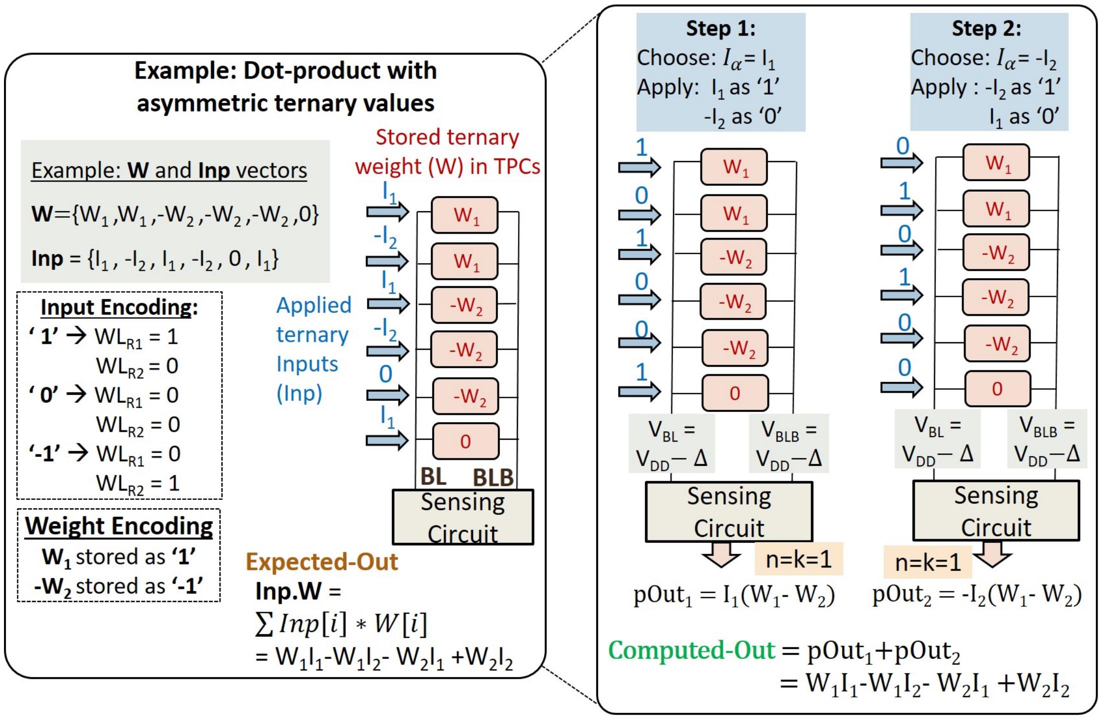
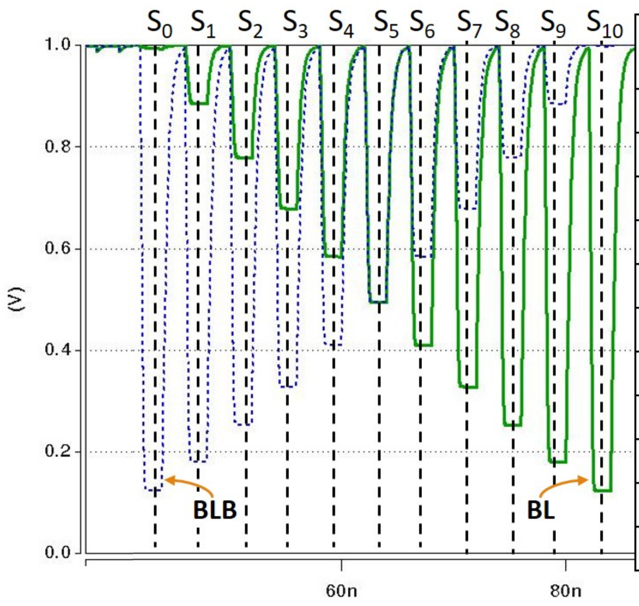

Example: Dot-product with asymmetric ternary values
Example: W and Inp vectors
W={W1,W1,-W2,-W2,-W2,0}
Inp = {I1, -I2, I1, -I2, 0, I1}
Input Encoding:
'1' → WLR1 = 1, WLR2 = 0
'0' → WLR1 = 0, WLR2 = 0
'-1' → WLR1 = 0, WLR2 = 1
Weight Encoding
W1 stored as '1'
-W2 stored as '-1'
Stored ternary weight (W) in TPCs
Applied ternary Inputs (Inp)
BL BLB
Sensing Circuit
Expected-Out
Inp.W = ∑Inp[i] * W[i] = W1I1-W1I2- W2I1 +W2I2
Step 1: Choose: Iα= I1 Apply: I1 as '1' -I2 as '0'
Step 2: Choose: Iα= -I2 Apply : -I2 as '1' I1 as '0'
VBL = VDD− Δ
VBLB = VDD−Δ
n=k=1
pOut1 = I1(W1- W2)
pOut2 = -I2(W1- W2)
Computed-Out = pOut1+pOut2 = W1I1-W1I2- W2I1 +W2I2

Fig. 7.5.: Example dot-product computation with asymmetric weighted ternary values

To validate the dot-product operation, we perform a detailed SPICE simulation to determine the possible final voltages at BL (V_{BL}) and BLB (V_{BLB}). Figure 7.6 shows various BL states (S_0 to S_{10}) and the corresponding value of V_{BL} and 'n'. Note that the possible values for V_{BLB} ('k') and V_{BL} ('n') are identical, as BL and BLB are symmetric. The state S_i refers to the scenario where 'i' out of 'L' TPCs compute an output of '1'. We observe that from S_0 to S_7 the average sensing margin (Δ) is 96mv. The sensing margin decreases to 60-80mv for states S_8 to S_{10}, and beyond S_{10} the bitline voltage (V_{BL}) saturates. Therefore, we can achieve a maximum of 11 BL states (S_0 to S_{10}) with sufficiently large sensing margin required for sensing reliably under process variations [114]. The maximum value of 'n' and 'k' is thus 10, which in turn determines the number of TPCs ('L') that can be enabled simultaneously. Setting $L = n_{max} = k_{max}$ would be a conservative choice. However, exploiting the weight and input sparsity of ternary DNNs [153–155], wherein 40% or more of the elements are zeros, and the fact that non-zero outputs are distributed between '1' and '-1', we choose a design with $n_{max} = 8$, and $L = 16$. Our experiments indicate that this choice had no effect on the final DNN accuracy compared to the conservative case. In this thesis, we also evaluate the impact of process variations on the dot-product operations realized using TPCs, and provide the experimental results on variations in Section 7.3.6.

7.1.3 TiM tile

We now present the TiM tile, *i.e.*, a specialized memory array designed using TPCs to realize massively parallel vector-matrix multiplications with ternary values. Figure 7.7 details the tile design, which consists of a 2D array of TPCs, a row decoder and write wordline driver, a block decoder, Read Wordline Drivers (RWDs), column drivers, a sample and hold (S/H) unit, a column mux, Peripheral Compute Units (PCUs), and scale factor registers. The TPC array contains 'L*K*N' TPCs, arranged in 'K' blocks and 'N' columns, where each block contains 'L' rows. As shown in



State	V (BL)	ADC-Out (n)
S_0	1	0
S_1	0.89	1
S_2	0.78	2
S_3	0.68	3
S_4	0.58	4
S_5	0.49	5
S_6	0.40	6
S_7	0.32	7
S_8	0.24	8
S_9	0.18	9
S_{10}	0.12	10

Fig. 7.6.: Dot-product circuit simulation

the Figure, TPCs in the same row (column) share wordlines (bitlines and source-lines). The tile supports two major functions, (i) programming, *i.e.*, row-by-row write operations, and (ii) a vector-matrix multiplication operation. A write operation is performed by activating a write wordline (WL_W) using the row decoder and driving the bitlines and source-lines. During a write operation, 'N' ternary words (TWs) are written in parallel. In contrast, to the row-wise write operation, a vector-matrix multiplication operation is realized at the block granularity, wherein 'N' dot-product operations each of vector length 'L' are executed in parallel. The block decoder selects a block for the vector-matrix multiplication, and RWDs apply the ternary inputs. During the vector-matrix multiplication, TPCs in the same row share the ternary input (Inp), and TPCs in the same column produce partial sums for the same output. As discussed in section 7.1.2, accumulation is performed in the analog domain using the bitlines (BL and BLB). In one access, TiM can compute the vector-matrix product **Inp**.**W**, where **Inp** is a vector of length L and **W** is a matrix of dimension LxN stored in TPCs. The accumulated outputs at each column are stored using a sample and hold (S/H) unit and get digitized using PCUs. To attain higher

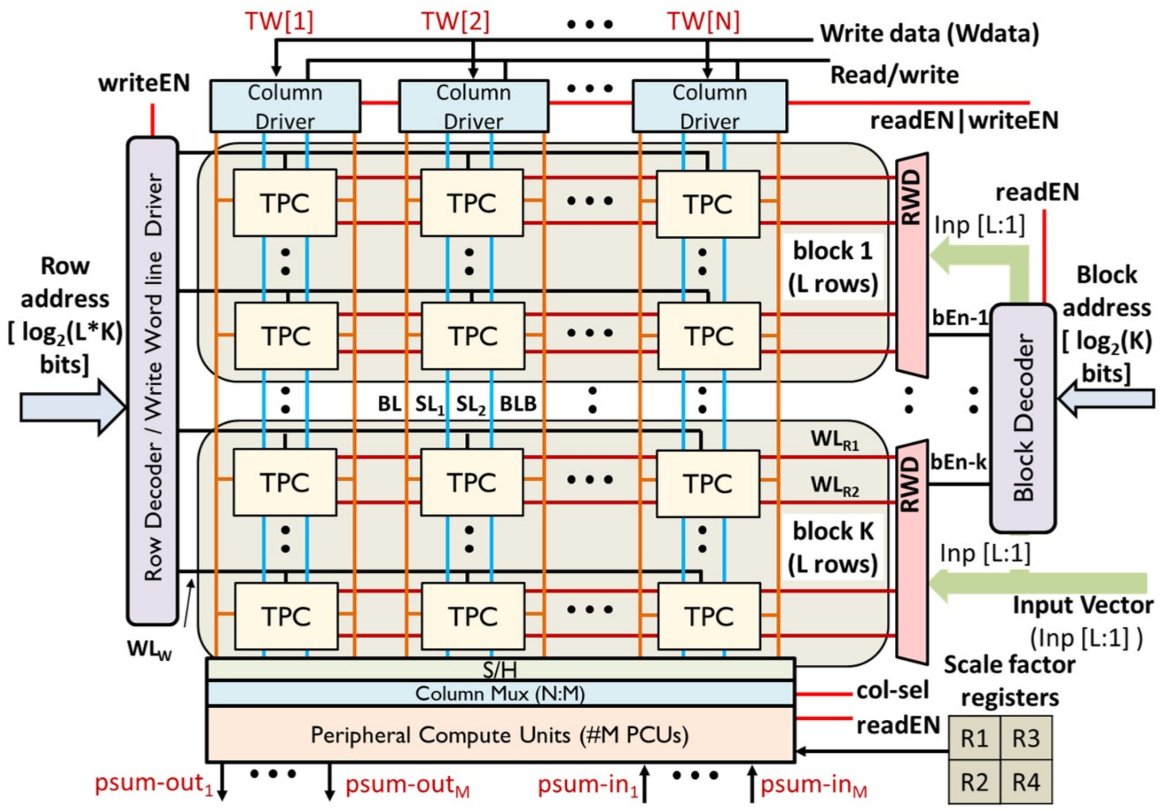


Fig. 7.7.: Ternary in-Memory processing tile

area efficiency, we utilize ‘M’ PCUs per tile (‘M’ $<$ ‘N’) by matching the bandwidth of the PCUs to the bandwidth of the TPC array and operating the PCUs and TPC array as a two-stage pipeline. Next, we discuss the TiM tile peripherals in detail.

Read Wordline Driver (RWD). Figure 7.8(a) shows the RWD logic that takes a ternary vector (**Inp**) and block enable (bEN) signal as inputs and drives all ‘L’ read wordlines (WL_{R1} and WL_{R2}) of a block. The block decoder generates the bEN signal based on the block address that is an input to the TiM tile. WL_{R1} and WL_{R2} are activated using the input encoding scheme shown in Figure 7.2 (Table on the right).

Peripheral Compute Unit (PCU). Figure 7.8(b) shows the logic for a PCU, which consists of two ADCs and a few small arithmetic units (adders and multipliers). The primary function of PCUs is to convert the bitline voltages to digital values using ADCs. However, PCUs also enable other key functions such as partial sum reduction,

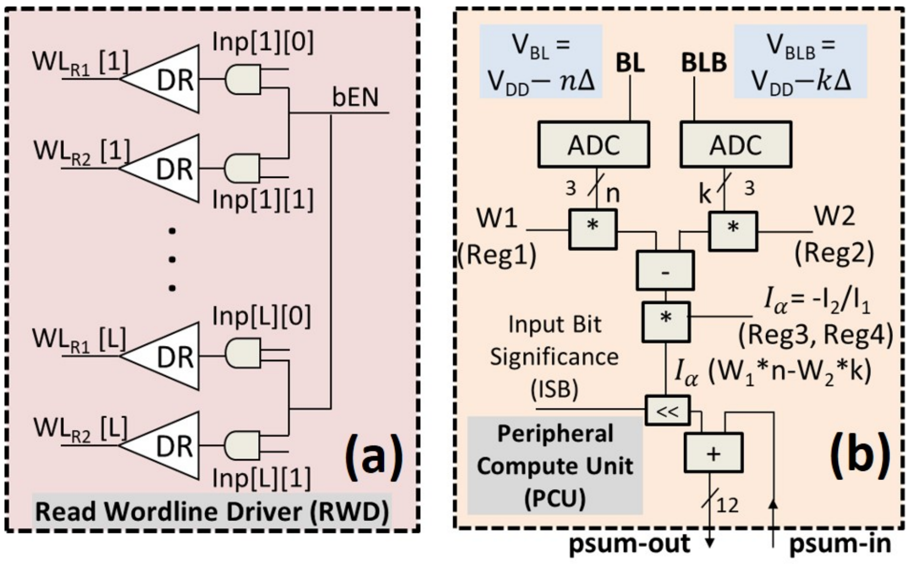

WL_{R1} [1]
DR
Inp[1][0]
bEN
WL_{R2} [1]
Inp[1][1]
Inp[L][0]
WL_{R1} [L]
WL_{R2} [L]
Inp[L][1]
(a)
Read Wordline Driver (RWD)
$V_{BL} = V_{DD} - n\Delta$
BL
BLB
$V_{BLB} = V_{DD} - k\Delta$
ADC
3
n
k
W1 (Reg1)
W2 (Reg2)
$I_\alpha = -I_2/I_1$ (Reg3, Reg4)
Input Bit Significance (ISB)
I_α (W_1*n-W_2*k)
<<
+
12
Peripheral Compute Unit (PCU)
(b)
psum-out
psum-in

Fig. 7.8.: (a) Read Wordline Driver (RWD), (b) Peripheral and Compute Unit (PCU)

and weight (input) scaling for weighted ternary encoding (-W_2,0,W_1) and (-I_2,0,I_1). Although the PCU can be simplified if W_2=W_1=1 or/and I_2=I_1=1, in this work, we target a programmable TiM tile that can support various state-of-the-art ternary DNNs. To further generalize, we use a shifter to support DNNs with ternary weights and higher precision activations [153, 155]. The activations are evaluated bit-serially using multiple TiM accesses. Each access uses an input bit, and we shift the computed partial sum based on the input bit significance using the shifter. TiM tiles have scale factor registers (shown in Figure 7.7) to store the weight and the activation scale factors that vary across layers within a network.

7.1.4 TiM-DNN accelerator architecture

Figure 7.9 shows the proposed TiM-DNN accelerator, which has a hierarchical organization with multiple banks, wherein each bank comprises of several TiM tiles, an activation buffer, a partial sum (Psum) buffer, a global Reduce Unit (RU), a Special Function Unit (SFU), an instruction memory (Inst Mem), and a Scheduler. The compute time and energy in Ternary DNNs are heavily dominated by vector-matrix multiplications which are realized using TiM tiles. Other DNN functions, *viz.*, ReLU, pooling, normalization, Tanh and Sigmoid are performed by the SFU. The partial sums produced by different TiM tiles are reduced using the RU, whereas the partial sums produced by separate blocks within a tile are reduced using PCUs (as discussed in section 7.1.3). TiM-DNN has a small instruction memory and a Scheduler that read instructions and orchestrates operations inside a bank. TiM-DNN also contains activation and Psum buffers to store activations and partial sums, respectively.

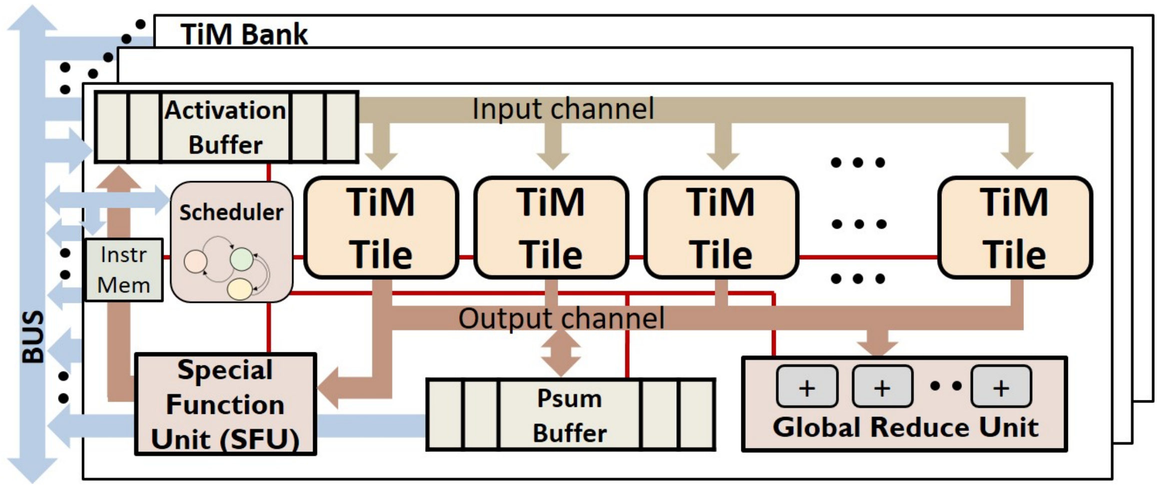


Fig. 7.9.: TiM-DNN accelerator architecture

Mapping. DNNs can be mapped to TiM-DNN both temporally and spatially. The networks that fit on TiM-DNN entirely are mapped spatially, wherein the weight matrix of each convolution (Conv) and fully-connected (FC) layer is partitioned and mapped to dedicated (one or more) TiM tiles, and the network executes in a pipelined fashion. In contrast, networks that cannot fit on TiM-DNN at once are executed using the temporal mapping strategy, wherein we execute Conv and FC layers sequentially over time using all TiM tiles. The weight matrix (W) of each CONV/FC layer could be either smaller or larger than the total weight capacity (TWC) of TiM-DNN. Figure 7.10 illustrates the two scenarios using an example workload (vector-matrix multiplication) that is executed on two separate TiM-DNN instances differing in the number of TiM tiles. As shown, when (W < TWC) the weight matrix partitions (W_1 & W_2) are replicated and loaded to multiple tiles, and each TiM tile computes on input vectors in parallel. In contrast, when (W > TWC), the operations are executed sequentially using multiple steps.

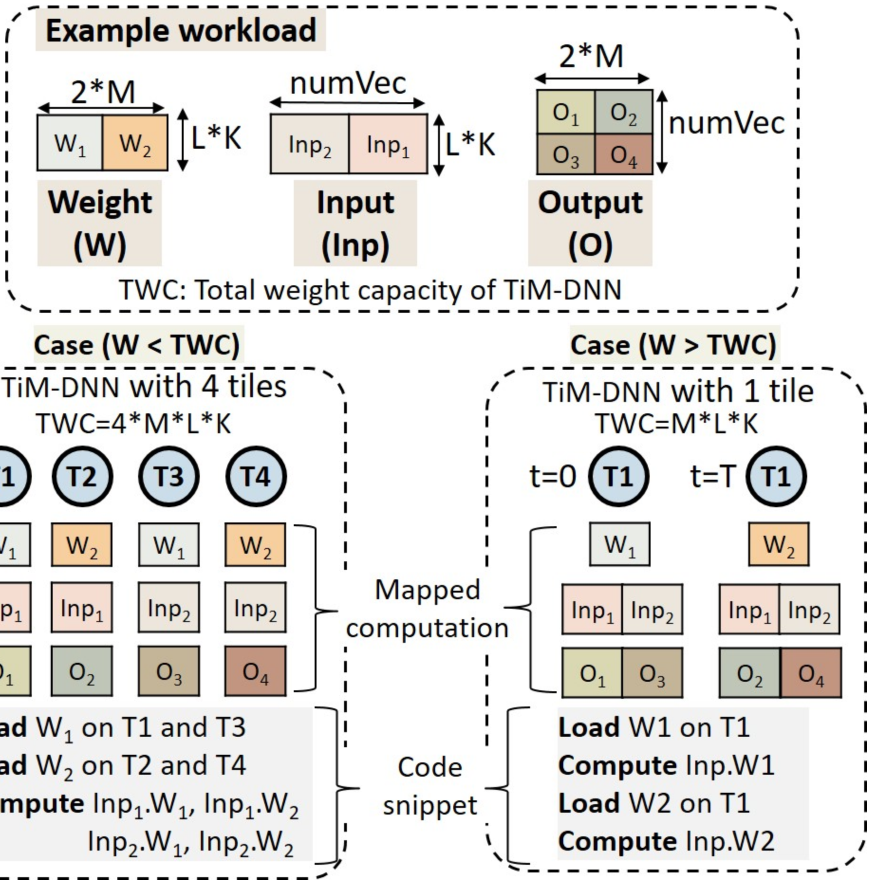


Fig. 7.10.: TiM-DNN mapping: Example

7.2 Experimental Methodology

In this section, we present our experimental methodology for evaluating TiM-DNN.

TiM tile modeling. We perform detailed SPICE simulations to estimate the tile-level energy and latency for the write and vector-matrix multiplication operations. The simulations are performed using 32nm bulk CMOS technology and PTM models.

We use 3-bit flash ADCs to convert bitline voltages to digital values. To estimate the area and latency of digital logic both within the tiles (PCUs and decoders) and outside the tiles (SFU and RU), we synthesized RTL implementations using Synopsys Design Compiler and estimated power consumption using Synopsys Power Compiler. We performed the TPC layout (Figure 7.11) to estimate its area, which is about $720F^2$ (where F is the minimum feature size). We also performed variation analysis to estimate error rates due to incorrect sensing by considering variations in transistor V_T (σ/μ=5%) [161].

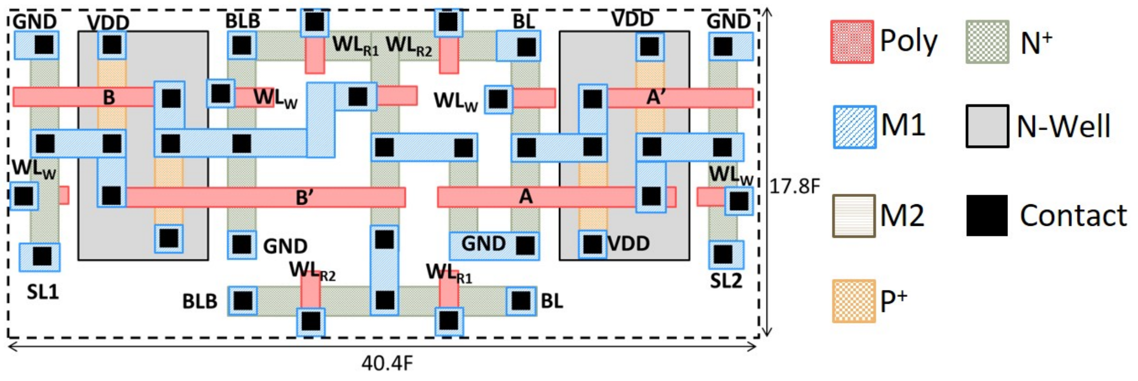


Fig. 7.11.: Ternary Processing Cell (TPC) layout

Table 7.1.: TiM-DNN micro-architectural parameters

Components	**Values**
No. of processing tiles	32 TiM tiles
TiM tile	256x256 TPCs, 32 PCUs, (M=32, N=256, L=K=16)
Buffer (Activation + Psum)	16 KB + 8 KB
I-Mem	128 entries
Global Reduce Unit (RU)	256 adders (12-bit)
Special function unit (SFU)	64 ReLUs, 8 vPE with 4 lanes, 20 SPEs, 32 QUs
Main memory	HBM2 (256 GB/s)

System-level simulation. We developed an architectural simulator to estimate application-level energy and performance benefits of TiM-DNN. The simulator maps various DNN operations, *viz.*, vector-matrix multiplications, pooling, Relu, etc. to TiM-DNN components and produces execution traces consisting of off-chip accesses, write and in-memory operations in TiM tiles, buffer reads and writes, and RU and SFU operations. Using these traces and the timing and energy models from circuit simulation and synthesis, the simulator computes the application-level energy and performance.

TiM-DNN parameters. Table 7.1 details the micro-architectural parameters for the instance of TiM-DNN used in our evaluation, which contains 32 TiM tiles, with each tile having 256x256 TPCs. The SFU consists of 64 Relu units, 8 vector processing elements (vPE) each with 4 lanes, 20 special function processing elements (SPEs), and 32 Quantization Units (QU). SPEs computes special functions such as Tanh and Sigmoid. The output activations are quantized to ternary values using QUs. The latency of the dot-product operation is 2.3 ns. TiM-DNN can achieve a peak performance of 114 TOPs/sec, consumes ~0.9 W power, and occupies ~1.96 mm^2 chip area.

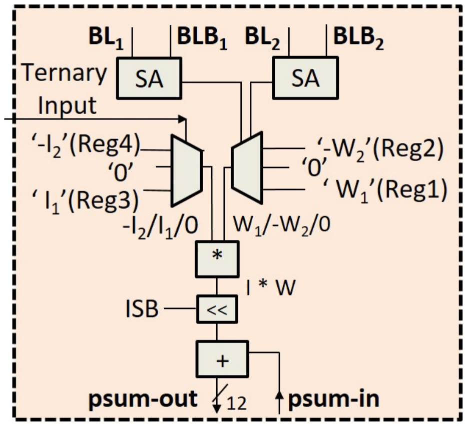


Fig. 7.12.: Near-memory compute unit for the baseline design

Baseline. The processing efficiency (TOPS/W) of TiM-DNN is 300X better than NVIDIA's state-of-the-art Volta V100 GPU [158]. This is to be expected, since the GPU is not specialized for ternary DNNs. In comparison to previous near-memory ternary accelerators [97], TiM-DNN achieves 55.2X improvement in TOPS/W. To perform a fairer comparison and to report the benefits exclusively due to in-memory computations enabled by the proposed TPC, we design a well-optimized near-memory ternary DNN accelerator. This baseline accelerator differs from TiM-DNN in only one aspect — tiles consist of regular SRAM arrays (256x512) with 6T bit-cells and near-memory compute (NMC) units (shown in Figure 7.12), instead of the TiM tiles. Note that, to store a ternary word using the SRAM array, we require two 6T bit-cells. The baseline tiles are smaller than TiM tiles by 0.52x, therefore, we use two baselines designs. (i) An iso-area baseline with 60 baseline tiles and the overall accelerator area is same as TiM-DNN.(ii) An iso-capacity baseline with the same weight storage capacity (2 Mega ternary words) as TiM-DNN. We note that the baseline is well-optimized, and our iso-area baseline can achieve 21.9 TOPs/sec, reflecting an improvement of 17.6X in TOPs/sec over near-memory accelerator for ternary DNNs proposed in [97].

Table 7.2.: DNN benchmarks for evaluating TiM-DNN [153, 154]

Application	CNN Network	FP32 Accuracy	Ternary network		
			Precision [A,W]	**Accuracy**	Quantization method
Image Classification on ImageNet	AlexNet	56.5%	[2,T]	55.8%	WRPN [153]
	ResNet-34	73.59%	[2,T]	73.32%	WRPN [153]
	Inception	71.64%	[2,T]	70.75%	WRPN [153]

Application	RNN Network	PPW* FP32	Ternary network		
			Precision [A,W]	**PPW**	Quantization method
Language Modeling on PTB	LSTM	97.2	[T,T]	110.3	HitNet [154]
	GRU	102.7	[T,T]	113.5	HitNet [154]

*PPW: Perplexity per word (Lower is better)

DNN Benchmarks. We evaluate the system-level energy and performance benefits of TiM-DNN using a suite of DNN benchmarks. Table 7.2 details our benchmark applications. We use state-of-the-art convolutional neural networks (CNN), viz., AlexNet, ResNet-34, and Inception to perform image classification on ImageNet. We also evaluate popular recurrent neural networks (RNN) such as LSTM and GRU that perform language modeling task on the Penn Tree Bank (PTB) dataset [160]. Table 7.2 also details the activation precision and accuracy of these ternary networks.

7.3 Results

In this section, we present various results that quantify the improvements obtained by TiM-DNN. We also compare TiM-DNN with other state-of-the-art DNN accelerators.

7.3.1 Comparison with prior DNN accelerators

We first quantify the advantages of TiM-DNN over prior DNN accelerators using processing efficiencies (TOPS/W and TOPS/mm^2) as our metric. Table 7.3 details 3 prior DNN accelerators including - (i) Nvidia Tesla V100 [158], a state-of-the-art GPU, (ii) Neural-Cache [116], a design using bitwise in-memory operations and bit-serial near-memory arithmetic to realize dot-products, (iii) BRien [97], a near-memory accelerator for ternary DNNs. As shown, TiM-DNN achieves substantial improvements in both TOPS/W and TOPS/mm^2. GPUs, near-memory accelerators [97], and binary in-memory accelerators [116] are less efficient than TiM-DNN as their efficiency is still limited by the on-chip memory bandwidth, wherein they can simultaneously access one [97, 158] or at most two [116] memory rows. In contrast, TiM-DNN offers addition parallelism by simultaneously accessing 'L' (L=16) memory rows to compute in-memory vector-matrix multiplications.

Table 7.3.: Comparison of TiM-DNN with other DNN accelerators

	Brein [97]	Neural Cache [116]	Nvidia GPU (Tesla V100) [158]	TiM-DNN (This work)
Precision	Binary/Ternary	8 bits	8-32 bit	Ternary
Technology	65nm	22nm	12nm	32nm
TOPS/W	2.3	0.529	0.42	127
TOPS/mm^2	0.365	0.2	0.15	58.2

7.3.2 Analysis of performance benefits

We analyze the performance benefits of TiM-DNN over our two baselines (Iso-capacity and Iso-area near-memory accelerators). Figure 7.13 shows the two major components of the normalized inference time which are MAC-Ops (vector-matrix multiplications) and Non-MAC-Ops (other DNN operations) for TiM-DNN (TiM) and the baselines. Overall, we achieve 5.1x-7.7x speedup over the Iso-capacity baseline and 3.2x-4.2x speedup over the Iso-area baseline across our benchmark applications. The speedups depend on the fraction of application runtime spent on MAC-Ops, with DNNs having higher MAC-Ops times attaining superior speedups. This is expected as the performance benefits of TiM-DNN over the baselines derive from accelerating MAC-Ops using in-memory computations. Iso-area (baseline2) is faster than Iso-capacity (baseline1) due to the higher-level of parallelism available from the additional baseline tiles. The 32-tile instance of TiM-DNN achieves 4827, 952, 1834, $2*10^6$, and $1.9*10^6$ inference/sec for AlexNet, ResNet-34, Inception, LSTM, and GRU, respectively. Our RNN benchmarks (LSTM and GRU) fit on TiM-DNN entirely, leading to better inference performance than CNNs.

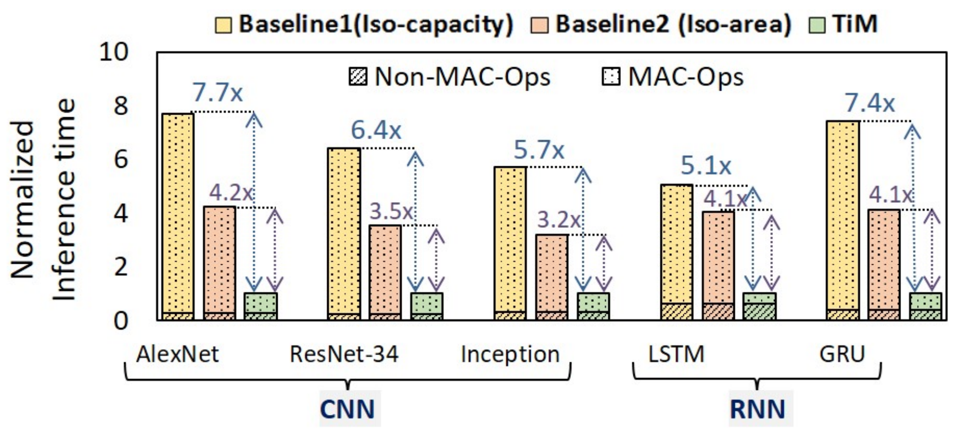


Fig. 7.13.: Performance benefits of TiM-DNN

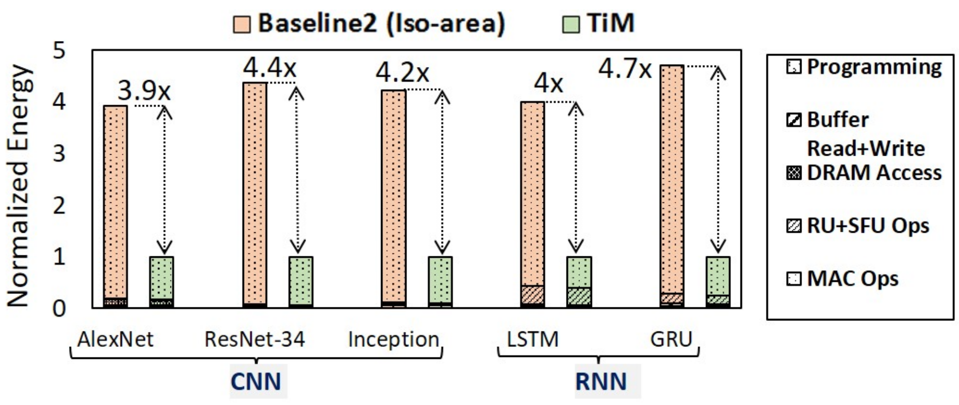


Fig. 7.14.: Energy benefits of TiM-DNN

7.3.3 Analysis of energy benefits

We now analyze the application level energy benefits of TiM-DNN over the superior of the two baselines (Baseline2). Figure 7.14 shows major energy components for TiM-DNN and Baseline2, which are programming (writes to TiM tiles), DRAM accesses, reads (writes) from (to) activation and Psum buffers, operations in reduce units and special function units (RU+SFU Ops), and MAC-Ops. As shown, TiM reduces the MAC-Ops energy substantially and achieves 3.9x-4.7x energy improvements across our DNN benchmarks. The primary cause for this energy reduction is that TiM-DNN computes on 16 rows simultaneously per array access.

7.3.4 Kernel-level benefits

To provide more insights on the application-level benefits, we compare the TiM tile and the baseline tile at the kernel-level. We consider a primitive DNN kernel, *i.e.*, a vector-matrix computation (Out = Inp*W, where Inp is a 1x16 vector and W is a 16x256 matrix), and map it to both TiM and baseline tiles. We use two variants of TiM tile, (i) TiM-8 and (ii) TiM-16, wherein we simultaneously activate 8 wordlines and 16 wordlines, respectively. Using the baseline tile, the vector-matrix multiplication operation requires row-by-row sequential reads, resulting in 16 SRAM accesses. In contrast, TiM-16 and TiM-8 require 1 and 2 accesses, respectively. Figure 7.15 shows that the TiM-8 and TiM-16 designs achieve a speedup of 6x and 11.8x respectively, over the baseline design. Note that the benefits are lower than 8x and 16x, respectively, as SRAM accesses are faster than TiM-8 and TiM-16 accesses.

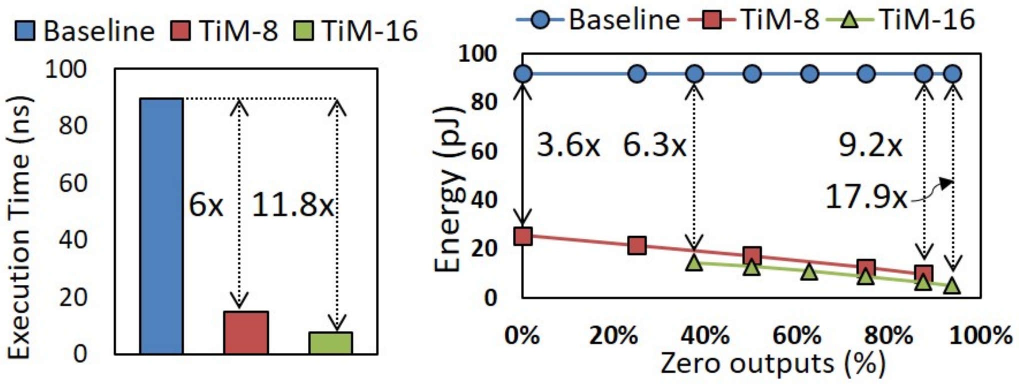


Fig. 7.15.: Kernel-level benefits of TiM tile

Next, we compare the energy consumption in TiM-8, TiM-16, and baseline designs for the above kernel computation. In TiM-8 and TiM-16, the bit-lines are discharged twice and once, respectively, whereas, in the baseline design the bit-lines discharge multiple (16*2) times. Therefore, TiM tiles achieve substantial energy benefits over the baseline design. The additional factor '2' in (16*2) arises as the SRAM array uses two 6T bit-cells for storing a ternary word. However, the energy benefits of TiM-8 and TiM-16 is not 16x and 32x, respectively, as TiM tiles discharge the bitlines by a larger amount (multiple Δs). Further, the amount by which the bitlines get discharged in TiM tiles depends on the number of non-zero scalar outputs. For example, in TiM-8, if 50% of the TPCs output in a column are zeros the bitline discharges by 4Δ, whereas if 75% are zeros the bitline discharges by 2Δ. Thus, the energy benefits over the baseline design are a function of the output sparsity (fraction of outputs that are zero). Figure 7.15 shows the energy benefits of TiM-8 and TiM-16 designs over the baseline design at various output sparsity levels.

7.3.5 TiM-DNN area breakdown

We now discuss the area breakdown of various components in TiM-DNN. Figure 7.16 shows the area breakdown of the TiM-DNN accelerator, a TiM tile, and a baseline tile. The major area consumer in TiM-DNN is the TiM-tile. In the TiM and baseline tiles, area mostly goes into the core-array that consists of TPCs and 6T bit-cells, respectively. Further, as discussed in section 7.2, TiM tiles are 1.89x larger than the baseline tile at iso-capacity. Therefore, we use the iso-area baseline with 60 tiles and compare it with TiM-DNN having 32 TiM tiles.

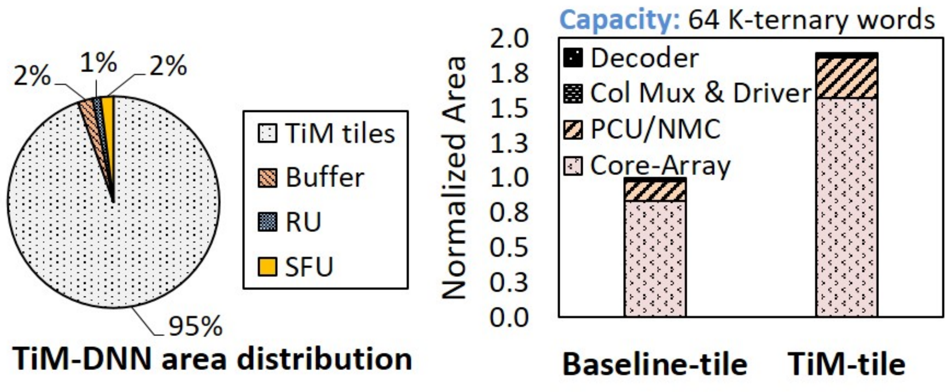


Fig. 7.16.: TiM-DNN area breakdown

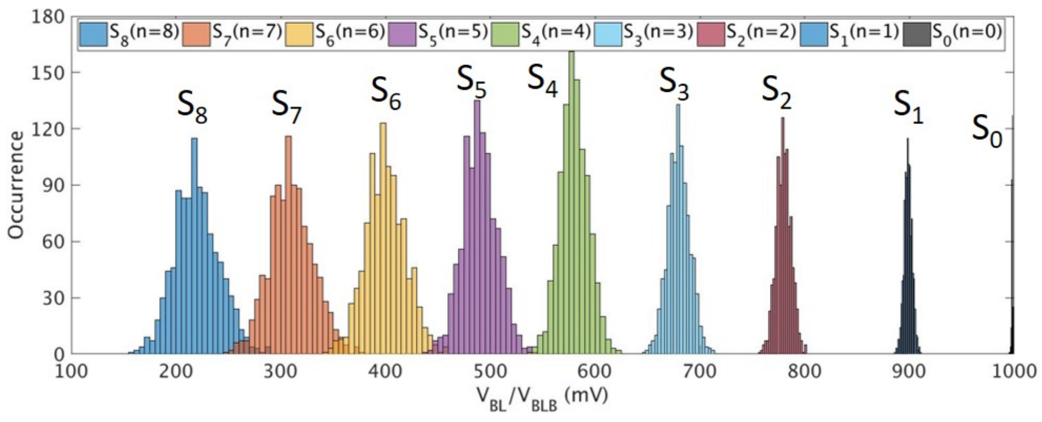


Fig. 7.17.: Histogram of the bit-line voltages (V_{BL}/V_{BLB}) under process variations

7.3.6 Impact of under process variations

Finally, we study the impact of process variations on the computations (*i.e.*, ternary vector-matrix multiplications) performed using TiM-DNN. To that end, we first perform Monte-Carlo circuit simulation of ternary dot-product operations executed in TiM tiles with n_{max} = 8 and L = 16 to determine the sensing errors under random variations. We consider variations ($\sigma/\mu = 5\%$) [161] in the threshold voltage (V_T) of all transistors in each and every TPC. We evaluate 1000 samples for every possible BL/BLB state (S_0 to S_8) and determine the spread in the final bitline voltages (V_{BL}/V_{BLB}). Figure 7.17 shows the histogram of the obtained V_{BL} voltages of all possible states across these random samples. As mentioned in section 7.1.2, the state S_i represents n = i, where 'n' is the ADC Output. We can observe in the figure that some of the neighboring histograms slightly overlap, while the others do not. For example, the histograms S_7 and S_8 overlap but S_1 and S_2 do not. The overlapping areas in the figure represent the samples that will result in sensing errors (SEs). However, the overlapping areas are very small, indicating that the probability of the sensing error (P_{SE}) is extremely low. Further, the sensing errors depend on 'n', and we represent this dependency as the conditional sensing error probability

[P_{SE}(SE/n)]. It is also worth mentioning that the error magnitude is always ±1, as only the adjacent histograms overlap.

$$P_E = \sum_{n=0}^{8} P_{SE}(SE/n) * P_n \tag{7.1}$$

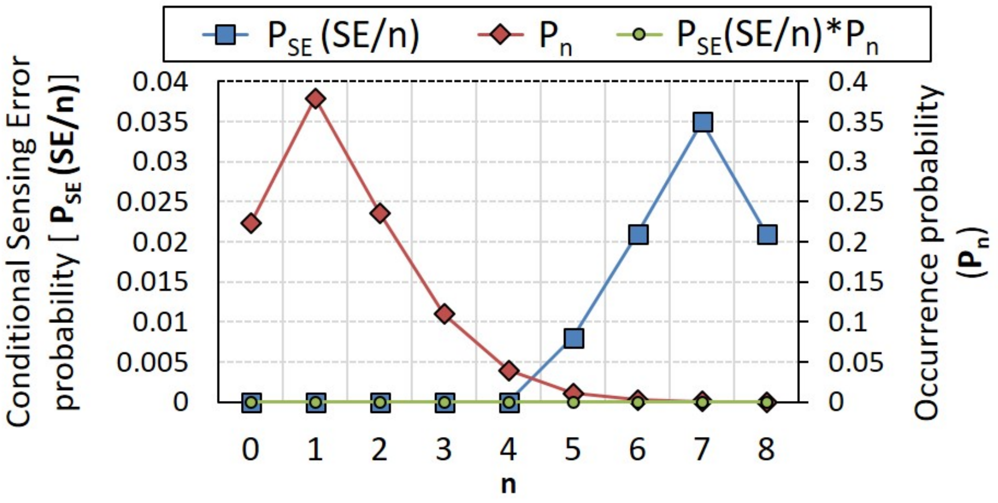


Fig. 7.18.: Error probability during vector-matrix multiplications

Equation 7.1 details the probability (P_E) of error in the ternary vector-matrix multiplications executed using TiM tiles, where $P_{SE}(SE/n)$ and P_n are the conditional sensing error probability and the occurrence probability of the state S_n (ADC-Out = n), respectively. Figure 7.18 shows the values of $P_{SE}(SE/n)$, P_n, and their product ($P_{SE}(SE/n)$*P_n) for each n. $P_{SE}(SE/n)$ is obtained using the Monte-Carlo simulation (described above), and P_n is computed using the traces of the partial sums obtained from sample ternary DNNs [153,154]. As shown in Figure 7.18, P_n is maximum at n=1 and drastically decreases with higher values of n. In contrast, $P_{SE}(SE/n)$ shows an opposite trend, wherein the probability of sensing error is higher for larger n. Therefore, we find the product $P_{SE}(SE/n)$*P_n to be quite small across all values on n. In our evaluation, the P_E is found to be 1.5*10^{-4}, reflecting an extremely low probability of error. In other words, we have roughly 2 errors of magnitude (±1) for every 10K ternary vector matrix multiplications executed using TiM-DNN. In our experiments, we found that $P_E = 1.5*10^{-4}$ has no impact on the application level accuracy. We note that this is due to the low probability and magnitude of error as well as the ability of DNNs to tolerate errors in their computations [87].

7.4 Summary

Ternary DNNs are extremely promising due to their ability to achieve accuracy similar to full-precision networks on complex machine learning tasks, while enabling DNN inference at low energy. In this work, we present TiM-DNN, an in-memory accelerator for executing state-of-the-art ternary DNNs. TiM-DNN is programmable accelerator designed using TiM tiles, *i.e.*, specialized memory arrays for realizing massively parallel signed vector-matrix multiplications with ternary values. TiM tiles consist of a new Ternary Processing Cell (TPC) that functions as both a ternary storage unit and a scalar multiplication unit. We evaluate an embodiment of TiM-DNN with 32 TiM tiles and demonstrate that TiM-DNN achieves significant energy and performance improvements over a well-optimized near-memory accelerator baseline.

8. CONCLUSION

The high computation and storage demands of modern computing workloads pose significant challenges to the efficiency of the computing platforms on which they are deployed. Traditional Von Neumann systems such as multi-cores and GPUs have been the mainstay computing platforms for past decades. However, their computing performance and efficiency on modern workloads (*e.g.*, DNNs) are limited by the on-chip data-transfer and memory bandwidth. In-memory computing is an emerging computing paradigm that enables massively parallel functions to be computed within the memory array itself. Consequently, reducing the time and energy spend in data-transfers and memory accesses significantly to attain performance and energy efficiency beyond current Von Neumann systems.

8.1 Thesis summary

In this dissertation, we propose new in-memory architectures to boost the system-level performance and energy efficiency of modern workloads. We also address key challenges associated with existing resistive crossbar based in-memory computing fabrics to enable this future adoption. The key contributions of this dissertation are summarized below.

- In this dissertation, we explore in-memory computing using STT-MRAM that is a promising candidate for future on-chip memories. The thesis proposes STT-CiM, an enhanced STT-MRAM that can perform a range of arithmetic, logic, and vector compute-in-memory operations. It addresses a key challenge associated with these in-memory operations, i.e., reliable computation under process variations. The thesis utilizes the proposed design (STT-CiM) as a scratchpad in the memory hierarchy of a programmable processor using ISA extensions and on-chip bus enhancements to support in-memory computations. To further the system-level performance and energy improvements due to STT-CiM, it proposes architectural optimizations and data mapping techniques. A device-to-architecture simulation framework was used for evaluating the benefits of STT-CiM. The experiments indicate that STT-CiM achieves substantial improvements in energy and performance, and shows considerable promise in alleviating the processor-memory gap.

- The thesis addresses a key challenge in resistive crossbar based in-memory computing fabrics, i.e., they suffer from numerous device and circuit-level non-idealities leading to degradation in application-level accuracy of large-scale DNNs. To quantify the impact of crossbar non-idealities on the accuracy of large-scale DNNs, a fast and scalable framework that can be integrated into state-of-the-art DNN software frameworks is needed. The dissertation addresses this need and proposes FCM, i.e., a fast and accurate functional crossbar model that abstracts crossbar non-idealities to achieve orders of magnitude speedup over the existing circuit simulations. It also presents RxNN, a software simulation framework for evaluating and re-training large-scale DNNs on resistive crossbar systems. The thesis evaluated several state-of-the-art neural networks and found that the errors due to non-idealities can degrade the overall accuracy of large scale DNNs considerably, thereby highlighting the need for further research in error correction and compensation schemes to enable adoption of resistive crossbar system.

- In this thesis, we also exemplify a compensation scheme for resistive crossbar system. We propose CxDNN, a hardware-software methodology comprising of a conversion algorithm, a hardware-instance-independent re-training mechanism, and a low-overhead hardware compensation scheme to design high accuracy DNNs on resistive crossbars. CxDNN preserves the train-once-deploy-anywhere tenet of current DNN implementations, wherein DNNs are re-trained once to obtain a common model that can be deployed on many resistive crossbar systems. Further, CxDNN utilizes low-overhead hardware to mitigate instance-specific errors. Our evaluations reveal that CxDNN can largely restore the accuracy loss due to non-idealities over a range of large-scale DNN benchmarks. Therefore, CxDNN is a promising approach to compensate for errors due to crossbar non-idealities and addresses a key bottleneck to the adoption of resistive crossbar based neural fabrics.

- Finally, the dissertation also proposes TiM-DNN, an in-memory accelerator for executing state-of-the-art ternary DNNs. TiM-DNN is programmable accelerator designed using TiM tiles, i.e., specialized memory arrays for realizing massively parallel signed vector-matrix multiplications with ternary values. TiM tiles consist of a new Ternary Processing Cell (TPC) that functions as both a ternary storage unit and a scalar multiplication unit. The evaluation of an embodiment of TiM-DNN with 32 TiM tiles and demonstrate that TiM-DNN achieves significant energy and performance improvements over well-optimized near-memory accelerators.

REFERENCES

REFERENCES

[1] R. Parloff, "The AI Revolution: Why Deep Learning Is Suddenly Changing Your Life. http://fortune.com/ai-artificial-intelligence-deep-machine-learning/ ," Online. Accessed Sept. 17, 2017. [Online]. Available: http://fortune.com/ai-artificial-intelligence-deep-machine-learning/

[2] C. Metz, "Google, Facebook and Microsoft are remaking themselves around AI. https://wired.com/2016/11/google-facebook-microsoft-remaking-around-ai/ ," Online. Accessed Sept. 17, 2017.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[4] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent ram (IRAM): Chips that remember and compute," in *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*. IEEE, 1997, pp. 224–225.

[5] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A computation model for intelligent memory," in *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, Jun 1998, pp. 192–203.

[6] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, no. 6, pp. 68–74, 2001.

[7] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang *et al.*, "The architecture of the DIVA processing-in-memory chip," in *Proc. ICS*. ACM, 2002, pp. 14–25.

[8] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[9] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan, "Near-Memory Data Services," *IEEE Micro*, vol. 36, no. 1, pp. 6–13, Jan 2016.

[10] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 380–392.

[11] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 283–295.

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

[13] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600213

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

[15] Q. Zhu, K. Vaidyanathan, O. Shacham, M. Horowitz, L. Pileggi, and F. Franchetti, "Design automation framework for application-specific logic-in-memory blocks," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, July 2012, pp. 125–132.

[16] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips*, vol. 23, 2011.

[17] D. Lee et al., "25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *Proc. ISSCC*. IEEE, 2014, pp. 432–433.

[18] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[19] M. Kang, M. S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 8326–8330.

[20] J. Wang and J. Harms, "General structure for computational random access memory (CRAM)," Nov. 13 2014, uS Patent App. 14/259,568. [Online]. Available: https://www.google.je/patents/US20140334216

[21] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 173:1–173:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898064

[22] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, July 2016.

[23] J. Reuben, R. Ben Hur, N. Wald, N. Talati, A. Haj Ali, P. Emmanuel, and S. Kvatinsky, "Memristive Logic: A framework for evaluation and comparison," in *Proceeding of the IEEE International Symposium on Power and Timing Modeling, Optimization and Simulation*, Sep, 2017.

[24] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, July 2015.

[25] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "AC-DIMM: Associative Computing with STT-MRAM," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485939

[26] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, "In-Memory Processing Paradigm for Bitwise Logic Operations in STT-MRAM," *IEEE Transactions on Magnetics*, vol. PP, no. 99, pp. 1–1, 2017.

[27] J. Zhang, Z. Wang, and N. Verma, "A machine-learning classifier implemented in a standard 6T SRAM array," in *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on.* IEEE, June 2016, pp. 1–2.

[28] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, and J. Yang, "RENO: A high-efficient reconfigurable neuromorphic computing accelerator design," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[29] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "SPINDLE: SPINtronic Deep Learning Engine for large-scale neuromorphic computing," in *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*, Aug 2014, pp. 15–20.

[30] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 27–39.

[31] "http://www.everspin.com/ ."

[32] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM)," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 13:1–13:35, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2463585.2463589

[33] "http://www.avalanche-technology.com/ ."

[34] H. Yoda, S. Fujita, N. Shimomura, E. Kitagawa, K. Abe, K. Nomura, H. Noguchi, and J. Ito, "Progress of stt-mram technology and the effect on normally-off computing systems," in *2012 International Electron Devices Meeting*, Dec 2012, pp. 11.3.1–11.3.4.

[35] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 243–252.

[36] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy Reduction for STT-RAM Using Early Write Termination," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09, Nov. 2009, pp. 264–268. [Online]. Available: http://doi.acm.org/10.1145/1687399.1687448

[37] S. Chatterjee, M. Rasquinha, S. Yalamanchili, and S. Mukhopadhyay, "A Scalable Design Methodology for Energy Minimization of STTRAM: A Circuit and Architecture Perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 5, pp. 809–817, May 2011.

[38] Y. Kim, S. K. Gupta, S. P. Park, G. Panagopoulos, and K. Roy, "Write-optimized Reliable Design of STT MRAM," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333664

[39] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita, "7.5 A 3.3ns-access-time 71.2 uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture," in *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, Feb 2015, pp. 1–3.

[40] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy, "Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture," in *Proceedings of the Design Automation Conference*, Jun. 2012, pp. 492–497.

[41] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2011, pp. 50 –61.

[42] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of Last-Level On-Chip Cache Using Spin-Torque Transfer RAM (STT RAM)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 3, pp. 483–493, March 2011.

[43] K. W. Kwon, X. Fong, P. Wijesinghe, P. Panda, and K. Roy, "High-Density and Robust STT-MRAM Array Through Device/Circuit/Architecture Interactions," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 1024–1034, Nov 2015.

[44] B. D. Bel, J. Kim, C. H. Kim, and S. S. Sapatnekar, "Improving STT-MRAM density through multibit error correction," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[45] W. Kang, W. Zhao, Z. Wang, Y. Zhang, J.-O. Klein, Y. Zhang, C. Chappert, and D. Ravelosona, "A low-cost built-in error correction circuit design for STT-MRAM reliability improvement," *Microelectronics Reliability*, vol. 53, no. 9, pp. 1224–1229, 2013.

[46] W. Kang, L. Zhang, W. Zhao, J.-O. Klein, Y. Zhang, D. Ravelosona, and C. Chappert, "Yield and Reliability Improvement Techniques for Emerging Nonvolatile STT-MRAM," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 5, no. 1, pp. 28–39, March 2015.

[47] X. Fong, Y. Kim, S. Choday, and K. Roy, "Failure Mitigation Techniques for 1T-1MTJ Spin-Transfer Torque MRAM Bit-cells," *IEEE Trans. VLSI Systems*, vol. 22, no. 2, pp. 384–395, Feb 2014.

[48] G. S. Kar, W. Kim, T. Tahmasebi, J. Swerts, S. Mertens, N. Heylen, and T. Min, "Co/Ni based p-MTJ stack for sub-20nm high density stand alone and high performance embedded memory application," in *2014 IEEE International Electron Devices Meeting*, Dec 2014, pp. 19.1.1–19.1.4.

[49] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate Storage for Energy Efficient Spintronic Memories," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015, pp. 195:1–195:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744799

[50] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das, "Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 69–80.

[51] K. Lee and S. H. Kang, "Development of Embedded STT-MRAM for Mobile System-on-Chips," *IEEE Transactions on Magnetics*, vol. 47, no. 1, pp. 131–136, Jan 2011.

[52] A. Nigam, C. W. Smullen, V. Mohan, E. Chen, S. Gurumurthi, and M. R. Stan, "Delivering on the promise of universal memory for spin-transfer torque RAM (STT-RAM)," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug 2011, pp. 121–126.

[53] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug 2011, pp. 79–84.

[54] Y. Zhang, W. Zhao, J. O. Klein, W. Kang, D. Querlioz, C. Chappert, and D. Ravelosona, "Multi-level cell Spin Transfer Torque MRAM based on stochastic switching," in *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, Aug 2013, pp. 233–236.

[55] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2012, pp. 81–87.

[56] W. Xu, Y. Chen, X. Wang, and T. Zhang, "Improving STT MRAM Storage Density Through Smaller-than-worst-case Transistor Sizing," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 87–90. [Online]. Available: http://doi.acm.org/10.1145/1629911.1629936

[57] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, "An energy efficient cache design using Spin Torque Transfer (STT) RAM," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug 2010, pp. 389–394.

[58] A. Aziz, N. Shukla, S. Datta, and S. K. Gupta, "COAST: Correlated material assisted STT MRAMs for optimized read operation," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2015, pp. 1–6.

[59] S. Ikeda, J. Hayakawa, Y. Ashizawa, Y. Lee, K. Miura, H. Hasegawa, M. Tsunoda, F. Matsukura, and H. Ohno, "Tunnel magnetoresistance of 604% at 300 K by suppression of Ta diffusion in Co Fe B/ Mg O/ Co Fe B pseudo-spin-valves annealed at high temperature," *Applied Physics Letters*, vol. 93, no. 8, p. 082508, 2008.

[60] W. Kang, L. Zhang, J. O. Klein, Y. Zhang, D. Ravelosona, and W. Zhao, "Reconfigurable Codesign of STT-MRAM Under Process Variations in Deeply Scaled Technology," *IEEE Transactions on Electron Devices*, vol. 62, no. 6, pp. 1769–1777, June 2015.

[61] N. N. Mojumder, X. Fong, C. Augustine, S. K. Gupta, S. H. Choday, and K. Roy, "Dual Pillar Spin-transfer Torque MRAMs for Low Power Applications," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 14:1–14:17, May 2013. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/2463585.2463590

[62] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano letters*, vol. 10, no. 4, pp. 1297–1301, 2010.

[63] W. H. Chen, W. S. Khwa, J. Y. Li, W. Y. Lin, H. T. Lin, Y. Liu, Y. Wang, H. Wu, H. Yang, and M. F. Chang, "Circuit design for beyond von Neumann applications using emerging memory: From nonvolatile logics to neuromorphic computing," in *2017 18th International Symposium on Quality Electronic Design (ISQED)*, March 2017, pp. 23–28.

[64] A. Sengupta, Y. Shim, and K. Roy, "Proposal for an All-Spin Artificial Neural Network: Emulating neural and synaptic functionalities through domain wall motion in ferromagnets," *IEEE transactions on biomedical circuits and systems*, vol. 10, no. 6, pp. 1152–1160, 2016.

[65] D. Fan, Y. Shim, A. Raghunathan, and K. Roy, "STT-SNN: A Spin-Transfer-Torque Based Soft-Limiting Non-Linear Neuron for Low-Power Artificial Neural Networks," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 1013–1023, Nov 2015.

[66] X. Liu, M. Mao, B. Liu, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, J. Yang, H. Li, and Y. Chen, "Harmonica: A Framework of Heterogeneous Computing Systems With Memristor-Based Neuromorphic Computing Accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 5, pp. 617–628, May 2016.

[67] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "TIME: A Training-in-memory Architecture for Memristor-based Deep Neural Networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 26:1–26:6. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/3061639.3062326

[68] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of BSB recall function using memristor crossbar arrays," in *DAC Design Automation Conference 2012*, June 2012, pp. 498–503.

[69] 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*, March 2017, pp. 19–24.

[70] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[71] I. Kataeva, F. Merrikh-Bayat, E. Zamanidoost, and D. Strukov, "Efficient training algorithms for neural networks based on memristive crossbar circuits," in *Neural Networks (IJCNN), 2015 International Joint Conference on.* IEEE, 2015, pp. 1–8.

[72] B. Yan, J. Yang, Q. Wu, Y. Chen, and H. Li, "A closed-loop design to enhance weight stability of memristor based neural network chips," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 541–548.

[73] P.-Y. Chen, B. Lin, I.-T. Wang, T.-H. Hou, J. Ye, S. Vrudhula, J.-s. Seo, Y. Cao, and S. Yu, "Mitigating Effects of Non-ideal Synaptic Device Characteristics for On-chip Learning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 194–199. [Online]. Available: http://dl.acm.org/citation.cfm?id=2840819.2840848

[74] Y. Wang, W. Wen, B. Liu, D. M. Chiarulli, and H. H. Li, "Group scissor: Scaling neuromorphic computing design to big neural networks," *CoRR*, vol. abs/1702.03443, 2017. [Online]. Available: http://arxiv.org/abs/1702.03443

[75] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[76] B. Liu, M. Hu, H. Li, Z.-H. Mao, Y. Chen, T. Huang, and W. Zhang, "Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–6.

[77] B. Li, Y. Wang, Y. Wang, Y. Chen, and H. Yang, "Training itself: Mixed-signal training acceleration for memristor-based neural network," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 361–366.

[78] B. Liu, H. Li, Y. Chen, X. Li, T. Huang, Q. Wu, and M. Barnell, "Reduction and IR-drop compensations techniques for reliable neuromorphic computing systems," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2014, pp. 63–70.

[79] L. Xia, B. Li, T. Tang, P. Gu, X. Yin, W. Huangfu, P. Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation platform for memristor-based neuromorphic computing system," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 469–474.

[80] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang, "Technological exploration of RRAM crossbar array for matrix-vector multiplication," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 106–111.

[81] W. Wen, C. R. Wu, X. Hu, B. Liu, T. Y. Ho, X. Li, and Y. Chen, "An EDA framework for large scale hybrid neuromorphic computing systems," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[82] B. Liu, W. Wen, Y. Chen, X. Li, C.-R. Wu, and T.-Y. Ho, "EDA Challenges for Memristor-Crossbar Based Neuromorphic Computing," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 185–188. [Online]. Available: http://doi.acm.org/10.1145/2742060.2743754

[83] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[84] S. Kim, T. Gokmen, H. Lee, and W. E. Haensch, "Analog CMOS-based Resistive Processing Unit for Deep Neural Network Training," *CoRR*, vol. abs/1706.06620, 2017. [Online]. Available: http://arxiv.org/abs/1706.06620

[85] T. Li, X. Bi, N. Jing, X. Liang, and L. Jiang, "Sneak-Path Based Test and Diagnosis for 1R RRAM Crossbar Using Voltage Bias Technique," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, June, 2017, pp. 38:1–38:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062318

[86] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-mann: A crossbar based architecture for memory augmented neural networks," in *Proceedings of the 56th Annual Design Automation Conference*

2019, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 130:1–130:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317935

[87] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient Neuromorphic Systems Using Approximate Computing," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ser. ISLPED '14. New York, NY, USA: ACM, 2014, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2627369.2627613

[88] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *CoRR*, vol. abs/1502.02551, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551

[89] H. Tann, S. Hashemi, I. Bahar, and S. Reda, "Hardware-Software Codesign of Accurate, Multiplier-free Deep Neural Networks," *CoRR*, vol. abs/1705.04288, 2017. [Online]. Available: http://arxiv.org/abs/1705.04288

[90] S. Jain, S. Venkataramani, and A. Raghunathan, "Approximation through logic isolation for the design of quality configurable circuits," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 612–617.

[91] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[92] T. Hanyu, "Challenge of MTJ/MOS-hybrid logic-in-memory architecture for nonvolatile VLSI processor," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May 2013, pp. 117–120.

[93] M. Natsui, D. Suzuki, N. Sakimura, R. Nebashi, Y. Tsuji, A. Morioka, T. Sugibayashi, S. Miura, H. Honjo, K. Kinoshita, S. Ikeda, T. Endoh, H. Ohno, and T. Hanyu, "Nonvolatile Logic-in-Memory LSI using cycle-based power gating and its application to motion-vector prediction," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 2, pp. 476–489, Feb 2015.

[94] S. Matsunaga, J. Hayakawa, S. Ikeda, K. Miura, T. Endoh, H. Ohno, and T. Hanyu, "MTJ-based nonvolatile logic-in-memory circuit, future prospects and issues," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 433–435.

[95] M.-F. Chang, A. Lee, C.-C. Lin, M.-S. Ho, P.-C. Chen, C.-C. Kuo, M.-P. Chen, P.-L. Tseng, T.-K. Ku, C.-F. Chen, K.-S. Li, and J.-M. Shieh, "Read circuits for resistive memory (ReRAM) and memristor-based nonvolatile Logics," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 569–574.

[96] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," *CoRR*, vol. abs/1609.00222, 2016. [Online]. Available: http://arxiv.org/abs/1609.00222

[97] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, T. Kuroda, and M. Motomura, "Brein memory: A 13-layer 4.2 k neuron/0.8 m synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm cmos," in *2017 Symposium on VLSI Circuits*, June 2017, pp. C24–C25.

[98] S. Jain, S. Sapatnekar, J. Wang, K. Roy, and A. Raghunathan, "Computing-in-memory with spintronics," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1640–1645.

[99] S. K. Thirumala, S. Jain, A. Raghunathan, and S. K. Gupta, "Non-volatile memory utilizing reconfigurable ferroelectric transistors to enable differential read and energy-efficient in-memory computation," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2019, pp. 1–6.

[100] D. Lee, X. Fong, and K. Roy, "R-MRAM: A ROM-Embedded STT MRAM Cache," *IEEE Electron Device Letters*, vol. 34, no. 10, pp. 1256–1258, Oct 2013.

[101] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[102] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 912–925, June 2019.

[103] S. Sen and A. Raghunathan, "Approximate computing for long short term memory (lstm) neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2266–2276, Nov 2018.

[104] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *CoRR*, vol. abs/1511.00363, 2015. [Online]. Available: http://arxiv.org/abs/1511.00363

[105] Z. Lin, M. Courbariaux, R. Memisevic, Y. Bengio, "Neural Networks with Few Multiplications," *CoRR*, vol. abs, 2015. [Online]. Available: http://arxiv.org/abs/1510.03009

[106] S. H. et al., "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[107] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 13–26, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080244

[108] B. R. et al., "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. ISCA*, June 2016.

[109] X. Sun, S. Yin, X. Peng, R. Liu, J. Seo, and S. Yu, "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1423–1428.

[110] X. Sun, X. Peng, P. Chen, R. Liu, J. Seo, and S. Yu, "Fully parallel RRAM synaptic array for implementing binary neural network with (+1, -1) weights and (+1, 0) neurons," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2018, pp. 574–579.

[111] J. Zhang, Z. Wang, and N. Verma, "In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, April 2017.

[112] A. Biswas and A. P. Chandrakasan, "Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, Feb 2018, pp. 488–490.

[113] R. Liu, X. Peng, X. Sun, W.-S. Khwa, X. Si, J.-J. Chen, J.-F. Li, M.-F. Chang, and S. Yu, "Parallelizing sram arrays with customized bit-cell for binary neural networks," in *Proceedings of the 55th Annual Design Automation Conference*. New York, NY, USA: ACM, 2018, pp. 21:1–21:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3196089

[114] Z. Jiang, S. Yin, M. Seok, and J. Seo, "XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks," in *2018 IEEE Symposium on VLSI Technology*, June 2018, pp. 173–174.

[115] A. Agrawal, A. Jaiswal, B. Han, G. Srinivasan, and K. Roy, "Xcel-RAM: Accelerating Binary Neural Networks in High-Throughput SRAM Compute Arrays," *CoRR*, vol. abs/1807.00343, 2018. [Online]. Available: http://arxiv.org/abs/1807.00343

[116] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 383–396. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00040

[117] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy, "8t SRAM cell as a multi-bit dot product engine for beyond von-neumann computing," *CoRR*, vol. abs/1802.08601, 2018. [Online]. Available: http://arxiv.org/abs/1802.08601

[118] M. Kang, S. Gonugondla, M.-S. Keel, and N. R. Shanbhag, "An energy-efficient memory-based high-throughput VLSI architecture for convolutional networks," vol. 2015, pp. 1037–1041, 08 2015.

[119] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *CoRR*, vol. abs/1705.06963, 2017. [Online]. Available: http://arxiv.org/abs/1705.06963

[120] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Letters*, vol. 10, no. 4, pp. 1297–1301, 2010, pMID: 20192230.

[121] 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*, vol. 100, no. 6, pp. 1951–1970, 2012.

[122] M. Sharad, G. Panagopoulos, and K. Roy, “Spin neuron for ultra low power computational hardware,” *70th Device Research Conference*, pp. 221–222, 2012.

[123] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. Hwu, J. P. Strachan, K. Roy, and D. Milojicic, “PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[124] P. Chen, X. Peng, and S. Yu, “Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[125] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 14–26.

[126] I. Chakraborty, D. Roy, and K. Roy, “Technology aware training in memristive neuromorphic systems based on non-ideal synaptic crossbars,” *CoRR*, vol. abs/1711.08889, 2017. [Online]. Available: http://arxiv.org/abs/1711.08889

[127] T. Gokmen and Y. Vlasov, “Acceleration of deep neural network training with resistive cross-point devices: Design considerations,” *Frontiers in Neuroscience*, vol. 10, p. 333, 2016. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2016.00333

[128] T. Gokmen, O. M. Onen, and W. Haensch, “Training deep convolutional neural networks with resistive cross-point devices,” *Frontiers in Neuroscience*, vol. 11, p. 538, 2017. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2017.00538

[129] B. Feinberg, S. Wang, and E. Ipek, “Making memristive neural network accelerators reliable,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 52–65.

[130] A. Avizienis, “Arithmetic error codes: Cost and effectiveness studies for application in digital system design,” *IEEE Transactions on Computers*, vol. C-20, no. 11, pp. 1322–1331, Nov 1971.

[131] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 248–255.

[132] “Nios II Processor, Intel Corporation.”

[133] D. Strukov, “The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories,” in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, Oct 2006, pp. 1183–1187.

[134] X. Fong, S. H. Choday, P. Georgios, C. Augustine, and K. Roy, “Spice models for magnetic tunnel junctions based on monodomain approximation https://nanohub.org/resources/19048,” Aug 2016. [Online]. Available: https://nanohub.org/resources/19048

[135] S. Ikeda, K. Miura, H. Yamamoto, K. Mizunuma, H. Gan, M. Endo, S. Kanai, J. Hayakawa, F. Matsukura, and H. Ohno, "A perpendicular-anisotropy CoFeB–MgO magnetic tunnel junction," *Nature materials*, vol. 9, no. 9, pp. 721–724, 2010.

[136] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. Washington, DC, USA: IEEE Computer Society, 2007, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.30

[137] A. F. Vincent, J. Larroque, N. Locatelli, N. B. Romdhane, O. Bichler, C. Gamrat, W. S. Zhao, J. O. Klein, S. Galdin-Retailleau, and D. Querlioz, "Spin-Transfer Torque Magnetic Memory as a Stochastic Memristive Synapse for Neuromorphic Systems," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, no. 2, pp. 166–174, April 2015.

[138] L. Gao, I.-T. Wang, P.-Y. Chen, S. Vrudhula, J.-s. Seo, Y. Cao, T.-H. Hou, and S. Yu, "Fully parallel write/read in resistive synaptic array for accelerating on-chip learning," *Nanotechnology*, vol. 26, p. 455204, 10 2015.

[139] J. Li, C. I. Wu, S. C. Lewis, J. Morrish, T. Y. Wang, R. Jordan, T. Maffitt, M. Breitwisch, A. Schrott, R. Cheek, H. L. Lung, and C. Lam, "A Novel Reconfigurable Sensing Scheme for Variable Level Storage in Phase Change Memory," in *2011 3rd IEEE International Memory Workshop (IMW)*, May 2011, pp. 1–4.

[140] D. Rules, "Mosis scalable cmos (scmos)."

[141] P. Moon, V. Chikarmane, K. Fischer, R. Grover, T. A. Ibrahim, D. Ingerly, K. J. Lee, C. Litteken, T. Mule, and S. Williams, "Process and Electrical Results for the On-die Interconnect Stack for Intel's 45nm Process Generation." *Intel Technology Journal*, vol. 12, no. 2, 2008.

[142] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, "Rx-caffe: Framework for evaluating and training deep neural networks on resistive crossbars," *CoRR*, vol. abs/1809.00072, 2018. [Online]. Available: http://arxiv.org/abs/1809.00072

[143] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

[144] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Hardware-software codesign of accurate, multiplier-free deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 28:1–28:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062259

[145] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[146] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, K. Gopalakrishnan, and L. Chang, "Biscaled-dnn: Quantizing long-tailed datastructures with two scale factors for deep neural networks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 201:1–201:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317783

[147] M. Hu, C. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, Q. Xia, and J. P. Strachan, "Memristor-based analog computation and neural network classification with a dot product engine," *Advanced Materials*, 2018.

[148] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, "A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb 2013, pp. 468–469.

[149] 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.

[150] Y. He and S. Han, "ADC: automated deep compression and acceleration with reinforcement learning," *CoRR*, vol. abs/1802.03494, 2018. [Online]. Available: http://arxiv.org/abs/1802.03494

[151] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *CoRR*, vol. abs/1603.05279, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

[152] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016. [Online]. Available: http://arxiv.org/abs/1612.01064

[153] A. K. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "WRPN: wide reduced-precision networks," *CoRR*, vol. abs/1709.01134, 2017. [Online]. Available: http://arxiv.org/abs/1709.01134

[154] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "Hitnet: Hybrid ternary recurrent neural network," in *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 604–614. [Online]. Available: http://papers.nips.cc/paper/7341-hitnet-hybrid-ternary-recurrent-neural-network.pdf

[155] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary neural networks with fine-grained quantization," *CoRR*, vol. abs/1705.01462, 2017. [Online]. Available: http://arxiv.org/abs/1705.01462

[156] Q. He, H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou, and Y. Zou, "Effective quantization methods for recurrent neural networks," *CoRR*, vol. abs/1611.10176, 2016. [Online]. Available: http://arxiv.org/abs/1611.10176

[157] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *CoRR*, vol. abs/1805.06085, 2018. [Online]. Available: http://arxiv.org/abs/1805.06085

[158] "NVIDIA Tesla V100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/tesla-v100/," Online. Accessed March 15, 2019.

[159] "Google Edge TPU. https://cloud.google.com/edge-tpu/ ," Online. Accessed March 15, 2019.

[160] A. Taylor, M. Marcus, and B. Santorini, "The penn treebank: an overview," in *Treebanks.* Springer, 2003, pp. 5–22.

[161] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai, "Process technology variation," *IEEE Transactions on Electron Devices*, vol. 58, no. 8, pp. 2197–2208, Aug 2011.

VITA

VITA

Shubham Jain is currently pursuing a Ph.D. degree in the School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA. His primary research interests include exploring the circuit and architectural techniques for emerging post-CMOS devices, in-memory computing, approximate computing, and energy efficient hardware architecture for deep learning. He has a B.Tech (Hons.) degree in Electronics and Electrical Communication Engineering from the Indian Institute of Technology, Kharagpur, India, in 2012.

Previously, Shubham worked as a design engineer in the Bangalore Design Center, Qualcomm, Bangalore, India from 2012 to 2014. He also worked as a summer intern at IBM T.J Watson Research Center, Yorktown Heights, in 2017 and 2018.

Shubham has received the Mitacs Globalink scholarship from Mitacs, in 2011, the Andrews Fellowship from Purdue University, in 2014, and the A. Richard Newton Young Student Fellowship from DAC in 2015. His research has received the best technical paper award in DAC 2018, and a best-in session award in TECHCON 2016.