HARDWARE-SOFTWARE CODESIGN FOR EFFICIENT MACHINE LEARNING USING IN-MEMORY COMPUTING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Aayush Ankit

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Kaushik Roy, Chair

School of Electrical and Computer Engineering

Dr. Anand Raghunathan

School of Electrical and Computer Engineering

Dr. Vijay Raghunathan

School of Electrical and Computer Engineering

Dr. Shreyas Sen

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the School Graduate Program

Dedicated to Maa, Papa, Pratyush, family and friends without whose unconditional love and support none of this would have been possible.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Kaushik Roy, who has been an exemplary role model for me in all aspects, whether in his intelligence and integrity as a researcher, his passion and clarity as a teacher, or his humility and compassion as a person. Kaushik always puts people first and is concerned about their well-being before being concerned about their technical progress, always willing to listen and always willing to help. Undoubtedly, over the years, his insightful and critical suggestions, work ethics, and inspiration to pursue exciting problems have shaped me as a researcher. His immense energy for research and constant drive for pursuing challenging problems, deeply motivated me and showed the path towards overcoming numerous challenges throughout my PhD. His strong emphasis on crisp and precise communication and presentation skills, and high-quality writing helped me significantly grow my non-technical skills. I deeply thank him for being an outstanding mentor, critic, and advisor to me through the course of my PhD. He continues to inspire us who work with him to strive for high standards of academic excellence and character through the example that he sets.

I would also like to thank the rest of my committee, Professors Anand Raghunathan, Shreyas Sen, and Vijay Raghunathan, for going above and beyond their duty in their responsiveness and guidance throughout my PhD. My work is much stronger because of their contributions.

I would also like to thank Dr. Dejan Milojicic, Dr. John Paul Strachan, and Dr. Sai Rahul Chalamalasetti from Hewlett Packard Labs. Getting to know them has helped my in my career because of the opportunities they opened up for me and the confidence they instilled in me by believing in me.

Lastly, I would like to thank the members of Nanoelectronics Research Laboratory for their feedback and companionship throughout this journey and for being a

fun bunch of people. I especially would like to thank Abhronil Sengupta, Indranil Chakraborty, Deboleena Roy, and Mustafa Fayez Ali for their brilliance, eagerness to help others, numerous brain-storming and fun discussions, and remarkable work ethic.

Lastly, I am not who I am without my parents, my sibling, my dear friends Carol Martin, Abhinav Gupta, Aamir Raihan, and Chankyu Lee. I thank them wholeheartedly for their love and support. They are my biggest blessing.

TABLE OF CONTENTS

Page

Page

Page

Page

ABSTRACT

Ankit, Aayush Ph.D., Purdue University, December 2020. Hardware-Software Codesign for Efficient Machine Learning using In-Memory Computing. Major Professor: Kaushik Roy.

General-purpose computing systems have benefited from technology scaling for several decades but are now hitting a performance/energy wall. This trend has led to a growing interest in domain-specific accelerators. Machine Learning (ML) workloads in particular have received tremendous attention because of their pervasiveness across applications. ML workloads tend to be data-intensive and perform many matrix operations. Their execution on digital CMOS hardware is typically characterized by high data movement costs. To overcome this limitation, in-memory computing primitives (CMOS, NVM) have been demonstrated to perform matrix operations with high efficiency by overcoming the low memory bandwidth and high memory energy issues. While such primitives have shown tremendous potential at the device-circuit levels, the system-level implications remain unclear, as they are not a drop-in replacement for traditional memory structures (register file, caches etc.).

First, this thesis explores the potential of in-memory computing towards domain-specific accelerators for inference, sparse inference, and training. To improve inference efficiency PUMA is proposed, which is a spatial architecture built with Non-Volatile Memory (NVM) crossbars. PUMA leverages the high on-chip storage density and analog computation capabilities of NVM to accelerate a wide range of ML applications. It supplements NVM crossbars with CMOS digital units, an instruction execution pipeline, and a specialized ISA to enable efficient coverage across different ML workloads and enhance programmability. It also includes a hardware-software codesign to optimize data movement. The evaluations show that PUMA achieves significant

energy and latency improvements for ML inference compared to the state-of-the-art GPUs, CPUs, and ASICs. Subsequently, the proposed accelerator is extended for sparse inference. The over-parametrized nature of typical ML models has motivated model-compression techniques such as network pruning to improve the inference efficiency. However, sparse models obtained by typical network pruning lead to inefficient execution particularly on in-memory hardware. To address this, TraNNsformer is proposed which prunes the connectivity matrix while forming clusters with the remaining connections during the training process to transform the original model into an optimally pruned and maximally clustered mapping. Next, the applicability of the proposed in-memory accelerator (PUMA) is explored for ML training. Despite the storage density and analog computing benefits, NVMs suffer from high write cost, which is detrimental for ML training because of weight updates being a common case. To address this, a bit-slicing technique is proposed that enables performing high-precision analog outer-products on NVM crossbars to realize the weight update without using typical reads and writes. This technique is incorporated into an ISA-programmable architecture namely PANTHER, which enables different training algorithms and different layers types.

Despite the effectiveness of domain-specific accelerators for ML acceleration, general-purpose systems such as General Purpose Graphics Processing Unit (GPGPU) have continued to play an important role in ML inference and training owing to their wider availability and lower engineering costs for hardware-software developments. Subsequently, modern GPGPUs have introduced domain-specific units for matrix multiplication namely tensor core to meet the growing performance demands of ML. However, even with the improved throughput, tensor cores are inherently limited by the bandwidth of large register file sizes in GPGPU. To this effect, the implication of in-memory computing based tensor cores on GPGPU is explored. We show that GPGPU augmented with in-memory tensor cores enables overcoming the bandwidth limitation of register file, and provides the required flexibility for emerging application trends such as quantization and sparsity.

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

1. INTRODUCTION

1.1 Machine Learning and its Computational Challenges

Machine Learning (ML) is the scientific study of computation models that learn to perform specific tasks without requiring explicit instructions, as defined by Arthur Samuel in 1959. This is inspired from the magnificent ability of the human brain to perform myriad tasks by just learning from the environment. On the contrary, traditional computer programs built for specific purpose exhibit a behavior defined by the hand-crafted heuristics that statically define their behavior. Consequently, the overarching goal of ML has been to achieve human brain-level intelligence.

The past decade has seen a tremendous surge in the usage of the ML algorithms across different application domains such as image and video processing [1, 2], speech and language processing [3, 4], medical imaging [5], gameplay [6] and robotics [7]. More recently ML algorithms have also superseded human brain capabilities in specific applications, for example Google's AlphaGo [1] and Baidu's DeepSpeech2 [8]. The increasing abilities in terms of accuracy has led to the successful proliferation of ML based personal assistants in commercial products used in day-to-day life such as Siri, Alexa and Google Now.

While these algorithmic feats are remarkable, the ever increasing hardware costs driving these developments are alarming. For instance, the AlphaGo algorithm runs on hundreds of KiloWatts of power which is about four orders of magnitude higher than the human brain. Consequently, the algorithm performance is moving closer to the human brain whereas the hardware cost is moving farther. Furthermore, aspirations have always grown faster than the technology available to satisfy them. To this effect, there is a need to look beyond today's general purpose systems in order to bridge the huge disparity in the hardware costs of ML algorithms and the brain.

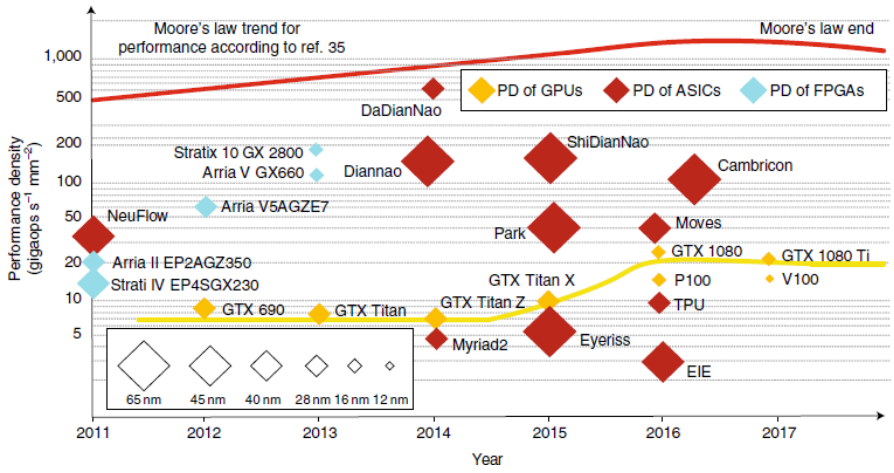


Fig. 1.1. Increase in the performance density of CMOS hardware [13]

1.2 Analog In-memory Computing

Owing to the high computational needs of ML particularly Matrix Vector Multiplication (MVM) operations, there has been active research in ML accelerator architectures built with CMOS digital technology. Several projects such as DianNao [9], Tensor Processing Unit [10] and Brainwave [11] have proposed ML hardware that achieve orders of magnitude higher efficiency compared to general purpose systems (CPU, GPU). However, the trends in increase in the performance density of digital CMOS hardware (Figure 1.1) and the computation requirements of ML (Figure 1.2), clearly highlight that ML models are growing at a faster rate than conventional CMOS hardware (or digital CMOS). Furthermore, ML workloads tend to be data-intensive and their execution on digital CMOS hardware is typically characterized by high data movement costs relative to compute [12]. To this end, analog in-memory computing using memristive and SRAM technologies have shown tremendous potential owing to their efficient dot product computation capabilities.

Memristive crossbars can store a matrix with high storage density and perform MVM operations with very low energy and latency [14–20]. Each crosspoint in the crossbar stores a multi-bit value in one memristor device, which enables high storage

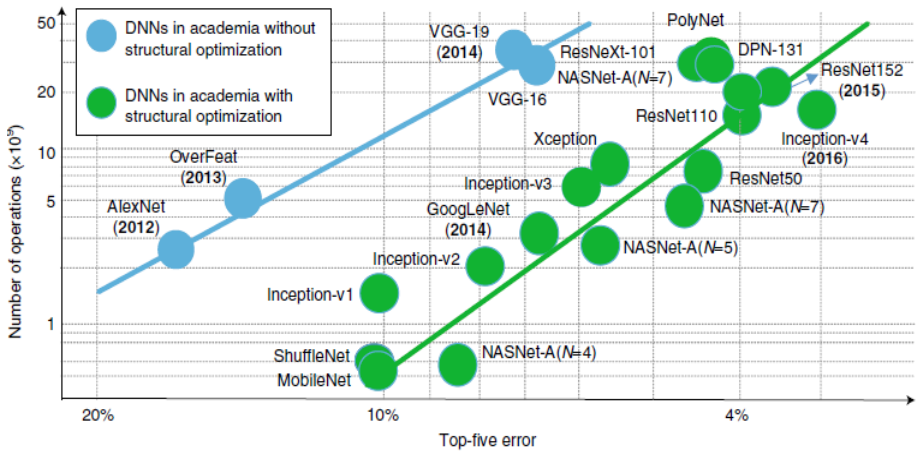


Fig. 1.2. Increase in the computational requirements of ML models [13]

density [21]. Upon applying an input voltage at the crossbar's rows, we get the MVM result as output current at the crossbar's columns based on Kirchhoff's law. A crossbar thus performs MVM in one computational step – including $O(n^2)$ multiplications and additions for an $n \times n$ matrix – which typically takes many steps in digital logic. It also combines compute and storage in a single device to alleviate data movement, thereby providing intrinsic suitability for data-intensive workloads [22, 23]. Several recent works have also demonstrated such in-memory MVM capability in SRAM technology [24, 25]. While SRAM technology has lower storage density compared to memristive technology, its high endurance, reliability and efficient writes makes it an attractive choice for augmenting existing systems such as CPU, GPU with in-memory computing capabilities.

1.3 Dissertation Overview

Thesis Statement

This dissertation proposes an ISA-programmable accelerator architecture built with memristive crossbars for ML inference called ***PUMA*** and shows that hybrid

CMOS-memristive technologies can achieve orders of magnitude higher efficiency than CMOS systems (CPUs, GPUs and ASICs) [26]. In addition to that, it also proposes a software optimization called ***TraNNsformer*** to enable efficient inference of emerging ML workloads namely sparse DNNs [27]. Subsequently, it proposes ***PANTHER*** which extends the PUMA architecture for performing ML training as well. Lastly, it proposes an in-memory tensor core (***TIMON***) for General Purpose Graphics Processing Unit (GPGPU) to improve the efficiency of conventional architectures for ML workloads.

Layout

The rest of the dissertation is organized as follows. Chapter 2 discusses the prior works in ML accelerators, software optimization (weight pruning) for ML, and GPGPU tensor cores. Chapters 3 and 4 cover the proposed PUMA accelerator architecture [26, 28] and TraNNsformer software optimization [27, 29]. Chaper 5 covers the proposed extensions for training - PANTHER [30]. Chaper 6 covers in-memory computing in GPGPU register file - TIMON.

2. RELATED WORK AND BACKGROUND

In this section we review the research on ML accelerator architectures, software optimizations for ML, and GPGPU tensor cores and register file. We also look into how the proposed *PUMA: inference accelerator architecture*, *TraNNsformer: software optimization*, *PANTHER: training accelerator architecture*, and *TIMON: in-memory tensor core* distinguish from the related works.

2.1 Inference Accelerator Architectures

Owing to the ever increasing computation demands of machine learning and its growing pervasiveness across application domains, past years have seen significant body of research in accelerators for machine learning. In this section, we review some of these implementations for CMOS Digital, and Memristor-based Analog accelerators. Finally, we discuss the manufacturability and non-ideality concerns associated with memristor technology.

2.1.1 CMOS Digital In-/Near-memory Accelerators

Sze et al. [31] provide a thorough survey of the many neural network accelerators and the differences between them. In the digital realm, accelerators can be classified as weight stationary spatial architectures (such as NeuFlow [32], Sankaradas et al. [33], Chakradhar et al. [34], Gokhale et al. [35], Origami [36], and Paer et al. [37]), output stationary spatial architectures (such as Peemen et al. [38], ShiDianNao [39], and Gupta et al. [40]), spatial architectures with no local reuse (such as DianNao [9], DaDianNao [41], and Zhang et al. [42]), and row stationary spatial architectures (such as Eyeriss [43]). Many accelerator designs have also proposed

support for optimizations and features including weight pruning and exploiting sparsity such as in Minerva [44], Cnvlutin [45], EIE [46], SCNN [47], Cambricon-X [48], Chung et al. [49], Pragmatic [50], and CirCNN [51], reducing precision such as in Stripes [52] and YodaNN [53], layer fusing such as in Alwani et al. [54], leveraging stochastic computing such as in Kim et al. [55] and SC-DNN [56], and adaptation to meet QoS and QoR requirements such as ELNA [57]. Accelerators vary in their degree of flexibility, ranging from custom accelerators specialized for a particular field such as Murray et al. [58] and Yazdani et al. [59] to accelerators that are fully programmable via an instruction set architecture such as PuDianNao [60], Cambricon [61], Cambricon-X [48], ScaleDeep [62], and Google's TPU [10]. Besides ASICs, accelerators have also been generated for FPGAs such as in CNP [63], Peemen et al. [38], Zhang et al. [42], DeepBurning [64], Tabla [65], DNNWeaver [66], and Shen et al. [67]. All these works remain in the digital domain, while PUMA focuses on hybrid digital-analog computing.

Chung et al. [11] propose Brainwave, which is a spatial accelerator built with FPGAs. Compared to Brainwave, a PUMA core performs 0.26 million 16-bit ops, equivalent to 1.04 million 8-bit ops, per coalesced MVM instruction. A Brainwave NPU performs 1.3million 8-bit ops per instruction. Therefore, PUMA and Brainwave have comparable control granularity while PUMA has 40.8x higher storage-density (Brainwave Stratix10 estimate).

Near-memory computing using DRAM has been proposed in past accelerators such as Neurocube [68], TETRIS [69], DRISA [70], and RAMANN [71]. PUMA uses non-volatile memristive crossbars for acceleration.

2.1.2 Memristor-based Analog In-/Near-memory Accelerators

The use of memristive crossbars for accelerating neural networks has received much attention in recent years [72–75]. Accelerators leveraging memristive crossbars include Hu et al. [76], SPINDLE [77], DNP [78], RENO [79], Memristive Boltz-

mann Machine [80], ISAAC [23], Prime [22], PipeLayer [81], TIME [82], Group Scissor [83], and RESPARC [84]. These accelerators have been demonstrated on several types of networks including BSBs [76], MLPs [77, 79, 82], SNNs [78, 84–87], Boltzman Machines [80], and CNNs [22, 23, 81–83]. Some accelerators support inference only [22, 23, 76, 77, 79, 84] while others also support training [78, 80–82]. NEUTRAMS [88] provides a framework for transforming neural networks to configure such accelerators. These accelerators vary in their degrees of flexibility, but even the most flexible rely on datapath configuration and have only been demonstrated on a few types of networks. PUMA is the first memristor-based accelerator for machine learning inference that is ISA-programmable and general-purpose.

Fujiki et al. [89] propose an ISA-programmable memristor-based accelerator. Their accelerator is a data-parallel accelerator whereas PUMA is a data-flow accelerator with more capability for producer-consumer synchronization. Moreover, their accelerator optimizes crossbars for vector operations in general-purpose workloads whereas PUMA optimizes crossbars for MVM operations prevalent in machine learning and uses digital VFUs for vector operations rather than crossbars.

2.1.3 Memristor Manufacturability and Non-ideality

There have been concerns in the community about memristor manufacturability. We distinguish between medium-density embedded memristor applications and high-density storage-class memory (SCM). Memristors in PUMA use 1T1R configuration which have been shown to have good manufacturability [90]. They are very different from SCM, where the selector transistor may be replaced with an in-line two-terminal selector device for higher density which complicates manufacturability. Panasonic formed a joined venture with UMC foundry in 2017 to enable integration of memristors to UMC 40nm CMOS process with first samples planned in 2018 [91]. TSMC also completed development of their own memristor technology that entered risk production in 40nm ULP CMOS process node at the end of 2017 [92].

Memristor technology suffers from non-idealities such as read/write noise, asymmetric non-linearity in conductance vs programming pulses, IR-drop, source and sense resistances; stuck-at-faults; and process variations. These can lead to significant degradation in the classification accuracy [93]. Past research has shown that the iterative nature of DNN training and careful re-training enables recovering the accuracy loss from non-idealities [94–97], faults [98] and variations [99]. Furthermore, PUMA uses low bit-resolution memristor cells to gain higher noise margin for mitigating the accuracy degradation from non-idealities.

2.2 Software Optimizations (Weight Pruning)

Weight pruning has been proposed to achieve significant reduction in DNN size, while maintaining the required accuracy [12, 100–103]. A reduction in the number of weights at the algorithm level opens up the potential to leverage this weight sparsity for efficient DNN execution at the hardware level. Consequently, this leads to two orthogonal research directions geared towards CMOS architectures 1) specialized accelerators for sparse DNNs, and 2) software transformations. Several specialized accelerators have been proposed to maximize the DNN efficiency for resource and energy constrained platforms [46, 48, 104–107], which have explored novel techniques to store the sparse matrix for reducing the memory requirements as well as exploiting the weight sparsity to save unnecessary computations. The software transformation approaches have explored techniques to obtain sparsity in a structured or regular manner, such that these sparse DNNs can be executed on CPU, GPU as well as general-purpose ML accelerators [105, 108–113] efficiently.

While the past works on designing specialized accelerators have shown significant improvements for DNN execution on CMOS hardware, these techniques are not amenable to crossbar-based architectures. The separability between storage and computation available on CMOS systems enables designing specialized CMOS accelerators to leverage sparsity. Herein, the sparse matrix can be stored in a compressed format

(in off-chip memory), which can be later decompressed during computations to enable correct computations (in SIMD units). However, a crossbar performs in-memory processing, where the computations are performed by the same crossbar which stores the weight matrix. This does not involve explicit memory fetches for the weight matrix, thereby removing the flexibility to store data in a compressed manner. Further, owing to the difference in the computation nature of CMOS and crosbar-based hardware, the software transformations for CMOS systems are not suitable to MCA systems.

Recently, software transformation approaches for crosbsar-based architectures have been proposed [114–116]. [114] focuses on clustering the synapses after the training process finishes i.e. offline clustering. [115] proposes pruning of convolution layers to reduce the throughput overhead by implementing pruning based on semi-folded mapping scheme. [116] uses the approach of hierarchical clustering to map DNNs on 3D ICs. TraNNsformer distinguishes from the prior works as it proposes an integrated training framework for optimizing fully connected layers of DNNs.

2.3 Training Accelerator Architectures

Various ReRAM-based training accelerators [81,82] have been proposed, but they rely on expensive serial reads and writes to accomplish weight updates. We avoid these reads and writes by leveraging the in-crossbar OPA operations [117, 118], and extending their precision for practical trainability. Our crossbar architecture can be used to enhance existing accelerators.

ReRAM-based accelerators have also been proposed for DNN inference [22, 23, 26, 80], graph processing [119], scientific computing [120], and general purpose data parallel applications [89]. Our work focuses on DNN training.

Analog [121, 122] and DRAM-based [68–70] accelerators have been proposed as alternatives to digital-CMOS accelerators. Our work uses ReRAM as an alternative.

Many accelerators use digital CMOS technology for accelerating DNNs, including those that mainly target inference [43,45,46,48,52,54,57,59,104,123–131] or also target

training [41, 61, 132–135]. Frameworks that assist with designing such accelerators deal with challenges such as power optimizations [44], stochastic computing [55, 56], pruning [49], targeting FPGAs [64–66, 136, 137], and scaling [138].

Many accelerators use digital CMOS technology for accelerating DNNs, including those that mainly target inference [31] or also target training [135]. Our work uses hybrid digital-analog computation based on ReRAM crossbars, not just CMOS.

Recent works have explored training DNNs with reduced precisions in floating-point arithmetic domain such as bfloat16 [139], float8 [140] as well as fixed-point arithmetic domain [141, 142]. While floating-point arithmetic is not amenable to ReRam-based hardware (without modifications), the reductions in fixed-point precision can be exploited in PANTHER by reducing the MCU width (number of slices) to improve training energy and time.

ReRAM technology suffers from imprecise writes due to non-idealities (noise and non-linearity) and manufacturability issues (stuck-at-faults and process variations). However, the iterative nature of DNN training and careful re-training helps recover the accuracy loss from non-idealities [94], faults [98], and variations [143]. Re-training is a fine-tuning process (typically 1 epoch) with insignificant cost compared to training.

2.4 GPGPU Tensor Core

Inference Accelerators: Domain-specific architectures for inference have been explored both in industry [10, 144] and academia [31, 43, 129]. Many works also support model compression techniques such as quantization [46, 145, 146], and sparsity [104, 147–152]. Alike Nvidia GPU tensor cores[1], majority of these accelerators leverage data reuse to overcome the memory bandwidth/energy limitations. However, the key difference lies in the amount of data reuse obtained at the area (thereby power) cost. GPU tensor cores operate at computational intensity of about 2 $Ops/Byte$ (Section 6.3) and consume $0.21mm^2$ die area per sub-core at 45nm node (Section 6.7.5).

[1]leverages data reuse by using local buffers in tensor cores to reduce register file access (Section 6.3).

On the other hand, Eyeriss, a custom architecture, achieves an order of magnitude higher computation intensity, and consumes 5.87 mm^2 die area at 45nm node [153]. The 28× higher area requirement is because the higher data reuse comes at the cost of significantly higher buffer size and computation units. *As a result, the smaller area versions of such accelerators incorporated as tensor core in GPGPUs have low data reuse, and are limited by low bandwidth and high energy of GPGPU register file.*

Register File Optimizations: Several past works have explored optimizations to reduce dynamic energy and leakage energy in GPGPU register files by caching [154], virtualization [155], emerging memory technologies [156, 157], and fine-grained access control [158]. These techniques reduce energy consumption, but do not improve inference performance.

Past research have also explored compiler optimizations such as register packing [159], and register coalescing [160] to reduce the register file bandwidth requirements for workloads with lower bit-width operands such as 1/2/3 bytes. These techniques can improve inference on quantized workloads, but the improvements obtained are linear at best due to linear reduction in register file accesses with bit-width reduction. Since quantization reduces the number of computations quadratically (bit-serial multiplier [146]), it would be ideal to obtain quadratic benefits in inference performance as well.

GPGPU Tensor Cores: Nvidia GPUs recently introduced tensor cores for inference acceleration in 2018 (Turing [161]). As mentioned before, they suffer from register file bandwidth/energy limitations. Further, existing tensor cores do not offer the required flexibility for model compression techniques. Recently, [162] explored accelerating workloads with *structured* sparity on tensor cores. However, it does not improve the performance on dense GEMMs, or support the more commonly used model compression techniques such as quantization, and *unstructured* sparsity.

TIMON leverages in-memory computing to overcome GPGPU register file bandwidth and energy limitations to achieve higher efficiency compared to existing GPGPU

tensor core on dense GEMMs. Further, TIMON enables flexible support for *irregular* quantization, and *unstructured* sparsity.

3. PUMA - INFERENCE ACCELERATOR ARCHITECTURE

3.1 Introduction

ML workloads tend to be data-intensive and perform a large number of Matrix Vector Multiplication (MVM) operations. Their execution on digital CMOS hardware is typically characterized by high data movement costs relative to compute [12]. To overcome this limitation, memristive crossbars can store a matrix with high storage density and perform MVM operations with very low energy and latency [14–19]. Each crosspoint in the crossbar stores a multi-bit value in one memristor device, which enables high storage density [21]. Upon applying an input voltage at the crossbar's rows, we get the MVM result as output current at the crossbar's columns based on Kirchhoff's law. A crossbar thus performs MVM in one computational step – including $O(n^2)$ multiplications and additions for an $n \times n$ matrix – which typically takes many steps in digital logic. It also combines compute and storage in a single device to alleviate data movement, thereby providing intrinsic suitability for data-intensive workloads [22, 23].

Memristive crossbars have been used to build special-purpose accelerators for Convolutional Neural Networks (CNN) and Multi Layer Perceptrons (MLP) [22, 23, 79], but these designs lack several important features for supporting general ML workloads. First, each design supports one or two types of neural networks, where layers are encoded as state machines. This approach is not scalable to a larger variety of workloads due to increased decoding overhead and complexity. Second, existing accelerators lack the types of operations needed by general ML workloads. For example, Long Short-Term Memory (LSTM) [163] workloads require multiple vector linear and transcendental functions which cannot be executed on crossbars efficiently

and are not supported by existing designs. Third, existing designs do not provide flexible data movement and control operations to capture the variety of access and reuse patterns in different workloads. Since crossbars have high write latency [164], they typically store constant data while variable inputs are routed between them in a spatial architecture. This data movement can amount to a significant portion of the total energy consumption which calls for flexible operations to optimize the data movement. Fourth, some designs use deep pipelines [23] which are not suitable for general workloads with control flow. Finally, other designs [22] make optimistic assumptions about precision which may not be suitable for all types of workloads.

To address these limitations, we present PUMA, a Programmable Ultra-efficient Memristor-based Accelerator. PUMA is a spatial architecture designed to preserve the storage density of memristive crossbars to enable mapping ML applications using on-chip memory only. It supplements crossbars with an instruction execution pipeline and a specialized ISA that enables compact representation of ML workloads with low decoder complexity. It employs temporal SIMD units and a ROM-Embedded RAM [165] for area-efficient linear and transcendental vector computations. It includes a microarchitecture and ISA co-designed to optimize data movement and maximize area and energy efficiency. To the best of our knowledge, PUMA is the first programmable and general-purpose ML inference accelerator built with hybrid CMOS-memristor technology.

A naïve approach to generality is not viable because of the huge disparity in compute and storage density between the two technologies. CMOS digital logic has an order of magnitude higher area requirement than a crossbar for equal output width ($\sim$20$\times$, see Table 3.5). Moreover, a crossbar's storage density (2-bit cells) is $160MB/mm^2$, which is at least an order of magnitude higher than SRAM (6T, 1-bit cell) [23]. A 90mm^2 PUMA node can store ML models with up to 69MB of weight data. Note that the PUMA microarchitecture and ISA are equally suitable to crossbars made from emerging technologies other than memristors such as STT-MRAM [20], NOR Flash [166], etc.

We make the following contributions:

- A programmable and highly efficient architecture exposed by a specialized ISA for scalable acceleration of a wide variety of ML applications using memristive crossbars.
- A detailed simulator which incorporates functionality, timing, and power models of the architecture.
- An evaluation across ML workloads showing that PUMA can achieve promising performance and energy efficiency compared to state-of-the-art CPUs, GPUs, TPU, and application-specific memristor-based accelerators.

3.2 Workload Characterization

This section characterizes different ML inference workloads with a batch size of one. The characteristics are summarized in Table 3.1. The section's objective is to provide insights on the suitability of memristive crossbars for accelerating ML workloads and highlight implications on the proposed architecture.

3.2.1 Multi-Layer Perceptron (MLP)

MLPs are neural networks used in common classification tasks such as digit-recognition, web-search, etc. [167, 168]. Each layer is fully-connected and applies a nonlinear function to the weighted-sum of outputs from the previous layer. The weighted-sum is essentially an MVM operation. Equation 3.1 shows the computations in a typical MLP (act is nonlinear):

$$O[y] = act(B[y] + \sum_{x=0}^{n-1} I[x] \times W[x][y]) \tag{3.1}$$

MLPs are simple, capturing the features common across the ML workloads we discuss: dominance of MVM operations, high data parallelism, and use of nonlinear operations.

Table 3.1.
Workload Characterization

Characteristic	MLP	LSTM	CNN
Dominance of MVM	Yes	Yes	Yes
High data parallelism	Yes	Yes	Yes
Nonlinear operations	Yes	Yes	Yes
Linear operations	No	Yes	No
Trancendental operations	No	Yes	Yes
Weight data reuse	No	Yes	Yes
Input data reuse	No	No	Yes
Bounded resource	Memory	Memory	Compute
Sequential access pattern	Yes	Yes	No

Dominance of MVM

MVM operations are $O(n^2)$ in space and computational complexity, whereas the nonlinear operations are $O(n)$, where n is the matrix dimension (layer size). MVMs are therefore the dominant operation in MLPs (and other ML workloads). This property makes memristive crossbars suitable for acceleration since they perform analog MVMs with low energy/latency.

High data parallelism

MLPs (and other ML workloads) have massive amounts of data parallelism. Moreover, practical model sizes are larger than the typical on-chip storage that can be provided by SRAM. For this reason, CMOS implementations suffer from costly DRAM accesses which are particularly taxing due to the absence of data reuse to amortize them. On the other hand, crossbars have extremely high area efficiency which allows deploying many of them on a single chip. Doing so not only captures the high data parallelism in these workloads, but it also provides high storage density to fit models on-chip and bypass costly DRAM accesses.

Nonlinear operations

In addition to MVM operations, MLPs (and other ML workloads) perform nonlinear vector operations (e.g., ReLU). Since these operations cannot be performed in crossbars, an implication on the architecture is the need to provide digital functional units to support them. Such functional units consume significantly more area ($\sim$20$\times$) than crossbars for equal output width (see Table 3.5). The challenge is to size these units appropriately to provide sufficient throughput without offsetting crossbar area/energy efficiency.

3.2.2 Long Short-Term Memory (LSTM)

LSTMs are the state-of-the-art technique for sequence processing tasks like speech processing, language modelling, etc. [163]. Each layer is fully connected and performs linear and nonlinear operations on the weighted-sum of outputs and the previous state. These operations translate into two MVMs followed by multiple (typically four) vector arithmetic operations and (typically four) nonlinear functions. Equations 3.2 to 3.4 show the computations in a typical LSTM:

$$F_{\mathrm{t}}[y] = act(B[f] + \sum_{x=0}^{n-1}(H, I)[x] \times W_{\mathrm{f}}[x][y]) \tag{3.2}$$

$$C_{\mathrm{t}}[y] = \sum_{x=0}^{n-1}(f_{\mathrm{t}}[y] \times C_{\text{t-1}}[y] + g_{\mathrm{t}}[y] \times Cp_{\text{t-1}}[y]) \tag{3.3}$$

$$H_{\mathrm{t}}[y] = \sum_{x=0}^{n-1}(h_{\mathrm{t}}[y] \times C_{\mathrm{t}}[y]) \tag{3.4}$$

To the best of our knowledge, PUMA is the first memristor-based accelerator demonstrated with LSTMs.

Linear and transcendental operations

Unlike MLPs, LSTMs also perform linear vector operations. Moreover, the typical nonlinear vector operations in LSTMs are transcendental (e.g. tanh, sigmoid). Sup-

porting these operations has the same implication on the architecture as discussed in Section 3.2.1 for nonlinear operations. Transcendental operations are particularly challenging due to their high complexity.

Weight reuse

Another key distinction of LSTMs compared to MLPs is data reuse. LSTM inputs consist of a sequence of vectors processed across multiple time-steps with the same weights. This feature benefits CMOS architectures by amortizing DRAM accesses for loading weights, but is not advantageous to memristive crossbars. That said, the scope of weight reuse in LSTMs is only over a few inputs so the workload remains memory-bound. It still suffers in CMOS hardware from insufficient amortization of DRAM accesses.

3.2.3 Convolutional Neural Network (CNN)

CNNs are widely used for image recognition and classification [169]. They typically include several convolutional, pooling, and response normalization layers. A convolution layer consists of weight kernels strided across the input image in a sliding window fashion. It exhibits a non-sequential memory access pattern since a window of the input consists of parts of the input image from different rows. Equation 3.5 shows the computations in a typical convolutional layer of a CNN:

$$O[m][x][y] = act(B[m]+ \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} I[k][Ux+i][Uy+j]\times W[m][k][i][j]) \tag{3.5}$$

Input reuse and compute intensity

Convolution layers exhibit both weight and input data reuse. They can be mapped to matrix-matrix multiplications which successively apply weight kernels on different

input windows. Matrix-matrix multiplications are compute-bound which makes them well-suited for CMOS hardware since there is enough data reuse to amortize DRAM access cost. However, memristive crossbars can still perform well on matrix-matrix operations by treating them as successive MVMs. An implication on architecture is the opportunity to take advantage of input reuse to minimize data movement within the chip. Another implication is that iterating over inputs creates the need for control flow to represent the workload compactly without code bloat.

Non-sequential access

Unlike MLPs and LSTMs, CNNs exhibit non-sequential accesses due to the way inputs are traversed as well as the behavior of non-convolutional layers such as pooling and response-normalization. An implication on the architecture is the need to support fine-grain/random access to memory, which is not needed for MLPs and LSTMs where it is sufficient to access data at the granularity of the input/output vectors to each layer.

3.2.4 Other ML Workloads

Other workloads, both supervised and unsupervised, can be represented using a combination of the patterns in the three applications in this section. *Logistic Regression* [170] and *Linear Regression* [171] compute weighted-sums which are passed to activation functions to generate probabilities and continuous values respectively. *Support Vector Machine (SVM)* [172] and *Recommender Systems* [173] compute weighted-sums followed by nonlinear functions. Their computations are similar to MLP. *Recurrent Neural Networks (RNNs)* [174] used for sequence processing compute weighted-sums on input and previous state. They are similar to LSTMs but without vector operations. *Generative Adversarial Networks (GANs)* are composed of two neural networks (MLP, LSTM, CNN, etc.) which compete to reach equilibrium [175]. *Restricted Boltzmann Machines (RBM)* [176] and *Boltzmann Machines (BM)* [177] are

commonly used in unsupervised learning tasks for energy-minimization. While RBM involves weighted-sums of previous state and inputs, BM uses inputs only. Their computations have similarities to MLPs and LSTMs as well.

3.3 Core Architecture

We propose a programmable architecture and ISA design that leverage memristive crossbars for accelerating ML workloads. PUMA is a spatial architecture organized in three-tiers: cores, tiles, and nodes. Cores consist of analog crossbars, functional units, and an instruction execution pipeline. Tiles consist of multiple cores connected via a shared memory. Nodes consist of multiple tiles connected via an on-chip network. Subsequently, nodes can be connected together via a chip-to-chip interconnect for large-scale execution.

While this hierarchical organization is common in related work [23, 41], our key contributions lie in the core architecture (this section) and tile architecture (Section 3.4) that bring programmability and generality to memristive crossbars without compromising their energy and area efficiency. An overview of the core architecture is shown in Figure 3.1. The following subsections discuss the components of the core architecture and the insights behind their design.

3.3.1 Instruction Execution Pipeline

Existing memristor-based accelerators [22, 23, 79] are limited to one or two ML workloads. They use state machines that can be configured to compose a small set of functional blocks (e.g., convolution block, pooling block, etc.). While this approach works well when the scope of workloads is small, supporting a larger variety of workloads creates high decoding complexity. For this reason, our core architecture breaks functionality down to finer-grain instructions and supplements memristive crossbars with an instruction execution pipeline. Our approach is based on the observation

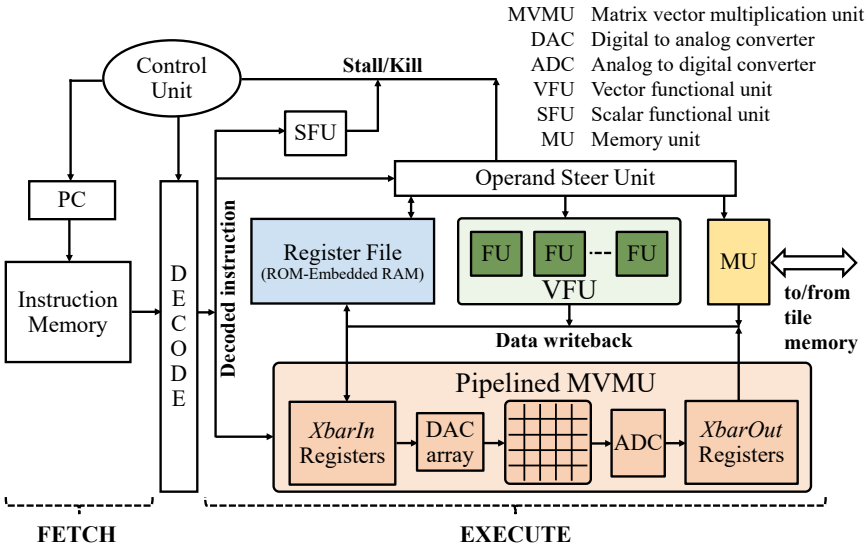


Fig. 3.1. Core Architecture

Table 3.2.
Instruction Set Architecture Overview

Category	Instruction	Description	Operands
Compute	MVM	Matrix-Vector Multiplication	mvm, mask, -, filter, stride, -, -
	ALU	Vector arithmetic/logical (add, subtract, multiply, divide, shift, and, or, invert) Vector non-linear (relu, sigmoid, tanh, log, exponential) Other (random vector, subsampling, min/max)	alu, aluop, dest, src1, src2, src3, vec-width
	ALUimm	Vector arithmetic immediate (add, subtract, multiply, divide)	alui, aluop, dest, src1, immediate, vec-width
	ALUint	Scalar arithmetic (add, subtract) - Compare (equal, greater than, not equal)	alu-int, aluop, dest, src1, src2, -, -
Intra-Core Data Movement	set	Register initialization	set, -, dest, immediate, -, -
	copy	Data movement between different registers	copy, -, dest, src1, -, ld-width, vec-width
Intra-Tile Data Movement	load	Load data from shared memory	load, -, dest, immediate, -, -
	store	Store data to shared memory	store, -, dest, src1, count, st-width, vec-width
Intra-Node Data Movement	send	Send data to tile	send, memaddr, fifo-id, target, send-width, vec-width
	receive	Receive data from tile	receive, memaddr, fifo-id, count, rec-width, vec-width
Control	jmp	Unconditional jump	jmp, -, -, -, -, pc
	brn	Conditional jump	brn, brnop, -, src1, src2, pc

in Section 3.2 that despite the large variety of ML workloads, these workloads share many low-level operations.

The instruction execution pipeline is an in-order pipeline with three stages: fetch, decode, and execute. Keeping the pipeline simple saves area to avoid offsetting the crossbars' area efficiency. The ISA executed by the pipeline is summarized in Ta-

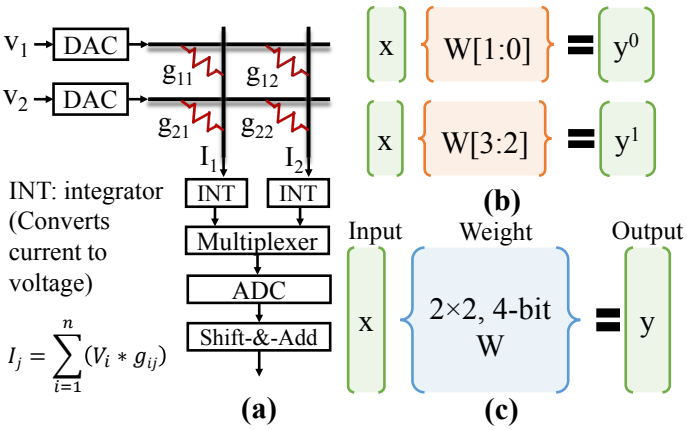


Fig. 3.2. MVM with Crossbars

ble 3.2. Instructions are 56-bits wide. The motivations for wide instructions are discussed in Sections 3.3.3 and 3.3.4. The ISA instruction usage is shown in Section 3.3.6. More ISA details are discussed in Section 5.5.2.

The instruction execution pipeline supports control flow instructions (*jmp* and *brn* in Table 3.2), as motivated in Section 3.2.3. It also includes a *Scalar Functional Unit (SFU)* that performs scalar integer operations (*ALUint* in Table 3.2) to support the control flow instructions.

3.3.2 Matrix-Vector Multiplication Unit (MVMU)

The MVMU (illustrated in Figure 3.1) consists of memristive crossbars that perform analog MVM operations, and peripherals (DAC/ADC arrays) that interface with digital logic via the *XbarIn* and *XbarOut* registers. *XbarIn* registers provide digital inputs to the DACs which feed analog voltages to the crossbar. ADCs convert crossbar output currents to digital values which are stored in the *XbarOut* registers. This crossbar design is similar to ISAAC [23].

Figure 3.2(a) shows how memristive crossbars can be used to perform analog MVM operations. DACs convert the input vector to analog voltages applied at crossbar

rows. The matrix is encoded as conductance states (g_{ij}) of the devices that constitute the crossbar. The currents at crossbar columns constitute the MVM result. They are integrated (converted to voltage) then converted to digital values with ADCs.

Precision Considerations

Practically realizable crossbars provide 2-6 bits of precision per device [16]. We conservatively use 2 bits per device, and realize 16-bit MVM operations by combining 8 crossbars via bit-slicing [23], illustrated in Figure 3.2(b). ADCs are reused across columns to save area. The impact of precision on inference accuracy is evaluated in Section 3.8.6.

Crossbar Co-location and Input Sharing

Crossbar peripherals have an order of magnitude higher area than the crossbar. Since all eight 2-bit crossbars of a 16-bit MVM operation are used simultaneously on the same input, we co-locate these 2-bit crossbars on the same core in the same MVMU, which allows us to use the same *XbarIn* registers and DAC array to feed them with minimal routing congestion. This co-location and input reuse is provided transparently in the ISA, which exposes a full 16-bit MVM operation in a single instruction (*MVM* in Table 3.2).

Input Shuffling

As motivated in Section 3.2.3, ML workloads with sliding window computations typically reuse large portions of the input across successive MVM operations (∼80% for convolutional layers with filter size 5 and unit stride). However, reused input values may come at different positions in the input vector. To avoid moving data around in *XbarIn*, the MVM instruction provides operands (*filter*/*stride* in Table 3.2) that

re-route XbarIn registers to different DACs, enabling logical input shuffling without physical data movement.

Multiple MVMUs per Core

A core may have multiple MVMUs, in which case it is desirable to activate them in parallel since MVMs are heavy operations. The in-order pipeline does not capture the Instruction-Level Parallelism (ILP) between MVM instructions automatically. Instead, the ISA exposes an operand (*mask* in Table 3.2) to allow a single MVM instruction to activate multiple MVMUs at once. Compiler optimizations that use this operand are discussed in Section **??**.

Crossbar Writes

PUMA is an inference accelerator, so crossbars are initialized with weights using serial writes at configuration time prior to execution and are not written to throughout execution. In this sense, PUMA is a spatial architecture where data is routed between crossbars, each crossbar storing a different portion of the model. Larger models therefore require more area, and may scale to multiple nodes.

3.3.3 Vector Functional Unit (VFU)

The VFU executes linear and nonlinear vector operations (ALU and $ALUimm$ in Table 3.2), as motivated by Sections 3.2.1 and 3.2.2. An important aspect of designing vector instructions and the VFU is choosing the vector width. Since ML workloads have high data parallelism, they execute wide vector operations, which motivates having wide vector instructions. Wide vector instructions have the benefit of reducing instruction count, and consequently, fetch, decode, and instruction storage overhead. On the other hand, hardware considerations motivate having narrow VFU vector width to avoid offsetting the area efficiency of crossbars as discussed in Section 3.2.1.

To balance the tension between workloads favoring wide vector width and hardware favoring narrow vector width, we propose a VFU design based on *temporal SIMD*. Temporal SIMD uses a narrow width VFU to execute wide vectors over multiple cycles. The vector instruction operands specify the starting address of the vectors in the register file as well as the vector width (*vec-width* in Table 3.2). The *operand steer unit* holds the decoded instruction and reads the operands from the register file over subsequent cycles to feed the VFU. The additional *vec-width* operand required by temoral SIMD motivates our wide instruction design.

Provisioning the adequate width for VFUs maintains crossbar area efficiency benefits without the VFU becoming a bottleneck and compromising throughput. A narrow VFU is possible because typical ML workloads compute $O(n)$ more operations per MVM instruction than per vector instruction. Section 3.8.6 evaluates the impact of VFU width on efficiency.

3.3.4 Register File

We propose a register file design that uses *ROM-Embedded RAM* [165] to accomplish two objectives: (1) harboring general purpose registers, and (2) providing area-efficient transcendental function evaluations as motivated in Section 3.2.2.

Implementing transcendental functions

Area-efficient function evaluations are crucial for preserving crossbar storage density. For this reason, we use a ROM-Embedded RAM structure [165] which adds a wordline per row to embed a ROM that is used to store look-up tables without increasing the array area or RAM latency. Alternative digital implementations to support transcendental functions are prohibitive due to their high area requirements, especially in light of the large number of transcendental function types used in ML. Transcendental function evaluations also use temporal SIMD (Section 3.3.3) to minimize fetch/decode energy consumption.

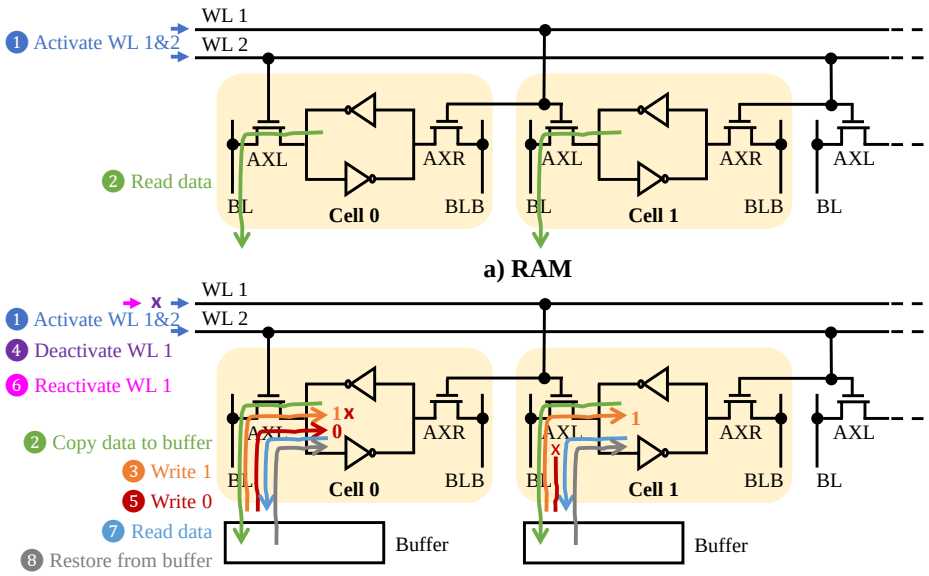


Fig. 3.3. ROM-Embedded RAM

Figure 3.3 details the operation of a ROM-Embedded RAM. In RAM mode, both wordlines ($WL1$ and $WL2$) are activated, followed by precharging or driving the bitlines for read or write operations, respectively (similar to typical SRAM). In ROM mode, since a ROM access overwrites the RAM data, the first step is to buffer the RAM data. Subsequently, 1 is written to all cells with both wordlines activated. Next, 0 is written to all cells while keeping the WL1 deactivated, which writes 0 to a cell only if its AXL is connected to WL2. Therefore, cells with AXL connected to WL1 and WL2, will store a ROM value of 1 and 0, respectively. A read with both wordlines activated is done to retrieve the ROM data, followed by restoring the original RAM contents.

Sizing the register file

The register file enables buffering data in general purpose registers to avoid higher-cost access to shared memory. However, if the register file were too large, it would

degrade core storage density. A key point to observe is that the majority of ML kernels are such that data is consumed within 1-2 instructions after being produced. This property is preserved via proper instruction scheduling by the compiler to reduce register pressure (Section **??**). Therefore, we provision a per-core register file size of *2*(crossbar dimension)*(# crossbars per core)*. This size retains storage density while addressing the buffering requirements in the common case as shown in Section 3.8.6. For window-based computations such as pooling layers that have a large number of intervening instructions (due to non-sequential data access across rows), the compiler spills registers to tile memory (Section **??**).

ISA implications

To accommodate the large register file required to match crossbar size, long operands are used in ISA (*src* and *dest* in Table 3.2), which is another motivation for the wide instruction design. To accommodate moving data between general purpose registers and *XbarIn*/*XbarOut* registers, a *copy* instruction is included.

3.3.5 Memory Unit (MU)

The MU interfaces the core with the tile memory via *load* and *store* instructions. These instructions can be executed at 16-bit word granularity to support random access as motivated in Section 3.2.3. However, the instructions also take a *vec-width* operand for wide vector loads caused by sequential access patterns. Vector loads also use temporal SIMD (Section 3.3.3) to minimize fetch/decode energy consumption.

3.3.6 Static Instruction Usage

Figure 3.4 shows the breakdown of the static instruction count for six different ML workloads. The breakdown demonstrates that MVM alone is insufficient to support all types of workloads, and that the ISA and functional units proposed can be used

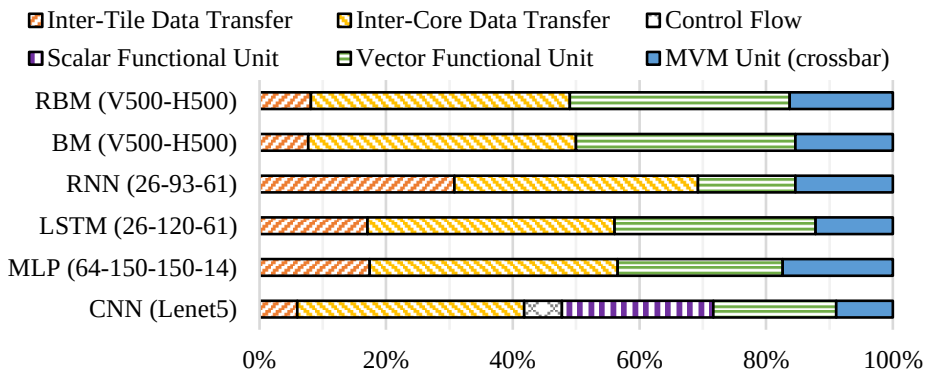


Fig. 3.4. Static instruction usage showing the importance of different execution units.

to bridge that gap. The ratio of instructions requiring MVMU versus VFU varies depending on the number of matrix versus vector transformation layers in the network. CNNs additionally use control flow instructions as discussed in Section 3.2.3. Deeper (or wider) versions of the same networks tend to have a similar instruction breakdown, except for data movement instructions which tend to be higher to implement larger matrices spanning multiple cores and tiles.

3.3.7 Summary

In summary, the core architecture provides programmability while maintaining crossbar area efficiency. It features an instruction pipeline exposed by an ISA to support a wide variety of ML workloads. The use of temporal SIMD and ROM-Embedded RAM enable linear, nonlinear, and transcendental vector operations. Data movement optimizations are enabled via input shuffling, proper sizing of the register file, and flexible memory access instructions.

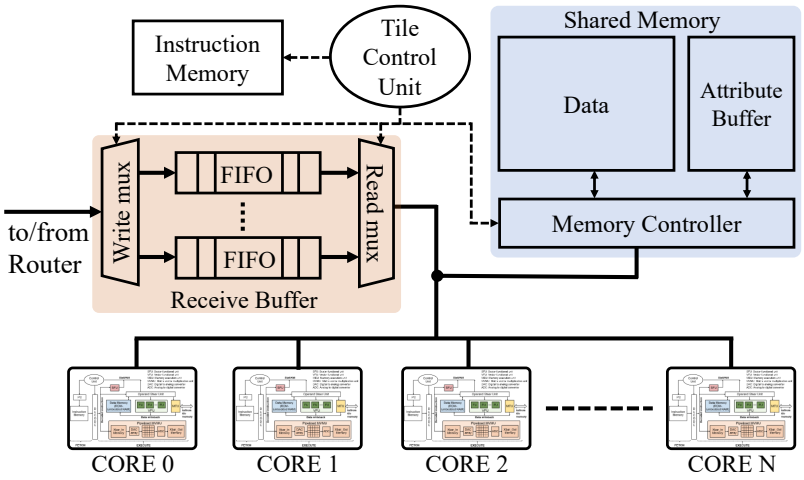


Fig. 3.5. Tile Architecture

3.4 Tile Architecture

Figure 3.5 illustrates the architecture of a tile. A tile is comprised of multiple cores connected to a shared memory. The tile instruction memory holds *send* and *receive* instructions that move data between tiles. The shared memory and receive buffer are described in the following subsections.

3.4.1 Shared Memory

The shared memory facilitates communication across cores and tiles. Our shared memory design follows two key principles: (1) enabling inter-core synchronization, and (2) sizing the shared memory to preserve storage density.

Inter-core synchronization

Synchronization between cores happens when the output of one layer is sent as input to the next layer. It also happens within a layer if a large weight matrix is partitioned across multiple cores and tiles and partial MVM results need to be

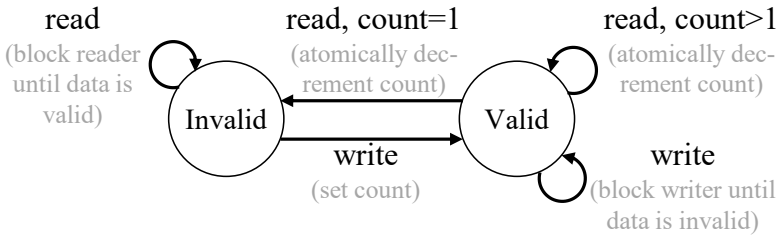


Fig. 3.6. Inter-core synchronization mechanism

Table 3.3.
Shared memory sizing

Workload	**Type**	**Required size without inter-tile pipelining**	**Required size with inter-tile pipelining**
MLP-L4	MLP	6 kB	6 kB
NMT-L5	LSTM	101 kB	2 kB
LSTM-2048	LSTM	216 kB	10 kB
Vgg19	CNN	1,176 kB	74 kB

aggregated together. To enable synchronization, we augment the shared memory with an attribute buffer that has two attributes per data entry: valid and count. The use of valid and count is illustrated in Figure 3.6. This mechanism enables consumer cores to block until producer cores have written their values, and ensures that producer cores do not overwrite data until it is consumed.

Sizing the shared memory

ML workloads may require buffering large number of inputs to utilize their weight reuse pattern. However, a large shared memory degrades the crossbar storage density. PUMA's spatial architecture enables programming *inter-core/tile pipelines* that exploit the inter-layer parallelism in ML workloads with weight reuse. These pipelines can be used to maintain throughput while keeping the shared memory size small. The

pipeline parallelism is based on the observation that we do not require all the outputs of previous layer to start the current layer computation. For example, LSTMs process a sequence of vectors with $S * N$ inputs per layer, where S is the number of vectors per input sequence and N is vector size. A layer can begin its computation as soon as its first N inputs are available. Table 3.3 shows the shared memory size requirement for four state-of-the-art ML workload types (described later in Section 3.7) with and without inter-tile pipelining. Note that MLP does not benefit from inter-tile pipelining because it does not exhibit any weight reuse (Section 3.2.1). Section 3.8.5 discusses the impact of shared memory sizing on energy consumption.

3.4.2 Receive Buffer

The receive buffer is an $N \times M$ structure with N FIFOs, each with M entries. FIFOs ensure that data being sent from the same source tile is received in the same order. Having multiple FIFOs enables multiple source tiles to send data concurrently using different FIFOs. It also enables data to be received through the network independently of receive instruction ordering in the program. This independence is important because receive instructions are executed in program order in a blocking manner for hardware simplicity.

Each send and receive instruction has a *fifo-id* operand that specifies the receiving FIFO to be used for incoming data. Using the FIFO ID instead of the sender tile ID provides additional flexibility for the compiler to apply FIFO virtualization, where a FIFO can be used by different sender tiles in different programs or program phases while keeping the number of physical FIFOs small. The key insight is that a typical ML layer will receive inputs from the tiles mapped to the previous layer only. Therefore, using 16 FIFOs (despite the node having 138 tiles) supports workloads with up to *(16 tiles)*(8 cores)*(2 MVMU)*128* previous layer activations, which suffices for large-scale ML workloads.

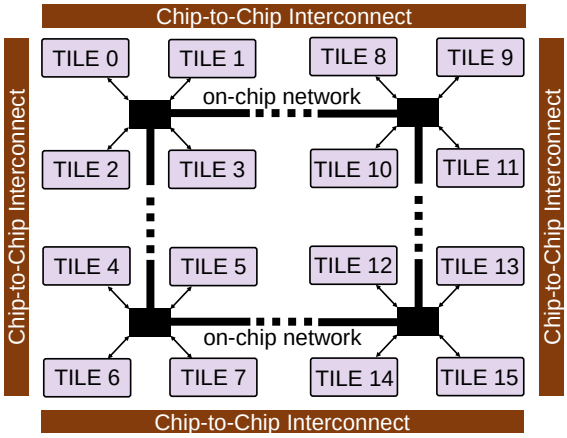


Fig. 3.7. Node architecture

3.4.3 Summary

In summary, PUMA tiles enable inter-core synchronization, inter-core/tile pipelines to contain shared memory size, and FIFO virtualization for efficient inter-tile communication.

3.5 Node Architecture

Figure 3.7 illustrates the organization of a PUMA node, consisting of multiple tiles connected by an on-chip network. Multiple PUMA nodes can be connected with a suitable chip-to-chip interconnect, such as CCIX [178], Gen-Z [179] or OpenCAPI [180], thereby enabling scalable acceleration of workloads with varying sizes. Note that the data movement cost in PUMA across the spatial hierarchies (core, tile, node) is analogous to data movement across the memory hierarchy in traditional systems (CPUs, GPUs). Here, the cost increases from intra-core to intra-node data movements as shown in Table 3.4.

Table 3.4.
Data Movement Cost

16-bit Data Transfer	**Energy (pJ)**	**Normalized cost**
Intra-core (copy)	0.954	1
Intra-tile (load, store)	3.009	3
Intra-node (send, receive)	18.045	19

3.6 Instruction Set Architecture

PUMA's instruction set is a domain specific ISA for implementing ML workloads on architectures with memristive crossbars, consisting of two components: core instructions and tile instructions (Table 3.2).

3.6.1 Core Instructions

Core instructions are categorized into compute, data movement, and control flow, and use a 56-bit format (Figure 3.8).

Computation Instructions

Figure 3.8(a) shows the ISA encoding of the MVM instruction. The MVM instruction invokes the MVMU(s) with the matrix preloaded on the crossbar and the vector in Xbar In memory (see Figure 3.1). The *xb-nma* operand specifies a mask for the MVMUs in the core that will be active during the MVM operation (recall that a core can have multiple MVMUs). Since MVMs are long operations, this mask allows MVMUs to execute in parallel when multiple matrices are used. It also allows MVMUs to be disabled when not used to save energy. The *src1* and *src2* register addresses are used to specify the kernel and stride sizes for input sharing across adjacent convolutions in CNNs.

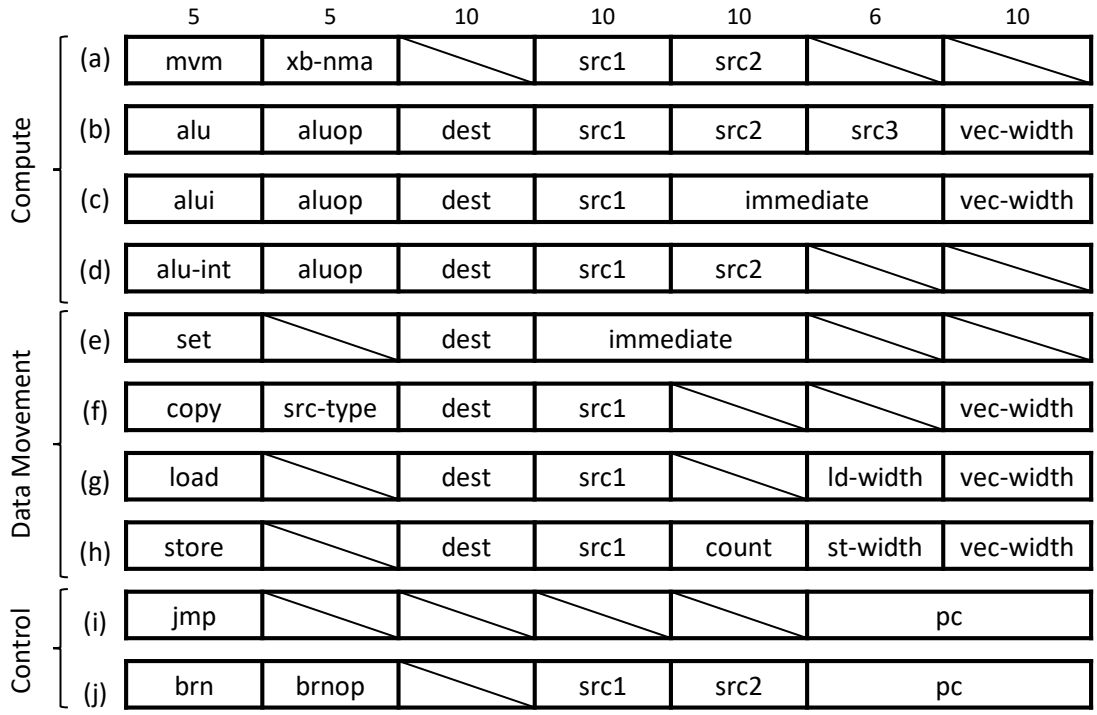


Fig. 3.8. ISA Encoding of Core Instructions

Figure 3.8(b) shows an *alu* instruction that either uses the VFU for vector arithmetic/logical operations or the ROM-embedded-RAM for transcendental operations. The *aluop* operand specifies the type of vector operation. Table **??** shows a list of all vector operations supported. The *dest*, *src1* and *src2* operands are references to the destination and two source operand locations in the core data memory respectively. The *src3* operand is used to specify the third operand for left and right shift operations. The *vec-width* operand specifies the vector width of the *alu* instruction. Successive computations within a vector instruction use contiguous destination and source addresses.

The *alui* instructions (Figure 3.8(c)) also run on the VFU or ROM-embedded-RAM depending on the *aluop* similar to *alu* instructions, but they use one immediate operand instead. The *alu-int* instructions (Figure 3.8(d)) run on the SFU.

Data Movement Instructions

Figure 3.8(e) shows a *set* instruction that writes an *immediate* to a data memory location. This instruction is used to initialize addresses used by load and store instructions as well as fixed program parameters such as loop bounds.

Figure 3.8(f) shows a *copy* instruction that moves data between memories within a core (data memory, Xbar In, Xbar Out). The *src-type* specifies if the source for data is Xbar In or Xbar Out. This distinction is needed because we alias the memory addresses for Xbar In and Xbar Out to reduce the total addressable locations on the core, and the instruction bits needed to address them. This aliasing is feasible because *mvm* is the only computation instruction that reads Xbar In and writes Xbar Out. No instruction ever reads both simultaneously, or writes both simultaneously.

Figures 3.8(g) and (h) show *load* and *store* instructions respectively for moving data between core memory and tile data memory. The *count* field in the *store* instruction contains the count attribute associated with a memory write used for synchronization. The *ld-width* and *st-width* together with the *vec-width* specify the length of data to be read or written.

Control Flow Instructions

Figures 3.8(i) and (j) respectively show the two supported control flow instructions, unconditional jump (*jmp*) and conditional branch (*brn*). Both take a *pc* with the target instruction address. Additionaly, *brn* takes a *brnop* that specifies the branch condition (equal, not-equal, etc.) and *src1* and *src2* which are operands for the condition evaluation.

3.6.2 Tile Instructions

We support two 56-bit instructions at the tile level to enable data movement between tiles, *send* and *receive*, shown in Figures 3.9(a) and (b). The *memaddr*

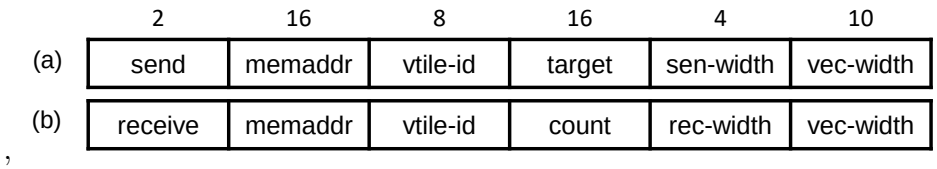


Fig. 3.9. ISA Encoding of Tile Instructions

operand specifies the tile data memory location where the sent data resides or the received is written. The *vtile-id* of a *send* instruction specifies the FIFO to which the data will be written to in the destination tile's receive buffer. In the corresponding *receive* instruction, the same *vtile-id* designates the FIFO where the received data is read from. The *target* operand in a *send* instruction specifies the target tile to which the data is sent. The *count* operand in the *receive* instruction specifies the count attribute for the tile data memory write performed by the receive. Finally, the *sen-width*, *rec-width*, and *vec-width* specify the number of data packets to be sent/received.

3.7 Evaluation Methodology

3.7.1 PUMA Simulator

We have implemented a detailed architectural simulator to evaluate the performance and energy consumption of PUMA. PUMA simulator runs applications compiled to the PUMA ISA and provides detailed traces of execution. The simulator incorporates functionality, timing, and power models of all system components. The datapath for the PUMA core and tile was designed at Register-Transfer Level (RTL) in Verilog HDL and mapped to IBM 45nm SOI technology using Synopsys Design Compiler which was used to measure the area and power consumption. Subsequently, these area and power numbers were added to the simulator for system-level evaluations of workloads. For fair comparison with other application-specific accelerators,

Table 3.5.
PUMA Hardware Characteristics

PUMA Tile at 1GHz on 32nm Technology node				
Component	**Power (mW)**	**Area (mm^2)**	**Parameter**	**Specification**
Control Pipeline	0.25	0.0033	# stages	3
Instruction Memory	1.52	0.0031	capacity	4KB
Register File	0.477	0.00192	capacity	1KB
MVMU	19.09	0.012	# per core dimensions	2 128×128
VFU	1.90	0.004	width	1
SFU	0.055	0.0006	-	-
Core	42.37	0.036	# per tile	8
Tile Control Unit	0.5	0.00145	-	-
Tile Instruction Memory	1.91	0.0054	capacity	8KB
Tile Data Memory	17.66	0.086	capacity technology	64KB eDRAM
Tile Memory Bus	7	0.090	width	384 bits
Tile Attribute Memory	2.77	0.012	# entries technology	32K eDRAM
Tile Receive Buffer	9.14	0.0044	# fifos fifo depth	16 2
Tile	373.8	0.479	# per node	138
On-chip Network	570.63	1.622	flit_size # ports conc	32 4 4
Node	62.5K	90.638	-	-
Off-chip Network (per node)	10.4K	22.88	type link bandwidth	HyperTransport 6.4 GB/sec

the datapath energy numbers have been scaled to the 32nm technology node. Memory modules are modelled in Cacti 6.0 [181] to estimate the area, power, and latency. The on-chip-network is modelled using the cycle-level Booksim 2.0 interconnection network simulator [182] and Orion 3.0 [183] for energy and area models. We use a chip-to-chip interconnect model similar to DaDianNao's [41] which has also been

Table 3.6.

Benchmarking Platforms

Name	**Platform Characteristics**
Haswell	Intel Xeon E5-2650v3, 10-cores per socket, Dual Socket, 128GB DDR4
Skylake	Intel Xeon 8180, 28-cores per socket, Dual Socket, 64GB DDR4
Kepler	Nvidia Tesla K80, 2496 CUDA Cores, Dual GPUs (only 1 used), 12GB GDDR5
Maxwell	Nvidia Geforce TitanX, 3072 CUDA Cores, 12GB GDDR5
Pascal	Nvidia Tesla P100, 3584 CUDA Cores, 16GB HBM2

Table 3.7.

Benchmarks

DNN Type	**Application**	**DNN Name**	**# FC Layers**	**# LSTM Layers**	**# Conv Layers**	**# Parameters**	**Sequence Size**
MLP	Object	MLP_{L4}	4	-	-	5M	-
	Detection	MLP_{L5}	5	-	-	21M	-
Deep	Neural Machine	NMT_{L3}	1	6 (3 Enc.,3 Dec., 1024 cells)	-	91M	50
LSTM	Translation	NMT_{L5}	1	10 (5 Enc.,5 Dec., 1024 cells)	-	125M	50
Wide	Language	BigLSTM	1	2 (8192 cell, 1024 proj)	-	856M	50
LSTM	Modelling	LSTM-2048	1	1 (8192 cell, 2048 proj)	-	554M	50
CNN	Image	Vgg16	3	-	13	136M	-
	Recognition	Vgg19	3	-	16	141M	-

adopted by other accelerators. The MVMU power and area models are adapted from ISAAC [23]. The memristors have a resistance range of $100k\Omega - 1M\Omega$ and read voltage of 0.5V. The ADC is based on the Successive Approximation Register (SAR) design, and its area and power were obtained from the ADC survey and analysis [184, 185].

In the present study, we do not compromise ML accuracy as we conservatively choose 2-bit memristor crossbar cells. Note that laboratory demonstrations have shown up to 6-bit capabilities [16]. We use 16 bit fixed-point precision that provides very high accuracy in inference applications [23, 41]. Table 3.5 shows the PUMA configuration used in our analysis and lists the area-energy breakdown of PUMA components.

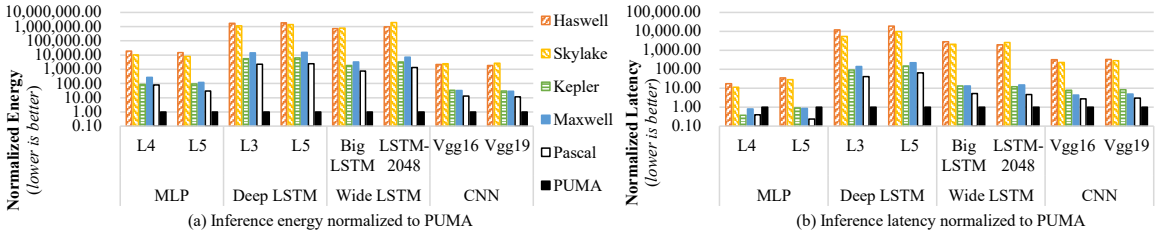


Fig. 3.10. Inference Energy and Latency Results

3.7.2 System and Workloads

We choose popular server grade CPUs and GPUs (listed in Table 3.6), Google TPU [168] (CMOS-based ASIC) and ISAAC [23] (application specific memristor-based accelerator) for evaluating PUMA. To measure power consumption of CPUs and GPUs, we used management tools such as board management control (BMC) and nvidia-smi respectively. For GPUs, we do not include full system power, just the board/device power. We run multiple iterations of the benchmarks on the GPU, discarding the longer warmup iterations and reporting results from the faster and more stable iterations.

Torch7 [186] was used to execute the ML models for CPUs and GPUs. The PUMA compiler was used to compile models for PUMA. These models used are listed in Table 3.7.

3.8 Results

3.8.1 Inference Energy

Figure 3.10(a) shows PUMA inference energy compared to other platforms. PUMA achieves massive energy reduction across all benchmarks for all platforms. Energy improvements come from two sources: lower MVM compute energy from crossbars and lower data movement energy by avoiding weight data movement.

CNNs show the least energy reduction over CMOS architectures (11.7×-13.0× over Pascal). Recall that CNNs have a lot of weight reuse because of the sliding window computation (discussed in Section 3.2.3). Hence, CMOS architectures can amortize DRAM accesses of weights across multiple computations. For this reason, PUMA's energy savings in CNNs come primarily from the use of crossbars for energy efficient MVM computation.

MLPs and LSTMs have little or no weight reuse (discussed in Section 3.2). Therefore, in addition to efficient MVM computation, PUMA has the added advantage of eliminating weight data movement. For this reason, we see much better energy reductions for MLPs (30.2×-80.1× over Pascal), Deep LSTMs (2,302×-2,446× over Pascal), and Wide LSTMs (758×-1336× over Pascal).

LSTMs (both deep and wide) show better reductions than MLPs because they have much larger model sizes (see # Parameters in Table 3.7). As model grows in size, weight data grows at $O(n^2)$ and input data grows at $O(n)$. For this reason, we see an increasing disparity between CMOS architectures which move both weight and input data, and PUMA which only moves input data.

Wide LSTMs have few layers (1-2) with very large matrices, whereas Deep LSTMs have many layers (6-10) with smaller matrices. The large matrices in Wide LSTMs span mutiple PUMA cores/tiles to compute one logical MVM, incurring higher intra-layer data movement overheads. Hence, Deep LSTMs show higher energy benefits than Wide LSTMs.

3.8.2 Inference Latency

Figure 3.10(b) shows PUMA inference latency compared to other evaluated platforms. PUMA achieves latency improvements across all platforms except MLPs on some GPUs. Latency improvements come from three sources: lower MVM compute latency from crossbars, no weight data access latency, and spatial architecture pipelining which exploits inter-layer parallelism.

CNNs show the least latency improvement over CMOS architectures (2.73×-2.99× over Pascal). Since CNNs are compute-bound, CMOS architectures can hide the data access latency. Thus, PUMA's primary latency improvements in CNNs come from the use of crossbars for low-latency MVM computation and spatial architecture pipelining.

LSTMs on the other hand are memory-bound. PUMA has the added advantage of eliminating weight data access latency in addition to low-latency MVM computation. For this reason, we see much better latency improvements for Deep LSTMs (41.6×-66.0× over Pascal) and Wide LSTMs (4.70×-5.24× over Pascal) than we see for CNNs. In comparing Deep and Wide LSTMs, Deep LSTMs have more layers than Wide LSTMs, hence more inter-layer parallelism to exploit spatial architecture pipelining (see #LSTM Layers in Table 3.7). Moreover, Deep LSTMs have less intra-layer communication than Wide LSTMs, hence lower data access latency.

MLPs show slowdown compared to some GPU datapoints (0.24×-0.40× compared to Pascal). The reason is that despite MLPs being memory-bound, the sizes of MLPs are typically small enough. Hence, the memory bandwidth bottleneck is not as pronounced, so they perform fairly well on GPUs. Moreover, MLPs have no inter-layer parallelism so they do not exploit spatial architecture pipelining (Section 3.4.1). Nevertheless, PUMA's order of magnitude energy reduction is still beneficial for MLPs in energy-constrained environments.

3.8.3 Batch Throughput and Energy

Inference applications are not usually intended for large batch sizes due to real-time application requirements. Nevertheless, CMOS architectures perform well with large batch sizes because of the weight reuse that data-batching exposes. For this reason, we compare PUMA's batch energy and throughput with the other platforms in Figure 3.11(c) and (d) respectively. Batch sizes of 16, 32, 64, and 128 are used.

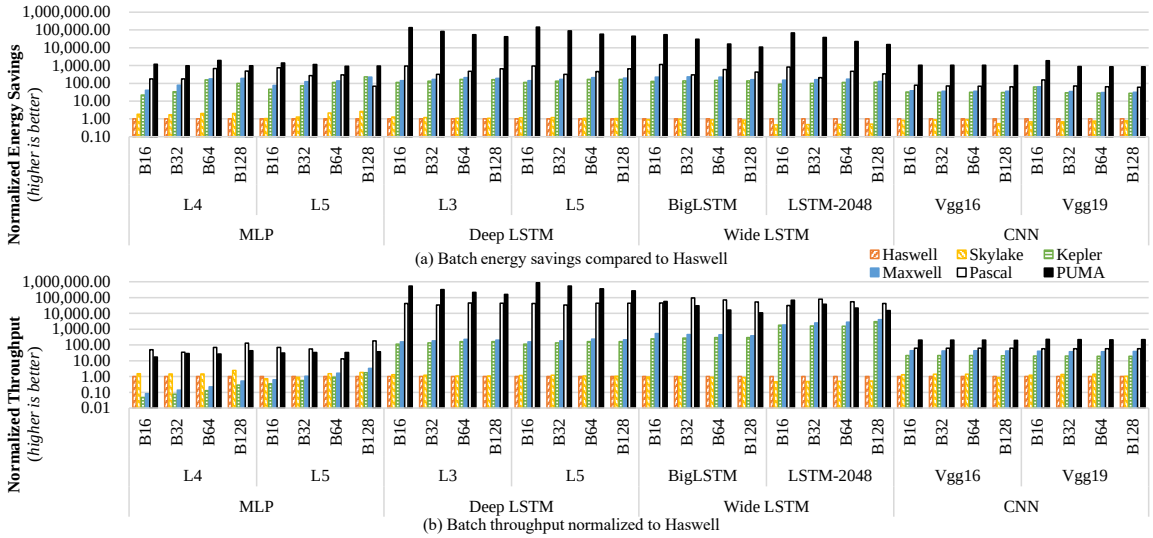


Fig. 3.11. Batch Inference Throughput and Energy Results

PUMA continues to have superior energy efficiency across all benchmarks and batch sizes. It also delivers better throughput in most cases, except when compared to Pascal on MLPs and Wide LSTMs. The benefits slightly decrease with larger batches because they expose more weight reuse which benefits CMOS architectures while PUMA's efficiency remains constant across batch sizes. Nevertheless, PUMA continues to perform well even when the batch size is very large.

3.8.4 Comparison with ML Accelerators

Google TPU

Table 3.8 compares key technology features and efficiency metrics for TPU [168] and PUMA. PUMA has 8.3× higher peak area-efficiency (TOPS/s/mm^2) than TPU. Since PUMA does not rely on weight reuse to improve throughput like CMOS architectures do, its area-efficiency remains constant across workloads and batch sizes. On the other hand, TPU's peak throughput is almost an order of magnitude lower for applications with low data reuse due to its inability to amortize weight data move-

Table 3.8.
Comparison with ML Accelerators

Platform	**PUMA**	**TPU** [168]	**ISAAC** [23]
Year	2018	2017	2016
Technology	CMOS(32nm)-Memristive	CMOS(28nm)	CMOS(32nm)-Memristive
Clock (MHz)	1000	700	1200
Precision	16-bit fixed point	16-bit fixed point	16-bit fixed point
Area (mm^2)	90.6	330*	85.4
Power (W)	62.5	45	65.8
Peak Throughput (TOPS/s†)	52.31	23‡	69.53
Peak AE (TOPS/s/mm^2)	**0.58**	**0.07**	**0.82**
Peak PE (TOPS/s/W)	0.84	0.51	1.06
Best AE - MLP	0.58	0.009	-
Best AE - LSTM	0.58	0.003	-
Best AE - CNN	0.58	0.06	0.82
Best PE - MLP	0.84	0.07	-
Best PE - LSTM	0.84	0.02	-
Best PE - CNN	0.84	0.48	1.06

* Less than or equal to half of Haswell's die area [168]

† Tera operations per second (multiply and add are counted as two separate operations)

‡ 92 TOPS for 8-bit arithmetic, scaled by 4 for 16-bit arithmetic [168]

Table 3.9.
Programmability Comparison with ISAAC

Platforms	**PUMA**	**ISAAC**
Architecture	Instruction execution pipeline, flexible inter-core synchronization	Application specific state machine
	Vector Functional Unit, ROM-Embedded RAM	Sigmoid unit
Programmability	Compiler-generated instructions (per tile & core)	Manually configured state machine (per tile)
Workloads	CNN, MLP, LSTM, RNN, GAN, BM, RBM, SVM, Linear Regression, Logistic Regression	CNN

ment. PUMA has 64.4×, 193×, and 9.7× higher area-efficiency than TPU for MLP, LSTM, and CNN respectively for the best TPU batch size.

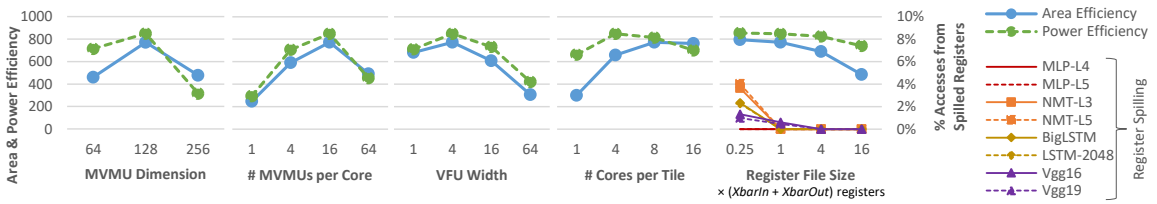


Fig. 3.12. Design Space Exploration: Tile Area Efficiency in GOPS/s/mm^2 and Tile Power Efficiency in GOPS/s/W

PUMA has 1.65× higher peak power-efficiency (TOPS/s/W) than TPU, with similar trends and reasoning as area-efficiency for specific workloads. We expect PUMA's power-efficiency advantage over TPU to grow by over 3×, as silicon processes scale from 32nm to 7nm and 5nm. Thanks to PUMA's higher peak throughput, we can follow the power reduction scaling curve at constant performance. Conversely, to narrow the performance gap, TPU would follow a scaling curve closer to the performance increase curve at constant power. Note that the former scaling curve is much faster (up to ∼40% power reduction per silicon node compared with ∼20% performance increase). Further, ADC power efficiency has also been following similar and very fast scaling trend with ∼2× power reduction per 1.8 years at the same sampling rate [187]. Consequently, we project that our power-efficiency advantage over TPU will increase by over 3× after scaling PUMA and TPU to 7nm.

ISAAC

Table 3.8 compares the peak area and power efficiency of PUMA with ISAAC [23], a memristor-based accelerator customized for CNNs. PUMA has 20.7% lower power efficiency and 29.2% lower area efficiency than ISAAC due to the overhead of programmability. Table 3.9 compares the programmability of PUMA and ISAAC.

Table 3.10.
Evaluation of Optimizations

Workload	**Input Shuffling** (energy reduction, lower is better)	**Shared Memory Sizing** (energy reduction, lower is better)	**Graph Partitioning** (energy reduction, lower is better)	**Register Pressure** (% accesses from spilled registers)	**MVM Coalescing** (latency reduction, lower is better)
MLP_{L4}	-	0.70×	0.81×	0%	0.60×
MLP_{L5}	-	0.66×	0.79×	0%	0.66×
NMT_{L3}	-	0.65×	0.65×	0%	0.63×
NMT_{L5}	-	0.63×	0.63×	0%	0.63×
BigLSTM	-	0.58×	0.61×	0%	0.76×
LSTM-2048	-	0.58×	0.62×	0%	0.84×
Vgg16	0.84×	0.75×	0.37×	1.96%	0.69×
Vgg19	0.85×	0.75×	0.43×	1.71%	0.71×

PUMA with Digital MVMUs

To demonstrate the importance of analog computing for MVMU efficiency, we compare PUMA with a hypothetical equivalent that uses digital MVMUs. A memristive 128×128 MVMU performs 16,384 MACs in 2304 ns consuming 43.97 nJ. A digital MVMU would require 8.97× more area to achieve the same latency and would consume 4.17× more energy. Using a digital MVMU would increase the total chip area of the accelerator by 4.93× for the same performance and would consume 6.76× energy (factoring in data movement energy due to increased area).

Tensor Cores

Nvidia V100 GPUs with tensor cores (FP16) can be up to 6× more energy-efficient (architecture, tensor cores, and half-precision) than Pascal GPUs. Therefore, PUMA can still achieve energy gains over GPUs with tensor cores, despite the technology difference (PUMA: 32nm, V100: 12nm).

3.8.5 Evaluation of Optimizations

Table 3.10 shows an evaluation of various optimizations described throughout the paper. ***Input shuffling*** reduces energy consumed by data movement within a core by leveraging input reuse in CNNs. ***Shared memory sizing*** keeps the shared memory small by leveraging inter-core/tile pipelining. The baseline here, sizes the shared memory with what would be needed without pipelining, which is 1×, 50.51×, 21.61×, and 15.91× larger for MLPs, Deep LSTMs, Wide LSTMs, and CNNs respectively. Note that MLP does not benefit from inter-tile pipelining because it does not exhibit any weight reuse (Section 3.2.1). ***Graph partitioning*** (compared to a baseline that partitions the graph randomly) reduces the number of loads, stores, sends, and receives, hence the overall energy. ***Register pressure*** is kept low by the compiler with little or no accesses from spilled registers across benchmarks. ***MVM coalescing*** runs MVMUs in parallel within a core which reduces latency.

3.8.6 Design Space Exploration

Figure 3.12 shows a PUMA tile's peak area and power efficiency swept across multiple design space parameters. For each sweep, all other parameters are kept at the sweetspot (PUMA configuration with maximum efficiency). Efficiency is measured using a synthetic benchmark: an MVM operation on each MVMU, followed by a VFU operation, then a ROM-Embedded RAM look-up.

Increasing the ***MVMU dimension*** increases the number of crossbar multiply-add operations quadratically and the number of peripherals linearly resulting in more amortization of overhead from peripherals. However, larger MVMUs also require ADCs with higher resolution and ADC overhead grows non-linearly with resolution, which counter-balances the amortization. Increasing the ***# MVMUs per core*** increases efficiency because of the high efficiency of memristive crossbars relative to CMOS digital components. However, with too many MVMUs, the VFU becomes a bottleneck which degrades efficiency. Increasing the ***VFU width*** degrades efficiency

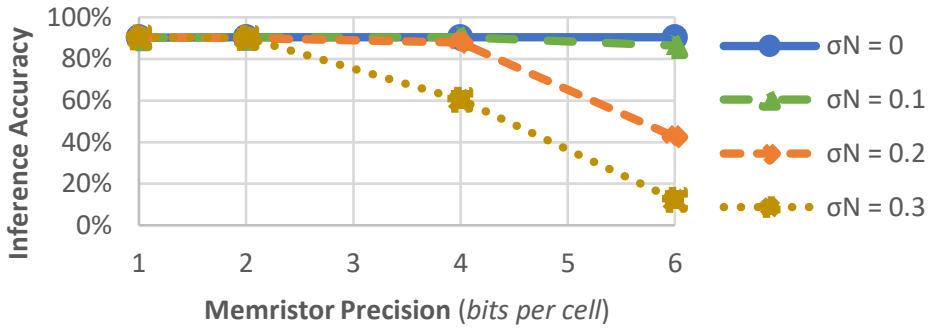


Fig. 3.13. Inference Accuracy

because of the low efficiency of CMOS relative to memristive crossbars. However, a VFU that is too narrow becomes a bottleneck. The sweetspot is found at 4 vector lanes. Increasing the ***# cores per tile*** improves efficiency until shared memory bandwidth becomes the bottleneck. Increasing the ***register file size*** results in lower efficiency, however a register file that is too small results in too many register spills.

Figure 3.13 shows PUMA's inference accuracy for different memristor bit precision (bits per device) and write noise levels (σ^N). Higher precision can lead to larger accuracy loss due to the reduction in noise margin. It can be seen that PUMA with 2-bit memristor performs well even at high noise levels. Real CMOS hardware follows the $\sigma^N = 0$ noise level. Further, recent research have explored coding schemes for reliable memristor computation at high precision [188, 189].

3.9 Conclusion

PUMA is the first ISA-programmable accelerator for ML inference that uses hybrid CMOS-memristor technology. It enhances memristive crossbars with general purpose execution units carefully designed to maintain crossbar area/energy efficiency and storage density. Our accelerator design comes with a complete compiler to transform high-level code to PUMA ISA and a detailed simulator for estimating performance

and energy consumption. Our evaluations show that PUMA can achieve significant improvements compared to state-of-the-art CPUs, GPUs, and ASICs for ML acceleration.

4. TRANNSFORMER - SOFTWARE OPTIMIZATION

4.1 Introduction

Deep Neural Networks (DNNs) are the most popular class of ML workloads and inspired from the hierarchy of neurons and synapses in the human brain. They have achieved outstanding classification accuracies across myriad cognitive applications including computer vision [169], speech recognition and natural language processing [174]. Consequently, they have been adopted in variety of applications across different computational platforms used in day-to-day life. For example, Siri, Google Now, Cortana are intelligent personal assistants developed by Apple, Google and Microsoft respectively and run DNNs to recognize external inputs (image, voice etc.). This tremendous pervasiveness has been possible due to the surge in the scale/size of the DNNs, which has enabled it to obtain human-level cognitive abilities. However, increasing DNN scale leads to high energy and resource requirements, thereby impeding their deployment in low-power and resource constrained platforms. This has motivated extensive research on DNN compression using weight pruning [12] and has shown upto $50\% - 90\%$ reduction in model sizes [12]. Subsequently, the reduction in model size achieves highly efficient inference in CMOS ASICs [100], FPGS [51] and general-prupose systems [110].

Weight pruning produces highly sparse DNNs, which can significantly reduce the memory, and computation requirements. However, such algorithmic approaches induce irregular sparsity which is incompatible with memristive crossbars due to the in-memory nature of MVM operations. A memristive crossbar maps the weights as well as performs the MVMs. Pruning a weight mapped onto a crossbar merely makes the crosspoint (of crossbar) unused. This is because each crosspoint physically serves as a possible connection channel between an input activation (mapped to the

row) and an output activation (mapped to the column). Consequently, realizations of sparse DNNs obtained from weight pruning onto crossbar based systems results in high area consumption. Typical crossbar based systems use several peripherals namely analog-to-digital converters, buffers, communication etc. in order to execute DNNs (discussed in Sections 3.3 and 3.4). Increasing the sparsity deteriorates the crossbar utilization, which results in a peripheral dominated energy profile. Eventually, weight sparsity does not translate to commensurate area and energy savings in a crossbar based system.

In this work, we present TraNNsformer which is an integrated training framework for dynamically learning clustered connections during training. This is motivated by the observation that pruning at the crossbar granularity, instead of a weight granularity, can preserve the benefits of weight sparsity at the hardware level for crossbar based systems. Our approach efficiently prunes the model by dynamically making pruning decisions from accuracy perspective (removing unnecessary connections to maximize sparsity) as well as clustering perspective (removing unclustered weights to maximize crossbar utilization) thereby producing trained models that can efficiently utilize the benefits of memristive technology. Further, our proposed transformation is technology agnostic as it enables mapping of connectivity structures using any MCA size permitted by the memristive technology for reliable operations.

Note that this work focuses on transformations for Fully Connected (FC) layers. Majority of the image processing and computer vision applications run CNNs, which are comprised of several convolution layers followed by a few FC layers. However, many CNNs have more than 96% of the weights are in the FC layers [46]. Furthermore, FC layers are widely used in different types of DNNs namely MLP, RNN, Long Short Term Memory (LSTM) [190] and SNN [191].

In summary, this work makes the following contributions:

1. A Size-Constrained Iterative Clustering (SCIC) algorithm for efficiently mapping dense matrices on crossbar sizes permissible by the memristive technology.

2. A training framework leveraging the SCIC algorithm to enable mapping large-scale DNNs on crossbar systems.

3. An evaluation on a range of image recognition benchmarks to study the area and energy improvements.

4.2 TraNNsformer Framework

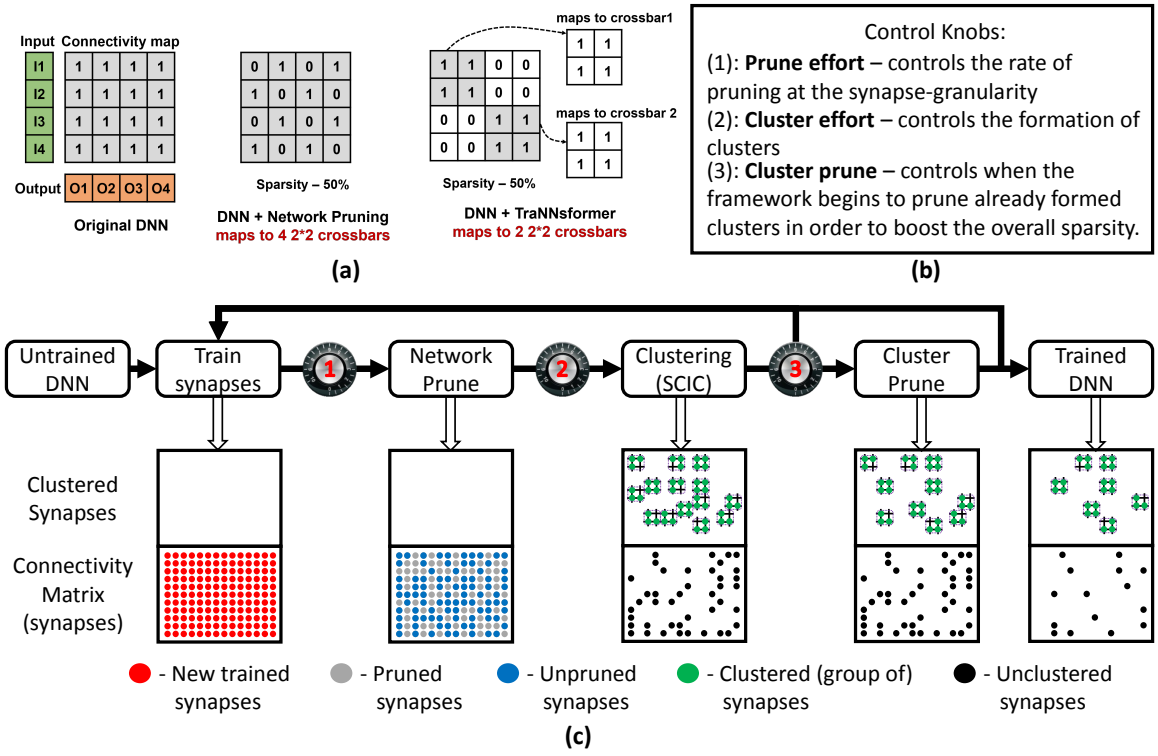


Fig. 4.1. (a) Illustration of Network pruning and TraNNsformer: pruning leads to irregular sparsity, TraNNsformer forms smaller clusters that are mapped efficiently to crossbars (1/0 only represents presence/absence of connection), (b) Description of control knobs in the framework, (c) Logical flow of TraNNsformer during DNN training.

Fig. 4.1 illustrates the logical flow of the framework with examples of how each execution stage affects the connectivity matrix of one particular layer during a training

iteration. In total there are 4 execution stages: (1) single training epoch of weights, (2) network pruning, (3) cluster formation, and (4) cluster pruning.

The first stage is the conventional way of training DNNs using feed-forward propagation of inputs and back-propagation of gradients to update weights for one training epoch. Each epoch results in a new set of weights, which forms the connectivity matrix of a layer (as shown in the example of Fig. 4.1(c)). Note that this can also result in the *revival* of weights pruned in earlier training epochs.

The second stage is network pruning which prunes some of the weights in the connectivity matrix based on the *prune effort* control knob. Here, the connectivity matrix is composed of both weights which were left after the pruning and clustering stages of the previous training iteration as well as the ones which were "revived" during the last training step. The blue and grey dots in Fig. 4.1(c) represent unpruned and pruned weights respectively. This stage is responsible for sparsity at weight granularity.

Cluster formation creates clusters from unpruned weights based on the *cluster effort* control knob. Cluster formation is implemented by the Size Constrained Iterative Clustering (SCIC) algorithm. The weights leading to cluster formation (shown as green dots in Fig. 4.1(c)) are masked such that they are less susceptible to pruning in the next training epoch than the unclustered weights (black dots in Fig. 4.1(c)).

Cluster pruning stage prunes the already formed clusters thereby, creating sparsity at the cluster granularity. The *Cluster prune* control knob manages the activation of this stage. We observed that initiating cluster pruning during the final $10 - 20\%$ of training epochs, achieves maximal benefits.

TraNNsformer framework is the iterative execution of stages 1-4 in each training epoch until the DNN training converges (in terms of accuracy) and an efficient mapping (in terms of crossbar requirements) is achieved. The following subsections discuss each stage in detail and discuss the impact of Trannsformer on crossbar-based architectures.

Algorithm 1 Spectral Clustering (SC)

Input: Similarity matrix $S \in R^{n \times n}$ for the graph (a row in S corresponds to a graph node), K clusters to construct

1: Compute the degree matrix: D

2: Compute the normaized laplacian matrix: L

3: Perform an eigenvalue decomposition: $L = U \Sigma U^{\mathrm{T}}$

4: Extract the K columns of U corresponding to the K smallest eigenvalues to form: $\widetilde{U}$

5: Cluster the row vectors of $\widetilde{U}$ using *K-means* algorithm

Output: K clusters - a row vector of $\widetilde{U}$ corresponds to a row of S

4.2.1 Spectral Clustering

Spectral Clustering (SC) [192] is a graph clustering algorithm that produces a set of disjoint graph nodes such that intra (inter) cluster associativity is maximized (minimized). We adopt spectral clustering to cluster the connectivity matrix for a FC layer of DNN where the input and output activations are the graph nodes and a weight corresponds to a graph edge. As shown in Algorithm 1, a symmetric matrix (L) is defined on the graph (connectivity matrix) using the adjacency matrix (S) and degree matrix (D). Subsequently, it undergoes an eigenvalue decomposition followed by dimensionality reduction. Finally, the remaining row vectors are clustered using the K-means algorithm. The intra-associativity maximization produces clusters that can be mapped to crossbars with high utilization factor. Furthermore, inter-associativity minimization enables low overhead or high throughput (number of activations computed per cycle) integration of crossbars outputs for generating the output activation. This is because the weights corresponding to a particular output activation can be spread across multiple clusters (mapped to multiple crossbars across cores/tiles). Eventually, the outputs from multiple crossbars are integrated to generate the output. Hence, minimizing the inter-cluster associativity results in a commensurate decrease in the inter crossbar interaction, thereby maximizing throughput.

4.2.2 Size Constrained Iterative Clustering (SCIC)

SCIC is an iterative algorithm based on SC, that minimizes the number of unclustered weights while ensuring cluster generation with higher utilization factors. Connectivity matrix (C) is a (0, 1)-matrix that represents the current morphology of a layer in the DNN such that a value C_{ij} being one corresponds to a non-zero weight between the i_{th} input activation and j_{th} output activation. Utilization factor is then defined as the fraction of used (mapped) cross-points in a memristive crossbar. A zero value in a cluster results in an unused cross-point. As shown in Algorithm 2, each iteration of SCIC algorithm runs SC on the remaining connectivity matrix based on the user defined parameters for crossbar and base/minimum utilization factors.

[Algorithm 2. Lines 1-3] Since SCIC is iterative algorithm, the cluster set is initialized to be the remaining connectivity matrix ($cluster\_set$). However, the clusters (in cluster set) are filtered to form the final set of sub clusters ($qual\_sub\_clusters$) based on two requirements: (1) cluster size $\leq$ crossbar size, and (2) clusters quality should be within an acceptable range. Note that the term sub clusters is used to differentiate the qualified clusters from the main pool of clusters.

[Algorithm 2. Line 4] Crossbar size ($crossbar\_size$) reflects the crossbar size. This parameter dictates the sub cluster size formed by SC (by dividing clusters of the initial cluster set) and allows the algorithm to perform technology aware mapping.

[Algorithm 2. Lines 5-17] Base ($base\_util\_factor$) and minimum (min_util_factor) utilization factors are used to set a range of quality thresholds for filtering the formed sub clusters. Since utilization factor is the fraction of mapped cross-points on a crossbar, high quality sub clusters are the sub clusters that map to crossbars with high utilization factor. Sub clusters formed by SC are greedily selected based on the quality threshold. The threshold starts from the base utilization factor and decays towards the minimum value, if no sub clusters satisfying the current threshold can be obtained. Since utilization factor is a normalized value, the value of base and minimum utilization factor as well as of quality thresholds lie in the range $[0, 1]$.

Algorithm 2 Size Constrained Iterative Clustering (SCIC)

Input: Connectivity matrix $C \in R^{m \times n}$ (m and n are number of input and output neurons respectively), crossbar_size, base_util_factor, min_util_factor, k

1: Initialize: cluster_set = C, qual_sub_clusters = $\emptyset$
2: **while** Δcluster_set $\neq 0$ **do**
3: **for** all *cluster* **in** *cluster_set* **do**
4: **if** size of (*cluster*) > crossbar_size **then**
5: s_clusters_found $\leftarrow \emptyset$
6: util_factor = base_util_factor
7: **while** s_clusters_found $\equiv \emptyset$ **do**
8: **if** util_factor < min_util_factor **then**
9: break
10: **else**
11: $\widetilde{C}$ = connectivity matrix of cluster
12: construct similarity matrix S from $\widetilde{C}$
13: s_clusters_temp = spectral_clustering(S, k)
14: **for** all *s_cluster* **in** *s_clusters_temp* **do**
15: **if** quality of (*s_cluster*) $\geq$ util_factor **then**
16: s_clusters_found $\leftarrow$ s_clusters_found + s_cluster
17: decay (util_factor)
18: update C to reflect unclustered weights
19: qual_sub_clusters $\leftarrow$ qual_sub_clusters +s_clusters_found

Output: *qual_sub_clusters* - the set of qualified sub clusters

[Algorithm 2. Lines 18-19] Such greedy and iterative approach synergistically ensures efficient clustering to produce high quality sub clusters. Other sub clusters (below quality threshold) are ignored and merged with the existing connectivity matrix to explore new clustering avenues in the subsequent iterations.

4.2.3 Integrated Training Approach

As shown in Algorithm 3, each training iteration (back-propagation) is accompanied with pruning followed by SCIC. Pruning removes the weights that do not affect the accuracy. The result of pruning is encoded as a *prunemap* which is a (0, 1)-matrix (similar to the connectivity matrix) where $0s'$ represent a pruned weight. $0s$ in the prune map correspond to Accuracy Don't Cares (ADCs). Subsequently, clusters produced in SCIC are used to form a *clustermap* that denotes if a weight in the connectivity matrix is part of a previously formed cluster. $0s$ in the cluster map represent Clustering Don't Cares (CDCs). The union of cluster map and prune map is masked from being pruned in the subsequent iterations. Weights belonging to both the ADC and CDC set, are aggressively pruned to maximize pruning without affecting the cluster quality. The subsequent training iteration tries to recover the accuracy loss incurred due to pruning.

Although SCIC generates high quality clusters, it leaves a large fraction of weights unclustered. This results in large number of crossbars with low utilization factors being mapped to the unclustered weights, thereby diminishing the benefits of SCIC. Consequently, a training algorithm based on offline clustering i.e. clustering the weights at the end of the training process will suffer from inefficiencies resulting from higher fraction of unclustered weights. However, it is interesting to note that subsequent pruning of the weights belonging to the intersection of ADC and CDC set removes several unclustered weights. Furthermore, this pruning exposes new avenues for SCIC to generate high quality clusters from the remaining unclustered weights. Thus, TraNNsformer allows to dynamically learn the DNN structure in a clustered manner during the training process in order to produce an optimized network for crossbar-based architectures.

It is worth noting that TraNNsformer favors cluster formation in the beginning to minimize the fraction of unclustered weights. Once the unclustered weights have been significantly reduced, TraNNsformer initiates cluster pruning (as shown in Algorithm

Algorithm 3 TraNNsformer

```
1: Train the connectivity for an epoch
2: if num_unclustered_weights < threshold then
3:   if training_error < training_error_previous then
4:     cluster_prune()
5: else
6:   if training_error < training_error_previous then
7:     Prune and update prune map
8:   Run SCIC on unclustered synpases and update cluster map
9: connectivity matrix ← (prune map ∪ cluster map)
10: Go to 1 if convergence is not reached
```

3). Cluster pruning incrementally prunes an entire cluster based on a combined score that quantifies cluster quality and the cluster's contribution to output accuracy. Although, cluster pruning may lead to accuracy degradation, subsequent training iteration ensures a graceful recovery of the lost accuracy. It is worth noting that cluster pruning does not affect the overall crossbar utilization as it entirely removes the mapped crossbar. Thus, cluster pruning allows achieving higher network sparsity (comparable to network pruning) while ensuring the clustered structure of the DNN's connectivity matrix.

4.3 Impact on Crossbar-based Architecture

Crossbar-based architectures are comprised of cores each of which consists of crossbars and peripherals associated namely buffers, communication and control logic (discussed in Section 3.3). Hence, the number of cores (*num_core*) has a linear dependence on the number of crossbars (*num_crossbar*) as shown in Equation 4.1 (where *"k"* is a microarchitecture dependent constant).

$$num\_core = \frac{num\_crossbar}{k} \tag{4.1}$$

TraNNsformer enables technology aware optimization to learn an optimally clustered network structure such that a learnt cluster can be mapped onto a crossbar with high utilization factor. Consequently, it ensures that the network sparsity efficiently translates to reduction in the number of crossbars required to map the transformed DNN with respect to the original DNN. This results in a commensurate reduction in the number of cores thereby leading to area savings.

The energy profile of a crossbar-based architecture is comprised of crossbar energy and peripheral energy components. Hence, the total energy consumption for a single inference can be defined as Equation 4.2.

$$\textit{Total Energy} = \sum_{i=0}^{\text{all cores}} \textit{Crossbar Energy} + \textit{Peripheral Energy} \tag{4.2}$$

An increase in weight sparsity results in a corresponding decrease in the crossbar energy component irrespective of the connectivity structure. However, the total peripheral energy component depends on the number of crossbars being used. As discussed before, weight pruning does not lead to significant reductions in the number of crossbars due to the irregular nature of sparsity pattern, thereby not affecting the total peripheral energy. This reduces the overall energy benefits that can be obtained by highly sparse connectivity structures. Additionally, the total energy profile becomes peripheral energy dominated, which would prevent harnessing the energy benefits from further efficient memristive technologies. On the contrary, TraNNsformer helps to obtain commensurate reductions in crossbar energy as well as the peripheral energy component, which in turn results in significant savings in total energy consumption. Consequently, the energy profile shows a favorable distribution between the crossbar and peripheral energy components.

Table 4.1.

MLP benchmarks

Application	Dataset	Layers	Neurons	Synapses
Digit Recognition	MNIST	3	2410	2392800
House Recognition	SVHN	4	3610	4120800
Object Recognition	CIFAR-10	4	3610	4120800

4.4 Experimental Methodology

TraNNsformer framework was designed using MATLAB by utilizing the relevant components from MATLAB Neural Network and Parallel Computing Toolboxes and a custom DeepLearn [193] Toolbox.

Algorithmic benefits of the framework were analyzed on MLP based Spiking Neural Networks (SNN). Recall, MLPs are comprised of fully connected layers only (Section 3.2). Applying TraNNsformer on MLPs updates the connectivity matrices between all layers of the network. In order to study the algorithm's scalability, we evaluate TraNNsformer on MLPs ranging in the number of layers and layer sizes. We use a range of applications, namely Digit Recognition (MNIST dataset [194]), House Number Recognition (SVHN dataset [195]) and Object Classification (CIFAR10 dataset [196]). For each application, an MLP architecture commensurate to the dataset complexity is chosen (shown in Table 4.1), to achieve high classification accuracy.

Although we analyze our results for SNN based benchmarks, the algorithmic benefits would be similar for Artificial Neural Networks (ANNs). This is because TraNNsformer works on optimizing the connectivity structure between layers in a DNN, which is similar for both SNN and ANN. Note that ANN and SNN (used in our case) differ only in the way inputs are transmitted between layers.

We used the architecture proposed in [84] for studying the system-level benefits of TraNNsformer on post-CMOS crossbar-based architectures. The crossbar size used to evaluate the proposed framework was 64×64 i.e. 64 rows and 64 columns. For the memristive devices, we used a resistance range of $20K\Omega - -200K\Omega$ with 16 levels (4 bits) for weight-discretization – typical of memristive technologies such as Phase Change Memory (PCM), Ag-Si [197]. We considered an operating voltage of $Vdd/2$ for the crossbar as it is interfaced with CMOS neurons [198]. To analyze the system level benefits on CMOS based general-purpose architectures, we use the energy numbers for arithmetic operations in a 45nm CMOS process shown in [12]. The memory for weight storage was modeled using CACTI [199].

TraNNsformer is targeted to improve the area and energy consumption of DNNs during inference phase but can have higher training effort in terms of time and energy consumption than typical DNN training (no pruning/clustering). This is because TraNNsformer can prune connections learned in the previous training epochs, to guide the training process towards crossbar aware connectivity structures. Thanks to the error-healing nature of training, TraNNsformer achieves iso-accuracy compared to non transformed networks. Network pruning also requires higher training effort than typical DNN training for the same reason. Note that, typical DNNs are trained very infrequently but used for testing/inference for much longer times. Hence, the significant benefits obtained for inference phase makes TraNNsformer extremely attractive for edge devices.

4.5 Results

In this section, we present the results of various experiments that demonstrate the benefits of TraNNsformer (at algorithm and system level) for DNN acceleration on post-CMOS crossbar-based systems. Note that we have used normalized values to report the area and energy benefits. This is because the benefits from our proposed

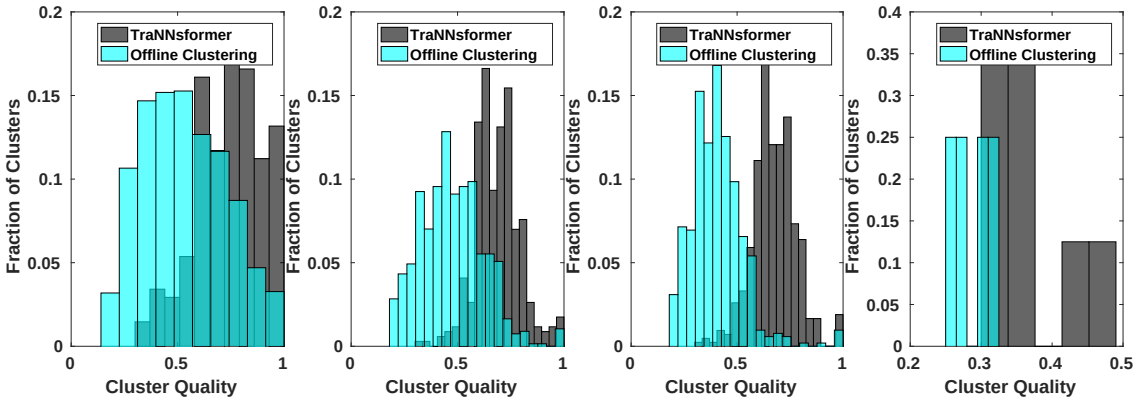


Fig. 4.2. Comparison of crossbar utilization (cluster quality) between Offline Clustering and TraNNsformer

framework are orthogonal to the benefits obtained from any particular choice/design of crossbar-based architecture.

4.5.1 Algorithm-level Analysis

Figure 4.2 shows the fractional distribution of clusters formed with respect to the cluster qualities for two different training approaches namely TraNNsformer and offline clustering on SVHN dataset. In both approaches, DNNs are trained to achieve iso-accuracies. Cluster quality represents the number of non-zero weights present in a cluster. Hence, a cluster with higher cluster quality will map to a crossbar with high utilization factor. Recall, higher crossbar utilization helps to achieve higher area and energy efficiency. Figure 4.2 shows the crossbar utilization for a DNN with 70% average sparsity (across all layers) for offline clustering. Utilization for a DNN trained with weight pruning (pruned DNN) is almost uniform across all crossbars required for mapping. Consequently, the pruned DNN maps to crossbars with 0.3 utilization factor. As shown in Figure 4.2, the pruned DNN upon undergoing offline clustering significantly improves the utilization across all the layers. It can also be seen that TraNNsformer further improves the utilization by a significant factor across all layers

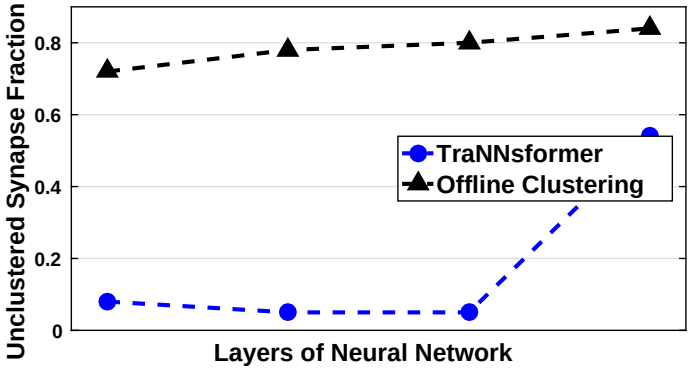


Fig. 4.3. Comparison of fraction of unclustered weights between Offline Clustering and TraNNsformer (the data points correspond to the layers of DNN). Note that the last fully connected layer consists of a small fraction of weights (<1%), thereby having insignificant effect on overall unclustered synapse comparison.

in comparison to the offline clustering approach. This underscores the fact that dynamic cluster formation during the training process helps to get highly structured DNN connectivities in comparison to both pruning and offline clustering.

Figure 4.3 shows that the fraction of unclustered weights remaining after the training process in offline clustering approach is significantly higher than TraNNsformer across all the layers. A large number of unclustered weights is undesirable, as their mapping results in large number of poorly utilized crossbars. However, TraNNsformer leads to much smaller fraction of unclustered weights across all the layers in the DNN (except the last layer).

Both lower fraction of unclustered weights and higher cluster quality are equally important factors in reducing the number of crossbars. A DNN with low cluster quality would map the clustered weights across a large number of crossbars. Similarly, a DNN with higher fraction of unclustered weights consumes a large number of crossbars to map the unclustered weights. Thus, TraNNsformer optimizes both these factors concurrently to reduce the total number of crossbars required. Note that similar results for cluster quality distribution and fraction of unclustered weights were obtained for MNIST and CIFAR10 as well thereby underscoring the scalability ben-

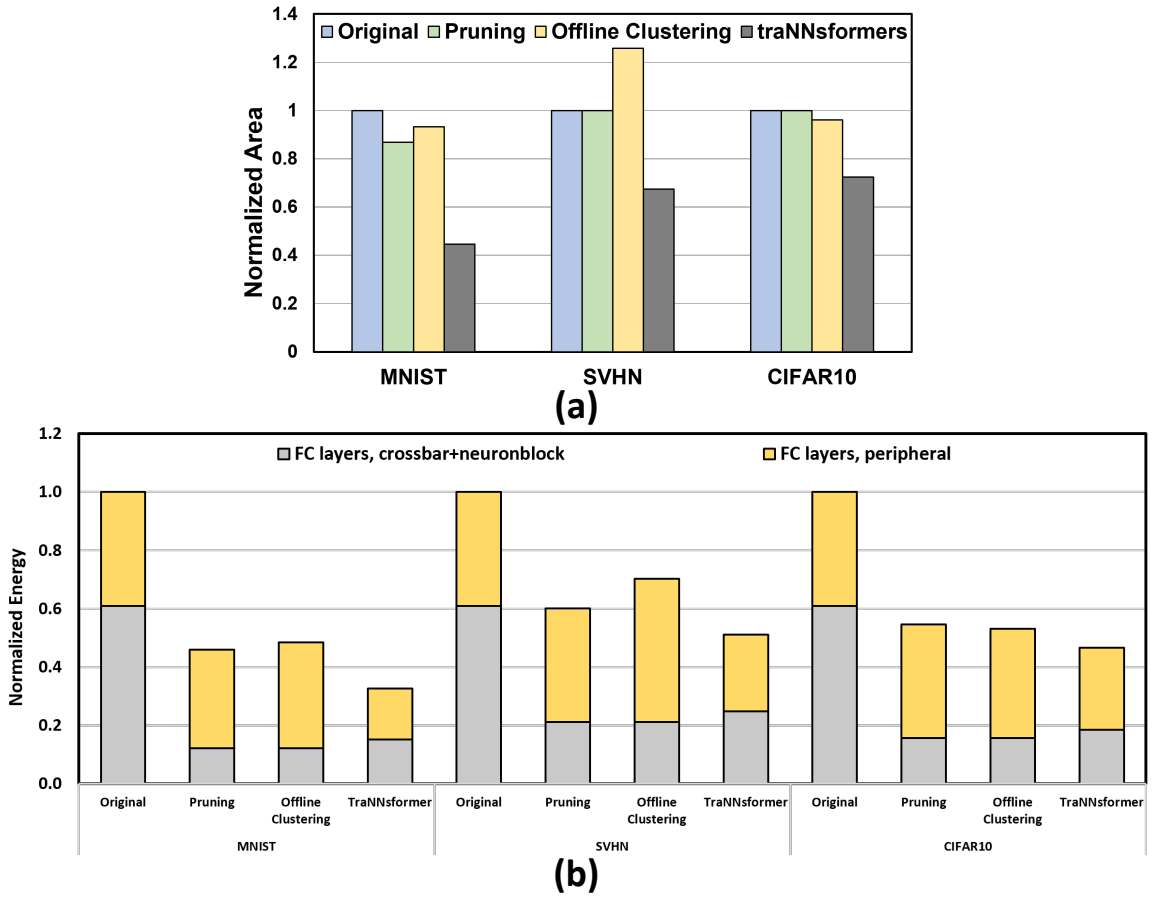


Fig. 4.4. Comparison of area consumption on crossbar based architecture for different DNN training approaches

efits of the proposed training framework with respect to DNN size (number of layers, number of weights in each layer).

4.5.2 Area and Energy Comparisons for Crossbar-based Architecture

Figure 4.4 (a) shows the area consumption on crossbar-based architecture for DNNs with iso-accuracies trained using four different approaches namely 1) Original i.e. typical back-propagation 2) Pruning (weight pruning) 3) Offline clustering and 4) TraNNsformer. The area consumption has been normalized with respect to the area consumption of the original DNN for each dataset. It can be seen that TraNNsformer achieves area savings of $28\% - 55\%$ (39% on average) compared to the original DNN

across all datasets. Furthermore, TraNNsformer also achieves significant area reductions of $28\% - 49\%$ (37% on average) with respect to the pruned DNNs across all datasets. This underscores the effectiveness of the proposed framework in preserving the benefits of sparsity at the hardware level. As mentioned before, DNN trained with weight pruning has irregular (unstructured) sparsity resulting in a large number of crossbars with low utilizations. Consequently, the benefits of weight sparsity does not improve a DNN's area efficiency. It can be seen that the area consumption for DNNs trained with offline clustering is significantly higher than TraNNsformer across all benchmarks. This justifies the importance of a clustering driven DNN training approach towards achieving efficient implementation on crossbar based systems. It is also worth noting that the area consumption in offline clustering case is irregular i.e. lower or comparable to network pruning in some cases (CIFAR10) while being higher in other cases (MNIST and SVHN). This is because the larger fraction of unclustered weights remaining after clustering in SVHN gets mapped to a large number of crossbars with very low utilization factors thereby, worsening the area consumption.

Figure 4.4 (b) shows the energy consumption per inference for the four training approaches. The energy consumption has been normalized with respect to the energy consumption of the original DNN for each dataset. It can be seen that TraNNsformer achieves significant energy improvements of $49\% - 67\%$ (56% on average) across all datasets. Furthermore, it also achieves $15\% - 29\%$ (20% on average) energy reduction with respect to the pruned DNNs across all datasets. As mentioned before, pruning decreases the crossbar energy component only while having minimal effect on the peripheral energy component. However, TraNNsformer based network sparsity translates to commensurate savings for both crossbar as well as peripheral energy components thereby, leading to greater energy savings. It can also be seen that offline clustering approaches have higher energy consumption (MNIST, SVHN) than the pruned DNNs owing to the higher fraction of unclustered weights.

4.6 Conclusions

The intrinsic compatibility of memristive technology with ML has ushered the usage of memristive crossbar based MVMUs to accelerate DNN inference with high efficiency. However, DNNs have multiple static (known before training) connectivity patterns (for instance, CNN, MLP, RNN) which are primarily application dependent. Further, techniques to optimize a DNN by obtaining highly sparse connectivity (network pruning) adds a high degree of dynamic variability (not known before training) to the connectivity pattern. This variability in connectivity pattern requires hardware-aware mapping algorithms to maximize the area and the energy benefits for crossbar-based architectures. While rule based mapping techniques can address the static variability, dynamic connectivity patterns are much more challenging to map owing to the large degree of irregularity. Additionally, an inefficient mapping algorithm prevents the algorithmic benefits of sparsity to be preserved at the hardware level. In this work, we proposed TraNNsformer an integrated training framework that learns connectivity structures, which can be efficiently mapped to crossbars while preserving the algorithmic benefits of weight sparsity. We also developed a technology-aware clustering approach to produce efficient mappings for any crossbar size, permissible by the technology for reliable operations. Our results on a range of image recognition applications suggest that TraNNsformer is a promising framework to implement DNNs, providing a scalable solution to designing large-scale neuromorphic systems.

5. PANTHER - TRAINING ACCELERATOR ARCHITECTURE

5.1 Introduction

Deep Neural Networks (DNNs) have seen wide adoption due to their success in many domains such as image processing, speech recognition, and natural language processing. However, DNN training requires substantial amount of computation and energy which has led to the emergence of numerous special-purpose accelerators [31]. These accelerators have been built using various circuit technologies, including digital CMOS logic [41, 128] as well as hybrid digital-analog logic based on ReRAM crossbars [22, 23].

ReRAM crossbars are circuits composed of non-volatile elements that can perform Matrix-Vector Multiplication (MVM) in the analog domain with low latency and energy consumption. Since MVM operations dominate the performance of DNN inference and training, various inference [22, 23, 26] and training [81, 82] accelerators have been built using these crossbars. However, while inference algorithms do not modify matrices during execution, training algorithms modify them during the weight gradient and update step (weight gradient computation followed by the weight update). For this reason, training accelerators [81, 82] require frequent reads and write to crossbar cells to realize weight gradient and update operations. These reads and writes to ReRAM crossbars are performed one row at a time (like a typical memory array), and are referred to as *serial reads and writes* in this chapter.

Figure 5.1 compares the energy and latency of CMOS and ReRAM technologies for various primitive operations. As shown, MVM consumes $\simeq 10.4\times$ less energy and has $\simeq 8.9\times$ lower latency with ReRAM over CMOS (at same area) for a 32 nm technology node. However, reading and writing the entire matrix consumes much higher energy

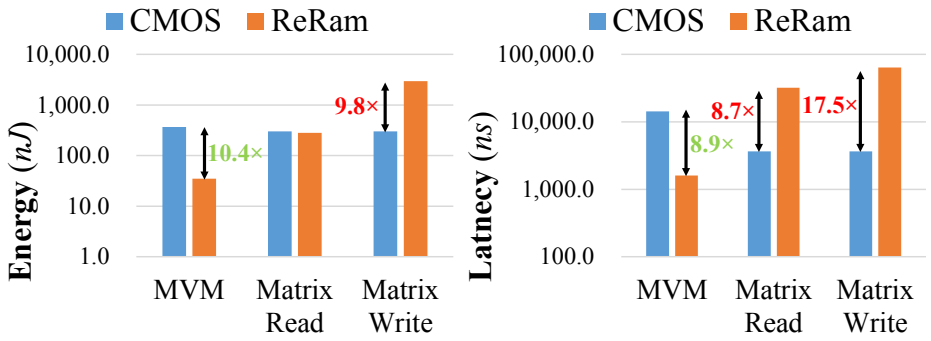


Fig. 5.1. Comparing CMOS and ReRAM Primitives

and latency with ReRAM. Particularly, ReRAM writing energy and latency are an order of magnitude higher due to the cost of the program-verify approach which requires tens of pulses [200]. Therefore, the use of serial reads and writes during training takes away the overall benefits gained from using ReRAM for acceleration.

To overcome this issue, recent demonstrations [117, 118] have shown that Outer Product Accumulate (OPA) operations can be performed in crossbars to realize the weight gradient and update operations without the use of serial reads and writes. The OPA operation is performed by applying two input vectors at the rows and the columns of a crossbar simultaneously, to update each cell depending on the inputs at the corresponding row and column. However, these demonstrations are limited to low-precision inputs/outputs (2-4 bits) and weights (2-5 bits) which is not sufficient for the typical training workloads [141,201]. Moreover, they are confined to Stochastic Gradient Descent (SGD) with batch size of one for fully-connected layers only.

To address these limitations, we propose a bit-slicing technique for achieving higher precision OPA operations by slicing the bits of the output matrix weights across multiple crossbars. While bit-slicing has previously been done for MVM operations [23], bit-slicing matrices to also support OPA operations is substantially different. For MVM, the rows and the crossbar cells are inputs and the columns are outputs, whereas for OPA, the rows and the columns are both inputs and the outputs are the crossbar cells themselves. Moreover, bit-slicing OPA presents additional constraints for the distribution of bits across the slices. First, weights are constant

during MVM, but they change during OPA, which necessitates support for overflow within each slice and accounting for saturation. Second, MVM favors fewer bits per slice to reduce analog-to-digital Converter (ADC) precision requirements [23], but we show that OPA favors more bits per slice. Third, MVM favors homogeneous slicing of bits (equal number of bits per slice), but we show that OPA favors heterogeneous slicing.

We incorporate our proposed technique for enhancing OPA precision into a crossbar architecture that performs both MVM and OPA operations at high precision. We present three variants of the crossbar architecture that are catered to different training algorithms: SGD, mini-batch SGD, and mini-batch SGD with large batches. Using this crossbar architecture, we build PANTHER, a Programmable Architecture for Neural Network Training Harnessing Energy-efficient ReRAM. We use PANTHER to evaluate our design on different layer types (fully-connected, convolutional, etc.) and training algorithms. Our design can also be integrated into existing training accelerators in the literature to enhance their efficiency. Our evaluation shows that PANTHER achieves up to 8.02×, 54.21×, and 2,358× energy reductions as well as 7.16×, 4.02×, and 119× execution time reductions compared to digital accelerators, ReRAM-based accelerators, and GPUs, respectively.

We make the following contributions:

- A bit-slicing technique for implementing high-precision OPA operations using ReRAM crossbars (Section 5.3)
- A crossbar-based architecture, that embodies this bit-slicing technique, with three variants for different training algorithms (Section 5.4)
- An ISA-programmable accelerator with compiler support to evaluate different types of layers in neural networks and training algorithms (Section 5.5)

We begin with a background on the use of ReRAM crossbars for DNN training (Section 5.2).

5.2 Background

5.2.1 Deep Neural Network Training

Typical DNN training comprises of iterative updates to a model's weights in order to optimize the loss based on an objective function. Equations 5.1–5.4 show the steps involved in DNN training based on the Stochastic Gradient Descent (SGD) algorithm [202]. Equation 5.1 constitutes the forward pass which processes an input example to compute the activations at each layer. Equation 5.2 computes the output error and its gradient based on a loss function using the activations of the final layer. Equations 5.3 constitutes the backward pass which propagates the output error to compute the errors at each layer. Finally, equation 5.4 computes the weight updates to minimize the error.

$$H^{(\mathrm{l}+1)} = W^{(\mathrm{l})}\ X^{(\mathrm{l})},\ X^{(\mathrm{l}+1)} = \sigma(H^{(\mathrm{l}+1)}) \tag{5.1}$$

$$E = Loss(X^{(\mathrm{L})}, y),\ \delta H^{(\mathrm{L})} = \nabla E \odot \sigma'(X^{(\mathrm{L})}) \tag{5.2}$$

$$\delta H^{(\mathrm{l})} = [(W^{\mathrm{l}})^{\mathrm{T}}\ \delta H^{(\mathrm{l}+1)}] \odot \sigma'(X^{(\mathrm{l})}) \tag{5.3}$$

$$\frac{\partial E}{\partial W^{\mathrm{l}}}\ (or\ \delta W^{\mathrm{l}}) = X^{(\mathrm{l})}\ (\delta H^{(\mathrm{l}+1)})^{\mathrm{T}},\ W^{\mathrm{l}} = W^{\mathrm{l}} - \eta * \frac{\partial E}{\partial W^{\mathrm{l}}} \tag{5.4}$$

5.2.2 Using Crossbars for Training

The most computationally intensive DNN layers that are typical targets for acceleration are the *fully-connected* layers and the *convolutional* layers. We use fully-connected layers as an example to show how ReRAM crossbars can be used to accelerate DNN training workloads.

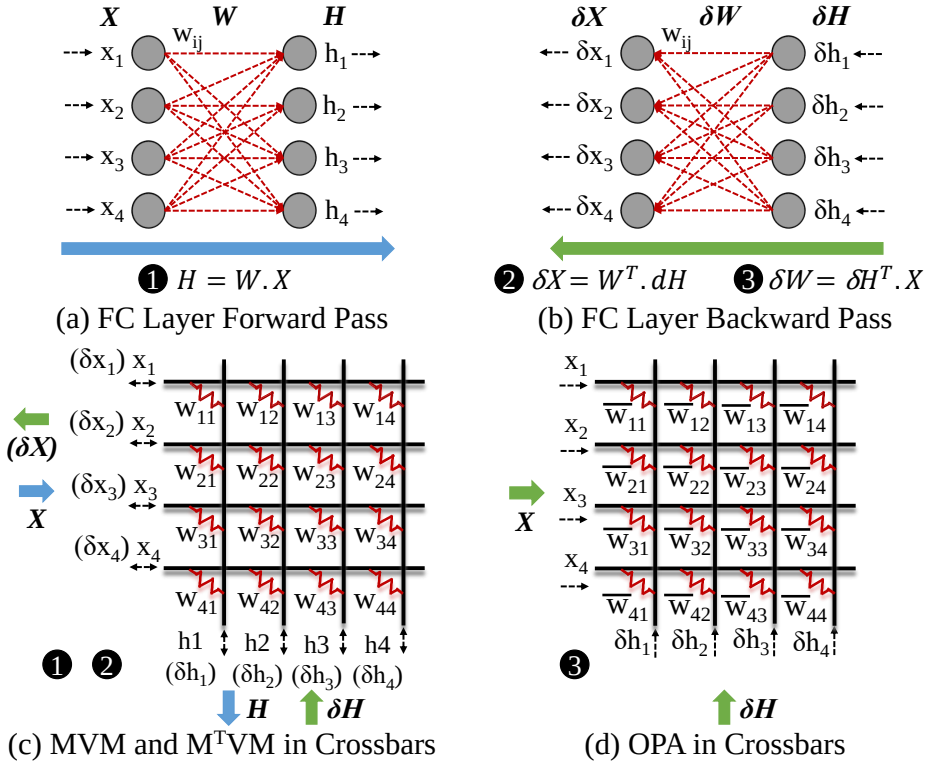


Fig. 5.2. FC Layer Matrix Operations in Crossbars

Overview of Fully Connected (FC) Layers

Figures 5.2(a) and (b) illustrate the operations involved during training in a FC layer. The training involves three types of matrix operations: ❶ *activation*, ❷ *layer gradients*, and ❸ *weight gradients*. Activation corresponds to an MVM operation with the weight matrix (W), as shown in Equation 5.1. Layer gradients correspond to an MVM operation with the transpose of the weight matrix (hereon denoted as M^TVM), as shown in Equation 5.3. Weight gradients correspond to an outer product operation, the result of which is accumulated to the weight matrix based on the learning rate (η), as shown in Equation 5.4. Therefore weight gradients and updates together can be viewed as an Outer Product Accumulate (OPA) operation on the weight matrix.

Activation and Layer Gradients in Crossbars

Figure 5.2(c) shows how a ReRAM crossbar can be used to compute activation and layer gradients. The weights of the matrix (W) are stored in the crossbar cells as the conductance state [16]. The MVM operation is realized by applying the input vector (X) as voltages on the rows of crossbar. Subsequently, the output vector (H) is obtained as currents from the columns. The $\mathrm{M^TVM}$ operation is realized by applying the input vector (δH) as voltages on the columns of the crossbar. Subsequently, the output vector (δX) is obtained as currents from the rows.

Both MVM and $\mathrm{M^TVM}$ operations execute $O(n^2)$ multiply-and-accumulate operations in one computational step in the analog domain (n is the crossbar size). Therefore, ReRAM crossbars can be leveraged to design highly efficient primitives for activation and layer gradient computations. For this reason, they have been extensively considered for DNN inference [22, 23, 26] and training [81, 82] accelerators.

Weight Gradients and Updates in Crossbars

Figure 5.2(d) shows how a ReRAM crossbar can be used to compute weight gradients. The OPA operation can be realized by applying the inputs (X and δH) as voltages on the crossbar's rows and columns, respectively. The change ($\overline{w}_{\mathrm{ij}} - w_{\mathrm{ij}}$) in the value stored at a cross-point (i, j) is equal to the product of the voltage on row i and column j (details in Section 5.3). Therefore, the outer product operation in the crossbar is naturally fused with the weight matrix accumulate operation.

The OPA operation executes $O(n^2)$ multiply-and-accumulate operations in one computational step in the analog domain. It avoids serial reads and writes to ReRAM crossbar cells, which is important because reads and writes have orders of magnitude higher cost (energy and latency) than in-crossbar computations (MVM, $\mathrm{M^TVM}$, OPA). Therefore, ReRAM crossbars can be leveraged to design highly efficient primitives for weight gradient computation and weight update.

The aforementioned technique has been demonstrated with low-precision inputs/outputs (2-4 bits) and weights (2-5 bits) on the SGD training algorithm for FC layers only [117, 118]. In this chapter, we enhance the technique with architecture support to increase its precision and cater to a multiple training algorithms and different layer types.

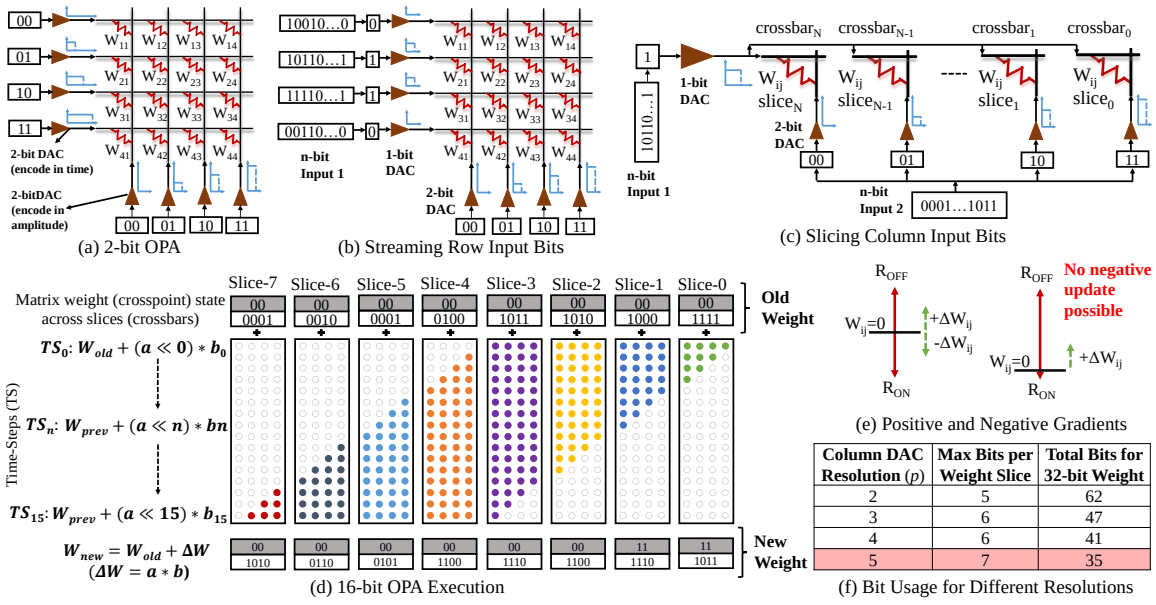
5.3 Enhancing ReRAM-based OPA Precision

DNN workloads require 16 to 32 bits of precision for training [141, 201]. However, input digital-to-analog converters (DACs), crossbar cells, and output ADCs cannot support such levels of precision due to technology limitations and/or energy considerations. For this reason, accelerators that use ReRAM crossbars for MVM/M^{T}VM operations typically achieve the required precision with bit-slicing [23, 26, 80], where matrix bits are sliced across the cells of multiple crossbars, input bits are streamed at the crossbar rows/columns, and shift-and-add logic is used to combine the output bits at each column/row across crossbars (slices).

Bit-slicing matrices to also support OPA operations is different because both the rows and columns are simultaneously applied as inputs and the outputs are the crossbar cells themselves. Moreover, bit-slicing for OPA operations presents additional constraints for the choice of bit distribution across slices. This section describes our technique for bit-slicing the OPA operation (Section 5.3.1), and discusses the constraints it adds to the choice of bit distribution and how we address them (Sections 5.3.2 to 5.3.4).

5.3.1 Bit Slicing the OPA Operation

Figure 5.3(a) illustrates how the OPA operation is performed when 2-bit inputs are applied at the rows and the columns. The digital row input is encoded in the time-domain using pulse-width modulation. The digital column input is encoded in the amplitude-domain using pulse-amplitude modulation. Both pulse-width and pulse-amplitude modulations can be implemented using DACs. The weight change in a cell



Column DAC Resolution (p)	Max Bits per Weight Slice	Total Bits for 32-bit Weight
2	5	62
3	6	47
4	6	41
5	7	35

Fig. 5.3. Bit Slicing OPA to Enhance its Precision

depends on the duration and the amplitude of the pulses applied on the corresponding row and column respectively, thereby realizing an analog OPA operation [117, 118].

To perform an OPA operation with 16-bit inputs, naively increasing the DAC resolution is infeasible because DAC power consumption grows rapidly with resolution (N) as:

$$P_{\text{DAC}} = \beta(2^N/N + 1)V^2 f_{\text{clk}} \text{ [203]} \tag{5.5}$$

Instead, we propose an architectural scheme to realize a 16-bit OPA operation by bit-streaming the row input bits, bit-slicing the column input bits, and bit-slicing the matrix weights across multiple crossbars.

Figure 5.3(b) illustrates how we stream row input bits, m bits at a time over $16/m$ cycles. Meanwhile column input bits are left-shifted by m-bits every cycle. Since the number of cycles decrease linearly with m while the cycle duration increases exponentially with m due to pulse-width modulation of row input, we choose $m = 1$ to minimize total latency. Using $m = 1$ also means that the row DACs are just inverters, thereby having low power consumption.

Figure 5.3(c) shows how we slice column input bits across crossbars. Only one weight W_{ij} is shown for clarity. In each cycle, the left-shifted column input is divided into chunks of p bits ($p = 2$ in this example) and each chunk is applied to the corresponding crossbar.

Figure 5.3(d) illustrates the steps for a 16-bit×16-bit OPA operation at one cross-point in the crossbar, resulting in a 32-bit output value for each matrix weight. It puts together the bit-streaming of the row input vector b and bit-slicing of the column input vector a with $p = 4$. Each dot represents a partial product ($a_{\mathrm{n}}.b_{\mathrm{n}}$), and the color corresponds to a specific weight slice (crossbar). Thus, the net accumulation to a slice is the result of all partial products of the specific color. The updated weight after a time step T_{n} can be expressed as:

$$W_{\mathrm{updated}} = W_{\mathrm{old}} + \sum_{n=0}^{n} (a << n) * b_{\mathrm{n}} \tag{5.6}$$

Crossbars store data in unsigned form. To enable positive and negative weight updates (δW), we represent inputs in the signed magnitude representation. To enable a symmetric representation of positive and negative weight updates, we bias each device such that, a zero weight (W_{ij}) is represented by the memory state ($R_{\mathrm{ON}} + R_{\mathrm{OFF}}$)/2, as shown in Figure 5.3(e). Hence, the signed magnitude computation and biased data representation enable both positive and negative updates to weights. This is important as both polarities of updates are equally important in DNN training. Such a biased-representation can be implemented by adding an extra column per crossbar (128 rows, 128 columns) with minimal area/energy cost [204].

5.3.2 Bits to Handle Overflow

For MVM/M$^{\mathrm{T}}$VM, the matrix weights are inputs to the operation and they do not change. In contrast, for OPA, the matrix weights are accumulated with the values resulting from the outer product. As a result, the weight slice stored in a crossbar cell may overflow, either from multiple accumulations within one OPA or over multiple

OPAs. We handle this overflow by provisioning weight slices with additional bits to store the carry (shaded bits shown in Figure 5.3(d)).

Propagating carry bits to other slices would require serial reads and writes which incur high overhead. For this reason, we do not propagate the carry bits immediately. Instead, they are kept in the slice and participate in future MVM/M^{T}VM and OPA operations on the crossbar.

The carry bits cannot be kept in the weight slice indefinitely because eventually the weight slice may get saturated i.e. crossbar cell at maximum/minimum state for positive/negative update. Saturation is detrimental for *trainability* (desirable loss reduction during training) because it freezes training progress due to the absence of weight change. For this reason, we employ a periodic *Carry Resolution Step* (CRS) which executes infrequently to perform carry propagation using serial reads and writes. We evaluate the impact of the number of bits provisioned per slice and the CRS frequency on saturation and accuracy in Section 5.7.1.

5.3.3 Number of Slices vs. Bits Per Slice

When slicing matrix bits across multiple crossbars, there is a tradeoff between the number of slices and the number of bits per cell in each slice. MVM operations favor using more slices and fewer bits per slice. The reason is that energy increases linearly with the number of crossbars, and non-linearly with the precision of a crossbar due to the increase in ADC precision required to support it. Therefore, using more slices with fewer bits each is better for energy consumption.

In contrast, OPA favors having fewer slices with more bits per slice. The reason is that OPA introduces carry bits to each slice and having more slices with fewer bits each increases the overhead from the carry bits. For example, Figure 5.3(f) shows that with 2 bits per slice, 62 total bits are required to represent the 32-bit weight while capturing the carry bits adequately.

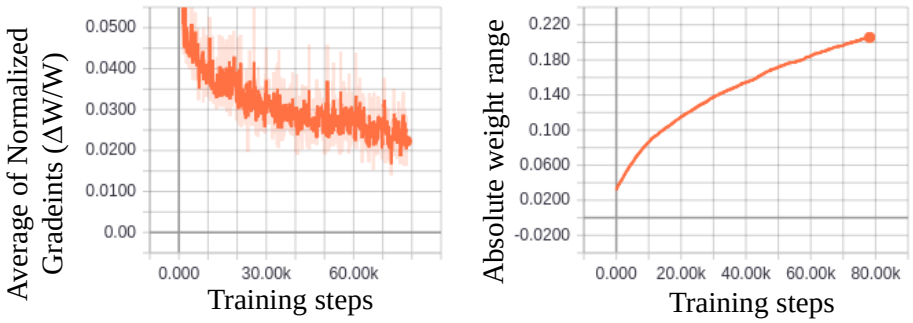


Fig. 5.4. Weight Gradients across Training Steps

To strike a balance, we choose $p = 4$, since $p > 4$ requires a device precision that exceeds ReRAM technology limits [16]. A 4-bit DAC resolution is feasible because DAC power does not increase rapidly at low resolution (Equation 5.5). By choosing $p = 4$, our MVM/$\mathrm{M^TVM}$ operations consume more energy than other ReRAM-based accelerators. However, our more energy efficient OPA operations compensate because they avoid the need for expensive serial reads and writes.

5.3.4 Heterogeneous Weight Slicing

MVM operations favor homogeneous bit-slicing. Increasing the precision of a slice while decreasing the precision of another is always an unfavorable tradeoff because energy increases nonlinearly with the precision of a crossbar. In contrast, for OPA operations where crossbar values change, provisioning more bits for slices that experience more weight updates helps reduce the frequency of saturation, thereby ensuring trainability at low CRS frequency.

Heterogeneous weight slicing provisions more bits for matrix slices that change more frequently. The frequency of change is impacted by two factors: OPA asymmetry and the small weight gradient range in DNNs. OPA asymmetry is illustrated in Figure 5.3(d) where the central slices receive more partial products (dots) than the edge slices, which motivates increasing precision for the central slices. Small weight gradient range is shown in Figure 5.4 where weight updates form a very small fraction

$(2\% - 5\%)$ of the overall weight range for $>= 95\%$ of training steps, which motivates increasing precision of the lower slices. We evaluate the impact of heterogeneous weight slicing on energy and accuracy in Section 5.7.2.

5.4 Matrix Computation Unit (MCU)

The techniques described in Section 5.3 are incorporated into a Matrix Computation Unit (MCU) for DNN training accelerators. This section first describes the MCU's organization (Section 5.4.1). It then describes the three variants of the MCU optimized for SGD (Section 5.4.2), mini-batch SGD (Section 5.4.3), and mini-batch SGD with large batches (Section 5.4.4).

5.4.1 MCU Organization

Figure 5.5 illustrates the organization of the MCU. Performing an MVM operation with the MCU is illustrated by the red arrow. Digital inputs stored in the *XBarIn* registers are fed to the crossbar rows through the *Input Driver*. The output currents from the crossbar columns are then then converted to digital values using *ADC* and stored in the *XBarOut* registers.

Performing a M^TVM operation in the MCU is illustrated by the purple arrow in Figure 5.5. The key difference compared to the MVM operation is the addition of multiplexers to supply inputs to crossbar columns instead of rows and to read outputs from crossbar rows instead of columns.

MVM and M^TVM operations require 16 to 32 bits of precision for training [41]. We use 16-bit fixed-point representation for input/output data and 32-bit fixed-point representation for weight data which ensures sufficient precision [201].

Performing an OPA operation in the MCU is illustrated by the blue arrow in Figure 5.5. Digital inputs stored in the *XBarIn* registers are fed to the crossbar rows through the *Input Driver*. Digital inputs stored in the *XBarOut* registers are fed to the crossbar columns through the *Input Driver*. The effect of this operation is that

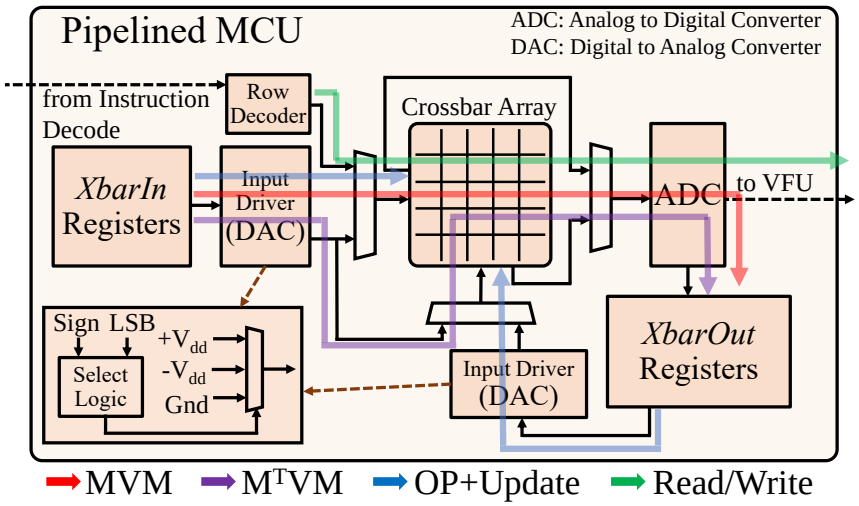


Fig. 5.5. Matrix Computation Unit

the outer product of the input vectors is accumulated to the matrix stored in the crossbar. To support positive and negative inputs, the input drivers in Figure 5.5 use the sign bit (MSB) to drive the crossbar rows and columns with positive or negative voltages.

5.4.2 Variant #1 for SGD Acceleration

SGD-based training performs example-wise gradient descent. First, an input example performs a forward pass (MVM) to generate activations - H^1. Next, the error computed with respect to the activation of the output layer is back propagated (M^TVM) to compute the layer gradients - δX^1. Finally, the activations and layer gradients are used to update (OPA) the weight matrix - W^1, before the next input example is supplied.

Table 5.1 illustrates the logical execution of matrix operations in three MCUs for a three-layer DNN with an input example $a0$. Each time step shows the operations executed on each MCU and their inputs/outputs. For example, at time step 0, MCU0 performs an MVM operation on input $a0$ to compute the output $a1$. The illustration assumes that each layer maps on one MCU and does not show the interleaved nonlin-

Table 5.1.

Dataflow for SGD

Time Step	**MCU0 (Layer1)**	**MCU1 (Layer2)**	**MCU2 (Layer3)**
0	**MVM** (a0) (a1)		
1		**MVM** (a1) (a2)	
2			**MVM** (a2) (a3)
3			$\mathbf{M^TVM}$ (δh3) (δh2)
4		$\mathbf{M^TVM}$ (δh2), (δh1)	**OP** (a2 , δh3) (∇W3)
5	**OP** (a0, δh1) (∇W1)	**OP** (a1, δh2) (∇W2)	

ear operations for clarity. For a layer size larger than one MCU capacity (128×128 matrix), the layer is partitioned across multiple MCUs (see Section **??**).

Variant #1 of the MCU uses a single crossbar to perform all three matrix operations: MVM, M^TVM, and OPA. This variant is suitable for SGD because, as shown in Table 5.1, the three matrix operations are data dependent and will never execute concurrently. However, this variant creates structural hazards for mini-batch SGD as described in Section 5.4.3.

5.4.3 Variant #2 for Mini-Batch SGD Acceleration

Mini-batch SGD performs batch-wise gradient descent. Like SGD, each input performs MVM, M^TVM, and OPA to compute activations, layer gradients, and weight gradients/updates, respectively. However, the weight update is only reflected at the end of a batch to be used by the inputs of the next batch.

Table 5.2 illustrates the logical execution of matrix operations for a batch of five inputs, where a_nm refers to the m^{th} activation of n^{th} input. MVM operations can be executed for multiple input examples concurrently in a pipelined fashion (**MVM** ($a_1$0) ($a_1$1), **MVM** ($a_0$1) ($a_0$2) in Table 5.2). Additionally, the MVM and M^TVM operations for different inputs in the batch can also execute in parallel during the

Table 5.2.
Dataflow for Mini-Batch SGD

Time Step	MCU0 (Layer1)	MCU1 (Layer2)	MCU2 (Layer3)
0	**MVM** ($a_0$0) ($a_0$1)		
1	**MVM** ($a_1$0) ($a_1$1)	**MVM** ($a_0$1) ($a_0$2)	
2	**MVM** ($a_2$0) ($a_2$1)	**MVM** ($a_1$1) ($a_1$2)	**MVM** ($a_0$2) ($a_0$3)
3	**MVM** ($a_3$0) ($a_3$1)	**MVM** ($a_2$1) ($a_2$2)	**MVM** ($a_1$2) ($a_1$3)
			M^{T}VM ($\delta h_0$3) ($\delta h_0$2)
4	**MVM** ($a_4$0) ($a_4$1)	**MVM** ($a_3$1) ($a_3$2)	**MVM** ($a_2$2) ($a_2$3)
		M^{T}VM ($\delta h_0$2), ($\delta h_0$1)	**M^{T}VM** ($\delta h_1$3) ($\delta h_1$2)
5		**MVM** ($a_4$1) ($a_4$2)	**MVM** ($a_3$2) ($a_3$3)
		M^{T}VM ($\delta h_1$2), ($\delta h_1$1)	**M^{T}VM** ($\delta h_2$3) ($\delta h_2$2)
6			**MVM** ($a_4$2) ($a_4$3)
		M^{T}VM ($\delta h_2$2), ($\delta h_2$1)	**M^{T}VM** ($\delta h_3$3) ($\delta h_3$2)
7			
		M^{T}VM ($\delta h_3$2), ($\delta h_3$1)	**M^{T}VM** ($\delta h_4$3) ($\delta h_4$2)
8			
		M^{T}VM ($\delta h_4$2), ($\delta h_4$1)	
9-12	**OP** (a_n0, δh_n1) (∇W_n1)	**OP** (a_n1, δh_n2) (∇W_n2)	**OP** (a_n2, δh_n3) (∇W_n3)
	Iterate for n=1 to 4		

same timestep, provided that there is no structural hazard on the MCU. The desire to eliminate such structural hazards motivates Variant #2.

Variant #2 of the MCU eliminates structural hazards in mini-batch SGD by storing two copies of the matrix on different crossbars, enabling the MCU to perform MVM and M^TVM in parallel. This replication improves the *energy-delay product* for a batch. With $< 2\times$ increase in area, we improve the batch latency by $O(L)$, where L is the number of layers. The ISA instruction for performing MVM/M^TVM (Section 5.5.2) is designed to enable the compiler (Section 5.5.3) to schedule these two operations in parallel on the same MCU.

The OPA operations are executed at the end of the mini-batch (steps 9-12 in Table 5.2) to reflect the weight updates for the entire batch. These OPA operations require that the vectors involved are saved until then. Variant #2 saves these vectors in shared memory. However, if the batches are large, this approach puts too much stress on the shared memory which motivates Vaiant #3 (Section 5.4.4).

5.4.4 Variant #3 for Mini-Batch SGD with Large Batches

For mini-batch SGD with very large batch sizes, saving the vectors in shared memory requires large shared memory size which degrades storage density. Variant #3 alleviates the pressure shared memory size by maintaining three copies of each crossbar. The first two copies enable performing MVM and $M^{T}VM$ in parallel, similar to Variant #2. The third copy is used to perform the OPA operation eagerly, as soon as its vector operands are available, without changing the matrices being used by the MVM and $M^{T}VM$ operations.

Performing OPA eagerly avoids saving vectors until the end, reducing the pressure on the shared memory. However, using a third crossbar for OPA requires serial reads and writes to commit the weight updates to the first and the second crossbars for MVM and $M^{T}VM$ in the next batch. Section 5.7.6 discusses the impact of these design choices.

5.5 Programmable Accelerator

The MCU described in Section 5.4 can be integrated with prior ReRAM-based training accelerators [81, 82] to improve their efficiency. We develop a programmable training accelerator named PANTHERto evaluate our design by extending the PUMA ReRAM-based inference accelerator [26]. This section describes PANTHER's organization (Section 5.5.1), ISA considerations (Section 5.5.2), compiler support (Section 5.5.3), and an example of how to implement convolutional layers (Section 5.5.4).

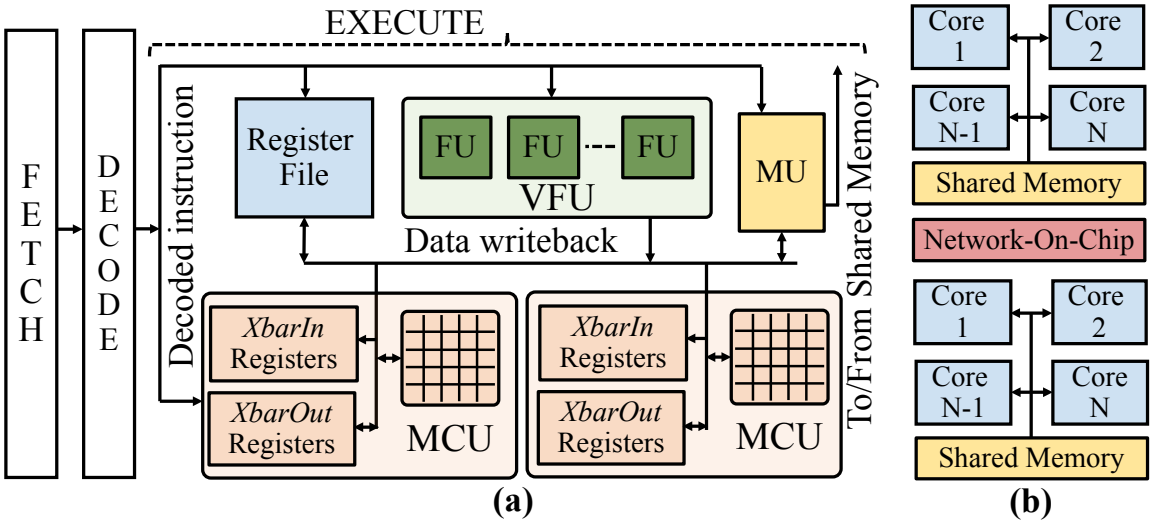


Fig. 5.6. Architecture Overview

5.5.1 Accelerator Organization

PANTHERis a spatial architecture organized in three tiers: nodes, tiles, and cores. A node consists of multiple tiles connected via an on-chip network, and a tile consists of multiple cores connected to a shared memory, as illustrated in Figure 5.6(b). A core consists of multiple MCUs for executing matrix operations, a digital CMOS-based vector functional unit (VFU) for executing arithmetic operations and non-linear functions, a register file, and a load/store memory unit. A core also features an instruction execution pipeline making the accelerator ISA-programmable. To support DNNs whose model storage exceeds a node's total MCU capacity, multiple nodes can be connected via an interconnect. This organization is similar to PUMA's [26] and is not a contribution of this chapter. The key distinction from PUMA is the MCU which supports $M^{T}VM$ and OPA operations, not just MVM operations, as described in Section 5.4.

5.5.2 ISA Considerations

The PUMA [26] ISA includes *mvm* instructions executed by crossbars, arithmetic/logic/nonlinear instructions executed by the VFU, load/store instructions to access shared memory, send/receive instructions to communicate with other tiles, and control flow instructions. We extend the PUMA ISA to also include a *mcu* instruction for executing all three matrix operations (MVM, M^TVM, OPA) on the MCU.

The *mcu* instruction takes six 3-bit masks, where each mask corresponds to one of the MCUs on the core (up to six). The three bits in the mask correspond to the three supported matrix operations (MVM, M^TVM, OPA). If multiple bits are set, then the instruction executes the operations concurrently. For example, if mask 0 is set to '110' and mask 1 is set to '011', then MCU 0 will execute MVM and M^TVM simultaneously and MCU 1 will execute M^TVM and OPA simultaneously. The incorporation of all three operations into a single instruction is important for being able to execute them concurrently in order to leverage the parallelism in batch-wise training on Variant #2 (Section 5.4.3) and Variant #3 (Section 5.4.4). Furthermore, having separate masks for each MCU within a core helps leverage the parallelism across matrix operations [26]. The mask is generated by fusing different MCU operations as discussed in Section 5.5.3. The *mcu* instruction does not take source and destination operands since these are implied to by *XBarIn* and *XBarOut*.

The semantic of the OPA operation is that it takes effect at the end of the execution when a special *halt* instruction is invoked. This semantic allows the same code to work for any of the three MCU variants, making the choice of variant a microarchitectural consideration and the ISA agnostic to it. The implementation of the OPA semantic on each of the variants is as follows. Consider the case when all three bits of an MCU's mask are set. In Variant #1, MVM and M^TVM will be serialized on the same crossbar, while the operands of OPA will be saved to shared memory then applied to that crossbar when *halt* is invoked. In Variant #2, MVM and M^TVM will be executed in parallel on the two crossbar copies, while the operands of OPA will

be treated like in Variant #1. In Variant #3, MVM and M^TVM will be executed in parallel on the first two crossbar copies, while the operands of OPA will be applied to the third crossbar. The values of the third crossbar will then be copied to the first two crossbars when *halt* is invoked.

5.5.3 Compiler Support

The PUMA [26] compiler provides a high-level programming interface in C++ that allows programmers to express models in terms of generic matrix and vector operations. The compiler is implemented as a runtime library that builds a computational graph when the code is executed then compiles the graph to PUMA ISA code. The compiler partitions matrices into sub-matrices and maps these sub-matrices to different MCUs, cores, and tiles. It then maps the operations in the graph to different MCUs, cores, and tiles accordingly, inserting communication operations where necessary. The compiler then linearizes the graph, creating an instruction sequence for each core. It performs register allocation for each sequence, spilling registers to shared memory if necessary. Finally, it generates ISA code for each core, collectively comprising a kernel that runs on the accelerator.

We make the following extensions to the PUMA compiler to support PANTHER. We extend the application programming interface (API) to allow programmers to define training matrices that support MVM, M^TVM, and OPA operations. We extend the intermediate representation to represent these matrices and include them in the partitioning. We also add an analysis and transformation pass for identifying MCU operations in the graph that can be fused and fusing them. This pass fuses MCU operations that do not have data dependences between them and that use different MCUs on the same core or use the same MCU but are different types of operations (MVM, M^TVM, OPA). The fusing process is iterative because every time operations are fused, new dependences are introduced to the graph. Finally, we extend the code generator to support the new *mcu* ISA instruction.

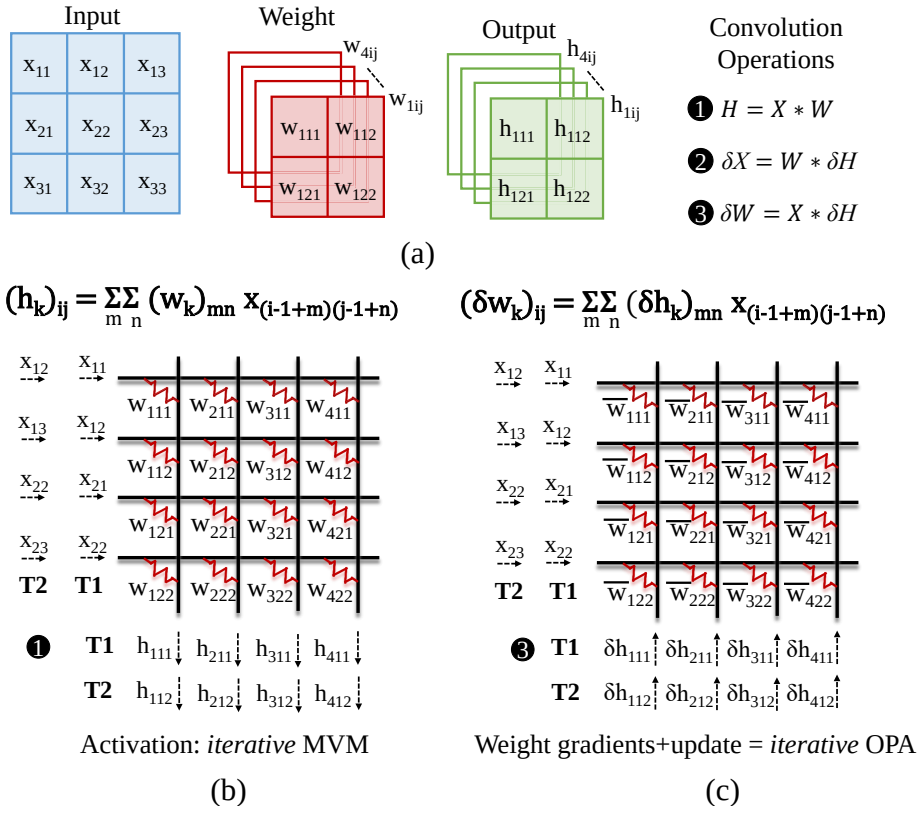


Fig. 5.7. Convolutional Layer Matrix Operations in Crossbars

Note that since the model weights are not updated until the *halt* instruction at the end, the scope of a kernel is a single batch. Multiple batches are executed by invoking the kernel multiple times on different input data.

5.5.4 Implementing Convolutional Layers

ReRAM-based OPA has one-to-one correspondence to the weight gradient/update operation for FC layers (discussed in Section 5.2.2). By integrating this technique into a programmable accelerator with compiler support, we enable the mapping of more complex layers on top of it such as convolutional layers. This section describes how convolutional layers can be implemented in our accelerator.

Figure 5.7(a) shows a typical convolution layer and the associated operations during training. Like with FC layers, convolutional layers performs three types of

matrix operations: ❶ *activation*, ❷ *layer gradients*, and ❸ *weight gradients*. Unlike FC layers, these operations are all convolutions ($*$).

Activation and Layer Gradients

Figure 5.7(b) shows how the convolution operation for activation is implemented in the crossbar on top of the MVM primitive. This approach is similar to that used in existing accelerators [81]. The crossbar stores the *convolution kernel* in the form of linearized filters (w_{k}), where each column corresponds to the weights associated with a specific output channel (h_{k}). The convolution operation to compute activations is implemented as an *iterative* MVM operation. An iteration is represented as a time step (T1/T2) in Figure 5.7(b), and corresponds to a specific (i,j) pair. A block of input features (X) is applied to the crossbar's rows as *convolution data* in each iteration. In a similar manner, the convolution operation for layer gradients (not shown in the figure) is realized using iterative $\mathrm{M^TVM}$. The next layer's errors (δH) are used as the convolution data and flipped filters (vertically and horizontally) are used as the convolution kernel.

Weight Gradients

Figure 5.7(c), shows our proposed technique for implementing the weight gradients convolution operation and weight update in the crossbar on top of the OPA primitive. The weight gradient computation uses input features (X) as the convolution data and output feature's errors (δH) as the convolution kernel. Each iteration is represented as a time step (T1, T2) in Figure 5.7(c), and corresponds to a specific (i,j) pair. On every iteration, the output feature's errors are applied on the columns, in a depth major order. Simultaneously, by applying the portion of input features that generate the corresponding activations (H) on the rows, a *partial convolution* is obtained between X and δH. Striding across the output feature's errors and input features for n^2 time steps, where n is size of one output feature map, realizes the full convolution

operation. Convolutions for different output feature maps are performed in parallel across the crossbar's columns, using the same weight data layout as used in MVM and M^TVM operations. To the best of our knowledge, our work is the first to formulate the weight gradients convolution operation in terms of outer products.

Comparison with Other Accelerators

Existing ReRAM-based training accelerators such as PipeLayer [81] do not compute the weight gradient convolutions using outer products, but rather, they compute them using MVM operations. This requires writing the convolution kernel (δH) on the crossbar because the convolution operation here uses *non-stationary* data (δH) as the convolution kernel. The drawback of this approach is that the latency and energy consumption of the serial reads and writes is very high, taking away from the overall efficiency provided by ReRAM-based MVMs.

5.6 Methodology

5.6.1 Architecture Simulator

We extend the PUMA [26] simulator to model the MCU unit and its associated instructions. The PUMA simulator is a detailed cycle-level architecture simulator that runs applications compiled by the compiler, in order to evaluate the execution of benchmarks. The simulator models all the necessary events that occur in an execution cycle, including compute, memory and NoC transactions. To estimate power and timing of the CMOS digital logic components, their RTL implementations are synthesized to the IBM 32nm SOI technology library, and evaluated using the Synopsys Design Compiler. For the on-chip SRAM memories, the power and timing estimates are obtained from Cacti 6.0. Subsequently, the power and timing of each component are incorporated in the cycle-level simulator in order to estimate the energy consumption.

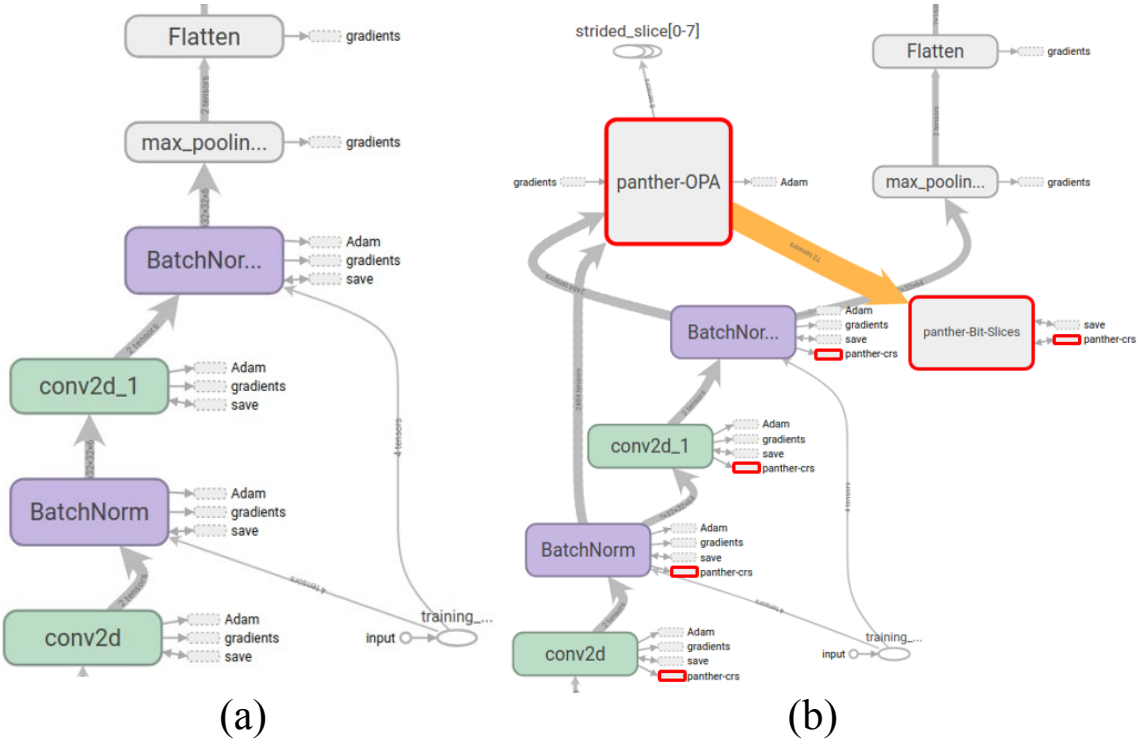


Fig. 5.8. Computational graph obtained using TensorBoard for (a) example model (b) example model with PANTHER OPA

MCU Modelling. Since the MCU is built with analog components and cannot be synthesized with publicly available libraries, we adopted the models from past works [23,117] and ADC survey [185]. We use the ReRam crossbar array and sample-and-hold circuit models in ISAAC [23]. We used capacitive DACs and Successive Approximation Register (SAR) ADCs. The DAC area and power are estimated using the equations described in Saberi et al. [205]. The ADCs for different precisions namely 8-12 bits operating at a sampling frequency of $1GHz$ are obtained from the ADC survey [185]. The ADC optimization technique in Newton [206] is incorporated to avoid unnecessary ADC conversions.

Table 5.3.
Summary of platforms

Parameter	PANTHER (1 node)	$Base_{digital}$ (1 node)	2080-Ti (1 card)
SIMD lanes	108 M	108 M	4352
Technology	CMOS-ReRam (32 nm)	CMOS (32 nm)	CMOS (12 nm)
Frequency	1 GHz	1 GHz	1.5 GHz
Area	117 mm^2	578 mm^2	750 mm^2
TDP	105 W	839 W	250 W
On-Chip Memory	72.4 MB	72.4 MB	29.5 MB

5.6.2 Functional Simulator

We implement a functional simulator using TensorFlow that models PANTHER's bit-sliced OPA technique. This simulator enables performing design space exploration (for accuracy) on large-scale DNNs to explore the bounds on heterogeneous weight slicing and CRS frequency for trainability. Here, a layer's weights are represented as a multi-dimensional tensor of shape $S \times M \times N$, where S corresponds to a weight slice (discussed in Figure 5.3 (d)), and M and N correspond to the weight matrix's dimensions respectively. Each weight slice can have a unique bit-precision, to model heterogeneous configurations (Section 5.3.4). The weight values beyond the range permissible by the bit-precision are clipped to model a slice's saturation. Subsequently, the weight update operation in native TensorFlow is modified to quantize and bit-slice the computed weight gradients and then update the previous copy of weights (already quantized and bit-sliced). Figures 5.8 (a) and (b) show the computational graphs for an example neural network model, and the example model augmented with PANTHER OPA operation (shown in red) respectively.

5.6.3 Baselines

We evaluate PANTHER against three weight-stationary ASIC baselines: $Base_{digital}$, $Base_{mvm}$, and $Base_{opa/mvm}$, as well as one NVIDIA GPU platform - Turing RTX 2080-Ti (2080-Ti).

$Base_{digital}$ uses a digital version of the MCU where weights are stored in an SRAM array within the core and matrix operations are performed with a digital VFU. $Base_{digital}$ is an adaptation of the digital baseline used in PUMA [26]. As shown in the PUMA work, this digital baseline is an optimistic estimate of the Google TPU [128]. It is optimistic because it uses weight-stationary MVM computations similar to TPU, but assumes that the entire model is mapped using on-chip SRAM, thereby avoiding the off-chip memory access costs in TPU. Therefore, our comparisons with $Base_{digital}$ also serve as a lower-bound on PANTHER's improvements compared to TPU. The objective of comparing with $Base_{digital}$ is to demonstrate the benefit of ReRAM-based computing over pure digital approaches.

$Base_{mvm}$ uses ReRAM for MVM and M^TVM, and a digital VFU for OPA with serial reads/writes to the crossbar. $Base_{opa/mvm}$ is a replication of PipeLayer's [81] approach described in Section 5.5.4 and only applies to convolutional layers. It uses ReRAM for MVM and M^TVM, and realizes OPA with ReRAM MVMs and serial reads/writes. The objective of comparing with $Base_{mvm}$ and $Base_{opa/mvm}$ is to demonstrate the benefit of ReRAM-based OPA operations.

Configurations. $Base_{mvm}$ and $Base_{opa/mvm}$ use 32-bit weights sliced across 16 slices with 2 bits each, which is optimal since crossbars only do MVM/M^TVM. PANTHER uses heterogeneous weight slicing with 32-bit weights represented using 39 bits sliced across 8 slices distributed from MSB to LSB like so: 44466555 (unless otherwise specified). For this reason, PANTHER consumes 17.5% higher energy for MVM/M^TVM than $Base_{mvm}$ and $Base_{opa/mvm}$ due to higher ADC precision. We also use a CRS frequency of 1024 steps (unless otherwise specified) which achieves similar accuracy as the software implementation. For all three ASIC baselines and PANTHER, the hierarchical organization uses 138 tiles per node, with 8 cores per tile and 2 MCUs per core. Table 5.3 summarizes the platforms. Note that both $Base_{mvm}$ and $Base_{opa/mvm}$ have same platform parameters as PANTHER.

Table 5.4.

Details of workloads

Layer	C	M	H/W	R/S	E/F	Wt (MB)	In (MB)	Ops/B
CNN-Vgg16								
Conv1	3	64	32	3	32	0.003	0.006	368.640
Conv2	32	64	32	3	16	0.035	0.063	92.160
Conv3	64	128	16	3	16	0.141	0.031	209.455
Conv4	128	128	16	3	8	0.281	0.063	52.364
Conv5	128	256	8	3	8	0.563	0.016	62.270
Conv6	256	256	8	3	8	1.125	0.031	62.270
Conv7	256	256	8	3	4	1.125	0.031	15.568
Conv8	256	512	4	3	4	2.250	0.008	15.945
Conv9	512	512	4	3	4	4.500	0.016	15.945
Conv10	512	512	4	3	2	4.500	0.016	3.986
Conv11	512	512	2	3	2	4.500	0.004	3.997
Conv12	512	512	2	3	2	4.500	0.004	3.997
Conv13	512	512	2	3	1	4.500	0.004	0.999
Dense14	512	4096	-	-	-	4.000	0.001	1.000
Dense15	4096	4096	-	-	-	32.000	0.008	1.000
Dense16	4096	100	-	-	-	0.781	0.008	0.990
MLP-L4								
Dense1	1024	256	-	-	-	0.500	0.002	0.996
Dense2	256	512	-	-	-	0.250	0.000	0.998
Dense3	512	512	-	-	-	0.500	0.001	0.998
Dense4	512	10	-	-	-	0.010	0.001	0.909

5.6.4 Workloads

We use a 4-layered MLP model and Vgg-16 CNN model on SVHN and CIFAR-100 datasets, respectively. Table 5.4 details the layer details of the two models and their computational intensity (operations to byte ratio). The individual layers of the chosen MLP and CNN models span a wide range of computational intensity observed across the spectrum of neural network workloads. Thus, our workloads are well representative of the large variety of layer types found in neural network models such as fully-connected, 2 D-convolution, point-wise convolution, etc.

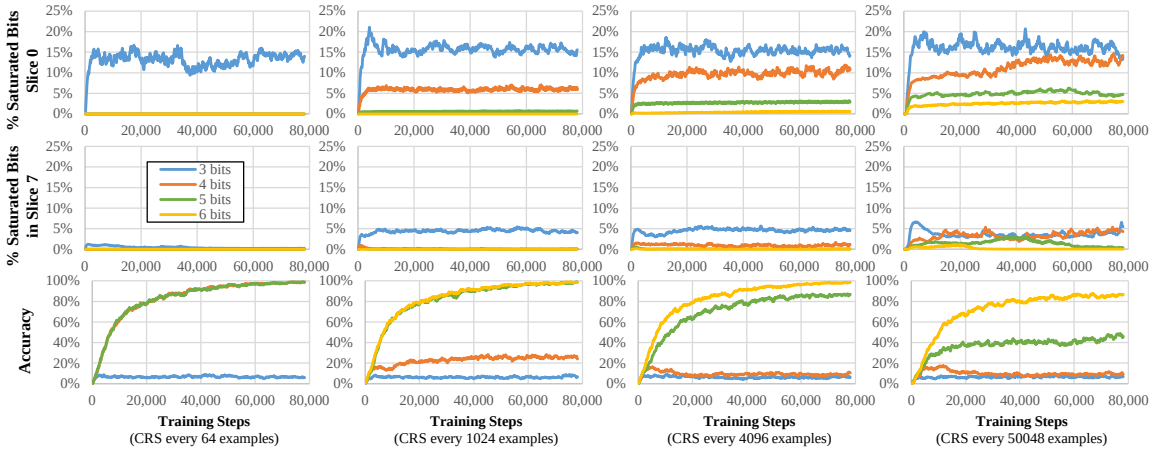


Fig. 5.9. Impact of Slice Bits and CRS Frequency on Accuracy

Similar to other ReRAM training accelerators [81, 82], we use fixed-point arithmetic which has been shown to be successful for training large DNNs [141]. We use the CIFAR-100 dataset for CNN which is comparable to the ImageNet dataset in terms of training difficulty [207, 208]. However, ImageNet's large image sizes make it difficult to run the training flow without actual hardware (CIFAR-100 requires 2 days and ImageNet requires 1 month on the simulator).

5.7 Evaluation

5.7.1 Impact of Slice Bits and CRS Frequency on Accuracy

Figure 5.9 shows the impact of the number of bits used per slice (uniform weight slicing) and CRS frequency for the CNN benchmark. We analyze the percentage of saturated cells per slice for a lower order and higher order slice, and their implications on CNN's Top-5 training accuracy.

Using 3 bits per slice shows significantly higher percentage of saturated cells for the lower order slice (Slice 0) than other configurations. Further, increasing the CRS frequency does not reduce the saturation fraction of Slice 0 at 3-bits. Consequently,

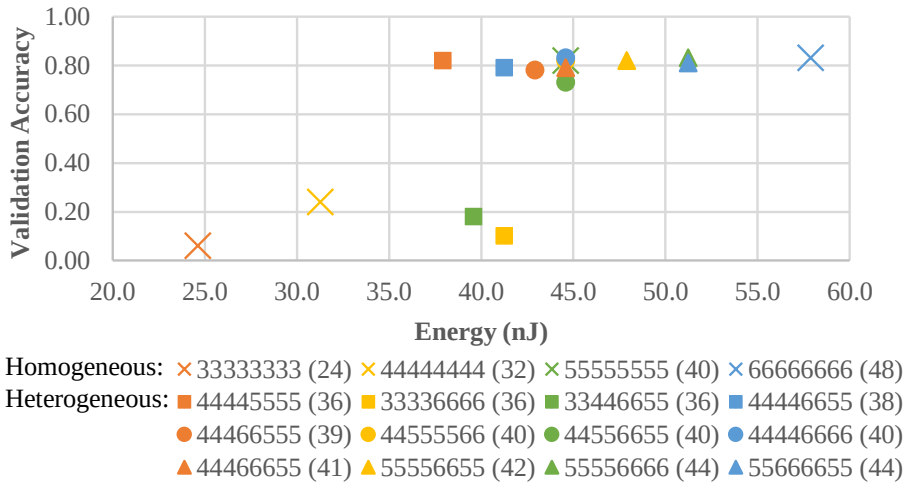


Fig. 5.10. Heterogeneous Weight-Slicing

the training accuracy with 3-bits slices remains very low throughout the training steps.

Using 4 bits per slice performs well at high CRS frequency (CRS every 64 steps), but does not scale well at lower CRS frequencies. A high CRS frequency is undesirable due to the high cost of serial reads and writes incurred during carry propagation between discrete slices.

Slices with 5-bits and 6-bits are robust to repeated weight updates as they exhibit lower saturation for both lower order and higher order slices even at low CRS frequencies (every 1024 or 4096 steps). Note that the cost of a CRS operation at low frequency (every 1024 steps) has negligible impact on overall energy and performance ($\leq 4.8\%$).

Figure 5.9 also motivates heterogeneous weight slicing because it shows that the higher order slice has significantly lower saturation in general than the lower order slice.

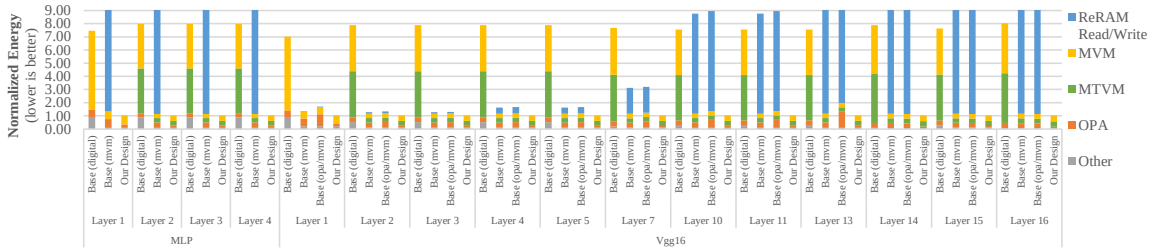


Fig. 5.11. SGD Energy (high bars are clipped)

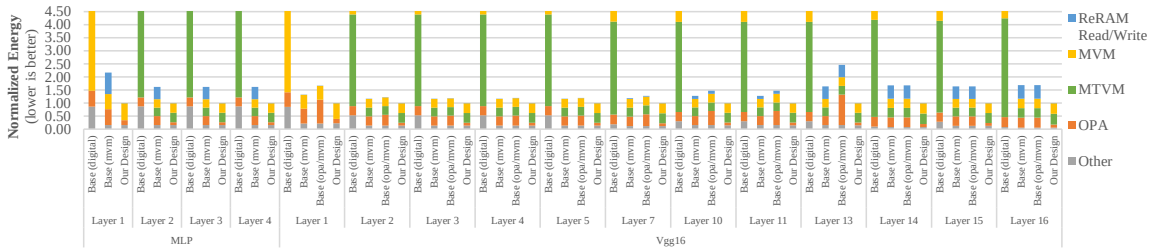


Fig. 5.12. Mini-batch SGD Energy (high bars are clipped)

5.7.2 Impact of Heterogeneous Weight Slicing

Figure 5.10 shows the accuracy and energy of sixteen slicing configurations. Generally speaking, increasing the total number of bits improves accuracy by reducing saturation, but it also increases energy because it requires higher precision ADCs for MVM and $M^{T}VM$. The graph shows that heterogeneous weight slicing enables favourable accuracy-energy tradeoffs, enabling lower energy at comparable accuracy or better accuracy at comparable energy. Provisioning ≥ 4 bits for the four higher order slices $(4-7)$ and ≥ 5 bits for the four lower order slices $(0-3)$ ensures desirable accuracy. Any configuration using 3 bit slices (irrespective of total bits) leads to significant accuracy degradation. Note that the configuration used in the rest of the evaluation (44466555) is not a Pareto-optimal one, so our energy numbers in the rest of the evaluation are underestimated.

5.7.3 Variant #1 SGD Energy Comparison

Figure 5.11 compares the layer-wise energy consumption of PANTHER's Variant #1 to that of all three baselines for SGD.

Base$_{digital}$. Compared to $Base_{digital}$, we achieve 7.01×–8.02× reduction in energy. This advantage is due to the energy efficiency of computing MVM, $M^{T}VM$, and OPA in ReRAM.

Base$_{mvm}$. Compared to $Base_{mvm}$, we achieve 31.03×–54.21× reductions in energy for FC layers (Layers 1-4 in MLP and 14-16 in CNN) and 1.47×–31.56× for convolution layers (Layers 1-13), with the later (smaller) convolution layers showing larger reductions. Recall that $Base_{mvm}$ uses serial reads and writes to perform the OPA operation with digital logic. While the large convolutional layers can amortize these reads and writes, the FC layers and small convolutional layers do not have enough work to do so which is why they suffer relatively. In contrast, PANTHER avoids these reads and writes by performing OPA in the crossbar (11.37 nJ).

Base$_{opa/mvm}$. $Base_{opa/mvm}$ behaves similarly to $Base_{mvm}$. Recall that both baselines perform serial reads and writes to crossbars for OPA, but $Base_{mvm}$ uses CMOS VFUs while $Base_{opa/mvm}$ uses ReRAM MVMs. Since ReRAM MVMs and CMOS OPAs have comparable energy consumption (35.10 nJ and 37.28 nJ respectively), the overall energy of the two baselines is similar.

5.7.4 Variant #2 Mini-Batch SGD Energy

Figure 5.12 compares the layer-wise energy consumption of Variant #2 of PANTHER to that of all three baselines for Mini-Batch SGD with batch size 64. Compared to SGD results (Figure 5.11), the key difference is that having multiple batches before weight updates amortizes the cost of serial reads and writes in $Base_{mvm}$ and $Base_{opa/mvm}$ (smaller blue bar). Our energy improvements therefore come mainly from reducing OPA energy. Energy is reduced by 1.61×–2.16× for fully connected

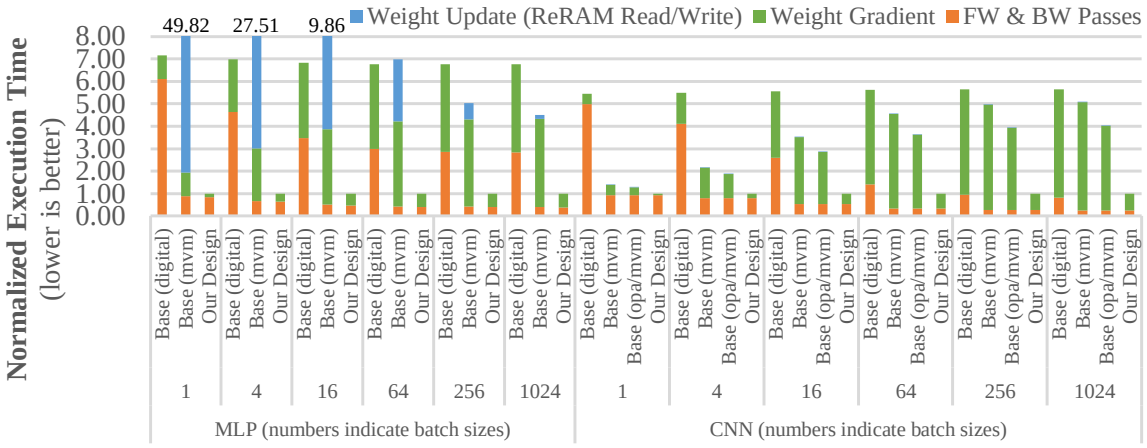


Fig. 5.13. Execution Time

layers for $Base_{mvm}$ and $Base_{opa/mvm}$. It is reduced by 1.18×–1.63× and 1.22×–2.45× for convolutional layers for $Base_{mvm}$ and $Base_{opa/mvm}$, respectively.

For very large batch sizes such as 1,024 (not shown in the figure), ReRAM writes can be completely amortized by $Base_{mvm}$ and $Base_{opa/mvm}$. In this case, PANTHER reduces energy by $\simeq$ 1.18× compared to $Base_{mvm}$ and $Base_{opa/mvm}$ due to reducing OPA energy. However, batch sizes preferred by ML practitioners for DNN training (32, 64) are typically smaller than what is required to amortize the ReRAM memory access costs because large batch sizes have adverse effects on DNN generalization [209].

5.7.5 Variant #2 Execution Time

Figure 5.13 compares the layer-wise execution time of Variant #2 to all three baselines for different batch sizes.

$Base_{digital}$. Compared to $Base_{digital}$, we have consistently lower execution time due to faster MVM, $M^{T}VM$, and OPA operations in ReRAM.

$Base_{mvm}$. For MLPs with small batch sizes, $Base_{mvm}$ significantly suffers because the ReRAM write latency is not amortized. However, for larger batch sizes and for CNNs, the ReRAM write latency is amortized. Nevertheless, we still outperform $Base_{mvm}$ across all batch sizes because of lower latency ReRAM OPA. In fact, our

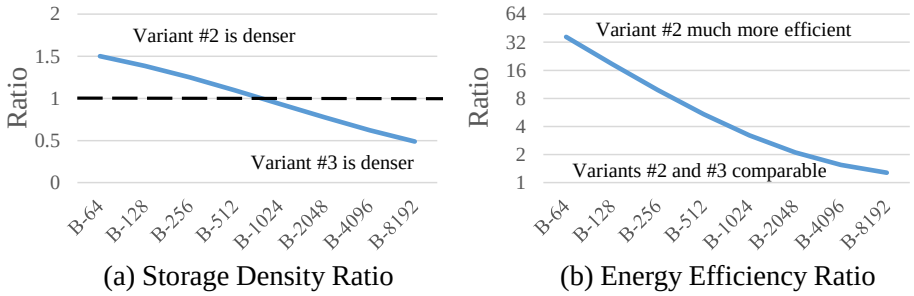


Fig. 5.14. Variant #2 vs. Variant #3

advantage grows with batch size because OPA consumes a larger percentage of the total time for larger batches since the forward and backward passes benefit from pipeline parallelism whereas OPA operations are serialized at the end.

***Base*$_{opa/mvm}$.** $Base_{opa/mvm}$ behaves similarly to $Base_{mvm}$ for convolutional layers.

5.7.6 Comparing Variants #2 and #3

Increasing the batch size for mini-batch SGD increases Variant #2's shared memory requirements for storing all activations and layer gradients in the batch, degrading its storage density. Variant #3 uses a third crossbar for eagerly computing and storing weight gradients, thereby keeping shared memory requirements low at the expense of higher energy to commit the updates to the other crossbars at the end. Figure 5.14 shows that Variant #2 has better storage density and energy efficiency for small batch sizes, while Variant #3 has better storage density for very large batch sizes at comparable energy efficiency.

5.7.7 Comparison with GPUs

Figure 5.15 compares the energy consumption and execution time of Variant #2 with a 2080-Ti GPU for SGD (batch size 1) and Mini-Batch SGD (batch sizes 64 and 1k). Our design significantly reduces energy consumption and execution time due to the use of energy-efficient and highly parallel ReRAM-based matrix operations.

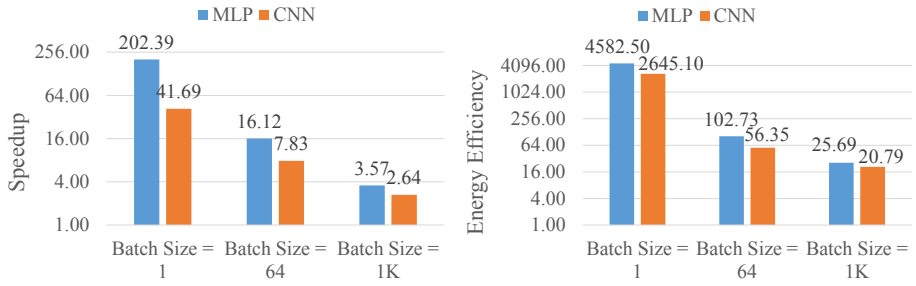


Fig. 5.15. PANTHER's speedup and energy-efficiency compared to GPU

GPUs rely on data reuse to hide memory access latency. For this reason, their relative performance is worse for MLP compared compared to CNN, and for smaller batch sizes compared to larger ones. Our design enables efficient training for a wide spectrum of batch sizes (small to large). Training based on small batch sizes is common in emerging applications such as lifelong learning [210] and online reinforcement learning [211], where training does not rely on any earlier collected dataset.

5.7.8 Sensistivity to ReRAM endurance

ReRAM devices have finite switching (1 to 0, 0 to 1) endurance of 10^9 conservative writes [212,213], which limits their applicability towards on-chip memories for typical workloads. However, the small magnitude of typical weight updates make ReRAM feasible for DNN training. Considering a 5% average conductance change per batch, the lifetime of a chip will be $\simeq$ 6 years (assuming 50% reduction from failed training flows), for 1,000 trainings per year where each training is comprised of 100 epochs, 64 batch-size and 1M training examples (typical parameters in state-of-the-art image recognition benchmarks [214]). While weight slicing makes lower order slices more prone to degradation arising from limited endurance, adding redundancy at lower order slices and higher endurance from technology improvements (currently shown in spintronics [215]) can make the chip more robust.

5.8 Conclusion

We propose a bit-slicing technique for enhancing the precision of ReRAM-based OPA operations to achieve sufficient precision for DNN training. We incorporate our technique into a crossbar architecture that performs high-precision MVM and OPA operations, and present three variants catered to different training algorithms: SGD, mini-batch SGD, and mini-batch SGD with large batches. Finally, to evaluate our design on different layer types and training algorithms, we develop PANTHER, an ISA-programmable training accelerator with compiler support. Our evaluation shows that PANTHER achieves up to 8.02×, 54.21×, and 103× energy reductions as well as 7.16×, 4.02×, and 16× execution time reductions compared to digital accelerators, ReRAM-based accelerators, and GPUs, respectively. The proposed accelerator explores the feasibility of ReRAM technology for DNN training by mitigating their serial read and write limitations, and can pave the way for efficient design of future machine learning systems.

6. TIMON - GPGPU TENSOR CORE

6.1 Introduction

The pervasiveness of deep learning (DL) in myriad of applications [169, 216] and their massive computational costs [10, 13] have led to a growing interest in domain-specific accelerators [10, 43, 144, 217], particularly for *inference* tasks. Such accelerators are designed to cater to the high latency-sensitivity and energy-efficiency demands for inference [144,218]. Such an approach is also resonated by multiple industry efforts such as Google TPU [10], Microsoft Brainwave [144], Nvidia Tensor Cores [161] etc. Further, the growing usage of resource/power constrained *edge* devices (smartphones, wearables) to improve user experience and cater to emerging applications (augmented/virtual reality) has amplified the need for inference efficiency [218].

DL inference tends to be data-intensive and performs a large number of General Matrix-Matrix Multiplication (GEMM) operations. As a result, GPGPUs are widely used for this task given their suitability for data parallelism and dense GEMMs. To meet the growing demands for efficiency, modern GPGPUs have introduced domain-specific units namely tensor cores [161, 219]. Tensor cores execute small matrix-multiplications (two 4×4 matrices) by buffering operands accessed from register file and reusing them across several multiply-and-accumulate (MAC) operations. This reduces the impact of low (high) register file bandwidth (energy) to enable GEMM execution with higher efficiency compared to SIMD cores [161]. Tensor cores have continually evolved with emerging inference trends such as 8-bit, 4-bit, 1-bit GEMMs [161].

Tensor cores consume nearly 62% of the overall runtime in inference (Section 6.3.1), and therefore are a ripe target for further acceleration. Unfortunately, despite the data reuse of tensor cores, they operate at nearly 50% of the peak throughput (Section 6.3.2), thereby being inherently limited by the lower bandwidth and high energy

cost of *large* register files in GPGPUs. The low bandwidth and high energy are implications of GPGPUs favoring large register file. This enables large number of active threads, which can help to hide the long latency of data accesses through multithreading.

Besides runtime limitations in GPGPU tensor cores, they provide limited support for model-compression techniques such as quantization and sparsity pertinent to edge inference. Often, leveraging the full potential of model compression results in irregular quantization [220–222] i.e. different precision across data structures (weight, activation) within and across layers; and unstructured sparsity [12,103,223]. Existing tensor cores neither support irregular quantization (for eg. 5 bits, 7 bits etc.), nor unstructured sparsity. Further, the inherent design of current tensor cores is based on digital CMOS based computation units accessing the large register file in GPGPUs. This limits their ability to leverage aggressive quantization and sparsity techniques (Section 6.3.3).

In this work, we propose TIMON, an in-memory computing based tensor core architecture which overcomes the limitations of GPGPU register files to execute dense GEMMs with high efficiency. Past research have shown in-memory computing can perform analog dot-product operations within the SRAM array [25, 224, 225]. Leveraging in-memory computing techniques, we develop a novel register file microarchitecture that executes GEMMs *within the register file* using bit-serial arithmetic, while preserving the conventional read/write functionality. We show that TIMON's bit-serial arithmetic coupled with in-memory GEMMs enables utilizing irregular and independent bit-precision reductions in activations and weights of DL models to reduce register file access cost. Further, we develop a hardware-software codesign technique that translates unstructured sparsity to reductions in peripheral (row and column peripherals) cost of SRAM sub-banks within the register file. We also present instruction extensions that expose TIMON to the programming model. Our key contributions are:

- Analysis of DL inference, and corresponding sources of inefficiency in GPGPU tensor cores (Section 6.3).
- Novel tensor core architecture named TIMON built by augmenting the conventional register file to execute on-demand GEMM operations within the memory itself, and corresponding instruction extensions for programmability (Sections 6.4, 6.5).
- Flexible and scalable support for model compression techniques used in emerging DL inference, namely irregular quantization and unstructured sparsity (Section 6.6).

6.2 Background

The key insight that motivates TIMON is the potential opportunity from *in-memory* computing based circuits to perform dot-product operations (fixed-point arithmetic) in the SRAM array itelf by activating multiple word-lines (K) in parallel. A typical register file access (read/write) activates only one word-line. Subsequently, the in-memory approach can potentially (i) reduce the number of serial accesses to register file (read/write), and (ii) improve the bandwidth by $K\times$ (upper bound). Thanks to the fixed-point arithmetic needs in inference (contrary to floating-point arithmetic in training), a potential in-memory tensor core can lead to significant performance and energy gains for inference.

6.2.1 Background on In-Memory Computing

In-memory computing has been widely explored at the circuits-level [24, 25, 226, 227] with the motivation of bringing computations closer to memory, thereby overcome memory bandwidth/energy issues. In a typical in-memory primitive (or circuit), multiple word-lines (WLDAC) are activated simultaneously by applying voltages (*analog* input) as shown in Figure 6.1. Typically, such primitives can operate

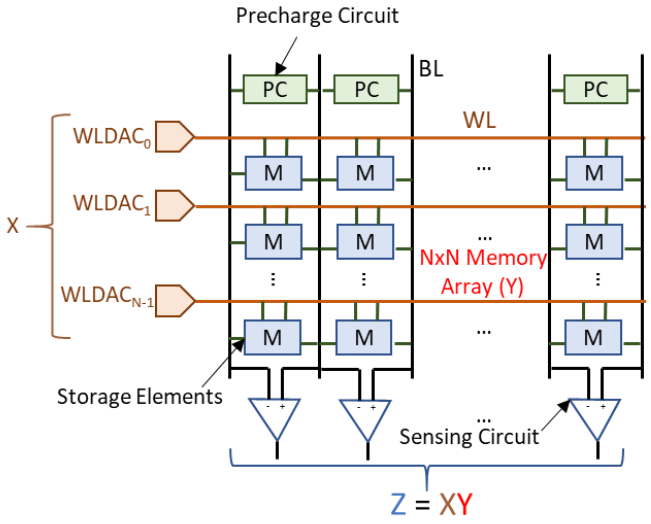


Fig. 6.1. Illustration of in-memory computing primitive.

in two modes: i) current based, or ii) charge based. Based on the voltage applied on WLDACs and the stored value in the memory cell (M), different amount of charge or current is accumulated on the bit-lines (*analog* output). The accumulated charge or current is sensed by a sense amplifier circuit. These primitives are particularly useful for dot-product computations, where an input vector X is applied on WLDACs and memory cells store the elements of matrix Y. The accumulated charge/current on bit-lines (BL) represent output vector Z.

Feasiblity: For CMOS technology, several kinds of in-memory computing primitives [24, 226, 227] have been demonstrated over the past few years, including implementations from industry such as TSMC [225] at scaled technology nodes (7nm). They have achieved high throughput and energy efficiency compared to digital CMOS units such as SIMD cores in GPGPU. Although most in-memory primitives have been explored in the analog computing domain, initial works had also explored performing digital operations [228–230]. Besides CMOS, in-memory primitives based on emerging technologies have also been demonstrated [231–233].

This work focuses on CMOS in-memory primitives that perform analog computations. We show that in-memory primitives performing digital operations (as explored

in [234, 235]) do not offer benefits over digital CMOS units for tensor core feature (Section 6.7.4). Emerging technology based primitives suffer from endurance, high write cost issues, thereby are not amenable for register file since they are written frequently.

6.3 GPGPU Characterization

In this section, we analyze the GEMM kernel's execution characteristics on GPGPU with tensor cores for modern deep learning applications. Subsequently, we discuss the corresponding inefficiency of existing tensor cores.

6.3.1 Workload Characterization

Runtime distribution of compute operations: Figure 6.2 shows the runtime distribution of different operations in inference of Transformer [236] and ResNet-18 [237] models on Nvidia RTX 2080 Ti GPU [161]. It can be observed that the compute operations running on the tensor cores namely convolution (conv2d) and matrix multiplication (matmul) constitute about 62% of the overall runtime. ***Thus, tensor core operations are key compute primitives to accelerate in hardware in order to improve the inference efficiency.***

Quantization in GEMMs: Deep learning applications have extensively leveraged quantization of weights and activations to improve the inference efficiency on edge devices. This is because quantization reduces the memory bandwidth and compute requirements, while preserving the classification accuracy [220,221,237,238]. The observed optimal quantization requirements (bit-widths) of weights and activations are highly irregular i.e. lie in a wide range of 1-16 bits, as well as varies across different layers of a model [220], different models [100], and datasets [238]. ***Thus, tensor cores need flexibility to support irregular quantization and scalability to efficiently trade-off energy and performance with bit-width.***

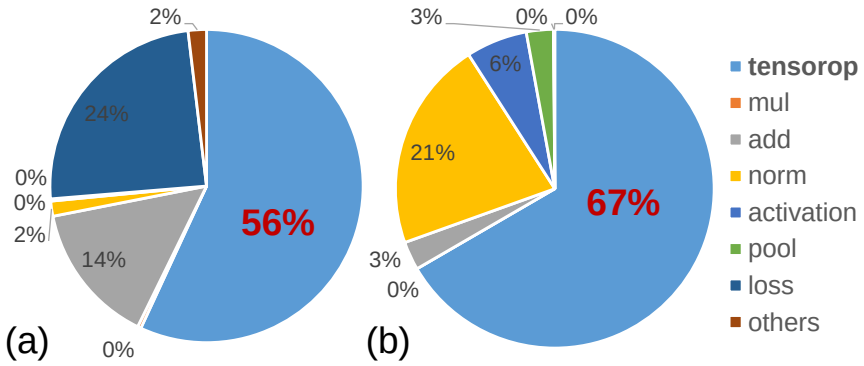


Fig. 6.2. Runtime distribution of tensor core operations (tensorop) compared to other compute operations in RTX 2080 Ti for (a) Transformer, and (b) ResNet-18.

Sparsity in GEMMs: Deep learning applications also leverage weight pruning techniques to obtain sparse models in order to reduce the memory and compute requirements, while retaining accuracy. Past research have shown models with upto 90% weight sparsity [103]. Activation sparsity is inherently present owing to the ReLU, dropout, and batch normalization layers. The commonly observed sparsity (weight and activation) in deep learning applications is of unstructured nature [223]. ***Thus, tensor cores need flexibility to leverage the irregular sparsity patterns in weights and activations.***

6.3.2 Inefficiency of GPU Tensor Cores

Background: Tensor cores are domain-specific units for matrix multiplication in Nvidia GPUs that provide 8× higher peak throughput than the SIMD cores [161]. We found that the two key aspects associated with tensor core that lead to higher throughput are data reuse enabled by a) CUDA WMMA (Warp Matrix Multiply and Accumulate) API, and b) corresponding machine-level instruction - HMMA. The data reuse from WMMA and HMMA overcome the shared memory bandwidth (64B/clock per sub-core), and register file bandwidth (128B/clock) limitations respectively. Subsequently, a high performance GEMM routine ($C = A * B$) using tensor cores such

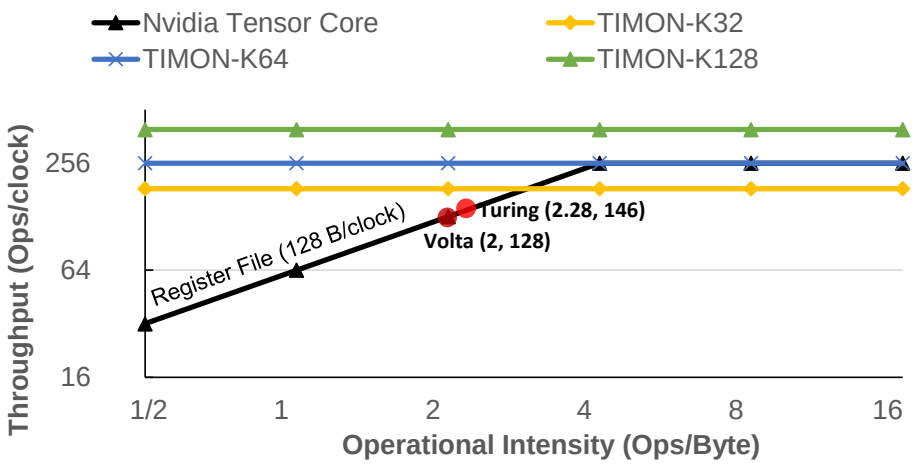


Fig. 6.3. Roofline of digital CMOS tensor core (Nvidia GPU), and in-memory tensor core (TIMON).

as Nvidia CUTLASS [239] leverages a 3-level tiling to overcome memory limitations from 1) global memory, 2) shared memory, and 3) register file.

Tensor Core Roofline: Figure 6.3 shows the roofline for SIMD core (extrapolated for FP16) and tensor core in Nvidia GPU, where the ridge point for tensor core is 4 Ops/Byte[1]. Subsequently, the HMMA throughput for pure FP16 mode was measured for V100 and RTX 2080 Ti GPUs using *clock*() function on a WMMA-based micro-benchmark. The micro-benchmark was compiled with CUDA Toolkit 10.1 with *gpu-architecture* flag being *sm_70* (Titan V) and *sm_75* (RTX 2080 Ti). We found that tensor cores in both Volta and Turing architectures are memory-bound by the register file, thereby operating at about half of the peak hardware throughput of 256 ops/clock. The measured throughputs closely match the theoretical estimations based on HMMA's data reuse. In the subsequent discussions, we will refer to a matrix multiply and accumulate operation of form $D_{M\times N} = A_{M\times K} * B_{K\times N} + C_{M\times N}$ as GEMM shaped $M \times N \times K$. On Volta architecture, a WMMA instruction (PTX level) computes a GEMM shaped $16 \times 16 \times 16$, and consists of 8 HMMA instructions (shown in disassembled SAAS instructions). Further, each HMMA instruction com-

[1]Volta HMMA with operation intensity of 2 Ops/Byte, runs at half of peak throughput.

putes a GEMM shaped $8 \times 8 \times 4$ and has one of the following instruction formats with data reuse of 1.33 and 2 respectively (shown in Table 6.1):

- `HMMA.884.F16.F16 RD,RA,RB,RC`
- `HMMA.884.F16.F16 RD,RA.reuse,RB.reuse,RC`

Here, *reuse* signifies operand forwarding between subsequent HMMA instructions to reduce register file accesses [240]. On Turing architecture, a WMMA instruction can compute one of these GEMM shapes - $16 \times 16 \times 16$ (A), $32 \times 8 \times 16$ (B), or $8 \times 32 \times 16$ (C) with each consisting of 4 HMMA instructions (shown in disassembled SAAS instructions). Further, each HMMA instruction computes a GEMM shaped $16 \times 8 \times 8$ with the following instruction format and data reuse of 2.28 (shown in Table 6.1):

- `HMMA.1688.F16 RD,RA,RB,RC`

Thus, our analysis (measurements and theoretical) concludes that digital CMOS based tensor cores in current GPGPU are memory-bound by the register file. While further improvement in such tensor cores may be obtained by increasing the HMMA shapes, this comes at the cost of increased buffer requirements (and area), thereby limiting its applicability for edge. It is worth noting that reaching the peak throughput with digital CMOS design would require HMMA shape $\geq 16 \times 16 \times 16$, which is similar to register file requirement per WMMA at which point it defeats the purpose of *reusing register file accesses by using smaller buffers.*

Quantization support: Tensor core in current GPGPUs support 16 bit, 8 bit, 4 bit, and 1 bit quantizations [161], where throughput increases nearly *linearly* with quantization. The linear increase is an outcome of linear decrease in memory bandwidth requirements with quantization. Furthermore, both weight and activations are required to have same quantization. ***Thus, quantization support in GPGPU tensor cores is restrictive in terms of 1) allowable bit-precisions, and 2) independent weight and activation bit-precision optimizations.***

Table 6.1.
HMMA data reuse and resulting WMMA cycles. Ops = $2 \times M \times N \times K$, Bytes = $2 \times (2 \times M \times N + M \times K + N \times K)$.

HMMA Type	#Ops	#Bytes	Data Reuse	Cycles (theoretical)	Cycles (measured)
884 w/o reuse	512	384	1.33	96	64
884 w reuse	512	256	2.0	64	
1688 - A	2048	896	2.28	56	59
1688 - B	2048	896	2.28	56	56
1688 - C	2048	896	2.28	56	54

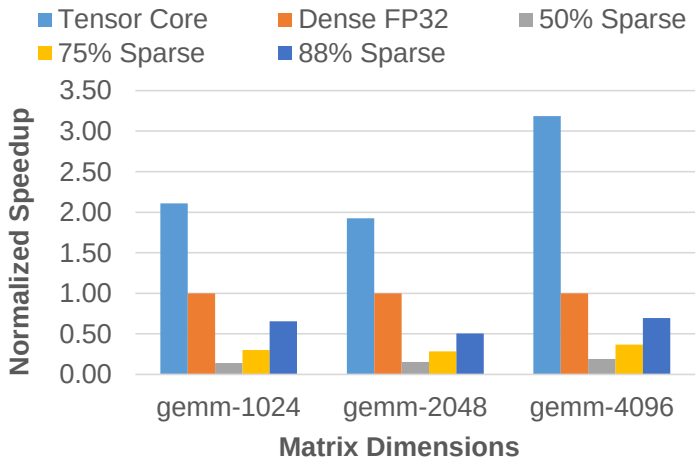


Fig. 6.4. Efficiency of sparse GEMMs on GPGPUs

Sparsity support: GPUs support sparse GEMMs (one sparse and one dense matrix) using NVIDIA cuSPARSE libraries which run on SIMD cores. Currently, tensor cores do not support unstructured sparse GEMMs. We analyzed the impact of unstructured sparsity on GEMM runtime on GPGPUs with 50%, 75%, and 88% sparsity in one of the matrix. It can be observed from Figure 6.4 that sparse GEMMs run at significantly lower efficiency compared to equivalent shaped dense GEMMs. ***Thus, sparsity support in current GPGPUs is not efficient towards leveraging unstructured sparsity for hardware efficiency improvements.***

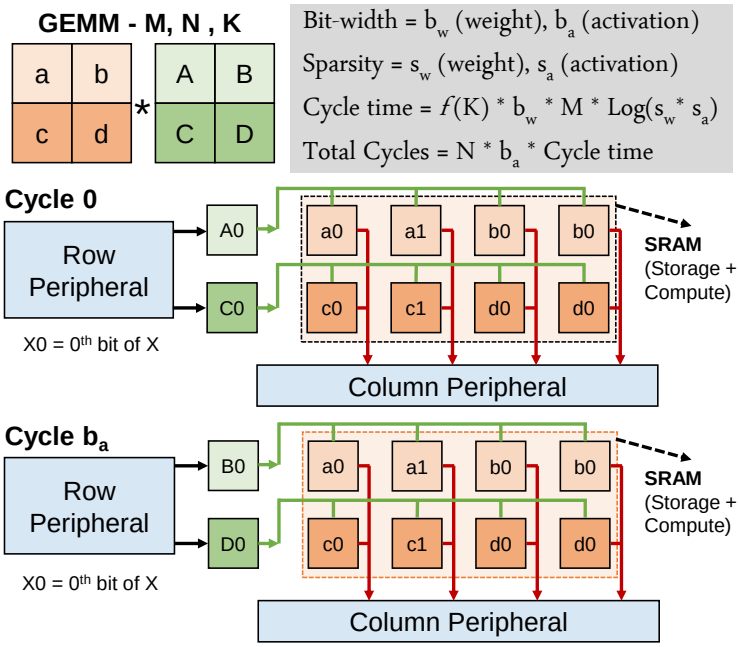


Fig. 6.5. Abstract view of TIMON's compute unit

6.3.3 Summary

GPGPU tensor cores suffer from inefficiencies in multiple scenarios owing to their digital CMOS design due to three key factors: a) register file bottleneck, b) limited support for quantization, and b) conflicting design requirements for dense and sparse GEMMs. First, low register file bandwidth limits the sustained throughput of tensor cores even in presence of data reuse in HMMA instruction. Second, irregular quantization resulting from a) irregular bit-widths (3, 5, 6, etc.), or b) different bit-widths of weights and activations do not inherently translate to reduction in bandwidth requirement. While compiler optimization such as register packing [159], and coalescing [160] can be used, such reductions with quantization is linear at best. Third, dense GEMMs favor large tiles (in HMMA) for data reuse, but sparse GEMMs favor small tiles for fine-grained skipping of computation. Note that while digital CMOS tensor core can be redesigned as bit-serial unit [145, 146], its performance gain will still be limited by register file bandwidth.

Figure 6.5 shows an abstract view of TIMON's compute unit which will be presented in detail in Section 6.4.

6.4 TIMON Architecture

In contrast to existing tensor core, TIMON uses bit-serial and in-memory computing in register file to execute GEMMs, and overcomes bandwidth/energy limitations (Figure 6.3) along with flexible support for irregular quantization and unstructured sparsity. We first discuss the baseline tensor core architecture in Nvidia GPU. Then, we go through the design aspects of in-memory tensor core. Subsequently, we present the modified register file architecture (*TIMON*).

6.4.1 Baseline Tensor Core Architecture (Turing GPU)

Figure 6.6 shows a Nvidia GPGPU sub-core which consists of an instruction cache, a warp scheduler, a dispatch unit connected to 4 SIMD datapaths, 2 tensor cores ($4 \times 4 \times 4$ tensor/clk), a queue for memory instructions (MIO queue), and 64KB register file. Four sub-cores connected to L1 instruction cache, data cache and shared memory compose a streaming multiprocessor (SM). A typical WMMA API that executes GEMM on tensor cores consists of 3 steps a) loading the data from shared memory to register file (*load_matrix_sync*), b) computation in tensor cores (*mma_sync*), and b) storing the data from register file to shared memory (*store_matrix_sync*). Subsequently, a warp's *mma_sync* runs concurrently on both tensor cores leading to a peak throughput of 128 tensor/clk or 2 GEMMs of shape $4 \times 4 \times 4$ per clk [240].

6.4.2 TIMON Dataflow

Figure 6.7 (a) shows the overview of GPGPU register file with enhancements for TIMON. TIMON re-purposes the register file to enable on-demand tensor core in addition to regular operand access (read/write) by augmenting the sub-bank with row

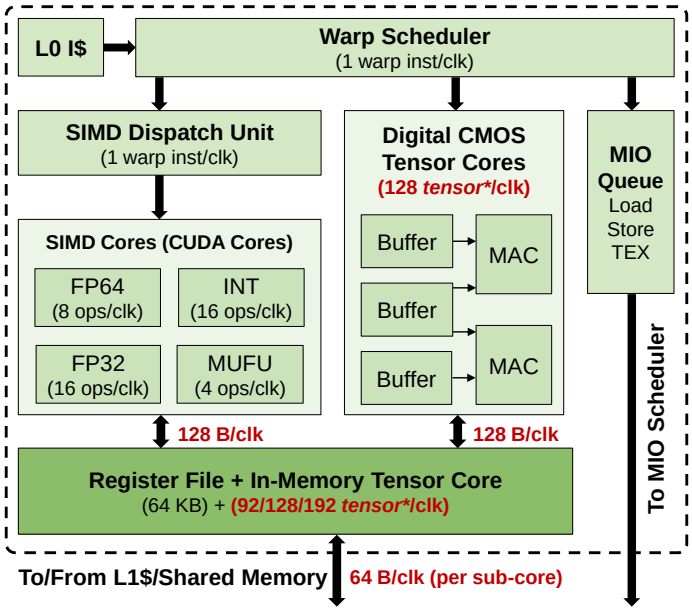


Fig. 6.6. Nvidia GPGPU Sub-Core [241] enhanced with TIMON (dark green). *tensor* consists of 2 ops - multiply, add.

and column peripherals as shown in Figure 6.7 (c). A regular register file read/write accesses an entry across all 8 sub-banks of a bank and follows the conventional datapath within a sub-bank: row decoder, SRAM access, sense amplifier. Next, we discuss the mapping, and execution of tensor computation which depend on three parameters (i) in-memory cell (IMC) bits, (ii) dac bits, and (iii) dot product (DP) width.

GEMM Mapping:

We illustrate the mapping using a GEMM shaped $1 \times 64 \times 4$ on the example configuration shown in Figure 6.7 (b)). The mapping is determined by (i) in-memory cell (IMC) bits, (ii) dac bits, and (iii) dot product (DP) width. In this illustration, we consider identical activation and dac bits; and identical weight and IMC bits for simplicity. As shown in Figure 6.7 (c), the multiplier $B_{4\times64}$ which remains *stationary* during execution is mapped to the SRAM array, while the multiplicand $A_{1\times4}$ is kept *streaming* and is stored in the row peripheral. A row of B is mapped spatially across all IMCs or *slices* in a row (here 64 IMCs ranging from BL 1 - BL 128). Multiple

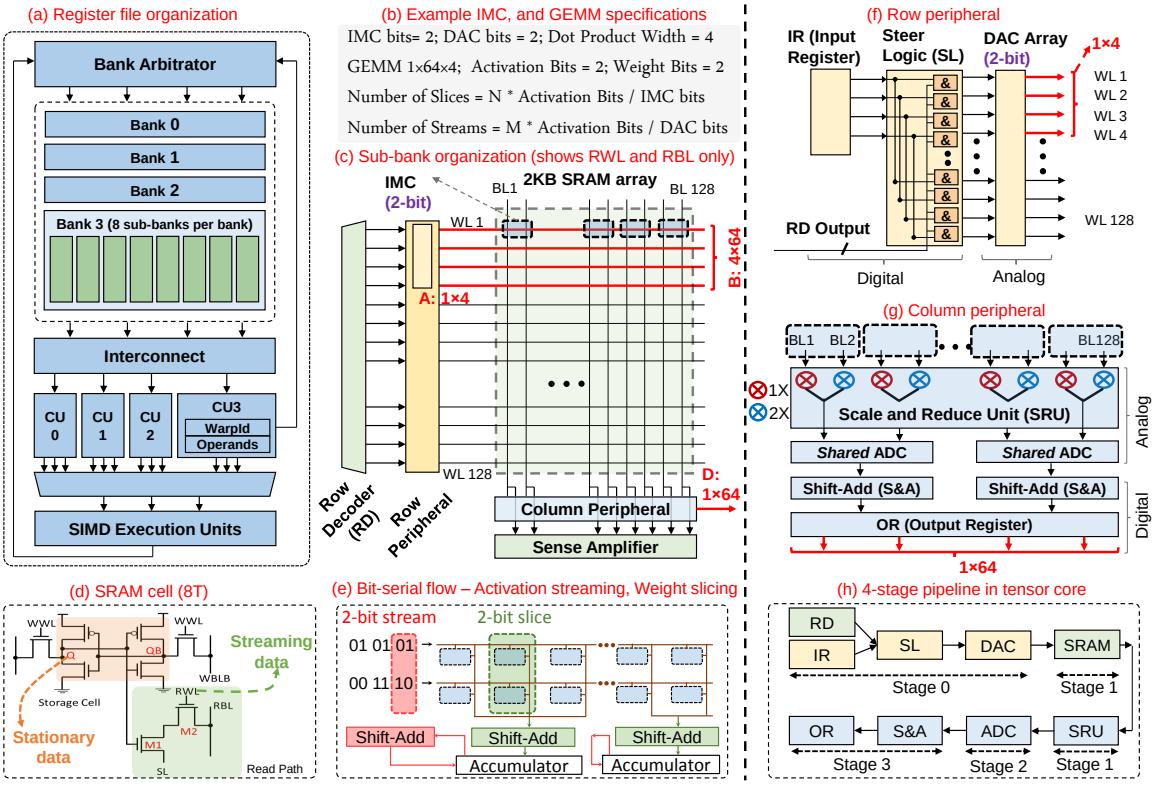


Fig. 6.7. (a) GPGPU Register File. (b) Sub-bank, and GEMM configurations used for illustration. (c) Organization of a 2KB sub-bank composed of 128 entries/wordlines (WL), and four 4B physical registers i.e. 128 bitlines (BL). Each sub-bank re-purposed to read/write an entry and do in-memory GEMM computation on a matrix stored in SRAM rows, and a matrix stored in row peripheral. (d) 8T SRAM cell showing where the stationary data resides and where streaming data is applied. (e) Bit-serial arithmetic for GEMM with generic bit-width for stationary and streaming data (f) Row peripheral for TIMON. (g) Column peripheral for TIMON. (h) 4-stage pipeline for parallel streams.

rows are mapped on adjacent word lines (here 4 ranging from WL 1 - WL 4). A row of A (here 4) is temporally mapped to the corresponding wordlines over multiple time-steps or *streams* (here 1). Subsequently, rows of A are processed one at a time. Figure 6.7 (d) shows the stationary and streaming data mapping within a SRAM cell. A storage cell consists of 1 bit, and multiple SRAM cells comprise an IMC (here 2), thereby leading to a single *slice* of stationary data. The DAC bits (here 2) worth of streaming data is applied (after conversion to analog voltage) to the RWL in read

path. Figure 6.7 (e) extends the mapping for generic bit-widths of stationary data (> slice width) and streaming data (> stream width) using shift and add logic commonly referred as bit-serial arithmetic [23, 234].

Tensor Core Execution:

A *TMMA* instruction (discussed in Section 6.5) executes a GEMM $m \times n \times k$ on a register file bank. The execution consists of 3 steps (i) ***load*** the streaming matrix from bank to input register (IR) in row peripheral, (ii) ***in-memory compute*** within bank, and (iii) ***store*** the output matrix from output register (OR) in column peripheral to bank. While load (to IR) and store (from OR) use conventional read/write datapath for the register file, in-memory compute uses the new datapath. The in-memory compute datapath performs tensor computations in hybrid digital-analog domain and is logically partitioned into 3 stages for each stream (i) analog vector generation - row decoder (RD), row peripheral, (ii) analog computation - SRAM, scale and reduce unit (SRU), and (iii) digital vector generation - column peripheral. DP width specifies the number of concurrently activated during in-memory compute. Here, we assume identical k and $DP\ width$ for simplicity. The operations performed in each stage are:

1. As shown in Figure 6.7 (f), a *digital* stream comprised of k elements is read from IR, and is expanded to *analog* vector consisting of 128 elements using steer logic and digital-to-analog converter (DAC) array. The row decoder output specifies the k rows corresponding to the stationary matrix and ensures their selective activation (other rows see 0 V).

2. Each SRAM cell on a BL computes an *analog* multiplication ($DAC\ bits \times 1\ bit$) followed by their *analog* reduction on the BL to yield a k-element dot product per BL (single cycle). The BLs corresponding to an IMC are scaled and reduced (*analog*) to obtain n dot-products. For example: $out(IMC1) = 1 \times BL1 + 2 \times BL2$ (shown in Figure 6.7 (g)).

3. As shown in Figure 6.7 (g), the analog-to-digital converter (ADC) produces n *digital* outputs, which are subsequently stored in OR after shift and add logic.

6.4.3 TIMON Microarchitecture

The previous section illustrates that TIMON *can increase* the register file bandwidth (upper bound $DP\ Width\times$) by activating multiple WLs in parallel within the SRAM array. However, the parallelism comes at a cost of significant increase in latency from row and column peripherals leading to longer critical path: $IR \rightarrow OR$ shown in Figure 6.7 (h). Even with the pipelined execution of streams (as illustrated in Figure 6.7 (h)), TIMON's latency can be significantly higher compared to a conventional register file access (single cycle). This is due to (i) higher cycle time from shared ADC to amortize the ADC area, and (ii) multiple cycles due to bit-serial processing of streams. Further, the row and column peripherals expend significantly higher energy in analog↔digital conversions compared to the conventional register file access. *Thus, designing an in-memory tensor core involves (i) balancing the latency and parallelism to truly overcome bandwidth limitations, and (ii) amortizing the peripheral overheads for energy-efficiency.*

Sub-bank:

Sub-bank is the key component that affects TIMON's dataflow and thereby its throughput and energy (Section 6.4.2). In-memory tensor core design possesses interdependent layers of complexity, namely, dot-product width, and input/weight data mapping. Hence, we explore the impact of different hardware parameters that affect sub-bank efficiency to understand their isolated impacts and interdependence.

Specification: The sub-bank (128 WL/BL) microarchitecture can be completely specified by 3 parameters $d - t - l$ where d, t, and l refer to DP, stream, and slice widths. For example, within a computation step (or per stream) TIMON (i) converts a digital stream of d elements (t bits each) to analog vector with 128 elements (ii)

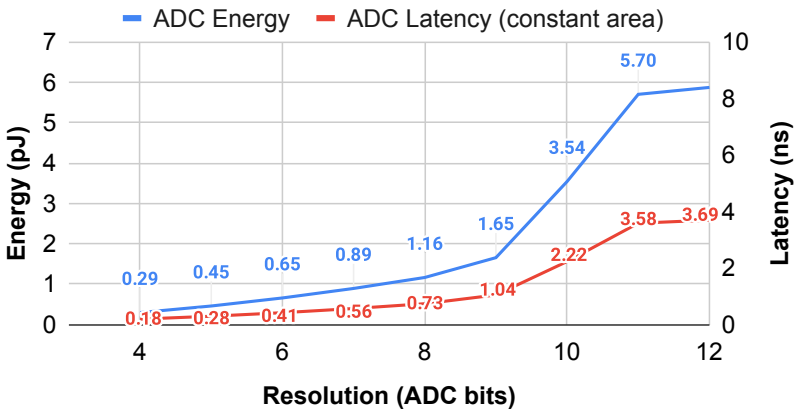


Fig. 6.8. ADC energy, latency for different bits at 45nm node

activates d WLs, 128 BLs, and scale-reduces groups of l BL outputs to generate $128/l$ analog dot-products, and (iii) converts $128/l$ dot-products to digital outputs using $log_2 d + t \times l$ resolution ADC.

Modelling: As the objective of this work is not focused on circuit techniques, we leverage the in-memory primitive (DAC, SRAM, SRU) from Jaiswal et al. [25] and obtained circuit-level measurements from HSPICE. We obtain the ADC trendlines (Figure 6.8) for energy, and latency (at constant area) per conversion using top-5 data points at each resolution ($4-12$ bits) from the widely used survey [185] that comprises of > 500 ADC designs published over the past 30 years. The digital components are modelled at RTL level (Section 6.7).

Impact of DP width: Figure 6.9 shows TIMON's energy/MAC and energy distribution with varying DP width while keeping stream and slice widths constant ($= 1$). It is evident that the energy/MAC is heavily dominated by column peripherals at lower DP widths, with ADC being the top contributor. Upon doubling the DP width the number of SRAM rows activated (or MACs) increase by $2\times$ (*linearly*), which leads to 1 bit increase in ADC resolution. The impact on row and column peripheral (except ADC) remains nearly unchanged since they primarily depend on stream and slice widths respectively for a fixed sub-bank dimensions (#WL and #BL). As shown in Figure 6.8, the ADC energy increase with resolution is nearly

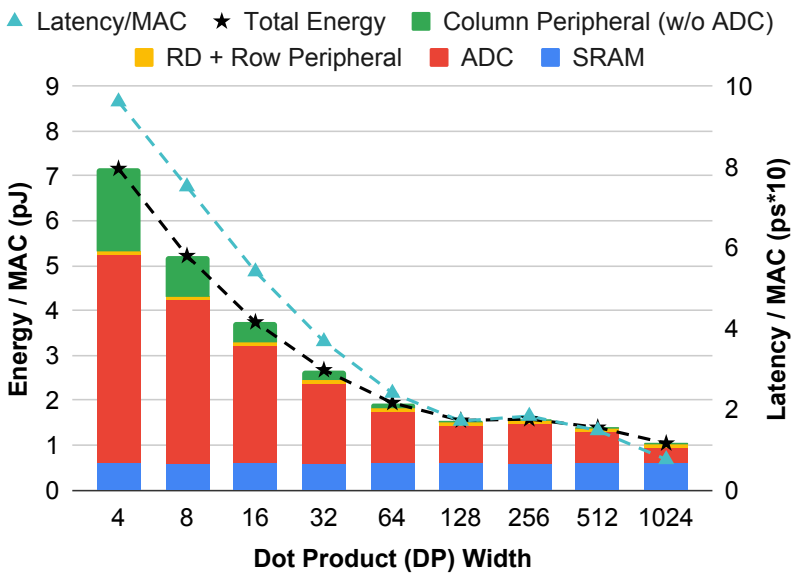


Fig. 6.9. Impact of DP width on energy, latency

quadratic at lower resolutions ($\leq$ 8), and subsequently *exponential*. Hence, higher DP widths are effective towards amortizing the high energy consumption of column peripherals thereby improving energy/MAC.

Figure 6.9 also shows latency/MAC with varying DP widths, with stream and slice widths kept constant (= 1). As shown in Figure 6.7 (h), TIMON's datapath is a 4-stage pipeline. Despite pipelining, the stage latency is significantly higher than a register file access latency owing to the ADC conversion being bottleneck. This is due to the **shared** ADC design necessitated by high area-cost of ADCs. For example, the area cost of a eight typical 8-bit ADCs is comparable to an entire 2KB SRAM (Section 6.7). Subsequently, a shared ADC is time-multiplexed across multiple analog dot-products to compute digital outputs. The latency per input vector in IR can be expressed as $(\#streams + 3) \times (\#slices/\#ADC) \times adc\ latency$. Alike ADC energy, ADC latency shows a *quadratic* growth at lower resolutions, and exponential growth subsequently (Figure 6.8). Consequently, higher DP widths are effective towards amortizing the high latency from shared ADC thereby improving latency/MAC.

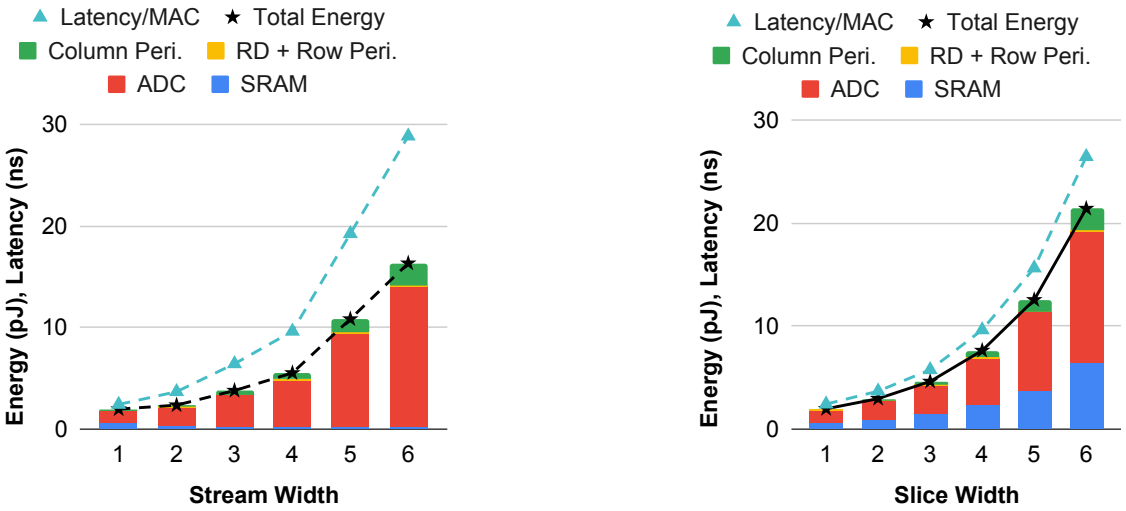


Fig. 6.10. Impact of stream,slice widths on energy, latency

Impact of Stream and Slice Widths: Figure 6.10 shows the individual impact of stream and slice widths on TIMON's energy/MAC (with distribution) and latency/MAC. It is evident that an increase in either stream or slice width makes the energy/MAC as well as latency/MAC worse. Increasing either the stream width or slice width leads to a linear reduction in the number of time steps required per input vector computation. However, at a constant DP width it also results in a commensurate increase in the ADC resolution thereby leading to quadratic (or *exponential*) increase in ADC energy and ADC latency, which eventually outweighs the impact of reduction in time steps. Thus, lower stream and slice widths are favorable for both optimal energy and throughput.

6.4.4 TIMON Register Mapping

A baseline register file in GPGPU interleaves the register mapping across banks to minimize bank-conflicts *across* warps. However, such register mapping is not amenable to TIMON. As discussed in Section 6.4.2, the stationary matrix ($K_t \times N_t$) is mapped on the register file such that the matrix columns are bit-aligned (same

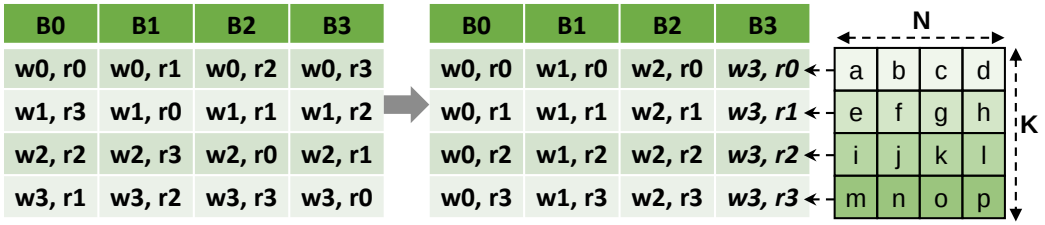


Fig. 6.11. Modified register mapping for bit-alignment

BL) in order to leverage in-memory computing. *Thus, while TIMON overcomes the bandwidth limitation of register file, it imposes strict constraint on register mapping.*

We enable the bit-alignment by (i) ensuring bank-conflicts within a warp, and (ii) mapping different matrix rows on different warp registers as shown in Figure 6.11. This may seem counter intuitive from the perspective of incurring more bank-conflicts for non tensor operations. Since the operand collector can only read one register per cycle, there is no loss in performance compared to the baseline where registers for a warp instruction reside in different banks. Note that operand collector based register file are common in GPGPUs [242]. Further, the new mapping has the same probability of bank-conflicts across warps ($1/\#Bank$) compared to the baseline on an average basis, but it may increase the occurrence of corner cases. For instance, two warps mapped to the same bank will always conflict if scheduled simultaneously. However, we do not observe such corner cases across our benchmarks. Note that such bank-conflict based register mapping have also been leveraged in past research for register file optimizations such as register packing, register coalescing etc. [159, 160].

6.5 TIMON Instruction Extension

SASS Extension: The in-memory datapath is exposed to the programming model (WMMA) by extending the tensor core instruction set with *TIMON MMA* (TMMA) instruction which performs 16-bit multiplication and 32-bit accumulation. Alike Nvidia GPU's HMMA instruction, TMMA is specified by the shape of the corresponding GEMM it executes - $M_t \times N_t \times K_t$. However, while HMMA shape

optimizes register file reuse, TMMA shape optimizes *register utilization*, and *adc reuse*.

$\mathbf{M_t}$: As discussed in Section 6.4.2, TIMON executes a GEMM as a sequence of matrix vector multiplication, where the row vectors of the streaming matrix are processed sequentially. Thus, we choose $M_t = 1$ without loss of generality.

$\mathbf{N_t}$: Each row of the stationary matrix ($N_t \times 16$ bits) is stored per warp register (1024 bits), thereby $N_t = 64$ ensures efficient register utilization. However, a higher value of M_t or N_t increases the register requirements per warp, and may lead to lower warp occupancy. In order to reduce N_t, we partition the $K_t \times N_t$ stationary matrix into s chunks each of dimensions $K_t/s \times N_t$ with a row across all chunks stored per warp register. Each chunk computes an independent K_t/s-element analog dot product, which are reduced by SRU (no scaling) before the ADC conversion. This ensures the same ADC resolution ($t * l * log_2 d$) and efficient register utilization, but increases the latency of Stage 1 in the TIMON pipeline (Figure 6.7 (h)) by a factor of s. However, this does not impact the pipeline latency at lower values of $s \leq 8$, owing to the latency bottleneck from shared ADC. We choose $s = 4$ as 4-wide SRUs are commonly demonstrated in in-memory circuits [25, 225], which leads to $N_t = 16$. Note that chunking can lead to increase in energy/MAC from row peripheral (chunks map across BLs), however the increase is insignificant at higher DP widths which is the common case.

$\mathbf{K_t}$: K_t represents DP width and higher DP widths achieve higher GEMM efficiency. Hence, we choose K_t=32, 64, and 128 which outperform digital CMOS tensor core's efficiency (discussed in Section 6.7.3). DP widths ≥ 256 are not realizable due to electromigration limits of CMOS technology [25].

Thus, TIMON has the following architectural variants (F32 denotes 32-bit accumulation):

- **K32:** `TMMA.1.16.32.F32 RD,RA,RB,RC`
- **K64:** `TMMA.1.16.64.F32 RD,RA,RB,RC`

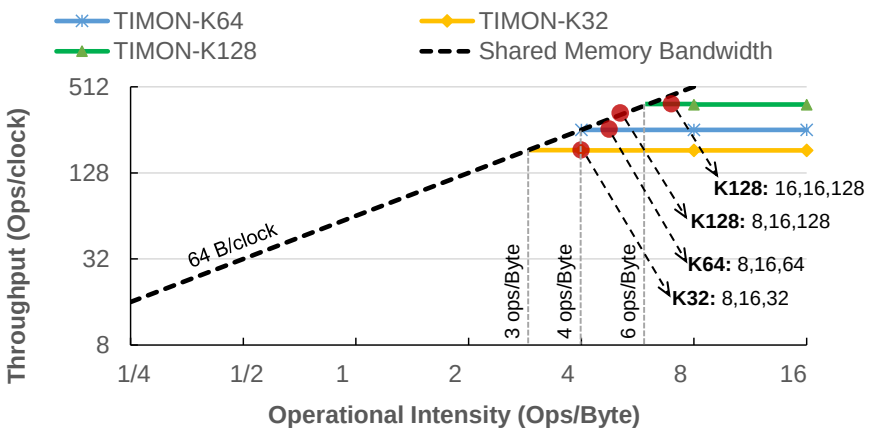


Fig. 6.12. Roofline of TIMON with respect to shared memory.

- **K128:** `TMMA.1.16.128.F32 RD,RA,RB,RC`

PTX Extension: The CUDA programming model uses WMMA API to expose tensor cores which is specified by the shape of the corresponding GEMM it executes - $M_w \times N_w \times K_w$. WMMA shape optimizes for shared memory reuse. To optimize TIMON's WMMA shape for shared memory reuse, we analyze the roofline model of the new datapath with respect to the shared memory bandwidth (Figure 6.12). We choose $N_w = N_t$ and $K_w = K_t$ for simplicity since it preserves the efficient register utilization enabled at TMMA. Subsequently, we optimize M_w to overcome the shared memory bandwidth limitations. It can be seen that the WMMA shapes shapes $8 \times 16 \times 32$, $8 \times 16 \times 64$, and $16 \times 16 \times 128$ enable operating TIMON at peak throughput for the three architectural variants K32, K64, K128 respectively.

Warp Occupancy: In general, GPGPUs prefer high warp occupancy, which enables efficient multithreading to hide the latency of data access. However, TIMON's WMMA shapes have 1×, 1.5×, 2× higher register requirements (registers/thread) compared to WMMA shapes for baseline tensor core. *Thus, improving TIMON's efficiency by increasing K_t or DP width comes at the cost of an increase in the register requirement, which may adversely affect warp occupancy.*

We observe that TIMON's higher register requirement does not degrade performance. Typical GEMM kernels used in DL require large shared memory tiles

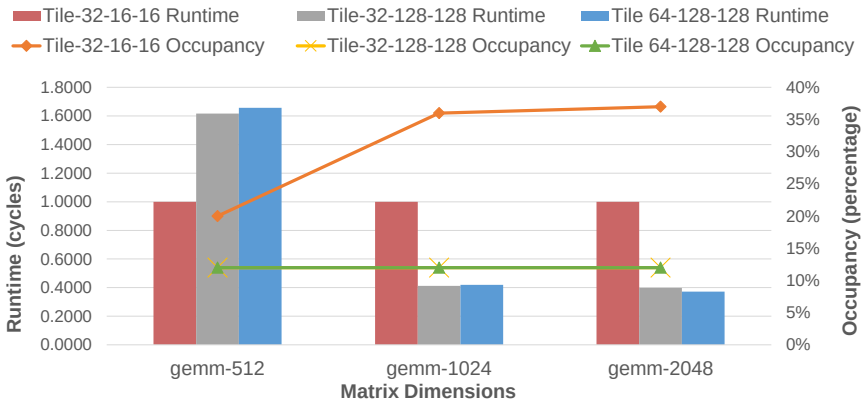


Fig. 6.13. GPGPU warp occupancy.

$(32 \times 128 \times 128, 64 \times 16 \times 128)$ to improve runtime (shown in Figure 6.13). A large tile overcomes the global memory bottleneck to utilize the high throughput of tensor cores, but requires larger shared memory per thread block. As a result, the warp occupancy are low and remain limited by the shared memory requirement, thereby largely unaffected by register requirement.

6.6 Supporting Model Compression

Model compression techniques namely quantization and sparsity reduce bit-widths and parameters respectively. This can potentially lower memory (footprint, bandwidth) and compute needs, thereby save resources, energy, and runtime. Thus, supporting model compression techniques are particularly important for edge. TIMON supports irregular quantization and unstructured sparsity which are widely used in DL algorithms.

6.6.1 Irregular Quantization

A reduction in the bit-widths of activation and weight may lead to reduction in the number of streams $(ceil(\frac{act\ width}{stream\ width}))$ and slices $(ceil(\frac{wt\ width}{slice\ width}))$ respectively per TMMA instruction. This enables independent optimizations to activation and weight

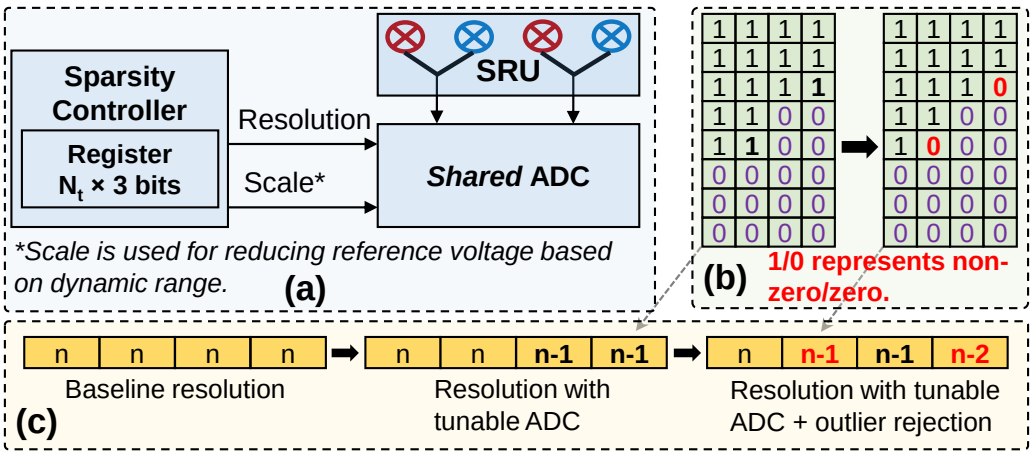


Fig. 6.14. (a) TIMON's column peripheral with sparsity controller. (b) Illustration of outlier rejection technique. (c) Impact of TIMON's hardware-software codesign on ADC resolution.

bit-widths. Further, low stream/slice widths are effective in translating bit-width reductions to commensurate stream/slice reductions. Thanks to low stream/slice widths leading to efficient TIMON configurations (Section 6.4.3), optimality is simultaneously achieved for efficiency and flexible support for irregular quantization. *Thus, TIMON's bit-serial aspect seamlessly (without datapath change) translates bit-width reduction to latency saving, and thereby energy saving.*

6.6.2 Unstructured Sparsity

A higher count of zero values in a column of the stationary matrix reduces the dynamic range of column's analog dot-product commensurately: $0 - nz \times d \times 2^{t \times l}$, where nz is the fraction of non-zero values. For example, 50% and 75% sparsity lead to dynamic range reductions of 1/2, and 1/4 respectively. A lower dynamic range of analog dot-product can reduce the effective bit-width of digital output (MSBs being 0), thereby be leveraged to lower the ADC resolution. Since TIMON's pipeline latency is limited by ADC latency, and energy is dominated by ADC energy, reduction in resolution can lead to significant latency, and energy savings. We propose a hardware-

Role of threshold (th) in outlier rejection

Sparsity of column before outlier rejection	Sparsity of column after outlier rejection	TIMON K64 ADC precision
0 to (0.5 - Th*0.5)	No change	8
(0.5 - Th*0.5) to 0.5	Increases to 0.5	7
0.5 to (0.75 - Th*0.25)	No change	7
(0.75 - Th*0.25) to 0.75	Increases to 0.75	6
0.75 to (0.875 - Th*0.125)	No change	6
(0.875 - Th*0.125) to 0.875	Increases to 0.875	5
And so on.		

Density to Sparsity reference.

Density	Sparsity = 1 - Density
1	0
1/2	0.5
1/4	0.75
1/8	0.875
And so on.	

Fig. 6.15. Illustration of outlier rejection methodology.

software codesign to leverage sparsity by using (i) tunable ADC, and (ii) outlier rejection.

Tunable ADC: Typical successive approximation register (SAR) based ADCs implement a binary search with n iterations to resolve the n output bits. Past research have adapted SAR-ADC design to perform conversions at varying resolutions by changing the reference voltage and subsequently terminating the search algorithm early ($n-1$ iterations for $n-1$ resolution) [243, 244]. This leads to *linear* reduction in ADC latency, and thereby ADC energy with resolution. We adopt tunable ADC approach in TIMON as shown in Figure 6.14 (a). The modified column peripheral consists of a sparsity controller, which stores a 3-bit value per column of the stationary matrix to encode 8 sparsity levels (50%, 75%, 87.5% and so on). At runtime, the sparsity value controls the resolution and reference voltage of shared ADC depending on the column being processed. The sparsity controller is initialized ($N_t \times 3$ bits) by the WMMA API before TMMA instructions' execution, and doesn't add to the TMMA's latency. Note that the sparsity values can be precomputed if the weight matrix is kept stationary (and activation matrix kept streaming) since weights remain constant in inference, thereby not resulting in overhead for computing sparsity values at runtime.

Outlier Rejection: We observed that typical unstructured pruning algorithms namely global [245], layer-wise magnitude [103] based pruning can lead to wide variations in sparsity across columns of the weight matrix. For example, Figure 6.14 (b) shows a toy example of 50% sparse matrix where the column sparsity varies from $38\% - 62\%$. As a result, even with tunable ADC, a 50% sparse matrix does not translate to ADC resolution reduction of all columns by 1 bit. This wastes ADC's dynamic range, which is an expensive component in TIMON. To address this, we augment unstructured pruning algorithms with a simple outlier rejection technique as illustrated in Figure 6.14 (b). Here, depending on a *threshold* and column's sparsity some lower magnitude weights (outliers) are pruned to further increase column sparsity if it improves utilization of ADC's dynamic range. As shown in Figure 6.14 (c), outlier rejection complements the benefits from tunable ADC. The detailed methodology is illustrated in Figure 6.15.

6.7 Evaluation Methodology and Results

6.7.1 Methodology

Tensor Core: For the baseline tensor core, we adapt the proposed microarchitecture by Raihan et al. [162, 240] for Turing [161] (128 MAC, 4.6 KB buffer per sub-core) and implement in RTL, synthesize at IBM 45nm SOI technology node, and evaluate the area, and energy using Synopsys Design Compiler. Timing was measured from RTX 2080 Ti GPU as discussed in Section 6.3.2. To incorporate the benefits of quantization, we use the bit-serial multiplier from BISMO [146].

For TIMON (at 45nm), we use the in-memory circuit (DAC, SRAM, SRU) from Jaiswal et al. [25] to obtain area, and perform circuit simulations to measure energy, and timing using HSPICE. ADC is modelled using the widely used survey by Murmann et al. [185] (Section 6.4.3). The digital components are implemented in RTL as in the baseline tensor core.

Table 6.2.
GEMM benchmarks.

Benchmarks	**Application**	**Shapes**		
		M	**N**	**K**
GNMT-1	Machine	128	2048	4096
GNMT-1	Translation	320	3072	4096
Transformer-1	Language	84	8457	2560
Transformer-2	Understanding	31999	1034	2560
DeepBench-1	Speech	2560	64	2560
DeepBench-2	Recognition	1760	128	1760
		W/H	**R/S**	**C/K**
Conv2d-1		14/14	3/3	256/512
Conv2d-2	Image	7/7	3/3	512/512
ConvPoint-1	Classification	14/14	1/1	256/512
ConvPoint-2		7/7	1/1	512/1024

For TIMON configurations, we use stream/slice widths = 1 as it achieves highest efficiency in terms of energy/MAC, latency/MAC (Section 6.4.3), and quantization (Section 6.6). We use 32 shared ADCs which leads to 33.2% increase in register file banks' area (4 banks). We partition TMMA's operation into 4 chunks as described in Section 6.5. Alike Turing, we execute TMMA instructions one warp at a time. Thus, for TMMA with 4 chunks, 1024 BLs are shared across 32 ADCs which leads to TIMON pipeline's stage latencies: Stage0/Stage1 being $\#chunk = 4$ cycles, Stage2 being $\#BL/\#ADC/\#chunk = 8$ cycles, and Stage 3 being $\#chunk = 4$ cycles. Note that Stage2 (ADC) latency is 2× higher than other stages.

Performance Model: We add the timing parameters for TIMON K32/64/128 configurations and the corresponding instructions in GPGPU-Sim simulator [240,246, 247]. We extend the WMMA API to support the new shapes. We use the simulator configuration for RTX 2080 Ti which has 68 SMs each with 4 sub-cores, 8 Tensor Cores, 256 KB register file [161].

Function Model: We develop a PyTorch [248] based function simulator to analyze the impact of TIMON's circuit non-idealities on inference accuracy. Here, the

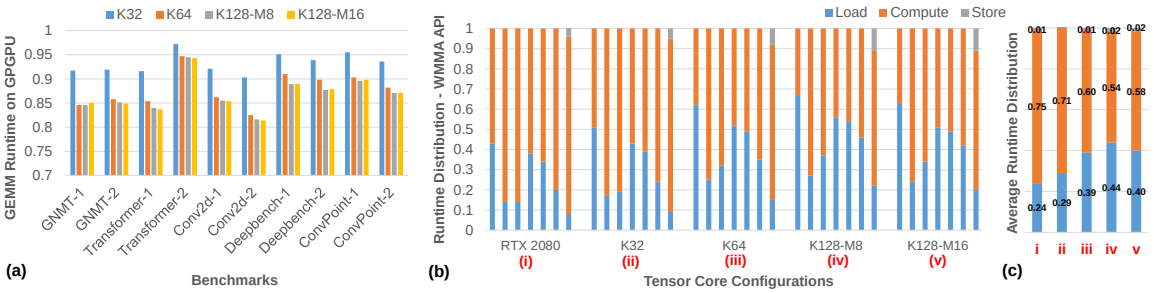


Fig. 6.16. (a) Comparison of GPGPU runtimes with different TIMON configurations normalized to GPGPU with Turing tensor cores. (b) Runtime distribution of WMMA API between Load, Compute, Store for seven instances in Conv2d-2 benchmark on Turing tensor cores and TIMON configurations. (c) Average runtime distribution on seven instances.

forward functions in conv2d, linear modules execute GEMM execution using TMMA's shapes. A TMMA execution uses the circuit-level model extracted from HSPICE [93].

DL Inference Benchmarks: We evaluate 10 benchmarks obtained from Transformer [236], GNMT [249], ResNet18 [237], DeepBench [250], and MobileNetV2 [251] to cover wide range of GEMM shapes (shown in Table 6.2).

6.7.2 GPGPU performance analysis

Figure 6.16 (a) shows the runtime of the basline GPGPU architecture with different TIMON configurations normalized to Turing tensor cores. K32, K64, and K128 refer to TIMON configurations with dot product widths 32, 64, and 128 respectively, and corresponding TMMA shapes discussed in Section 6.5. K128-M8 and K128-M16 refer to TIMON's K128 hardware configuration used with WMMA shapes $8\times16\times128$, and $16 \times 16 \times 128$ respectively (refer Figure 6.12).

K32 and K64: It can be seen that K32 and K64 achieve significant runtime improvements of upto 9.7%, and 17.5% respectively. K64 achieves higher benefits than K32 since increasing the DP width improves TIMON's performance (latency/MAC) as discussed in Section 6.4.3. Further, the reason for tensor core's performance improvements translating to overall runtime improvements can be observed in Fig-

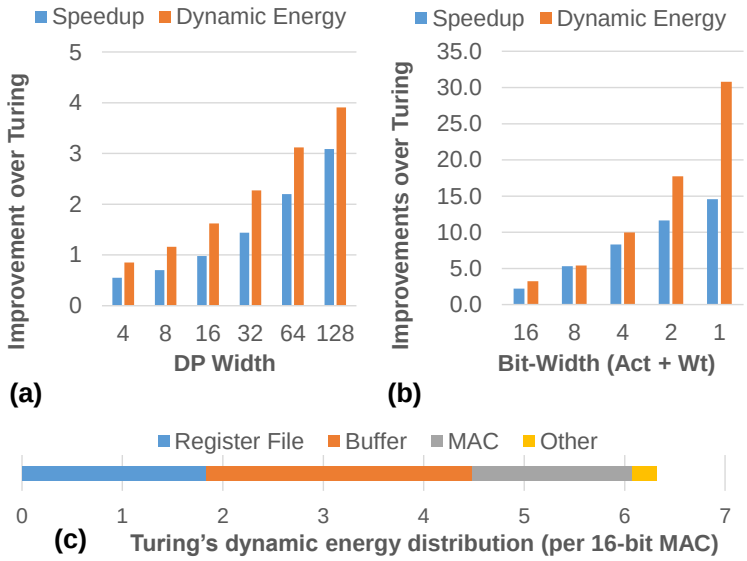


Fig. 6.17. TIMON's speedup and dynamic energy normalized to Turing tensor core for different (a) DP widths, (b) bit-widths of activation (Act) and weight (Wt). (c) Turing tensor core's dynamic energy distribution for 16-bit MAC.

ure 6.16 (b), which shows the runtime distribution of seven different WMMA API instances (load, store and compute) within the Conv2d-2 benchmark. Figure 6.16 (c) averages this distribution for ease of visualization. For the baseline tensor core (Turing), on an average the runtime for tensor computations is nearly 3× higher compared to loading data from shared memory to register file. Consequently, the improved tensor core throughput in K32 and K64 helps to reduce the runtime of WMMA API, thereby reducing the overall GEMM runtime.

K128: Figure 6.16 shows that further benefits in TIMON's throughput from K64 to K128-M8 leads to minimal (nearly 1%) improvement in GEMM runtime. Despite the higher peak throughput, K128-M8 is memory bound by the shared memory as shown in Figure 6.12. As a result, it operates at lower sustained throughput, thereby leading to rather small runtime benefits. This emphasizes the importance of optimizing the WMMA shapes towards utilizing the increased throughput offered by TIMON. Interestingly, even with optimized WMMA shape for K128 (K128-M16) the performance improvement over K64 is minimal. This is because increasing the tensor

core throughput is eventually limited by the global memory bandwidth, for a given shared memory tile size.

6.7.3 Tensor core analysis - TIMON versus Turning

Figure 6.17 (a) shows that TIMON's improvements over Turing increases from 0.55× - 3.09×, and 0.85× - 3.91× for speedup and dynamic energy respectively, when DP width goes from 4 - 128. This is because higher DP widths effectively amortize the cost of row and column peripherals in TIMON, which leads to high energy efficiency (energy/MAC) and latency (latency/MAC) (Section 6.4.3. This emphasizes that at lower DP widths, in-memory computing performs worse compared to digital CMOS, and thereby is not suitable for vector operations such as addition, or multiplication.

6.7.4 Model Compression on TIMON

Quantization: Figure 6.17 (b) shows TIMON's speedup and dynamic energy compared to Turing for different bit-widths of activation and weight. As discussed in Section 6.6, reduction in activation/weight bits leads to a commensurate reduction in #streams/#slices, which results in nearly *quadratic* reduction in TIMON's latency, and energy. As shown in Figure 6.17 (c), Turing's dynamic energy is dominated by memory components (register file, buffer). A reduction in bits leads to at most *linear* reduction in number of memory accesses (Section 6.3.2), thereby leading to *linear* reductions in memory energy, and register file bandwidth required. Consequently, Turing tensor core's energy efficiency, and performance increases nearly *linear* with bit-widths. Thus, TIMON's improvements over Turing increase significantly at low bit-widths.

Sparsity: Figure 6.18 shows the distribution of **column sparsity** for TMMA's GEMMs executed in the conv2d-2 benchmark with 0.5 (or 50%) **model sparsity**, and subsequently applying outlier rejection with thresholds - $0.25, 0.5, 1.0$. A column density ($1.0 - sparsity$) of 1/2, 1/4, 1/8 is ideal since it ensures full utilization

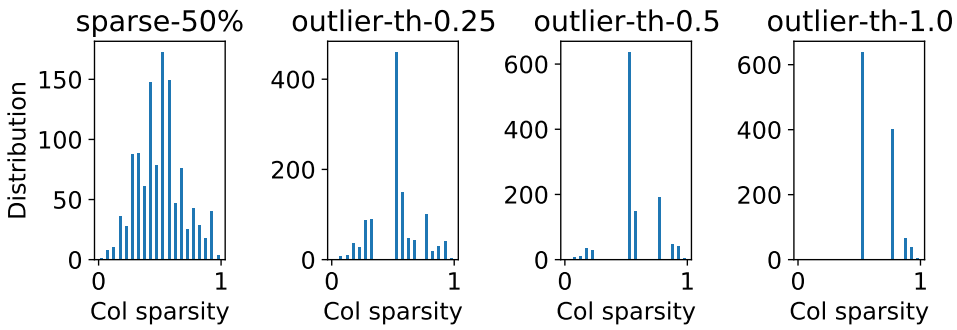


Fig. 6.18. Impact of outlier rejection on column sparsity

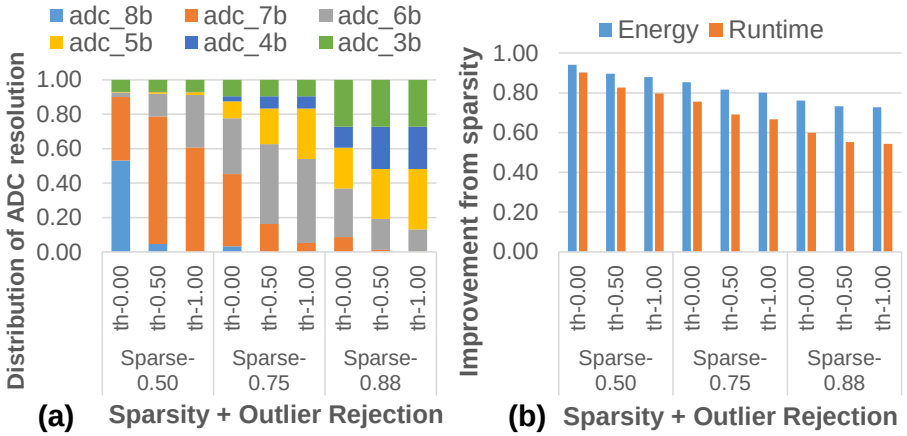


Fig. 6.19. Impact of tunable ADC and outlier rejection on (a) ADC resolution, (b) Tensor Core's energy and runtime.

of ADC's dynamic range. As discussed in Section 6.6, outlier rejection with higher threshold can increase a column's sparsity, thereby leading to higher utilization. Figure 6.18 shows that higher thresholds increase the fraction of columns with ideal density thereby, improving the utilization of ADC's dynamic range.

Figure 6.19 (a) shows the impact of tunable ADC on a sparse model (for eg. sparse-0.50 denotes 50% model sparsity), and its corresponding versions obtained using outlier rejection with different thresholds (*th*). We use TIMON's K64 variant since it achieves high efficiency over Turing for dense GEMMs (Section 6.7.2). A baseline without tunable ADC uses 8 bit (8b) resolution for all operations. It is

Table 6.3.
Area comparison of TIMON/Turing tensor cores.

TIMON	**IR**	**SL**	**DAC**	**SRU**	**ADC**	**S&A**	**OR**	**Total**
Area (mm^2)	0.00077	0.00006	0.00017	-[†]	0.10560	0.00192	0.00019	0.1087
Turing	**MAC**	**Buffer**	**Total**					
Area (mm^2)	0.2025	0.0042	0.2067					

* SRU is logical unit for slice width 1. In absence of scaling, analog reductions occur on BLs itself.

Table 6.4.
TIMON's accuracy under circuit non-idealities

Baseline Accuracy - 69.60% - ResNet20 - CIFAR-100		
Configuration	**w/o retraining**	**w/ retraining (#epochs)**
K32	65.36 %	69.51 (1)
K64	63.47 %	68.80 (1), 69.58 (2)

evident that tunable ADC enables leveraging sparsity to reduce ADC resolutions requirement. Further, outlier rejection complements tunable ADC to get further reductions in ADC resolution requirements. As discussed in Section 6.4.3, TIMON's energy consumption is dominated by ADC energy, and pipeline latency is limited by the ADC latency. For K64, ADC energy constitutes 59.6% of TIMON's energy (Figure 6.9), and the ADC stage has 2× higher latency than other pipeline stages (Section 6.7.1). Consequently, as shown in Figure 6.19 (b), the reduction in ADC energy/latency with sparsity translates to significant improvements in tensor core's energy consumption and runtime. We observe that outlier rejection incurs $\leq 1.01\%$ reduction in test accuracy compared to the baseline sparse models for all thresholds, thereby emphasizing its efficacy.

6.7.5 Area Overheads and Circuit Non-Idealities

Table 6.3 shows the area breakdown of TIMON and Turing tensor core. It can be seen that TIMON adds 53% lower area overhead to a baseline GPGPU (without

tensor cores) to enable inference acceleration, thereby emphasizing its effectiveness. Table 6.4 analyzes the impact of circuit non-idealities on TIMON's test accuracy. We observe that inference of a pretrained model (no retraining) on TIMON may incur undesirable accuracy degradation. Higher DP widths lead to higher degradation due to the higher volume of analog operations performed before conversion to digital data. However, simple retraining for 1-2 epochs recovers the accuracy within 0.09% and 0.02% of the baseline accuracy for K32 and K64 respectively. Thus, the impact of circuit non-idealities in TIMON can be reliably mitigated with minimal retraining overhead.

6.8 Conclusion

We propose TIMON, an "in-memory" tensor core which overcomes bandwidth and energy limitations of GPGPU register files to perform efficient dense GEMMs. TIMON leverages in-memory computing within the register files to execute GEMMs in a bit-serial fashion and reduces register file accesses even for irregular weight and activation precisions. Further, we present a hardware-software codesign to utilize sparsity, and develop instruction extensions to expose TIMON to the programming model. Our results show that a GPGPU augmented with TIMON can achieve upto 195 Tensor TOPS at 53% lower area overhead and improve IPC upto 18%. Further, these performance benefits are accompanied by a 74% reduction in tensor core energy. Finally, TIMON achieves nearly quadratic and logarithmic performance gains with bit-precision and sparsity, respectively.

7. SUMMARY

General-purpose computing systems have benefited from technology scaling for several decades but are now hitting a performance/energy wall. This trend has led to a growing interest in domain-specific accelerators. Machine Learning (ML) workloads in particular have received tremendous attention because of their pervasiveness across applications. ML workloads tend to be data-intensive and perform many matrix operations. Their execution on digital CMOS hardware is typically characterized by high data movement costs. To overcome this limitation, in-memory computing primitives (CMOS, NVM) have been demonstrated to perform matrix operations with high efficiency by overcoming the low memory bandwidth and high memory energy issues. While such primitives have shown tremendous potential at the device-circuit levels, the system-level implications remain unclear, as they are not a drop-in replacement for traditional memory structures (register file, caches etc.). This thesis explores the implications of in-memory computing on domain-specific and general-purpose architectures for ML. The key contributions are summarized below.

- In this thesis, we propose PUMA - the first ISA-programmable accelerator for ML inference that uses hybrid CMOS-memristor technology. It enhances memristive crossbars with general purpose execution units carefully designed to maintain crossbar area/energy efficiency and storage density. Our accelerator design comes with a complete compiler to transform high-level code to PUMA ISA and a detailed simulator for estimating performance and energy consumption. Our evaluations show that PUMA can achieve significant improvements compared to state-of-the-art CPUs, GPUs, and ASICs for ML acceleration.

- This thesis explores training approaches that can learn connectivity structures, which can be efficiently mapped to crossbars while preserving the algorithmic

benefits of weight sparsity. We developed TraNNsformer a technology-aware clustered pruning approach to produce efficient mappings for any crossbar size, permissible by the technology for reliable operations. Our results on a range of image recognition applications suggest that TraNNsformer is a promising framework to implement DNNs, providing a scalable solution to designing large-scale neuromorphic systems.

- In this thesis, we explore bit-slicing techniques for enhancing the precision of ReRAM-based OPA operations to achieve sufficient precision for DNN training. We incorporate our technique into a crossbar architecture that performs high-precision MVM and OPA operations, and present three variants catered to different training algorithms: SGD, mini-batch SGD, and mini-batch SGD with large batches. Finally, to evaluate our design on different layer types and training algorithms, we develop PANTHER, an ISA-programmable training accelerator with compiler support. The proposed accelerator explores the feasibility of ReRAM technology for DNN training by mitigating their serial read and write limitations, and can pave the way for efficient design of future machine learning systems.

- This thesis also proposes TIMON - an in-memory tensor core which overcomes bandwidth and energy limitations of GPGPU register files to perform efficient dense GEMMs. It leverages in-memory computing within the register files to execute GEMMs in a bit-serial fashion and reduces register file accesses even for irregular weight and activation precisions. Further, we present a hardware-software codesign to utilize sparsity, and develop instruction extensions to expose TIMON to the programming model. Finally, the proposed tensor core achieves nearly quadratic and logarithmic performance gains with bit-precision and sparsity, respectively.

REFERENCES

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[2] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in neural information processing systems*, 2014, pp. 568–576.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[4] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 8604–8608.

[5] H. Zeng, M. D. Edwards, G. Liu, and D. K. Gifford, "Convolutional neural network architectures for predicting dna–protein binding," *Bioinformatics*, vol. 32, no. 12, pp. i121–i127, 2016.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[7] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[8] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, 2016, pp. 173–182.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, 2014, pp. 269–284.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[11] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, 2018.

[12] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[13] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, pp. 216–222, 2018.

[14] F. Alibart, E. Zamanidoost, and D. B. Strukov, "Pattern classification by memristive crossbar circuits using ex situ and in situ training," *Nature communications*, vol. 4, 2013.

[15] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, Nov 2015.

[16] 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.

[17] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[18] P. M. Sheridan, F. Cai, C. Du, W. Ma, Z. Zhang, and W. D. Lu, "Sparse coding with memristor networks." *Nature nanotechnology*, 2017.

[19] S. Yu, Z. Li, P.-Y. Chen, H. Wu, B. Gao, D. Wang, W. Wu, and H. Qian, "Binary neural network with 16 mb rram macro chip for classification and online training," in *Electron Devices Meeting (IEDM), 2016 IEEE International*. IEEE, 2016, pp. 16–2.

[20] 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.

[21] R. Waser, R. Dittmann, G. Staikov, and K. Szot, "Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges," *Advanced materials*, vol. 21, no. 25-26, pp. 2632–2663, 2009.

[22] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 27–39. [Online]. Available: https://doi.org/10.1109/ISCA.2016.13

[23] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[24] 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)*. IEEE, 2018, pp. 488–490.

[25] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy, "8t sram cell as a multibit dot-product engine for beyond von neumann computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2556–2567, 2019.

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

[27] A. Ankit, A. Sengupta, and K. Roy, "Trannsformer: Neural network transformation for memristive crossbar based neuromorphic system design," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 533–540.

[28] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W. mei Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan, "Hardware-software co-design for an analog-digital accelerator for machine learning," in *Rebooting Computing (ICRC), 2018 IEEE International Conference on*. IEEE, 2018.

[29] A. Ankit, T. Ibrayev, A. Sengupta, and K. Roy, "Trannsformer: Clustered pruning on crossbar-based architectures for energy efficient neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[30] A. Ankit, I. El Hajj, S. Agarwal, M. Marinella, M. Foltin, J.-P. Strachan, D. S. Milojicic, W.-M. W. Hwu, K. Roy *et al.*, "Panther: A programmable architecture for neural network training harnessing energy-efficient reram," *IEEE Transactions on Computers*, 2020.

[31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[32] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 257–260.

[33] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53–60.

[34] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 247–257.

[35] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[36] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015.

[37] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "4.6 a1. 93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications," in *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*. IEEE, 2015, pp. 1–3.

[38] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 13–19.

[39] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 92–104.

[40] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.

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

[42] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170.

[43] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[44] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. IEEE Press, 2016, pp. 267–278.

[45] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ser. ISCA'16. IEEE, 2016, pp. 1–13.

[46] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[47] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40.

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

[49] J. Chung and T. Shin, "Simplifying deep neural networks for neuromorphic architectures," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[50] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 382–394.

[51] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 395–408.

[52] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[53] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultra-low power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[54] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Microarchitecture (MICRO), 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[55] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[56] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "SC-DCNN: Highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 405–418.

[57] Y. Wang, H. Li, and X. Li, "Real-time meets approximate computing: An elastic CNN inference accelerator with adaptive trade-off between QoS and QoR," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 33.

[58] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[59] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition," in *Microarchitecture (MICRO), 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[60] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 369–381.

[61] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 393–405. [Online]. Available: https://doi.org/10.1109/ISCA.2016.42

[62] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 13–26.

[63] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32–37.

[64] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of FPGA-based learning accelerators for the neural network family," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[65] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 14–26.

[66] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Microarchitecture (MICRO), 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[67] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 535–547.

[68] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ser. ISCA'16. IEEE, 2016, pp. 380–392.

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

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

[71] M. Ali, A. Agrawal, and K. Roy, "Ramann: in-sram differentiable memory computations for memory-augmented neural networks," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 61–66.

[72] K. Roy, I. Chakraborty, M. Ali, A. Ankit, and A. Agrawal, "In-memory computing in emerging memory technologies for machine learning: an overview," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[73] I. Chakraborty, M. Ali, A. Ankit, S. Jain, S. Roy, S. Sridharan, A. Agrawal, A. Raghunathan, and K. Roy, "Resistive crossbars as approximate hardware building blocks for machine learning: Opportunities and challenges," *Proceedings of the IEEE*, 2020.

[74] S. Jain, A. Ankit, I. Chakraborty, T. Gokmen, M. Rasch, W. Haensch, K. Roy, and A. Raghunathan, "Neural network accelerator design with resistive crossbars: Opportunities and challenges," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 10–1, 2019.

[75] A. Ankit, I. Chakraborty, A. Agrawal, M. Ali, and K. Roy, "Circuits and architectures for in-memory computing-based machine learning accelerators," *IEEE Micro*, vol. 40, no. 6, pp. 8–22, 2020.

[76] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 498–503.

[77] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Spindle: Spintronic deep learning engine for large-scale neuromorphic computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, pp. 15–20.

[78] Y. Kim, Y. Zhang, and P. Li, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *J. Emerg. Technol. Comput. Syst.*, vol. 11, no. 4, pp. 38:1–38:25, Apr. 2015.

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

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

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

[82] 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*. ACM, 2017, p. 26.

[83] Y. Wang, W. Wen, B. Liu, D. Chiarulli, and H. H. Li, "Group scissor: Scaling neuromorphic computing design to large neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 85.

[84] A. Ankit, A. Sengupta, P. Panda, and K. Roy, "Resparc: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 27.

[85] A. Sengupta, A. Ankit, and K. Roy, "Performance analysis and benchmarking of all-spin spiking neural networks (special session paper)," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 4557–4563.

[86] B. Han, A. Ankit, A. Sengupta, and K. Roy, "Cross-layer design exploration for energy-quality tradeoffs in spiking and non-spiking deep artificial neural networks," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 613–623, 2017.

[87] A. Ankit, A. Sengupta, and K. Roy, "Neuromorphic computing across the stack: Devices, circuits and architectures," in *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2018, pp. 1–6.

[88] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 21.

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

[90] Y. Hayakawa, A. Himeno, R. Yasuhara, W. Boullart, E. Vecchio, T. Vandeweyer, T. Witters, D. Crotti, M. Jurczak, S. Fujii *et al.*, "Highly reliable tao x reram with centralized filament for 28-nm embedded application," in *VLSI Technology (VLSI Technology), 2015 Symposium on*. IEEE, 2015, pp. T14–T15.

[91] R. Yu, "Panasonic and umc partner for 40nm reram process platform," *UMC Press Release*, Feb 2017.

[92] "Tsmc annual report 2017," *TSMC*, Mar 2018.

[93] I. Chakraborty, M. F. Ali, D. E. Kim, A. Ankit, and K. Roy, "Geniex: A generalized approach to emulating non-ideality in memristive xbars using neural networks," *arXiv preprint arXiv:2003.06902*, 2020.

[94] S. Agarwal, S. J. Plimpton, D. R. Hughart, A. H. Hsia, I. Richter, J. A. Cox, C. D. James, and M. J. Marinella, "Resistive memory device requirements for a neural algorithm accelerator," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 929–938.

[95] S. Agarwal, R. B. J. Gedrim, A. H. Hsia, D. R. Hughart, E. J. Fuller, A. A. Talin, C. D. James, S. J. Plimpton, and M. J. Marinella, "Achieving ideal accuracies in analog neuromorphic computing using periodic carry," in *2017 Symposium on VLSI Technology*. IEEE, 2017, pp. T174–T175.

[96] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, "RxNN: A Framework for Evaluating Deep Neural Networks on Resistive Crossbars," *CoRR*, vol. abs/1809.00072, 2018. [Online]. Available: http://arxiv.org/abs/1809.00072

[97] I. Chakraborty, D. Roy, and K. Roy, "Technology aware training in memristive neuromorphic systems for nonideal synaptic crossbars," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 5, pp. 335–344, 2018.

[98] C. Liu, M. Hu, J. P. Strachan, and H. H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 87.

[99] 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 *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 19–24.

[100] 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.

[101] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[102] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 1398–1406.

[103] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[104] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[105] B. Li, W. Wen, J. Mao, S. Li, Y. Chen, and H. H. Li, "Running sparse and low-precision neural network: When algorithm meets hardware," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*. IEEE, 2018, pp. 534–539.

[106] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy, "2.9 tops/w reconfigurable dense/sparse matrix-multiply accelerator with unified int8/inti6/fp16 datapath in 14nm tri-gate cmos," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 39–40.

[107] Z. Yuan, J. Yue, H. Yang, Z. Wang, J. Li, Y. Yang, Q. Guo, X. Li, M.-F. Chang, H. Yang *et al.*, "Sticker: A 0.41-62.1 tops/w 8bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 33–34.

[108] J. Park, J. Kung, W. Yi, and J.-J. Kim, "Maximizing system performance by balancing computation loads in lstm accelerators," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 7–12.

[109] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[110] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[111] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Ccr: A concise convolution rule for sparse neural network accelerators," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 189–194.

[112] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[113] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA'17. New York, NY, USA: ACM, 2017, pp. 548–560. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080215

[114] 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 *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[115] L. Liang, L. Deng, Y. Zeng, X. Hu, Y. Ji, X. Ma, G. Li, and Y. Xie, "Neural network pruning for crossbar architecture," *IEEE Access*, pp. 1–1, 2018.

[116] Q. Xu, S. Chen, B. Yu, and F. Wu, "Memristive crossbar mapping for neuromorphic computing systems on 3d ic," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, 2018, pp. 451–454.

[117] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, "Multiscale co-design analysis of energy, latency, area, and accuracy of a ReRAM analog neural training accelerator," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 86–101, 2018.

[118] P. Narayanan, A. Fumarola, L. L. Sanches, K. Hosokawa, S. Lewis, R. M. Shelby, and G. W. Burr, "Toward on-chip acceleration of the backpropagation algorithm using nonvolatile memory," *IBM Journal of Research and Development*, vol. 61, no. 4/5, pp. 11–1, 2017.

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

[120] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, ser. ISCA'18. IEEE, 2018, pp. 367–382.

[121] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "RedEye: analog ConvNet image sensor architecture for continuous mobile vision," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. IEEE Press, 2016, pp. 255–266.

[122] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "PROMISE: an end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA'18. IEEE Press, 2018, pp. 43–56.

[123] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 382–394. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123982

[124] R. Cai, A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang, "VIBNN: Hardware acceleration of bayesian neural networks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 476–488. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173212

[125] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA'10. New York, NY, USA: ACM, 2010, pp. 247–257. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[126] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[127] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA'15. New York, NY, USA: ACM, 2015, pp. 92–104. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750389

[128] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA'17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[129] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[130] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 553–564.

[131] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-Brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[132] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 395–408.

[133] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 369–381. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694358

[134] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 66–77.

[135] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: a scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA'17. New York, NY, USA: ACM, 2017, pp. 13–26. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080244

[136] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis *et al.*, "Spatial: a language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 296–311.

[137] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA'17. New York, NY, USA: ACM, 2017, pp. 535–547. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080221

[138] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 367–381.

[139] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[140] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7675–7684.

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

[142] Y. Yang, S. Wu, L. Deng, T. Yan, Y. Xie, and G. Li, "Training high-performance and large-scale deep neural networks with full 8-bit integers," *arXiv preprint arXiv:1909.02384*, 2019.

[143] 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*. IEEE, mar 2017.

[144] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[145] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[146] Y. Umuroglu, L. Rasnayake, and M. Själander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 307–3077.

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

[148] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[149] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 749–763.

[150] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 189–202.

[151] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 674–687.

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

[153] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[154] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 235–246.

[155] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 420–432.

[156] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 247–258.

[157] X. Liu, M. Mao, X. Bi, H. Li, and Y. Chen, "An efficient stt-ram-based register file in gpu architectures," in *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 490–495.

[158] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 412–423.

[159] X. Wang and W. Zhang, "Gpu register packing: Dynamically exploiting narrow-width operands to improve performance," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 745–752.

[160] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 701–714.

[161] Nvidia, "Nvidia turing gpu architecture," in *https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf*.

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

[163] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[164] 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 *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[165] D. Lee and K. Roy, "Area efficient rom-embedded sram cache," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 9, pp. 1583–1595, 2013.

[166] X. Guo, F. M. Bayat, M. Bavandpour, M. Klachko, M. Mahmoodi, M. Prezioso, K. Likharev, and D. Strukov, "Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded nor flash memory technology," in *Electron Devices Meeting (IEDM), 2017 IEEE International*. IEEE, 2017, pp. 6–5.

[167] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

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

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

[170] A. Agresti, *Logistic regression*. Wiley Online Library, 2002.

[171] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied linear statistical models*. Irwin Chicago, 1996, vol. 4.

[172] T. S. Furey, N. Cristianini, N. Duffy, D. W. Bednarski, M. Schummer, and D. Haussler, "Support vector machine classification and validation of cancer tissue samples using microarray expression data," *Bioinformatics*, vol. 16, no. 10, pp. 906–914, 2000.

[173] P. Resnick and H. R. Varian, "Recommender systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[174] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[175] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[176] I. Sutskever, G. E. Hinton, and G. W. Taylor, "The recurrent temporal restricted boltzmann machine," in *Advances in Neural Information Processing Systems*, 2009, pp. 1601–1608.

[177] T. Tanaka, "Mean-field theory of boltzmann machine learning," *Physical Review E*, vol. 58, no. 2, p. 2302, 1998.

[178] CCIX Consortium, "CCIX interconnect," https://www.ccixconsortium.com, 2017.

[179] Gen-Z Consortium, "Gen-Z interconnect," http://genzconsortium.org/about/, 2016.

[180] B. Wile, "Coherent accelerator processor interface (CAPI) for POWER8 systems," IBM, Tech. Rep., September 2014.

[181] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," HP Labs, HPL-2009-85, Tech. Rep., 2009.

[182] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W. J. Dally, *BookSim 2.0 User's Guide*, 2013.

[183] A. B. Kahng, B. Lin, and S. Nath, "Comprehensive modeling methodologies for NoC router estimation," UCSD, Tech. Rep., 2012.

[184] Q. Wang, Y. Kim, and P. Li, "Neuromorphic processors with memristive synapses: Synaptic interface and architectural exploration," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 12, no. 4, p. 35, 2016.

[185] B. Murmann, "ADC performance survey 1997-2020," *http://www. stanford. edu/˜ murmann/adcsurvey. html*, 2020.

[186] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[187] B. Murmann, "The race for the extra decibel: a brief review of current adc performance trajectories," *IEEE Solid-State Circuits Magazine*, vol. 7, no. 3, pp. 58–66, 2015.

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

[189] R. M. Roth, "Fault-tolerant dot-product engines," *arXiv preprint arXiv:1708.06892*, 2017.

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

[191] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8.

[192] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp. 849–856.

[193] R. B. Palm, "Prediction as a candidate for learning deep hierarchical models of data," *Technical University of Denmark*, vol. 5, 2012.

[194] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[195] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, no. 2, 2011, p. 5.

[196] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[197] B. Rajendran, Y. Liu, J.-s. Seo, K. Gopalakrishnan, L. Chang, D. J. Friedman, and M. B. Ritter, "Specifications of nanoscale devices and circuits for neuromorphic computational systems," *IEEE Transactions on Electron Devices*, vol. 60, no. 1, pp. 246–253, 2013.

[198] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" in *Neural Networks (IJCNN), The 2012 International Joint Conference on*. IEEE, 2012, pp. 1–5.

[199] 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*. IEEE Computer Society, 2007, pp. 3–14.

[200] E. J. Merced-Grafals, N. Dávila, N. Ge, R. S. Williams, and J. P. Strachan, "Repeatable, accurate, and high speed multi-level programming of memristor 1t1r arrays for power efficient analog computing applications," *Nanotechnology*, vol. 27, no. 36, p. 365202, 2016.

[201] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[202] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[203] Y. Kim, H. Kim, D. Ahn, and J.-J. Kim, "Input-splitting of large neural networks for power-efficient accelerator with resistive crossbar memory array," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2018, p. 41.

[204] S. N. Truong and K.-S. Min, "New memristor-based crossbar array architecture with 50-% area reduction and 48-% power saving for matrix-vector multiplication of analog neuromorphic computing," *Journal of semiconductor technology and science*, vol. 14, no. 3, pp. 356–363, 2014.

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

[206] A. Nag, R. Balasubramonian, V. Srikumar, R. Walker, A. Shafiee, J. P. Strachan, and N. Muralimanohar, "Newton: Gravitating towards the physical limits of crossbar acceleration," *IEEE Micro*, vol. 38, no. 5, pp. 41–49, 2018.

[207] Y. Geifman, "cifar-vgg," https://github.com/geifmany/cifar-vgg/blob/master/README.md, 2018.

[208] BVLC, "caffe," https://github.com/BVLC/caffe/wiki/Models-accuracy-on-ImageNet-2012-val, 2017.

[209] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[210] Z. Chen and B. Liu, "Lifelong machine learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 10, no. 3, pp. 1–145, 2016.

[211] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *arXiv preprint arXiv:1507.00814*, 2015.

[212] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Muraoka, S. Mitani, S. Fujii, K. Katayama, M. Iijima *et al.*, "Highly reliable taox reram and direct evidence of redox reaction mechanism," in *2008 IEEE International Electron Devices Meeting*. IEEE, 2008, pp. 1–4.

[213] J. J. Yang, M.-X. Zhang, J. P. Strachan, F. Miao, M. D. Pickett, R. D. Kelley, G. Medeiros-Ribeiro, and R. S. Williams, "High switching endurance in tao x memristive devices," *Applied Physics Letters*, vol. 97, no. 23, p. 232102, 2010.

[214] W. Yang, "pytorch-classification," https://github.com/bearpaw/pytorch-classification/blob/master/TRAINING.md, 2017.

[215] X. Fong, Y. Kim, R. Venkatesan, S. H. Choday, A. Raghunathan, and K. Roy, "Spin-transfer torque memories: Devices, circuits, and systems," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1449–1488, 2016.

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

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

[218] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[219] Nvidia, "Nvidia a100 tensor core gpu architecture," in *https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf*.

[220] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 8612–8620.

[221] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[222] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *International Conference on Machine Learning*, 2017, pp. 3007–3016.

[223] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *arXiv preprint arXiv:2003.03033*, 2020.

[224] X. Si, J.-J. Chen, Y.-N. Tu, W.-H. Huang, J.-H. Wang, Y.-C. Chiu, W.-C. Wei, S.-Y. Wu, X. Sun, R. Liu *et al.*, "A twin-8t sram computation-in-memory unit-macro for multibit cnn-based ai edge processors," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 189–202, 2019.

[225] Q. Dong, M. E. Sinangil, B. Erbagci, D. Sun, W.-S. Khwa, H.-J. Liao, Y. Wang, and J. Chang, "15.3 a 351tops/w and 372.4 gops compute-in-memory sram macro in 7nm finfet cmos for machine-learning applications," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020, pp. 242–244.

[226] H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma, "A 64-tile 2.4-mb in-memory-computing cnn accelerator employing charge-domain compute," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 6, pp. 1789–1799, 2019.

[227] X. Si, J. Chen, Y. Tu, W. Huang, J. Wang, Y. Chiu, W. Wei, S. Wu, X. Sun, R. Liu, S. Yu, R. Liu, C. Hsieh, K. Tang, Q. Li, and M. Chang, "24.5 a twin-8t sram computation-in-memory macro for multiple-bit cnn-based machine learning," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 396–398.

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

[229] A. Agrawal, A. Jaiswal, D. Roy, B. Han, G. Srinivasan, A. Ankit, and K. Roy, "Xcel-ram: Accelerating binary neural networks in high-throughput sram compute arrays," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 8, pp. 3064–3076, 2019.

[230] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy, "X-sram: Enabling in-memory boolean computations in cmos static random access memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4219–4232, 2018.

[231] C. Xue, W. Chen, J. Liu, J. Li, W. Lin, W. Lin, J. Wang, W. Wei, T. Chang, T. Chang, T. Huang, H. Kao, S. Wei, Y. Chiu, C. Lee, C. Lo, Y. King, C. Lin, R. Liu, C. Hsieh, K. Tang, and M. Chang, "24.1 a 1mb multibit reram computing-in-memory macro with 14.6ns parallel mac computing time for cnn based ai edge processors," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 388–390.

[232] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu, "A fully integrated reprogrammable memristor–cmos system for efficient multiply–accumulate operations," *Nature Electronics*, vol. 2, no. 7, pp. 290–299, 2019.

[233] S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, M. Giordano, M. Bodini, N. C. Farinha *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, no. 7708, pp. 60–67, 2018.

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

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

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

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

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

[239] "Nvidia cutlass 2.2."

[240] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 79–92.

[241] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *Ieee Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[242] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand collector architecture," Nov. 16 2010, uS Patent 7,834,881.

[243] M. Yip and A. P. Chandrakasan, "A resolution-reconfigurable 5-to-10b 0.4-to-1v power scalable sar adc," in *2011 IEEE International Solid-State Circuits Conference*. IEEE, 2011, pp. 190–192.

[244] ——, "A resolution-reconfigurable 5-to-10-bit 0.4-to-1 v power scalable sar adc for sensor applications," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 6, pp. 1453–1464, 2013.

[245] A. See, M.-T. Luong, and C. D. Manning, "Compression of neural machine translation models via pruning," *arXiv preprint arXiv:1606.09274*, 2016.

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

[247] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring modern gpu memory system design challenges through accurate modeling," *arXiv preprint arXiv:1810.07269*, 2018.

[248] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[249] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[250] Baidu, "Baidu deepbench," in *https://github.com/baidu-research/DeepBench*.

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

VITA

VITA

Aayush Ankit received the B.Tech. in Electronics Engineering from Indian Institute of Technology (BHU), Varanasi, India in 2015, and Ph.D. degree in Electrical and Computer Engineering from Purdue University in 2020 (expected). Currently, he is working on next-generation AI systems at Microsoft. His research interests lie in hardware-software codesign, computer architecture, deep learning and AI accelerator architectures.

He has worked as a ML Architect at HPE Labs, Palo Alto, CA in 2017; CPU Designer at Intel Corporation, Hillsboro, OR in 2017; and GPU Architect at Samsung ACL, San Jose, CA in 2019. He has also worked as a Mitacs Globalink Fellow at University of Alberta, Canada in 2014, and Research Intern at Hanyang University, South Korea in 2013. Aayush also serves on the Technical Program Committee of international conferences and workshops such as DAC, EMC2, and has been awarded IEEE SiPS Best Paper Award (2018).