NEURO-GENERAL COMPUTING
AN ACCELERATION-APPROXIMATION APPROACH

A Dissertation
Presented to
The Academic Faculty By

Amir Yazdanbakhsh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2018

NEURO-GENERAL COMPUTING
AN ACCELERATION-APPROXIMATION APPROACH

Approved by:

Dr. Hadi Esmaeilzadeh, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Nam Sung Kim
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Computer Science
Georgia Institute of Technology

Date Approved: May 3, 2018

Be humble or else you tumble.

Anonymous

This dissertation is dedicated to all members of my family, especially to my: father, Mohammad Esmaeil Yazdanbakhsh; mother, Shahla Roosta; and sister, Niloofar Yazdanbakhsh—my first teacher of my life—who have always loved and supported me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

ACKNOWLEDGEMENTS

This long venture of my PhD journey is a combined effort of many characters, including the scholars whom I had the privilege to meet and work with, family members who constantly supported me and evinced me to finish my PhD, and friends who were always accessible and guided me through this journey. All of them played a crucial role in one or another step of this journey and helped me to achieve my goals. None of my achievements, if any, would have been possible without the unconditional support and help from them. This thesis would therefore be incomplete without expressing my gratitude to all of them.

First and foremost, I want to thank my marvelous advisors, Hadi Esmaeilzadeh and Nam Sung Kim, for always trusting me, giving me enough resources and opportunities to improve my work and excel in my path of learning, as well as my presentation and writing skills. An special thank to Hadi Esmaeilzadeh (my direct advisor) whom I was honored to work with for almost four years. He undoubtedly had played a pivotal role in my life. He was always available and supportive through my PhD journey. He provided me unparalleled opportunities, supported me in both my academic life and personal life, and inspired me to work hard and maintain my determination. His commitment to guide me for carrying out high-impact and high-quality research and instilling his unique creative vision in research have helped me to form a visionary research personality and be always ambitious about my goals, something that I will be indebted to him for the rest of my life. I am proud to say my experience of working with him was intellectually exciting and fun, and has energized me to continue research. I am also grateful to the members of my PhD committee: Hyesoon Kim, Sudhakar Yalamanchili, and Milos Prvulovic for their valuable feedback and for making the final steps towards my PhD very smooth.

I am also grateful to all the current and previous members of Alternative Computing Technologies (ACT) Lab who taught me new things every day. I will not forget the all-nighters that we pulled together and the crazy nights that we spent in the lab crunching

numbers. Thanks for the fun and support which made my PhD extremely enjoyable and memorable. Thanks for being patient with me during our heated research discussions and openly listening to all of my ideas and providing me the most invaluable feedback and comments. I greatly look forward to having all of you as colleagues in the years ahead.

During my education journey, I was fortunate to be surrounded by some of the most wonderful people I know. Hooman Tahayori and Farzin Sobhanmanesh, my advisors in Shiraz University, who motivated me to pursue my studies at graduate level. My late advisor in University of Tehran, Sied Mehdi Fakhraie, a scholar and a true human being, who was the one that introduced me to the research world. His death, undoubtedly, is a loss in my life. Azadeh Davoodi, my advisor in University of Wisconsin-Madison, who gave me the opportunity to continue my studies in the USA and supported me unconditionally to become a successful researcher. Pejman Lotfi-Kamran who passionately taught me the underpinnings of research in Computer Architecture and assisted me to grow in my chosen career path. I am extremely grateful to work under Gennady Pekhimenko's close mentorship. He helped me to better form my ideas and clearly and coherently present them to others. Gennady was always available for offering his unsolicited advice which helped me to transcend in my career and become a more mature researcher. He also helped me to be a better citizen of the Computer Architecture community.

I would like to thank Doug Burger, Luis Ceze, Onur Mutlu, and Todd C. Mowry with whom I had the opportunity to collaborate. Working with these renowned scholars helped me to better understand the research in our field and taught me to always set the bar high in my career and follow my dreams. I am grateful for having the opportunity to work under the supervision of Karu Sankaralingam for a short period of time. He supported me and took me under his wing when I needed it the most in my career path. His unwavering support put me back on the track to pursue my dream in the academic world. Although, working with Karu was short, his impact on me and my career was tremendous. During working in his lab, I had the opportunity to work with many great scholars, such as Raghuraman

Balasubramanian and Anthony Nowatzki, from whom I learned much about research in Computer Architecture. I am grateful for having Mehdi Kamal as my mentor and now a reliable friend. He helped me to grow when I first joined University of Tehran for my Master degree and taught me the first steps to become a researcher. I also want to thank my first grade teacher, Mrs. Farzaneh, who I always remember for the rest of my life. She patiently taught me how to read and write despite me being a high-maintenance student. I wish I could see her one more time and thank her for all she has done for me.

During my PhD, I had the opportunity to work in many acclaimed companies as a research intern. I am grateful to my internship mentors for making my work in their companies mutually successful for both sides. At Information Sciences Institute, I had the chance to work with Michel Sika, Jonatan Ahlbin, and Bradley Kiddie. An special thank to Michel who has faithfully mentored me and taught me how to carry out high impact research. At Rambus, I had the fortunate to work with David Stork and Craig Hampel. At Microsoft Research, I had the privilege to closely work with leading researchers in the field, Madan Musuvathi, Todd Mytkowicz, Saeed Maleki. At NVIDIA Research, I had the privilege to closely work with Joel Emer, Michael Pellauer, Christopher Fletcher, Angshuman Parashar, Steve Keckler. I am also thankful to all the companies that financially supported my research and helped me to continue living my dream life. Specially, Microsoft for Microsoft Research PhD Fellowship and Qualcomm for the Qualcomm Innovation Fellowship.

During my PhD, I had the privilege to mentor and supervise many undergraduate and graduate students from multiple illustrious universities, including Michael Brzozowski, Fatemehsadat Mireshghallah, Hajar Falahati, Vahideh Akhlaghi, Manali Kumar, Oleg Filatov, Anandhavel Nagendrakumar, and Sindhuja Sethuraman. Thank you for letting me to be your mentor, helping me grow as a person and become a better leader, and contributing significantly to my research. Each day of working with you was a treasured lesson for me. I can not wait to see what you all do next.

I am also deeply grateful to my friends, colleagues, and mentors for their vital contri-

butions to my personal and academic life (in no particular order), Farshad Firouzi, Hamid Shojaee, Min Li, Mohammad Fattah, Iman Entezari, Amin Farmahini, Bashir Ebrahimi, Hamid Noori, Saeed Safari, Solmaz Asanfi, Mostafa E. Salehi, Ali Azarpeyvand, Kia Bazarga, Hamed Dorosti, Zeinalabedin Navabi, David Palframan, Mikko Lipasti, Ali Afzali-Kusha, Masoud Pedram, Renée St. Amant, Arjang Hassibi, Saba Amanollahi, Amir Rostami, Abbas Rahimi, Atieh Lotfi, Rajesh K. Gupta, Kambiz Samadi, Manu Rastogi, Behnam Khaleghi, Girish Varatkar, Sunjae Park, Ching-Kai Liang, Girish Mururu, Sarah Cannon, Mohmmadali Rahimian, Bita Darvish Rohani, Mehdi Ahmadi, Tushar Krishna, Santosh Pande, Sriseshan Srikanth, Amin Momeni, Amirhossein Davoodi, Abolghasem Beheshti, Mohammad Reza Karami Mehr, Vijay Thiruvengadem, Emmanuel Amaro, Jared Urban, Emily Faszold, Hadi Mirisaee, Ramtin Raji, Mehdi Askari.

I also thank the wonderful staff in the School of Computer Science as well as in other departments of Georgia Institute of Technology for always being so helpful and friendly. People here are genuinely nice and want to help you out and Im glad to have interacted with many.

Last but not the least, I especially thank my mom, dad, and sister. My hard-working parents have sacrificed their lives for my sister and myself and provided unconditional love and care. I love them so much, and I would not have made it this far without them. My sister has been my best friend all my life and I love her dearly and thank her for all her advice and support. I know I always have my family to count on when times are rough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Keywords: Approximate Computing; Machine Learning; Generative Adversarial Networks; Convolutional Neural Networks; CNN; Transposed Convolution; Access-Execute Architecture; GAN; DNN; MIMD; SIMD; Accelerator

SUMMARY

Neuro-General Computing
An Acceleration-Approximation Approach

A growing number of commercial and enterprise systems rely on compute and power intensive tasks. While the demand of these tasks is growing, the performance benefits from general-purpose platforms are diminishing. As the result of the Dark Silicon studies shows [1, 2, 3, 4, 5] the improvements in per-transistor speed and energy efficiency diminishing. Moreover, the current paradigm of microprocessor design falls significantly short of the historical cadence of performance improvements [1, 4, 5]. Performance has hit the power wall. These challenges have coincided with the big data era where the data is being generated in such an overwhelming rate that is beyond the capabilities of current computing systems to match. While data generation is quadrupling each year, modern processors have seen a performance improvement 15% every two years. Without continuous performance improvements, grand-challenge applications, such as enhanced cognition and immersive virtual reality, computer vision, machine learning, sensory data processing, stochastic optimization and big data analytics may stay out of reach due to their need for significantly higher compute capacity. To address these convoluted challenges, there is a need to move beyond traditional techniques and explore unconventional paradigms in computing. This thesis is set to introduce a new paradigm in computing, called neuro-general computing that leverages the approximibility in many emerging applications (e.g., machine learning, physical simulation, data visualization, big data analytics, sensory data processing, augmented reality, stochastic optimization, and computer vision) for delivering significant gains in performance and energy efficiency. Furthermore, in this thesis, I study the symbiosis between accelerator design and approximation in deep neural networks. Finally, I explore the challenges of accelerating generative adversarial networks, the frontier of deep networks, and introduce and develop an architecture which accelerate this new

class of deep networks. As such, this thesis consists of three main parts:

1. **Neuro-general computing.** We explore three different design points for this new paradigm of computing. First, we leverage the simplicity of the operations in neuro-general paradigm, add and multiplication, to design a mixed-signal accelerator. We introduce a novel architecture that carefully implement the microarchitectural components in analog or digital domain. Furthermore, we introduce a compiler-circuit co-design to mitigate the inherent imprecision in analog circuits. Then, we study the potential benefits of neuro-general computing in GPU throughput processors. Integrating neural accelerator units within GPUs is fundamentally different from doing so in a CPU, because of the hardware constraints and the many-thread SIMT execution model in GPUs. Finally, we observe that neurally accelerating GPU cores increase the pressure on the already-limited GPU memory bandwidth. As such, we study the integration of neural accelerators within DRAM. We introduce a novel DRAM architecture that integrates several low-overhead neural accelerators within DRAM while preserving the SIMT execution model of GPUs.
2. **Accelerator-approximation in deep neural networks.** Deep Convolutional Neural Networks (CNNs) are among the most widely used family of machine learning methods that have had a transformative effect on a wide range of applications. CNNs require ample amounts of computation even for a single input query. For instance, assigning a label to a relatively small RGB image requires billions of multiply-and-accumulate operations. In this part of thesis, we aim to reduce these copious amount of computation by exploiting both their runtime information and algorithmic structure. In convolutional layers of many modern CNNs, each convolution operation is commonly followed by an activation function called a Rectifying Linear Unit (ReLU) that returns zero for negative inputs and yields the input itself for the positive ones. Leveraging this insight, we introduce a holistic software-hardware solution, that cuts a large fraction of the computations short by identifying the zero intermediate values earlier during the runtime.

3. **Unsupervised learning acceleration.** Generative Adversarial Networks (GANs) are one of the most recent deep learning models that generate synthetic data from limited genuine datasets. GANs are on the frontier as further extension of deep learning into many domains (e.g., medicine, robotics, content synthesis) requires massive sets of labeled data that is generally either unavailable or prohibitively costly to collect. GANs leverage a new operator, called transposed convolution, that exposes unique challenges for hardware acceleration. This operator first inserts zeros within the multidimensional input, then convolves a kernel over this expanded array to augment information to the embedded zeros. Even though there is a convolution stage in this operator, the inserted zeros lead to underutilization of the compute resources when a conventional convolution accelerator is employed. We propose an architecture to alleviate the sources of inefficiencies associated with the acceleration of GANs using conventional convolution accelerators, making the first GAN accelerator design possible. We propose a reorganization of the output computations to allocate compute rows with similar patterns of zeros to adjacent processing engines, which also avoids inconsequential multiply-adds on the zeros. This compulsory adjacency reclaims data reuse across these neighboring processing engines, which had otherwise diminished due to the inserted zeros. The reordering breaks the full SIMD execution model, which is prominent in convolution accelerators. Therefore, we propose a unified MIMD-SIMD design that leverages repeated patterns in the computation to create distinct microprograms that execute concurrently in SIMD mode.

CHAPTER 1

LIMITED PRECISION NEURO-GENERAL COMPUTING

1.1 Summary

As improvements in per-transistor speed and energy efficiency diminish, radical departures from conventional approaches are becoming critical to improving the performance and energy efficiency of general-purpose processors. We propose a solution—from circuit to compiler—that enables general-purpose use of limited-precision, analog hardware to accelerate "approximable" code—code that can tolerate imprecise execution. We utilize an algorithmic transformation that automatically converts approximable regions of code from a von Neumann model to an "analog" neural model. We outline the challenges of taking an analog approach, including restricted-range value encoding, limited precision in computation, circuit inaccuracies, noise, and constraints on supported topologies. We address these limitations with a combination of circuit techniques, a hardware/software interface, neural-network training techniques, and compiler support. The results of this work show that using limited-precision analog circuits for code acceleration, through a neural approach, is both feasible and beneficial over a range of approximation-tolerant, emerging applications including financial analysis, signal processing, robotics, 3D gaming, compression, and image processing. This chapter is based on work presented in ISCA 2014 [6] and IEEE Micro Top Picks [7]. This work is a result of collaboration with Renée St Amant[1], Bradley Thwaites[2], Arjang Hassibi[1], Luis Ceze[3], Doug Burger, and Hadi Esmaeilzadeh[4].

[1]University of Texas at Austin
[2]Georgia Institute of Technoloy
[3]University of Washington
[4]University of California, San Diego

1.2 Introduction

Energy efficiency now fundamentally limits microprocessor performance gains. CMOS scaling no longer provides gains in efficiency commensurate with transistor density increases [1, 8]. As a result, both the semiconductor industry and the research community are increasingly focused on specialized accelerators, which provide large gains in efficiency and performance by restricting the workloads that benefit. The community is facing an "iron triangle"; we can choose any two of performance, efficiency, and generality at the expense of the third. Before the effective end of Dennard scaling, we improved all three consistently for decades. Solutions that improve performance and efficiency, while retaining as much generality as possible, are highly desirable, hence the growing interest in GPGPUs and FPGAs. A growing body of recent work [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] has focused on *approximation* as a strategy for the iron triangle. Many classes of applications can tolerate small errors in their outputs with no discernible loss in *QoR* (Quality of Result). Many conventional techniques in energy-efficient computing navigate a design space defined by the two dimensions of performance and energy, and traditionally trade one for the other. General-purpose approximate computing explores a third dimension—that of error.

Many design alternatives become possible once precision is relaxed. An obvious candidate is the use of analog circuits for computation. However, computation in the analog domain has several major challenges, even when small errors are permissible. First, analog circuits tend to be special purpose, good for only specific operations. Second, the bit widths they can accommodate are smaller than current floating-point standards (i.e. 32/64 bits), since the ranges must be represented by physical voltage or current levels. Another consideration is determining where the boundaries between digital and analog computation lie. Using individual analog operations will not be effective due to the overhead of A/D and D/A conversions. Finally, effective storage of temporary analog results is challenging in

current CMOS technologies. These limitations has made it ineffective to design analog von Neumann processors that can be programmed with conventional languages.

Despite these challenges, the potential performance and energy gains from analog execution are highly attractive. An important challenge is thus to architect designs where a significant portion of the computation can be run in the analog domain, while also addressing the issues of value range, domain conversions, and relative error. Recent work on Neural Processing Units (NPUs) may provide a possible approach [12]. NPU-enabled systems rely on an algorithmic transformation that converts regions of approximable general-purpose code into a neural representation (specifically, multi-layer perceptrons) at compile time. At run-time, the processor invokes the NPU instead of running the original code. NPUs have shown large performance and efficiency gains, since they subsume an entire code region (including all of the instruction fetch, decode, etc., overheads). They have an added advantage in that they convert many distinct code patterns into a common representation that can be run on a single physical accelerator, improving generality.

NPUs may be a good match for mixed-signal implementations for a number of reasons. First, prior research has shown that neural networks can be implemented in analog domain to solve classes of domain-specific problems, such as pattern recognition [20, 21, 22, 23]. Second, the process of invoking a neural network and returning a result defines a clean, coarse-grained interface for D/A and A/D conversion. Third, the compile-time training of the network permits any analog-specific restrictions to be hidden from the programmer. The programmer simply specifies which region of the code can be approximated, without adding any neural-network-specific information. Thus, no additional changes to the programming model are necessary.

In this work we evaluate an NPU design with mixed-signal components and develop a compilation workflow for utilizing the mixed-signal NPU for code acceleration. The goal of this study is to investigate challenges and define potential solutions to enable effective mixed-signal NPU execution. The objective is to both bound application error to

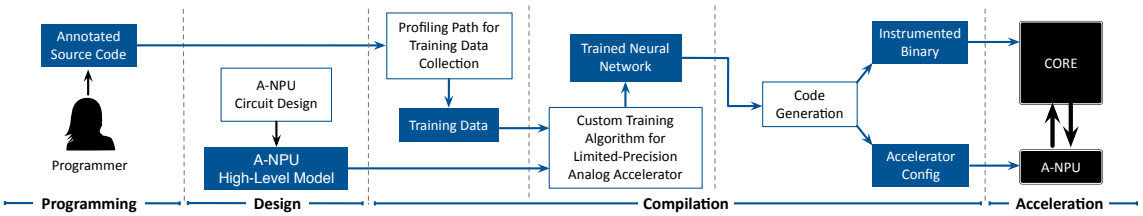


Figure 1.1: Framework for using limited-precision analog computation to accelerate code written in conventional languages.

sufficiently low levels and achieve worthwhile performance or efficiency gains for general-purpose approximable code. This study makes the following four findings:

1. Due to range limitations, it is necessary to limit the scope of the analog execution to a single neuron; inter-neuron communication should be in the digital domain.

2. Again due to range issues, there is an interplay between the bit widths (inputs and weights) that neurons can use and the number of inputs that they can process. We found that the best design limited weights and inputs to eight bits, while also restricting the number of inputs to each neuron to eight. The input count limitation restricts the topological space of feasible neural networks.

3. We found that using a customized continuous-discrete learning method (CDLM) [24], which accounts for limited-precision computation at training time, is necessary to reduce error due to analog range limitations.

4. Given the analog-imposed topology restrictions, we found that using a Resilient Back Propagation (RPROP) [25] training algorithm can further reduce error over a conventional backpropagation algorithm.

We found that exposing the analog limitations to the compiler allowed for the compensation of these shortcomings and produced sufficiently accurate results. The latter three findings were all used at training time; we trained networks at compile time using 8-bit values, topologies restricted to eight inputs per neuron, plus RPROP and CDLM for training. Using these techniques together, we were able to bound error on all applications but one to a 10% limit, which is commensurate with entirely digital approximation techniques. The

average time required to compute a neural result was 3.3× better than a previous digital implementation with an additional energy savings of 12.1×. The performance gains result in an average full-application-level improvement of 3.7× and 23.3× in performance and energy-delay product, respectively. This study shows that using limited-precision analog circuits for code acceleration, by converting regions of imperative code to neural networks and exposing the circuit limitations to the compiler, is both feasible and advantageous. While it may be possible to move more of the accelerator architecture design into the analog domain, the current mixed-signal design performs well enough that only 3% and 46% additional improvements in application-level energy consumption and performance are possible with improved accelerator designs. However, improving the performance of the analog NPU may lead to higher overall performance gains.

1.3 Overview and Background

Programming. We use a similar programming model as described in [12] to enable programmers to mark error-tolerant regions of code as candidates for transformation using a simple keyword, `approximable`. Explicit annotation of code for approximation is a common practice in approximate programming languages [26, 27]. A candidate region is an error-tolerant function of any size, containing function calls, loops, and complex control flow. Frequently executed functions provide a greater opportunity for gains. In addition to error tolerance, the candidate function must have well-defined inputs and outputs. That is, the number of inputs and outputs must be known at compile time. Additionally, the code region must not read any data other than its inputs, nor affect any data other than its outputs. No major changes are necessary to the programming language beyond adding the approximable keyword.

Exposing analog circuits to the compiler. Although an analog accelerator presents the opportunity for gains in efficiency over a digital NPU, it suffers from reduced accuracy and flexibility, which results in limitations on possible network topologies and limited-

precision computation, potentially resulting in a decreased range of applications that can utilize the acceleration. These shortcomings at the hardware level, however, can be exposed as a high-level model and considered in the training phase.

Four characteristics need to be exposed: (1) limited precision for input and output encoding, (2) limited precision for encoding weights, (3) the behavior of the activation function (sigmoid), (4) limited feasible neural topologies. Other low-level circuit behavior such as response to noise can also be exposed to the compiler. Section 1.6 describes this necessary hardware/software interface in more detail.

Analog neural accelerator circuit design. To extract the high-level model for the compiler and to be able to accelerate execution, we design a mixed-signal neural hardware for multilayer perceptrons. The accelerator must support a large enough variety of neural network topologies to be useful over a wide range of applications. As we will show, each applications requires a different topology for the neural network that is replacing its approximable regions of code. Section 1.5 describes a candidate A-NPU circuit design, and outlines the challenges and tradeoffs present with an analog implementation.

Compiling for analog neural hardware. The compiler aims to mimic approximable regions of code with neural networks that can be executable on the mixed-signal accelerator. While considering the limitation of the analog hardware, the compiler searches the topology space of the neural networks and selects and trains a neural network to produce outputs comparable to those produced by the original code segment.

1) Profile-driven training data collection. During a profiling stage, the compiler runs the application with representative profiling inputs and collects the inputs and outputs to the approximable code region. This step provides the training data for the rest of the compilation workflow.

2) Training for a limited-precision A-NPU. This stage is where our compilation workflow significantly deviates from the framework presented in [12] that targets digital NPUs. The compiler uses the collected training data to train a multilayer perceptron neural network, choosing a network topology, i.e. the number of neurons and their connectivity, and taking a gradient descent approach to find the synaptic weights of the network while minimizing the error with respect to the training data. This compilation stage does a neural topology search to find the smallest neural network that (a) adheres to the organization of the analog circuit and (b) delivers acceptable accuracy at the application level. The network training algorithm, which finds optimal synaptic weights, uses a combination of a resilient back propagation algorithm, RPROP [25], that we found to outperform traditional back propagation for restricted network topologies, and a continuous-discrete learning method, CDLM [24], that attempts to correct for error due to limited-precision computation. Section 1.6 describes these techniques that address analog limitations.

3) Code generation for hybrid analog-digital execution. Similar to prior work [12], in the code generation phase, the compiler replaces each instance of the original program code with code that initiates a computation on the analog neural accelerator. Similar ISA extensions are used to specify the neural network topology, send input and weight values to the A-NPU, and retrieve computed outputs from the A-NPU.

1.4 Analog Circuits for Neural Computation

This section describes how analog circuits can perform the computation of neurons in multi-layer perceptrons, which are widely used neural networks. We also discuss, at a high-level, how limitations of the analog circuits manifest in the computation. We explain how these restrictions are exposed to the compilation framework. The next section presents a concrete design for the analog neural accelerator.

As Figure 1.2a illustrates, each neuron in a multi-layer perceptron takes in a set of inputs

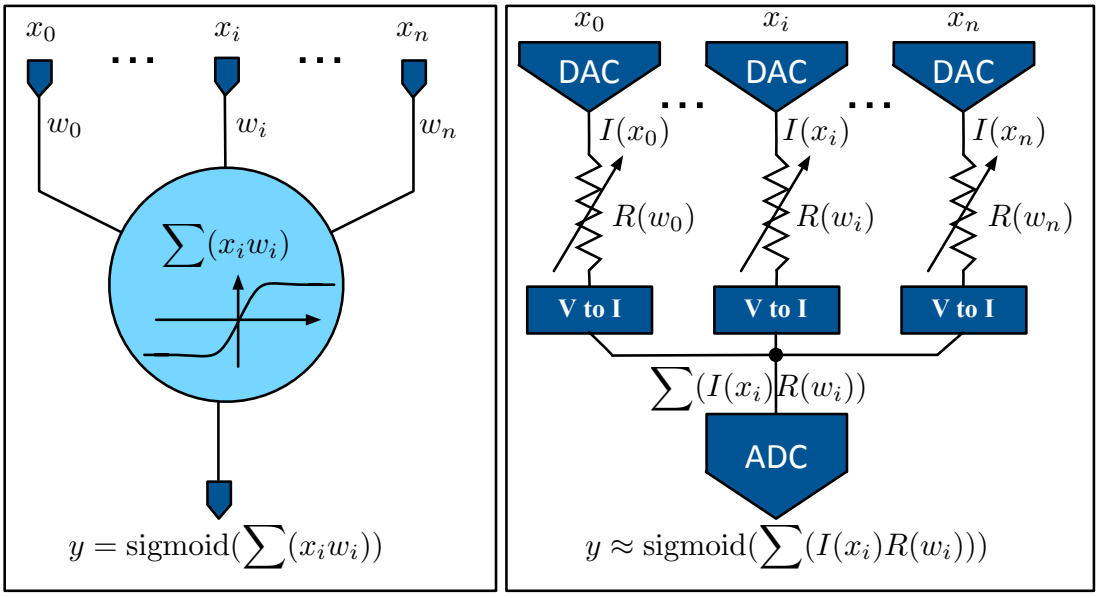


Figure 1.2: One neuron and its conceptual analog circuit.

(x_i) and performs a weighted sum of those input values ($\sum_i x_i w_i$). The weights (w_i) are the result of training the neural network on training data (compile time) and are constant during the recall phase (execution time). After the summation stage, which produces a linear combination of the weighted inputs, the neuron applies a nonlinearity function, *sigmoid*, to the result of summation.

Figure 1.2b depicts a conceptual analog circuit that performs the three necessary operations of a neuron: (1) scaling inputs by weight ($x_i w_i$), (2) summing the scaled inputs ($\sum_i x_i w_i$), and (3) applying the nonlinearity function (*sigmoid*). This conceptual design first encodes the digital inputs (x_i) as analog current levels ($I(x_i)$). Then, these current levels pass through a set of variable resistances whose values ($R(w_i)$) are set proportional to the corresponding weights (w_i). The voltage level at the output of each resistance ($I(x_i)R(w_i)$), is proportional to $x_i w_i$. These voltages are then converted to currents that can be summed quickly according to Kirchhoff's current law (KCL). Analog circuits only operate linearly within a small range of voltage and current levels, outside of which the transistors enter saturation mode with IV characteristics similar in shape to a non-linear sigmoid function. Thus, at the high level, the non-linearity is naturally applied to the result of summation when the final voltage reaches the analog-to-digital converter (ADC). Compared to a digital

implementation of a neuron, which requires multipliers, adder trees and sigmoid lookup tables, the analog implementation leverages the physical properties of the circuit elements and is orders of magnitude more efficient. However, it operates in limited ranges and therefore offers limited precision.

Analog-digital boundaries. The conceptual design in Figure 1.2b draws the analog-digital boundary at the level of an algorithmic neuron. As we will discuss, the analog neural accelerator will be a composition of these analog neural units (ANUs). However, an alternative design, primarily optimizing for efficiency, may lay out the entirety of a neural network with only analog components, limiting the D-to-A and A-to-D conversions to the inputs and outputs of the neural network and not the individual neurons. The overhead of conversions in the ANUs significantly limits the potential efficiency gains of an analog approach toward neural computation. However, there is a tradeoff between efficiency, reconfigurability (generality), and accuracy in analog neural hardware design. Pushing more of the implementation into the analog domain gains efficiency at the expense of flexibility, limiting the scope of supported network topologies and, consequently, limiting potential network accuracy. The NPU approach targets *code approximation*, rather than typical, simpler neural tasks, such as recognition and prediction, and imposes higher accuracy requirements. The main challenge is to manage this tradeoff to achieve acceptable accuracy for code acceleration, while delivering higher performance and efficiency when analog neural circuits are used for *general-purpose code acceleration*.

As prior work [12] has shown and we corroborate, regions of code from different applications require different topologies of neural networks. While a holistically analog neural hardware design with fixed-wire connections between neurons may be efficient, it effectively provides a fixed topology network, limiting the scope of applications that can benefit from the neural accelerator, as the optimal network topology varies with application. Additionally, routing analog signals among neurons and the limited capability of analog

circuits for buffering signals negatively impacts accuracy and makes the circuit susceptible to noise. In order to provide additional flexibility, we set the digital-analog boundary in conjunction with an algorithmic, sigmoid-activated neuron. where a set of digital inputs and weights are converted to the analog domain for efficient computation, producing a digital output that can be accurately routed to multiple consumers. We refer to this basic computation unit as an analog neural unit, or ANU. ANUs can be composed, in various physical configurations, along with digital control and storage, to form a reconfigurable mixed-signal NPU, or A-NPU.

One of the most prevalent limitations in analog design is the *bounded range* of currents and voltages within which the circuits can operate effectively. These range limitations restrict the bit-width of input and weight values and the network topologies that can be computed accurately and efficiently. We expose these limitations to the compiler and our custom training algorithm and compilation workflow considers these restrictions when searching for optimal network topologies and training neural networks. As we will show, one of the insights from this work is that even with limited bit-width (≤ 8), and a restricted neural topology, many general-purpose approximate applications achieve acceptable accuracy and significantly benefit from mixed-signal neural acceleration.

Value representation and bit-width limitations. One of the fundamental design choices for an ANU is the bit-width of inputs and weights. Increasing the number of bits results in an exponential increase in the ADC and DAC energy dissipation and can significantly limit the benefits from analog acceleration. Furthermore, due to the fixed range of voltage and current levels, increasing the number of bits translates to quantizing this fixed value range to fine granularities that practical ADCs can not handle. In addition, the fine granularity encoding makes the analog circuit significantly more susceptible to noise, thermal, voltage, current, and process variations. In practice, these non-ideal effects can adversely affect the final accuracy when more bit-width is used for weights and inputs. We design our ANUs

such that the granularity of the voltage and current levels used for information encoding is to a large degree robust to variations and noise.

Topology restrictions. Another important design choice is the *number of inputs* in the ANU. Similar to bit-width, increasing the number of ANU inputs translates to encoding a larger value range in a bounded voltage and current range, which, as discussed, becomes impractical. There is a tradeoff between accuracy and efficiency in choosing the number ANU inputs. The larger the number of inputs, the larger the number of multiply and add operations that can be done in parallel in the analog domain, increasing efficiency. However, due to the bounded range of voltage and currents, increasing the number of inputs requires decreasing the number of bits for inputs and weights. Through circuit-level simulations, we empirically found that limiting the number of inputs to eight with 8-bit inputs and weights strikes a balance between accuracy and efficiency. A digital implementation does not impose such restrictions on the number of inputs to the hardware neuron and it can potentially compute arbitrary topologies of neural networks. However, this unique ANU limitation restricts the topology of the neural network that can run on the analog accelerator. Our customized training algorithm and compilation workflow takes into account this topology limitation and produces neural networks that can be computed on our mixed-signal accelerator.

Non-ideal sigmoid. The saturation behavior of the analog circuit that leads to sigmoid-like behavior after the summation stage represents an approximation of the ideal sigmoid. We measure this behavior at the circuit level and expose it to the compiler and the training algorithm.

1.5 Mixed-Signal Neural Accelerator (A-NPU)

This section describes a concrete ANU design and the mixed-signal, neural accelerator, A-NPU.

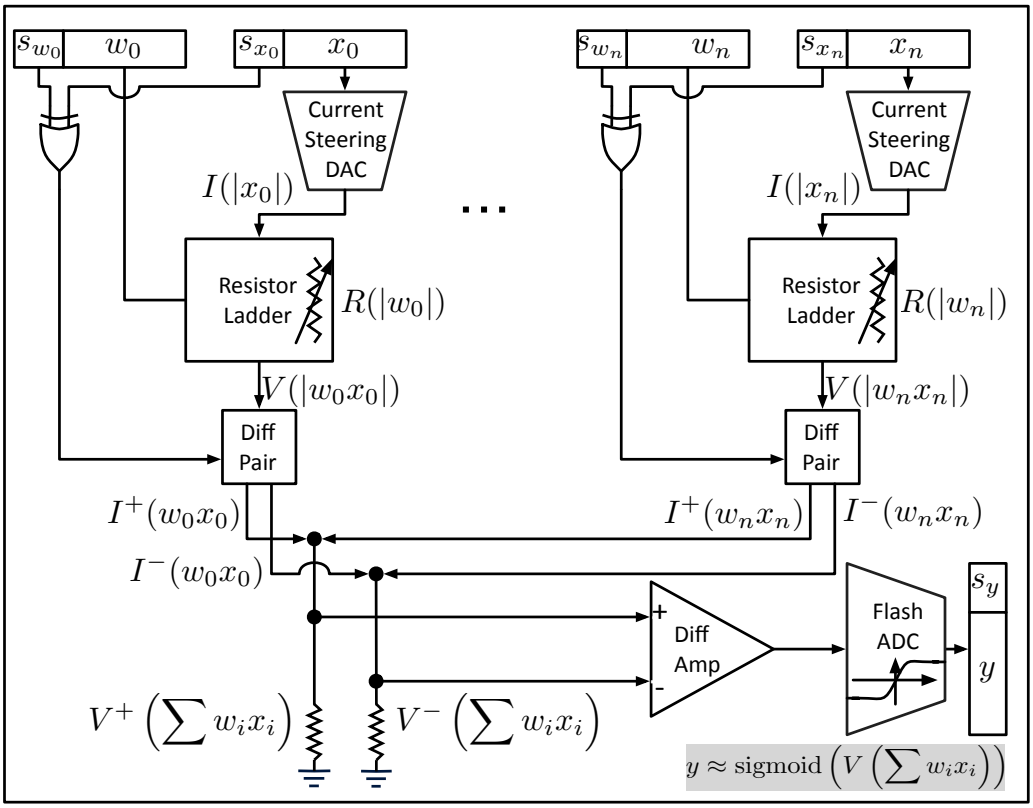


Figure 1.3: A single analog neuron (ANU).

1.5.1 ANU Circuit Design

Figure 1.3 illustrates the design of a single analog neuron (ANU). The ANU performs the computation of one neuron, or $y \approx sigmoid(\sum_i w_i x_i)$. We place the analog-digital boundary at the ANU level, with computation in the analog domain and storage in the digital domain. Digital input and weight values are represented in sign-magnitude form. In the figure, s_{w_i} and s_{x_i} represent the sign bits and w_i and x_i represent the magnitude. Digital input values are converted to the analog domain through current-steering DACs that translate digital values to analog currents. Current-steering DACs are used for their speed and simplicity. In Figure 1.3, $I(|x_i|)$ is the analog current that represents the magnitude of the input value, x_i. Digital weight values control resistor-string ladders that create a variable resistance depending on the magnitude of each weight ($R(|w_i|)$) . We use a standard resistor ladder thats consists of a set of resistors connected to a tree-structured set of switches. The digital weight bits control the switches, adjusting the effective resistance, $R(|w_i|)$, seen by the

input current ($I(|x_i|)$). These variable resistances scale the input currents by the digital weight values, effectively multiplying each input magnitude by its corresponding weight magnitude. The output of the resistor ladder is a voltage: $V(|w_i x_i|) = I(|x_i|) \times R(|w_i|)$. The resistor network requires 2^m resistors and approximately 2^{m+1} switches, where m is the number of digital weight bits. This resistor ladder design has been shown to work well for $m \leq 10$. Our circuit simulations show that only minimally sized switches are necessary.

$V(|w_i x_i|)$, as well as the XOR of the weight and input sign bits, feed to a differential pair that converts voltage values to two differential currents ($I^+(w_i x_i)$, $I^-(w_i x_i)$) that capture the sign of the weighted input. These differential currents are proportional to the voltage applied to the differential pair, $V(|w_i x_i|)$. If the voltage difference between the two gates is kept small, the current-voltage relationship is linear, producing $I^+(w_i x_i) = \frac{I_{bias}}{2} + \Delta I$ and $I^-(w_i x_i) = \frac{I_{bias}}{2} - \Delta I$. Resistor ladder values are chosen such that the gate voltage remains in the range that produces linear outputs, and consequently a more accurate final result. Based on the sign of the computation, a switch steers either the current associated with a positive value or the current associated with a negative value to a single wire to be efficiently summed according to Kirchhoff's current law. The alternate current is steered to a second wire, retaining differential operation at later design stages. Differential operation combats environmental noise and increases gain, the later being particularly important for mitigating the impact of analog range challenges at later stages.

Resistors convert the resulting pair of differential currents to voltages, $V^+(\sum_i w_i x_i)$ and $V^-(\sum_i w_i x_i)$, that represent the weighted sum of the inputs to the ANU. These voltages are used as input to an additional amplification stage (implemented as a current-mode differential amplifier with diode-connected load). The goal of this amplification stage is to significantly magnify the input voltage *range of interest* that maps to the linear output region of the desired sigmoid function. Our experiments show that neural networks are sensitive to the steepness of this non-linear function, losing accuracy with shallower, non-linear activation functions. This fact is relevant for an analog implementation because

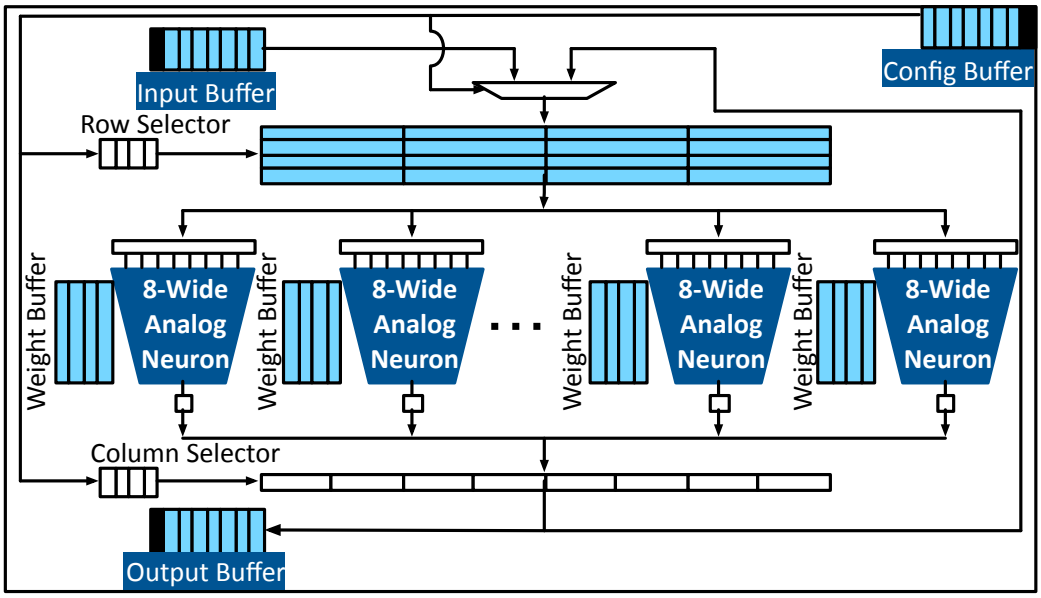


Figure 1.4: Mixed-signal neural accelerator, A-NPU. Only four of the ANUs are shown. Each ANU processes eight 8-bit inputs.

steeper functions increase range pressure in the analog domain, as a small range of interest must be mapped to a much larger output range in accordance with ADC input range requirements for accurate conversion. We magnify this range of interest, choosing circuit parameters that give the required gain, but also allowing for saturation with inputs outside of this range.

The amplified voltage is used as input to an ADC that converts the analog voltage to a digital value. We chose a flash ADC design (named for its speed), which consists of a set of reference voltages and comparators [28, 29]. The ADC requires 2^n comparators, where n is the number of digital output bits. Flash ADC designs are capable of converting 8 bits at a frequency on the order of one GHz. We require 2–3 mV between ADC quantization levels for accurate operation and noise tolerance. Typically, ADC reference voltages increase linearly; however, we use a non-linearly increasing set of reference voltages to capture the behavior of a sigmoid function, which also improves the accuracy of the analog sigmoid.

1.5.2 Reconfigurable Mixed-Signal A-NPU

We design a reconfigurable mixed-signal A-NPU that can perform the computation of a wide variety of neural topologies since each requires a different topology. Figure 1.4 illustrates the A-NPU design with some details omitted for clarity. The figure shows four ANUs while the actual design has eight. The A-NPU is a time-multiplexed architecture where the algorithmic neurons are mapped to the ANUs based on a static scheduling algorithm, which is loaded to the A-NPU before invocation. The multi-layer perceptron consists of layers of neurons, where the inputs of each layer are the outputs of the previous layer. The ANU starts from the input layer and performs the computations of the neurons layer by layer. The Input Buffer always contains the inputs to the neurons, either coming from the processor or from the previous layer computation. The Output Buffer, which is a single entry buffer, collects the outputs of the ANUs. When all of its columns are computed, the results are pushed back to the Input Buffer to enable calculation of the next layer. The Row Selector determines which entry of the input buffer will be fed to the ANUs. The output of the ANUs will be written to a single-entry output buffer. The Column Selector determines which column of the output buffer will be written by the ANUs. These selectors are FIFO buffers whose values are part of the preloaded A-NPU configuration. All the buffers are digital SRAM structures.

Each ANU has eight inputs. As shown in Figure 1.4, each A-NPU is augmented with a dedicated weight buffer, storing the 8-bit weights. The weight buffers simultaneously feed the weights to the ANUs. The weights and the order in which they are fed to the ANUs are part of the A-NPU configuration. The Input Buffer and Weight Buffers synchronously provide the inputs and weights for the ANUs based on the pre-loaded order.

A-NPU configuration. During code generation, the compiler produces an A-NPU configuration that constitutes the weights and the schedule. The static A-NPU scheduling algorithm first assigns an order to the neurons of the neural network, in which the neurons

will be computed in the ANUs. The scheduler then takes the following steps for each layer of the neural network: (1) Assign each neuron to one of the ANUs. (2) Assign an order to neurons. (3) Assign an order to the weights. (4) Generate the order for inputs to feed the ANUs. (5) Generate the order in which the outputs will be written to the Output Buffer. The scheduler also assigns a unique order for the inputs and outputs of the neural network in which the core communicates data with the A-NPU.

1.5.3 Architectural interface for A-NPU

We adopt the same FIFO-based architectural interface through which a digital NPU communicates with the processor [12]. The A-NPU is tightly integrated to the pipeline. The processor only communicates with the ANUs through the Input, Output, Config FIFOs. The processor ISA is extended with special instructions that can enqueue and dequeue data from these FIFOs as shown in Figure 1.4. When a data value is queued/dequeued to/from the Input/Output FIFO, the A-NPU converts the values to the appropriate representation for the A-NPU/processor.

1.6 Compilation for Analog Acceleration

As Figure 2.3 illustrates, the compilation for A-NPU execution consists of three stages: (1) profile-driven data collection, (2) training for a limited-precision A-NPU, and (3) code generation for hybrid analog-digital execution. In the profile-driven data collection stage, the compiler instruments the application to collect the inputs and outputs of approximable functions. The compiler then runs the application with representative inputs and collects the inputs and their corresponding outputs. These input-output pairs constitute the training data. Section 1.5 briefly discussed ISA extensions and code generation. While compilation stages (1) and (3) are similar to the techniques presented for a digital implementation [12], the training phase is unique to an analog approach, accounting for analog-imposed, topology restrictions and adjusting weight selection to account for limited-precision computa-

tion.

Hardware/software interface for exposing analog circuits to the compiler. As we discussed in Section 1.4, we expose the following analog circuit restrictions to the compiler through a hardware/software interface that captures the following circuit characteristics: (1) input bit-width limitations, (2) weight bit-width limitations, (3) limited number of inputs to each analog neuron (topology restriction), and (4) the non-ideal shape of the analog sigmoid. The compiler internally constructs a high-level model of the circuit based on these limitations and uses this model during the neural topology search and training with the goal of limiting the impact of inaccuracies due to an analog implementation.

Training for limited bit widths and analog computation. Traditional training algorithms for multi-layered perceptron neural networks use a gradient descent approach to minimize the average network error, over a set of training input-output pairs, by backpropagating the output error through the network and iteratively adjusting the weight values to minimize that error. Traditional training techniques, however, that do not consider limited-precision inputs, weights, and outputs perform poorly when these values are saturated to adhere to the bit-width requirements that are feasible for an implementation in the analog domain. Simply limiting weight values during training is also detrimental to achieving quality outputs because the algorithm does not have sufficient precision to converge to a quality solution.

To incorporate bit-width limitations into the training algorithm, we use a customized continuous-discrete learning method (CDLM) [24]. This approach takes advantage of the availability of full-precision computation at training time and then adjusts slightly to optimize the network for errors due to limited-precision values. In an initial phase, CDLM first trains a fully-precise network according to a standard training algorithm, such as backpropagation [30]. In a second phase, it discretizes the input, weight, and output values according the the exposed analog specification. The algorithm calculates the new

Table 1.1: The evaluated benchmarks, characterization of each offloaded function, training data, and the trained neural network.

Benchmark Name	Description	Type	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Evaluation Input Set	Training Input Set	Neural Network Topology	Fully Digital NN MSE	Analog NN MSE (8-bit)	Application Error Metric	Fully Digital Error	A-NPU Error
blackscholes	Mathematical model of a financial market	Financial Analysis	5	0	5	309	4,096 Data Point from PARSEC	16,384 Data Point from PARSEC	6 → 8 → 8 → 1	0.000011	0.00228	Average Relative Error	6.02%	10.2%
fft	Radix-2 Cooley-Tukey fast fourier	Signal-Processing	2	0	0	34	2,048 Random Floating Point Numbers	32,768 Random Floating Point Numbers	1 → 4 → 4 → 2	0.00002	0.00194	Average Relative Error	2.75%	4.1%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	4	0	0	100	10,000 (x, y) Random Coordinates	10,000 (x, y) Random Coordinates	2 → 8 → 2	0.000341	0.00467	Average Relative Error	6.2%	9.4%
jmeint	Triangle intersection detection	3D Gaming	32	0	23	1,079	10,000 Random Pairs of 3D Triangle Coordinates	10,000 Random Pairs of 3D Triangle Coordinates	18 → 32 → 8 → 2	0.05235	0.06729	Miss Rate	17.68%	19.7%
jpeg	JPEG encoding	Compression	3	4	0	1,257	220x220-Pixel Color Image	Three 512x512-Pixel Color Image	64 → 16 → 8 → 64	0.0000156	0.0000325	Image Diff	5.48%	8.4%
kmeans	K-means clustering	Machine Learning	1	0	0	26	220x220-Pixel Color Image	50,000 Pairs of Random (r, g, b) Values	6 → 8 → 4 → 1	0.00752	0.009589	Image Diff	3.21%	7.3%
sobel	Sobel edge detector	Image Processing	3	2	1	88	220x220-Pixel Color Image	One 512x512 Pixel Color Image	9 → 8 → 1	0.000782	0.00405	Image Diff	3.89%	5.2%

error and backpropagates that error through the fully-precise network using full-precision computation and updates the weight values according to the algorithm also used in stage 1. This process repeats, backpropagating the 'discrete' errors through a precise network. The original CDLM training algorithm was developed to mitigate the impact of limited-precision weights. We customize this algorithm by incorporating the input bit-width limitation and the output bit-width limitation in addition to limited weight values. Additionally, this training scheme is advantageous for an analog implementation because it is general enough to also make up for errors that arise due to an analog implementation, such as a non-ideal sigmoid function and any other analog non-ideality that behaves consistently. In essence, after one round of full-precision training, the compiler models an analog-like version of the network. A second, CDLM-based training pass adjusts for these analog-imposed errors, enabling the inaccurate and limited A-NPU as an option for a beneficial NPU implementation by maintaining acceptable accuracy and generality.

Training with topology restrictions. In addition to determining weight values for a given network topology, the compiler searches the space of possible topologies to find an optimal network for a given approximable code region. Conventional multi-layered perceptron networks are fully connected, i.e. the output of each neuron in one layer is routed to the input of each neuron in the following layer. However, analog range limitations restrict the number of inputs that can be computed in a neuron (eight in our design). Consequently, network connections must be limited, and in many cases, the network can not be fully connected.

We impose the circuit restriction on the connectivity between the neurons during the topology search and we use a simple algorithm guided by the mean-squared error of the network to determine the best topology given the exposed restriction. The error evaluation uses a typical cross-validation approach: the compiler partitions the data collected during profiling into a *training set*, 70% of the data, and a *test set*, the remaining 30%. The topology search algorithm trains many different neural-network topologies using the training set and chooses the one with the highest accuracy on the test set and the lowest latency on the A-NPU hardware (prioritizing accuracy). The space of possible topologies is large, so we restrict the search to neural networks with at most two hidden layers. We also limit the number of neurons per hidden layer to powers of two up to 32. The numbers of neurons in the input and output layers are predetermined based on the number of inputs and outputs in the candidate function.

To further improve accuracy, and compensate for topology-restricted networks, we utilize a Resilient Back Propagation (RPROP) [25] training algorithm as the base training algorithm in our CDLM framework. During training, instead of updating the weight values based on the backpropagated error (as in conventional backpropagation [30]), the RPROP algorithm increases or decreases the weight values by a predefined value based on the sign of the error. Our investigation showed that RPROP significantly outperforms conventional backpropagation for the selected network topologies, requiring only half of the number of training epochs as backpropagation to converge on a quality solution. The main advantage of the application of RPROP training to an analog approach to neural computing is its robustness to the sigmoid function and topology restrictions imposed by the analog design. Backpropagation, for example, is extremely sensitive to the steepness of the sigmoid function, and allowing for a variety of steepness levels in a fixed, analog implementation is challenging. Additionally, backpropagation performs poorly with a shallow sigmoid function. The requirement of a steep sigmoid function exacerbates analog range challenges, possibly making the implementation infeasible. RPROP tolerates a more shallow sigmoid

activation steepness and performs consistently utilizing a constant activation steepness over all applications. Our RPROP-based, customized CDLM training phase requires 5000 training epochs, with the analog-based CDLM phase adding roughly 10% to the training time of the baseline training algorithm.

1.7 Evaluations

Cycle-accurate simulation and energy modeling. We use the MARSSx86 x86-64 cycle-accurate simulator [31] to model the performance of the processor. The processor is modeled after a single-core Intel Nehalem to evaluate the performance benefits of A-NPU acceleration over an aggressive out-of-order architecture[5]. We extended the simulator to include ISA-level support for A-NPU queue and dequeue instructions. We also augmented MARSSx86 with a cycle-accurate simulator for our A-NPU design and an 8-bit, fixed-point D-NPU with eight processing engines (PEs) as described in [12]. We use GCC v4.7.3 with -o3 to enable compiler optimization. The baseline in our experiments is the benchmark run solely on the processor without neural transformation. We use McPAT [32] for processor energy estimations. We model the energy of an 8-bit, fixed-point D-NPU using results from McPAT, CACTI 6.5 [33], and [34] to estimate its energy. Both the D-NPU and the processor operate at 3.4GHz at 0.9 V, while the A-NPU is clocked at one third of the digital clock frequency, 1.1GHz at 1.2 V, to achieve acceptable accuracy.

Circuit design for ANU. We built a detailed transistor-level SPICE model of the analog neuron, ANU. We designed and simulated the 8-bit, 8-input ANU in the Cadence Analog Design Environment using predictive technology models at 45 nm [35]. We ran detailed Spectre SPICE simulations to understand circuit behavior and measure ANU energy con-

[5]**Processor:** Fetch/Issue Width: 4/5, INT ALUs/FPUs: 6/6, Load/Store FUs: 1/1, ROB Entries: 128, Issue Queue Entries: 36, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48 KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Load/Store Queue Entries: 48/48, Dependence Predictor: 4096-entry Bloom Filter, ITLB/DTLB Entries: 128/256 **L1:** 32 KB Instruction, 32 KB Data, Line Width: 64 bytes, 8-Way, Latency: 3 cycles **L2:** 256 KB, Line Width: 64 bytes, 8-Way, Latency: 6 cycles **L3:** 2 MB, Line Width 64 bytes, 16-Way, Latency: 27 cycles **Memory Latency:** 50 ns

Table 1.2: Area estimates for the analog neuron (ANU).

Sub-circuit	Area
8×8-bit DAC	3,096 T*
8×Resistor Ladder (8-bit weights)	4,096 T + 1 KΩ (≈ 450T)
8×Differential Pair	48 T
I-to-V Resistors	20 KΩ (≈ 30 T)
Differential Amplifier	244 T
8-bit ADC	2550 T + 1 KΩ (≈ 450 T)
Total	≈ 10,964 T

*Transistor with width/length = 1

sumption. We used CACTI to estimate the energy of the A-NPU buffers. Evaluations consider all A-NPU components, both digital and analog. For the analog parts, we used direct measurements from the transistor-level SPICE simulations. For SRAM accesses, we used CACTI. We built an A-NPU cycle-accurate simulator to evaluate the performance improvements. Similar to McPAT, we combined simulation statistics with measurements from SPICE and CACTI to calculate A-NPU energy. To avoid biasing our study toward analog designs, all energy and performance comparisons are to an 8-bit, fixed-point D-NPU (8-bit inputs/weights/multiply-adders). For consistency with the available McPAT model for the baseline processor, we used McPAT and CACTI to estimate D-NPU energy. Even though we do not have a fabrication-ready layout for the design, in Table 1.2, we provide an estimate of the ANU area in terms of number of transistors. T denotes a transistor with $\frac{width}{length} = 1$. As shown, each ANU (which performs eight, 8-bit analog multiply-adds in parallel followed by a sigmoid) requires about 10,964 transistors. An equivalent digital neuron that performs eight, 8-bit multiply-adds and a sigmoid would require about 72,456 T from which 56,000 T are for the eight, 8-bit multiply-adds and 16,456 T for the sigmoid lookup. With the same compute capability, the analog neuron requires 6.6× fewer transistors than its equivalent digital implementation.

Benchmarks. We use the benchmarks in [12] and add one more, blackscholes. These benchmarks represent a diverse set of application domains, including financial analysis, signal processing, robotics, 3D gaming, compression, image processing. Table 1.1 summarizes information about each benchmark: application domain, target code, neural-network

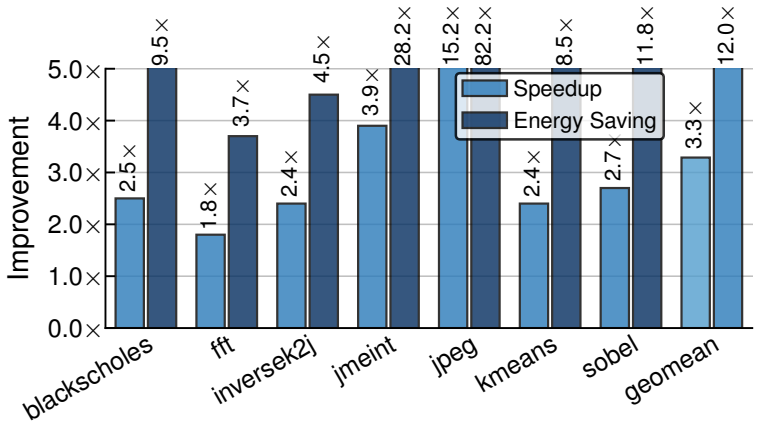
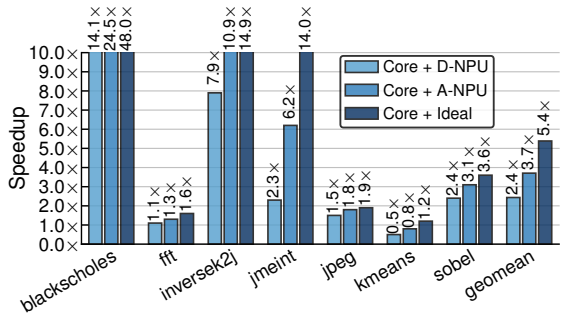


Figure 1.5: A-NPU with 8 ANUs vs. D-NPU with 8 PEs.

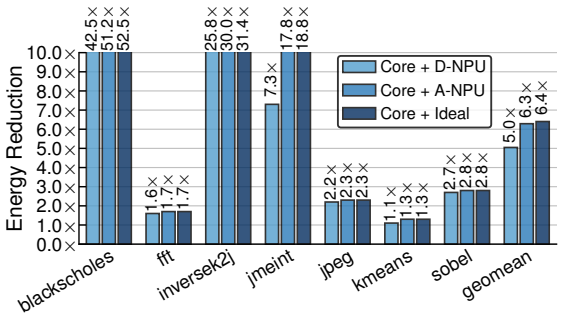
topology, training/test data and final application error levels for fully-digital neural networks and analog neural networks using our customized RPROP-based CDLM training algorithm. The neural networks were trained using either typical program inputs, such as sample images, or a limited number of random inputs. Accuracy results are reported using an independent data set, e.g, an input image that is different than the image used during training. Each benchmark requires an application-specific error metric, which is used in our evaluations. *As shown in Table 1.1, each application benefits from a different neural network topology, so the ability to reconfigure the A-NPU is critical.*

A-NPU vs 8-bit D-NPU. Figure 1.5 shows the average energy improvement and speedup for one invocation of an A-NPU over one invocation of an 8-bit D-NPU, where the A-NPU is clocked at $\frac{1}{3}$ the D-NPU frequency. On average, the A-NPU is 12.1× more energy efficient and 3.3× faster than the D-NPU. While consuming significantly less energy, the A-NPU can perform 64 multiply-adds in parallel, while the D-NPU can only perform eight.

This energy-efficient, parallel computation explains why jpeg–with the largest neural network (64→16→8→64)–achieves the highest energy and performance improvements, 82.2× and 15.2×, respectively. The larger the network, the higher the benefits from A-NPU. Compared to a D-NPU, an A-NPU offers a higher level of parallelism with low energy cost that can potentially enable using larger neural networks to replace more complicated code.



(a) Whole application speedup.



(b) Whole application energy saving.

	blackscholes	fft	inversek2j	jmeint	jpeg	kmeans	sobel
Percentage Instructions Subsumed	97.2%	67.4%	95.9%	95.1%	56.3%	29.7%	57.1%

(c) % dynamic instructions subsumed.

Figure 1.6: Whole application speedup and energy saving with D-NPU, A-NPU, and an Ideal NPU that consumes zero energy and takes zero cycles for neural computation.

Whole application speedup and energy savings. Figure 1.6 shows the whole application speedup and energy savings when the processor is augmented with an 8-bit, 8-PE D-NPU, our 8-ANU A-NPU, and an ideal NPU, which takes zero cycles and consumes zero energy. Figure 1.6c shows the percentage of dynamic instructions subsumed by the neural transformation of the candidate code. The results show, following the Amdahl's Law, that the larger the number of dynamic instructions subsumed, the larger the benefits from neural acceleration. Geometric mean speedup and energy savings with an A-NPU is 3.7× and 6.3× respectively, which is 48% and 24% better than an 8-bit, 8-PE NPU. Among the benchmarks, kmeans sees slow down with D-NPU and A-NPU-based acceleration. All benchmarks benefit in terms of energy. The speedup with A-NPU acceleration ranges from 0.8× to 24.5×. The energy savings range from 1.3× to 51.2×. *As the results show, the savings with an A-NPU closely follows the ideal case, and, in terms of "energy", there is little value in designing a more sophisticated A-NPU. This result is due to the fact that the energy cost of executing instructions in the von Neumann, out-of-order pipeline is much higher than performing simple multiply-adds in the analog domain. Using physics laws (Ohm's law for multiplication and Kirchhoff's law for summation) and analog properties of devices to perform computation can lead to significant energy and performance benefits.*

Table 1.3: Error with a floating point D-NPU, A-NPU with ideal sigmoid, and A-NPU with non-ideal sigmoid.

	blackscholes	fft	inversek2j	jmeint	jpeg	kmeans	sobel
Floating Point D-NPU	6.0%	2.7%	6.2%	17.6%	5.4%	3.2%	3.8%
A-NPU + Ideal Sigmoid	8.4%	3.0%	8.1%	18.4%	6.6%	6.1%	4.3%
A-NPU	10.2%	4.1%	9.4%	19.7%	8.4%	7.3	5.2%

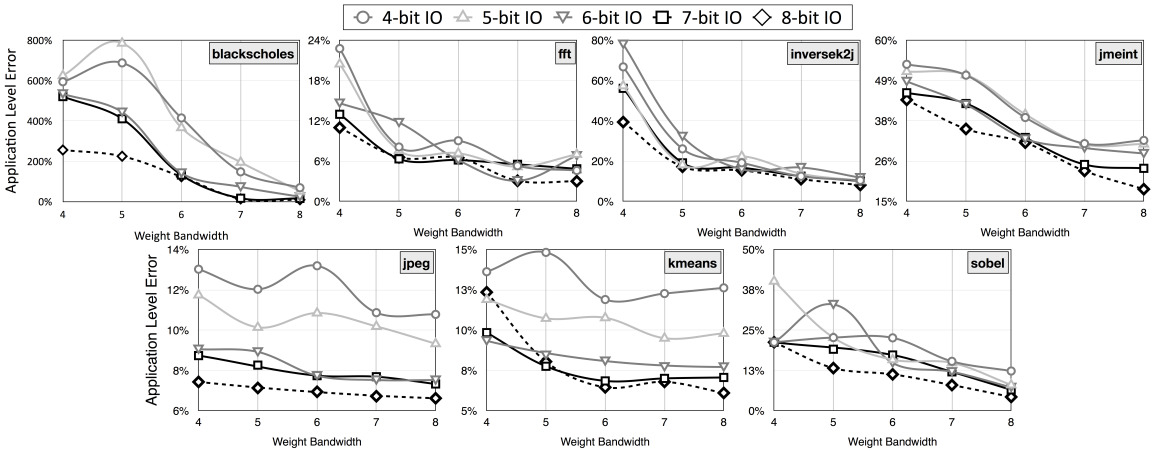


Figure 1.7: Application error with limited bit-width analog neural computation.

Application error. Table 1.3 shows the application-level errors with a floating point D-NPU, A-NPU with ideal sigmoid and our A-NPU which incorporates non-idealities of the analog sigmoid. Except for jmeint, which shows error above 10%, all of the applications show error less than or around 10%. Application average error rates with the A-NPU range from 4.1% to 10.2%. This quality-of-result loss is commensurate with other work on quality trade-offs. Among digital hardware approximation techniques, Truffle [13] and EnerJ [26] shows similar error (3–10%) for some applications and much greater error (above 80%) for others in a moderate configuration. Green [36] has error rates below 1% for some applications but greater than 20% for others. A case study [37] explores manual optimizations of the x264 video encoder that trade off 0.5–10% quality loss. *As expected, the quality-of-results degradation with an A-NPU is more than a floating point D-NPU. However, the quality losses are commensurate with digital approximate computing techniques.*

To study the application-level quality loss in more detail, 1.8 illustrates the CDF (cu-

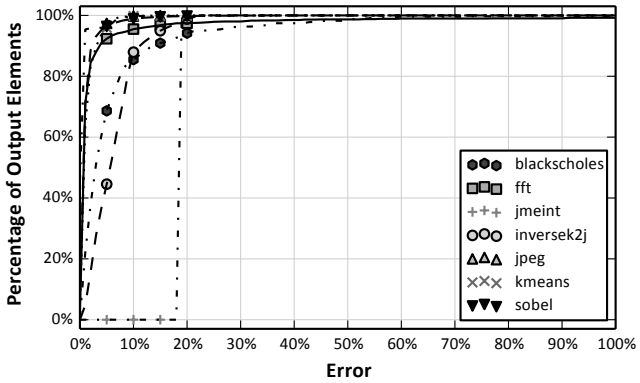
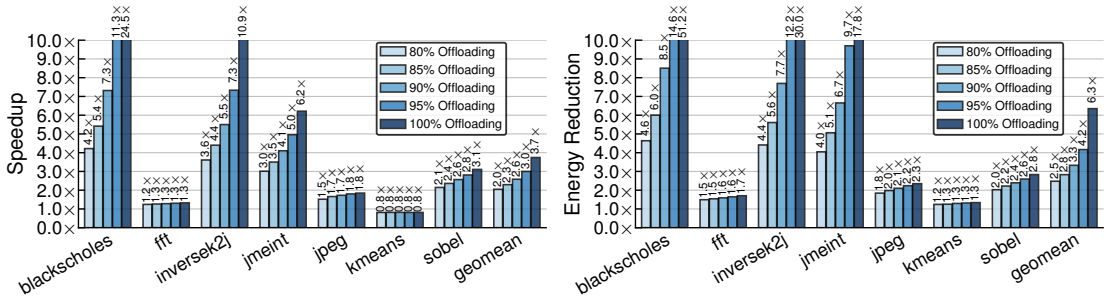


Figure 1.8: CDF plot of application output error. A point (x,y) indicates that y% of the output elements see error $\leq$ x%.

mulative distribution function) plot of final error for each element of application's output. Each benchmark's output consists of a collection of elements–an image consists of pixels; a vector consists of scalars; etc. This CDF reveals the distribution of error among an application's output elements and shows that only a small fraction of the output elements see large quality loss with analog acceleration. *The majority (80% to 100%) of each application's output elements have error less than 10% except for jmeint.*

Exposing circuit limitations to the compiler. Figure 1.7 shows the effect of bit-width restrictions on application-level error, assuming 8 inputs per neuron. As the results suggest, exposing the bit-width limitations and the topology restrictions to the compiler enables our RPROP-based, customized CDLM training algorithm to find and train neural networks that can achieve accuracy levels commensurate with the digital approximation techniques, using only eight bits of precision for inputs, outputs, and weights, and eight inputs to the analog neurons. Several applications show less than 10% error even with fewer than eight bits. *The results shows that there are many applications that can significantly benefit from analog acceleration without significant output quality loss.*

Limited analog acceleration. We examine the effects on the benefits when, due to noise or pathological inputs, only a fraction of the invocations are offloaded to the A-NPU. In this case, the application falls back to the original code for the remaining invocations. Figure 1.9 depicts the application speedup and energy improvement when only 80%, 85%, 90%,



(a) Speedup with limited A-NPU invocations. **(b)** Energy saving for limited A-NPU invocations.

Figure 1.9: Speedup/energy saving with limited A-NPU invocations.

95%, and 100% of the invocations are offloaded to the A-NPU. The results suggest that even limited analog accelerators can provide significant energy and performance improvements.

1.8 Limitations and Considerations

Applicability. Not all applications can benefit from analog acceleration; however, our work shows that there are many that can. More rigorous optimization at the circuit level, as well as broadening the scope off application coverage by continued advancements at the neural transformation step, may provide significant improvements in accuracy and generality.

Other design points. This study evaluates the performance and energy improvements of an A-NPU assuming integration with a modern, high-performance processor. If low-power cores are used instead, we expect to see, and preliminary results confirm, that the performance benefits of an A-NPU increase, and that the energy benefits decrease.

Variability and noise. We designed the circuit with variability and noise as first-order concerns, and we made several design decisions to mitigate them. We limit both the input and weight bit widths, as well as the analog neuron input count to eight to provide quantization margins for variation/noise. We designed the sigmoid circuit one order of magnitude more shallow than the digital implementation to provide additional margins for variation and noise. We used a differential design, which provides resilience to noise by

representing a value by the difference between two signals; as noise affects the pair of nearby signals similarly, the difference between the signals remains intact and the computation correct. Conversion to the digital domain after each analog neuron computation enforces computation integrity and reduces variation/noise susceptibility, while incurring energy and speed overheads. As mentioned in Section 1.7, to further improve the quality of the final result, we can refrain from A-NPU invocations and fall back to the original code as needed. An online noise-monitoring system could potentially limit the invocation of the A-NPU to low-noise situations. Incorporating a quantitative noise model into the training algorithm may improve robustness to analog noise.

Training for variability. A neural approach to approximate computing presents the opportunity to correct for certain types of analog-imposed inaccuracy, such as process variation, non-linearity, and other forms of non-ideality that are consistent for executions on a particular A-NPU hardware instance for some period of time. After an initial training phase that accounts for the predictable, compiler-exposed analog limitations, a second (shorter) training phase can adjust for hardware-specific non-idealities, sending training inputs and outputs to the A-NPU and adjusting network weights to minimize error. This correction technique is able to address inter and intra-chip process variation and hardware-dependent, non-ideal analog behavior.

Smaller technology nodes. This work is the start of using analog circuits for code acceleration. Providing benefits at smaller nodes may require using larger transistors for analog parts, trading off area for resilience. Energy-efficient performance is growing in importance relative to area efficiency, especially as CMOS scaling benefits continue to diminish.

1.9 Conclusions

For decades, before the effective end of Dennard scaling, we consistently improved performance and efficiency while maintaining generality in general-purpose computing. As the

benefits from scaling diminish, the community is facing an iron triangle; we can choose any two of performance, efficiency, and generality at the expense of the third. Solutions that improve performance and efficiency, while retaining as much generality as possible, are growing in importance. Analog circuits inherently trade accuracy for significant gains in energy-efficiency. However, it is challenging to utilize them in a way that is both programmable and generally useful. As this work showed, the neural transformation of general-purpose approximable code provides an avenue for realizing the benefits of analog computation while targeting code written in conventional languages. This work provided an end-to-end solution for utilizing analog circuits for accelerating approximate applications, from circuits to compiler design. The insights from this work show that it is crucial to expose analog circuit characteristics to the compilation and neural network training phases. The NPU model offers a way to exploit analog efficiencies, despite their challenges, for a wider range of applications than is typically possible. Further, mixed-signal execution delivers much larger savings for NPUs than digital. However, this study is not conclusive. The full range of applications that can exploit mixed-signal NPUs is still unknown, as is whether it will be sufficiently large to drive adoption in high-volume microprocessors. It is still an open question how developers might reason about the acceptable level of error when an application undergoes an approximate execution including analog acceleration. Finally, in a noisy, high-performance microprocessor environment, it is unclear that an analog NPU would not be adversely affected. However, the significant gains from A-NPU acceleration and the diversity of the studied applications suggest a potentially promising path forward.

CHAPTER 2

NEURO-GENERAL COMPUTING FOR GPU THROUGHPUT PROCESSORS

2.1 Summary

Graphics Processing Units (GPUs) can accelerate diverse classes of applications, such as recognition, gaming, data analytics, weather prediction, and multimedia. Many of these applications are amenable to approximate execution. This application characteristic provides an opportunity to improve GPU performance and efficiency. Among approximation techniques, neural accelerators have been shown to provide significant performance and efficiency gains when augmenting CPU processors. However, the integration of neural accelerators within a GPU processor has remained unexplored. GPUs are, in a sense, many-core accelerators that exploit large degrees of data-level parallelism in the applications through the SIMT execution model. This work aims to harmoniously bring neural and GPU accelerators together without hindering SIMT execution or adding excessive hardware overhead. We introduce a low overhead neurally accelerated architecture for GPUs, called NGPU, that enables scalable integration of neural accelerators for large number of GPU cores. This work also devises a mechanism that controls the tradeoff between the quality of results and the benefits from neural acceleration. This chapter is based on work presented in MICRO 2015 [38]. This work is a result of collaboration with Jongse Park[1], Hardik Sharma[1],Pejman Lotfi-Kamran[2], and Hadi Esmaeilzadeh[3].

[1]Georgia Institute of Technology
[2]Institute for Research in Fundamental Sciences
[3]University of California-San Diego

2.2 Introduction

The diminishing benefits from CMOS scaling [1, 8, 39] has coincided with an overwhelming increase in the rate of data generation. Expert analyses show that in 2011, the amount of generated data surpassed 1.8 trillion GB and by 2020, consumers will generate 50× this staggering figure [40]. To overcome these challenges, both the semiconductor industry and the research community are exploring new avenues in computer architecture design. Two of the promising approaches are acceleration and approximation. Among programmable accelerators, GPUs provide significant gains in performance and efficiency. GPUs that were originally designed to accelerate graphics functions, now are being used for a wide range of applications, including recognition, learning, gaming, data analytics, weather prediction, molecular dynamics, multimedia, scientific computing, and many more. The availability of programming models for GPUs and the advances in their microarchitecture have played a significant role in their widespread adoption. Many companies, such as Microsoft, Google, and Amazon use GPUs to accelerate their enterprise services. As GPUs play a major role in accelerating many classes of applications, improving their performance and efficiency is imperative to enable new capabilities and to cope with the ever-increasing rate of data generation. Improving GPU performance is challenging since they have a relatively high power consumption (e.g., power budget of Fermi GTX 480 is 250 Watts [41]).

Many of the applications that benefit from GPUs are also amenable to imprecise computation [42, 43, 9, 44]. For these applications, some variation in output is acceptable and some degradation in the output quality is tolerable. This characteristic of many GPU applications provides a unique opportunity to devise approximation techniques that trade small losses in the quality of results for significant gains in performance and efficiency. Among approximation techniques, neural acceleration provides significant gains for CPUs [45, 46, 47, 6, 12] and may be a good candidate for GPUs. Neural acceleration relies on an automated algorithmic transformation that converts an approximable segment of code[4] to a

[4]Approximable code is a segment that if approximated will not lead to catastrophic failures in execution (e.g.,

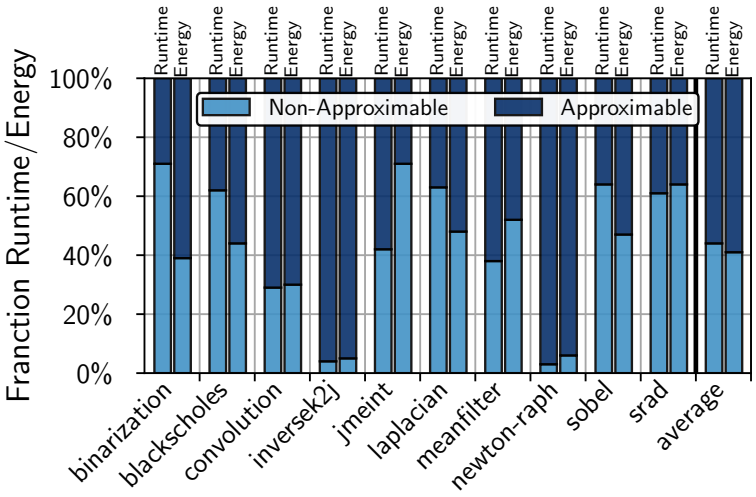


Figure 2.1: Runtime and energy breakdown between neurally approximable regions and the regions that cannot be approximated.

neural network. This transformation is called the neural transformation [12]. The compiler automatically performs the neural transformation and replaces the approximable segment with an invocation of a neural hardware that accelerates the execution of that segment.

To examine the potential benefits of neural acceleration in GPUs, we first study[5] its applicability to a diverse set of representative CUDA applications. Figure 2.1 illustrates the results and shows the breakdown of application runtime and energy dissipation between neurally approximable regions and the regions that cannot be neurally approximated[6]. The neurally approximable segments are the ones that can be approximated by a neural network. On average, applications spend 56% of their runtime and 59% of their energy in neurally approximable regions. Some applications such as inversek2j and newton-raph spend more than 93% of their runtime and energy in neurally approximable regions. These encouraging results demonstrate the significant potential of neural acceleration for GPU processors.

Why hardware acceleration? As previous work [48] suggested, it is possible to apply neural transformation with no hardware modifications and replace the approximable region with an efficient software implementation of the neural network that mimics the region. We explored this possibility and the results are presented in Figure 2.2. On average, the appli-

segmentation fault) and its approximation may only lead to graceful degradation of the application output quality.

[5]Section 2.7.1 presents our experimental methodology with the GPGPU-Sim cycle-accurate simulator.

[6]The annotation procedure is discussed in Section 2.3.

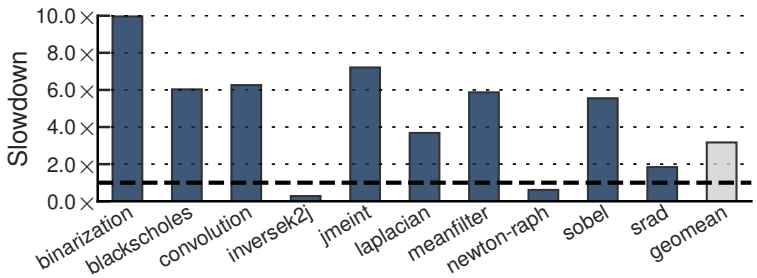


Figure 2.2: Slowdown with software-only neural transformation due to the lack of hardware support for neural acceleration.

cations suffer from 3.2× slowdown. Only inversek2j and newton-raph, which spend more than 93% of their time in the neurally approximable region, see 3.6× and 1.6× speedup, respectively. The slowdown with software implementation is due to (1) the overhead of fetching/decoding the instructions, (2) the cost of frequent accesses to the memory/register file, and (3) the overhead of executing the sigmoid function. The significant potential of neural transformation (Figure 2.1) and the slowdown with the software-only approach (Figure 2.2) necessities designing GPU architectures with integrated neural accelerators.

Why not reuse CPU neural accelerators? Previous work [12] proposes an efficient hardware neural accelerator for CPUs. One possibility is to use CPU Neural Processing Unit (NPU) in GPUs. However, compared to CPUs, GPUs contain (1) significantly larger number of cores (SIMD lanes) that are also (2) simpler. Augmenting each core with a NPU that harbors several parallel processing engines and buffers imposes significant area overhead. Area overhead of integrating NPUs to a GPU while reusing SIMD lanes' multiply-add units is 31.2%. Moreover, neural networks are structurally parallel. Hence, replacing a code segment with neural networks adds structured parallelism to the thread. In the CPU case, NPU's multiple multiply-add units exploit this added parallelism to reduce the thread execution latency. GPUs, on the other hand, already exploit data-level parallelism and leverage many-thread execution to hide thread latencies. One of the insights from this work is that the added parallelism is not the main source of benefits from neural acceleration in GPUs. Therefore, neural acceleration in GPUs leads to a significantly different hardware design as compared to CPUs.

Contributions. To this end, the following are the major contributions of this work.

- While this work is not the first to explore neural acceleration, it is the first to evaluate tight integration of neural acceleration within GPU cores. Integrating neural accelerators within GPUs is fundamentally different from doing so in a CPU because of the hardware constraints and the many-thread SIMT execution model in GPUs.
- We observe that, unlike CPUs, the added parallelism is not the main source of benefits from neural acceleration in GPUs. The gains in GPUs come from (1) eliminating the fetch/decode during neural execution, (2) reducing accesses to the memory/register file by storing the parameters and the partial results in small buffers within the SIMD lanes, and (3) implementing sigmoid as a lookup table. This insight leads to a low overhead integration of neural accelerators to SIMD lanes by limiting the number of ALUs in an accelerator to only the one that is already in a SIMD lane.
- Through a combination of cycle-accurate simulations and a diverse set of GPU applications from different domains (finance, machine learning, image processing, vision, medical imaging, robotics, 3D gaming, and numerical analysis), we rigorously evaluate the proposed NGPU design. Compared to the baseline GPU, NGPU achieves a 2.4× average speedup and a 2.8× average energy reduction within a 10% quality loss margin. These benefits are achieved with less than 1% area overhead.
- We also devise a mechanism that controls the tradeoff between the quality loss and performance and efficiency gains. The quality control mechanism retains a 1.9× average speedup and a 2.1× energy reduction while reducing the quality loss to 2.5%.

2.3 Neural Transformation for GPUs

To enable the integration of neural accelerators within GPUs, we need to develop a compilation workflow that automatically performs the neural transformation on GPU code. We also need to design a programming interface that enables developers to delineate approximable regions as candidates for the neural transformation.

2.3.1 Safe Programming Interface

Any practical approximation technique, including ours, needs to provide execution safety guarantees. That is, approximation should never lead to catastrophic failures such as out-of-bound memory accesses. In other words, approximation should never affect critical data and operations. The criticality of data and operations is a semantic property of the program and can only be identified by the programmer. The programming language must therefore provide a mechanism for programmers to specify where approximation is safe. This requirement is commensurate with prior work on safe approximate programming languages such as EnerJ [26], Rely [27], FlexJava [49], and Axilog [50]. To this end, we extend the CUDA programming language with a pair of `#pragma` annotations that mark the start and the end of a safe-to-approximate region of GPU code. The following example illustrates these annotations.

```
#pragma(begin_approx,"min_max")
mi = __min(r, __min(g, b));
ma = __max(r, __max(g, b));
result = ((ma + mi) > 127 * 2) ? 255 : 0;
#pragma(end_approx,"min_max")
```

This segment of the binarization benchmark is approximable and is marked as a candidate for transformation. The `#pragma(begin_approx,"min_max")` marks the segment's beginning and names it the `"min_max"` segment. The `#pragma(end_approx,"min_max")` marks the end of the segment that was named `"min_max"`.

2.3.2 Compilation Workflow

As discussed, the main idea of neural algorithmic transformation is to learn the behavior of a code segment using a neural network and then replace the segment with an invocation of an efficient neural hardware. To implement this algorithmic transformation, the compiler needs to (1) identify the inputs and outputs of the segment, (2) collect the training data

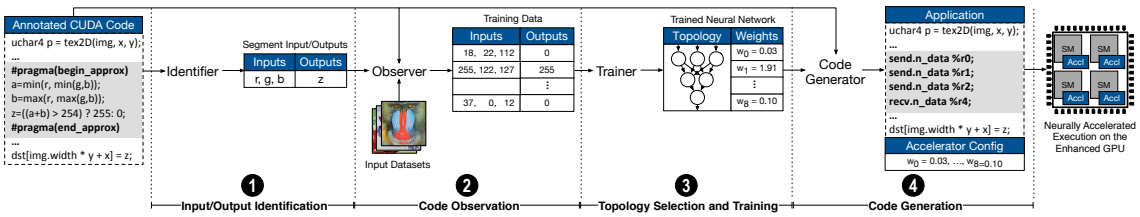


Figure 2.3: Overview of the compilation workflow for neural acceleration in GPU throughput processors.

by observing (logging) the inputs and outputs, (3) find and train a neural network that mimics the observed behavior, and finally (4) replace that region of code with instructions that configure and invoke the neural hardware. These steps are illustrated in Figure 2.3. Our compilation workflow is similar to the one described in prior work that targets neural acceleration in CPUs [12]. However, we specialize these steps for GPU applications and add the automatic *input/output identification step to the compilation workflow* to further automate the transformation.

❶ **Input/output identification.** To train a neural network that mimics a code segment, the compiler needs to collect the input-output pairs that represent the functionality of the region. The first step is identifying the inputs and outputs of the delineated segment. The compiler uses a combination of live variable analysis and Mod/Ref analysis [51] to automatically identify the inputs and outputs of the annotated segment. The inputs are the intersection of live variables at the location of `#pragma(begin_approx,...)` with the set of variables that are referenced within the segment. The outputs are the intersection of live variables at the location of `#pragma(end_approx,...)` with the set of variables that are modified within the segment. In the previous example, this analysis identifies `r`, `g`, and `b` as the inputs to the region and `result` as the output.

❷ **Code observation.** After identifying the inputs and outputs of the segment, the compiler instruments these inputs and outputs to log their values in a file as the program runs. The compiler then runs the program with a series of representative input datasets (such as the ones from a program test suite) and logs the pairs of input-output values. The collected set of input-output values constitutes the training data that captures the behavior of the

segment.

❸ **Topology selection and training.** This step needs to both find a topology for the neural network and train it. In finding the topology, the objective is to strike a balance between the network's accuracy and its efficiency. Theoretically, a larger, more complex network offers better accuracy potential but is likely to be slower and less efficient. However, enlarging the network does not improve its accuracy beyond a certain point. Thus, the compiler considers a search space for the neural topology and picks the smallest network that delivers comparable accuracy to the largest network in the space. The neural network of choice is Multilayer Perceptron (MLP) that consists of a fully-connected set of neurons organized into layers: input layer, a number of hidden layers, and output layer. The number of neurons in the input and output layers is fixed and corresponds to the number of inputs and outputs of the code segment. The challenge is finding the number of hidden layers and the number of neurons in each hidden layer.

The space of possible topologies is infinitely large. Therefore, we restrict the search space to the neural networks with at most two hidden layers. The number of neurons per hidden layer is also restricted to powers of two, up to 32 neurons. These choices limit the search space to 30 possible topologies. The maximum number of hidden layers and the maximum neurons per hidden layer are compilation options and can be changed if needed. These neural networks are trained independently in parallel. To find the best fitting neural network topology, we partition the application input datasets into a training dataset ($\frac{2}{3}$ of the programmer-provided application input datasets), and a selection dataset, (the remaining $\frac{1}{3}$). The training datasets are used during training, and the selection datasets are used to select the final neural network topology based on the application's desired quality loss. Note that we use completely separate input datasets to measure the final quality loss .

To train the networks for neural acceleration, we use the standard backpropagation [30] algorithm. Our compiler performs 10-fold cross-validation for training each neural network. The output from this phase consists of a neural network topology – specifying the

number of layers and the number of neurons in each layer – along with the weights for the inputs of each neuron that are determined by the backpropagation training algorithm.

❹ **Code generation.** After identifying the neural network and training it, the compiler replaces the code segment with special instructions to send the inputs to the neural accelerator and retrieve the results. The compiler also configures the neural accelerator. The configuration includes the weights and the schedule of the operations within the accelerator. This information is loaded into the integrated neural accelerators once when the program loads for execution.

2.4 Instruction Set Architecture Design

To enable neural acceleration, the GPU ISA should provide three instructions: (1) one for sending the inputs to the neural accelerator, (2) one for receiving outputs from the neural accelerator, and finally (3) one for sending the accelerator configuration and the weights. To this end, we extend the PTX ISA as follows:

1. **`send.n_data`** `%r`: This instruction sends the value of register `%r` to the neural accelerator as an input.
2. **`recv.n_data`** `%r`: This instruction retrieves a value from the accelerator and writes it to the register `%r`.
3. **`send.n_cfg`** `%r`: This instruction sends the value of register `%r` to the accelerator and indicates that the value is for configuration.

We use PTX ISA 4.2 which supports vector instructions that can read or write two or four registers instead of one. We take advantage of this feature and introduce two vector versions for each of our instructions. The **`send.n_data.v2`** `{%r0, %r1}` sends two register values to the accelerator and a single **`send.n_data.v4`** `{%r0, %r1, %r2, %r3}` sends the value of four registers to the neural accelerator. The vector versions for **`recv.n_data`** and **`send.n_cfg`** have similar semantics. These vector versions reduce the number of instructions that need to be fetched and decoded to communicate with the neural accelerator. This reduction lowers

the overhead of invoking the accelerator and provides more opportunities for speedup and efficiency gains. As follows, these instructions will be executed in SIMT mode as other GPU instructions. GPU applications typically consist of kernels and GPU threads execute the same kernel code. The neural transformation approximates segments of these kernels. That is, each corresponding thread will contain the aforementioned instructions to communicate with the neural accelerator. Each thread only applies different input data to the same neural network. GPU threads are grouped into cooperative thread arrays (a unit of thread blocks). The threads in different thread blocks are independents and can be executed in any order. The thread block scheduler maps them to GPU processing cores called the streaming multiprocessors (SMs). The SM divides threads of a thread block into smaller groups called warps, typically of size 32 threads. All the threads within a warp execute the same instruction in lock-step. The three new instructions, `send.n_data`, `recv.n_data`, and `send.n_cfg` follow the same SIMT model. That is, executing each of these instructions, conceptually, communicates data with 32 parallel neural accelerators.

A typical GPU architecture, such as Fermi [52], contains 15 SMs, each with 32 SIMD lanes. That is, to support hardware neural acceleration, 480 neural accelerators need to be integrated. Hence the GPU-specific challenge is designing a hardware neural accelerator that can be replicated many times within the GPU without imposing extensive hardware overhead.

2.5 Accelerator Design and Integration

To describe our neural accelerator design and its integration into the GPU architecture, we assume a GPU processor based on the Nvidia Fermi. Fermi's SMs contain 32 double-clocked SIMD lanes that execute two half warps (16 threads) simultaneously, where each warp executes in lock-step. Ideally, to preserve the data-level parallelism across the threads and preserve the default SIMT execution model, each SM needs to be augmented with 32 neural accelerators. Therefore, the objective is to design a neural accelerator that can be

replicated 32 times within each SM for a minimal hardware overhead. These two requirements fundamentally change the design space of the neural accelerator from prior work that aims at accelerating single-thread cores with only one accelerator.

A naïve approach is to replicate and add the previously proposed CPU neural accelerator to each SM [12]. These CPU specific accelerators harbor multiple processing engines and contain significant amount of buffering for weights and control. Such a design not only imposes significant hardware overhead, but also is an overkill for data-parallel GPU architectures as our results in Section 2.7.3 show. Instead, we tightly integrate a GPU specific neural network in every SIMD lane.

The neural algorithmic transformation uses Multilayer Perceptrons (MLPs) to approximate CUDA code segments. As Figure 2.5a depicts, an MLP consists of a network of neurons arranged in multiple layers. Each neuron in a layer is connected to all of the neurons in the next layer. Each neuron input is associated with a weight value that is generated after training. All neurons are identical and each neuron computes its output (y) based on $y = sigmoid(\sum_i (w_i \times x_i))$, where x_i is a neuron input and w_i is the input's associated weight. Therefore, all the computations of a neural network are a set of multiply-add operations followed by the nonlinear sigmoid operation. The neural accelerator only needs to support these two operations.

2.5.1 Integrating the Neural Accelerator

Each SM has 32 SIMD lanes, divided into two 16-lane groups that execute two half warps simultaneously. The ALU in each lane supports floating point multiply-add operation. We reuse these ALUs while enhancing the lanes for neural computation. We leverage the existing SIMT execution model to minimize the hardware overhead for the weights and control. We refer to the resulting SIMD lanes as neurally enhanced SIMD lanes.

In Figure 2.4, the added hardware components are numbered and highlighted in gray. The first component is the Weight FIFO (❶) that is a circular buffer and stores the synaptic

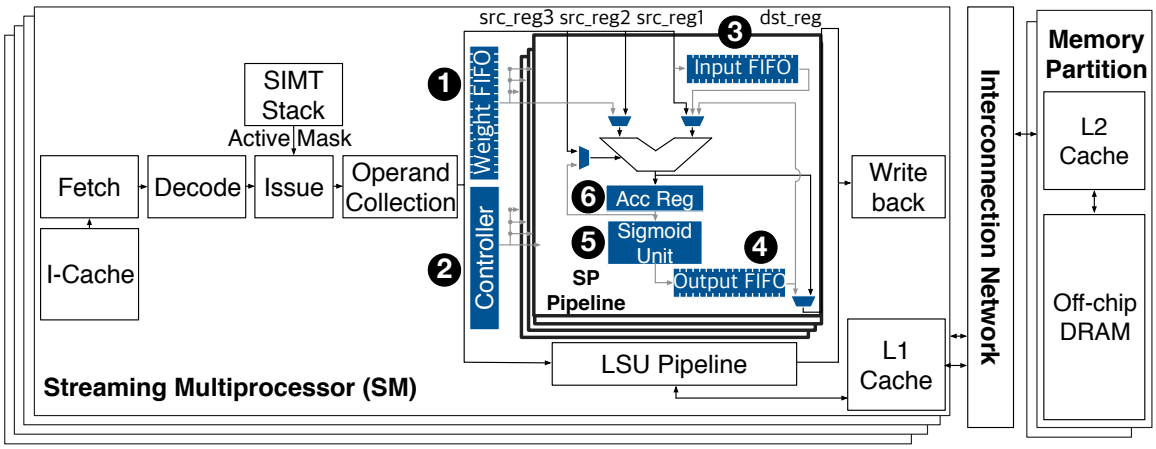


Figure 2.4: SM pipeline after integrating the neural accelerator within SIMD lanes. The added hardware is highlighted in gray.

weights. Since all of the threads are approximated by the same neural network, we only add one Weight FIFO, which is shared across all SIMD lanes. The Weight FIFO has two read ports corresponding to the two 16 SIMD lanes that execute two half warps. Each port supplies a weight to 16 ALUs. The second component is the Controller (❷) which controls the execution of the neural network across the SIMD lanes. Again, the Controller is shared across 16 SIMD lanes that execute a half warp (two controllers per SM). The Controller follows the SIMT pattern of execution for neural computation and enables the ALUs to perform the computation of the same input of the same neuron in the network.

We augment each of the SIMD lanes with an Input FIFO (❸) and an Output FIFO (❹). The Input FIFO stores the neural network inputs. The Output FIFO stores the output of the neurons including the output neurons that generate the final output. These two are small FIFO structures that are replicated for each SIMD lane. Each of the SIMD lanes also harbors a Sigmoid Unit (❺) that contains a read-only lookup table. This lookup table implements the nonlinear sigmoid function and is synthesized as combinational logic to reduce the area overhead. Finally, the Acc Reg (❻), which is the accumulator register in each of the SIMD lanes, retains the partial results of the sum of products ($\sum_i(w_i \times x_i)$) before passing it to the Sigmoid Unit.

One of the advantages of this design is that it limits all major modifications to the execution part of the SIMD lanes (pipelines). There is no need to change any other part of

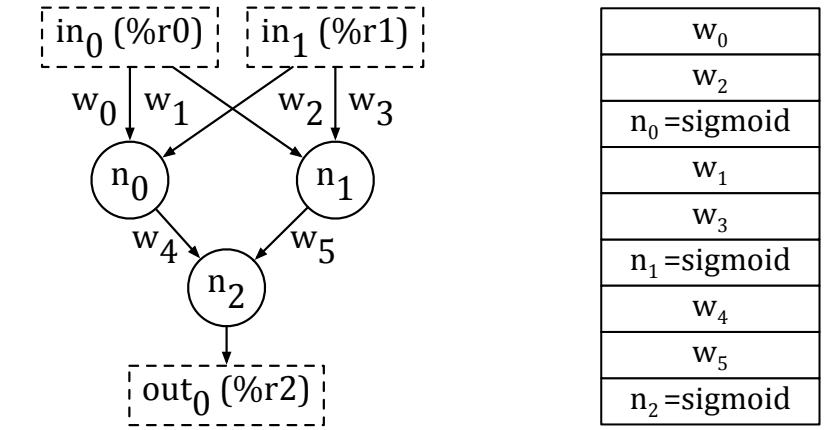
the SM except for adding support for decoding the ISA extensions that communicate data to the accelerator (i.e., input and output buffers). Scheduling and issuing these instructions are similar to arithmetic instructions and do not require specific changes.

2.5.2 Executing Neurally Transformed Threads

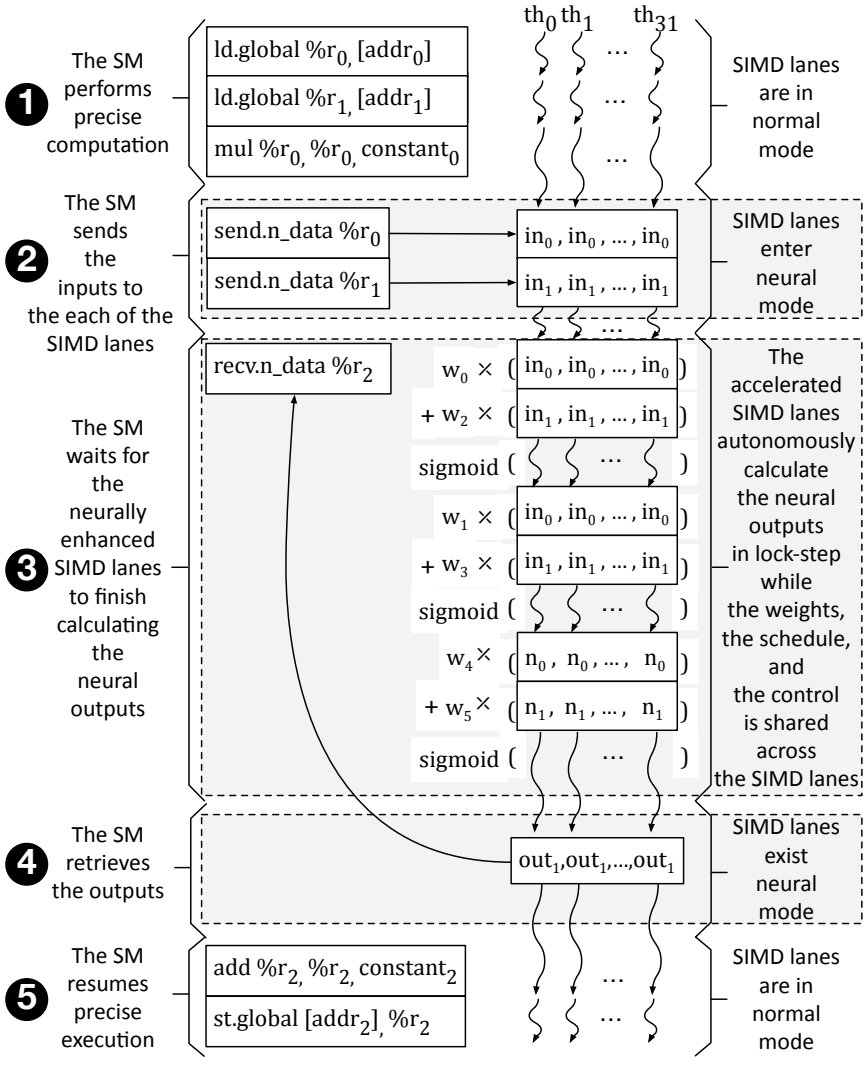
Figure 2.5c illustrates the execution of a neurally transformed warp, which contains normal precise and special approximate (i.e., `send.n_data`/`recv.n_data`) instructions, on its neurally enhanced SIMD lanes. The other simultaneously executing warp (similarly contains both normal and special instructions) is not shown for clarity. In the first phase (❶), SIMD lanes execute the precise instructions as usual before reaching the first `send.n_data` instructions. In the second phase (❷), SIMD lanes execute the two `send.n_data` instructions to copy the neural network inputs from the register file to their input buffers. These instructions cause SIMD lanes to switch to the neural mode. In the third phase (❸), the enhanced SIMD lanes perform the neural computation and store the results in their output buffers. At the same time, the SM issues `recv.n_data`, but since the output of the neural network is not ready yet, the SM stops issuing the next instruction and waits for the neurally-enhanced SIMD lanes to finish computing the neural network output. In the fourth phase (❹), once the neural network output is ready, `recv.n_data` instruction copies the results from the output buffer to the register file and then in the fifth phase (❺) normal execution resumes. As there is no control divergence or memory access in the neural mode, our design does not swap the running warp with another warp *in the neural mode* to avoid the significant overhead of dedicated input/output buffers or control logic per active warp (SMs support 48 ready-to-execute warps).

2.5.3 Orchestrating Neurally Enhanced Lanes

To efficiently execute neural networks on the neurally enhanced SIMD lanes, the compiler needs to create a static schedule for the neural computation and arrange the weights in



(a) Neural Network **(b)** Schedule



(c) Accelerated Execution on the Enhanced SM

Figure 2.5: (a) Neural network replacing a segment of a GPU code. (b) Schedule for the accelerated execution of the neural network. (c) Accelerated execution of the GPU code on the enhanced SM.

proper order. This schedule and the preordered weights are encoded in the program binary and are preloaded to the Weight FIFO (Figure 2.4 ❶) when the program loads for execution. The compiler generates the execution schedule based on the following steps:

1. The computations for the neurons in each layer are dependent on the output of the neurons in the previous layer. Thus, the compiler first assigns a unique order to the neurons starting from the first hidden layer down to the output layer. This order determines the execution of the neurons. In Figure 2.5a. n_0, n_1, and n_2 show this order.
2. Then, for each neuron, the compiler generates the order of the multiply-add operations, which are followed by a sigmoid operation. This schedule is shown in Figure 2.5b for the neural network in Figure 2.5a. The phase (❸) of Figure 2.5c illustrates how the neurally enhanced SIMD lanes execute this schedule in SIMT mode while sharing the weights and control.

The schedule that is presented in 2.5b constitutes the most of the accelerator configuration and the order in which the weights will be stored in Weight FIFO (❶ in 2.4). For each accelerator invocation, SIMD lanes go through these weights in lock-step and perform the neural computation autonomously without engaging the other parts of the SM.

2.6 Controlling Quality Tradeoffs

To be able to control the quality tradeoffs, any approximation technique including ours, needs to expose a quality knob to the compiler and/or runtime system. The knob for our design is *the accelerator invocation rate*. That is the fraction of the warps that are offloaded to the neural accelerator. The rest of the warps will execute the original precise segment of code and generate exact outputs. In the default case, without any quality control, all the warps that contain the approximable segment will go through the neural accelerator which translates to 100% invocation rate. With quality control, only a fraction of the warps will go through the accelerator. Naturally, the higher the invocation rate, the higher the benefits and the lower the quality. For a given quality target, the compiler predetermines the invocation

Table 2.1: Applications, accelerated regions, training and evaluation datasets, quality metrics, and approximating neural networks.

Benchmark Name	Description	Source	Domain	Quality Metric	# of Function Calls	# of Loops	# of ifs/ elses	# of PTX Insts.	Training Input Set	Evaluation Input Set	Digital NPU	
											Neural Network Topology	Quality Loss
binarization	Image binarization	Nvidia SDK	Image Processing	Image Diff	1	0	1	27	Three 512x512 pixel images	Twenty 2048x2048 pixel images	3 → 4 → 2 → 1	8.23%
blackscholes	Option pricing	Nvidia SDK	Finance	Average Relative Error	2	0	0	96	8,192 options	262,144 options	6 → 8 → 1	4.35%
convolution	Data filtering operation	Nvidia SDK	Machine Learning	Average Relative Error	0	2	2	886	8,192 data points	262,144 options	17 → 2 → 1	5.25%
inversek2j	Inverse kinematic for 2-joint arm	CUDA-based Kinematics	Robotics	Average Relative Error	0	3	5	132	8,192 2D coordinates	262,144 2D coordinates	2 → 16 → 3	8.73%
jmeint	Triangle intersection detection	jMonkey Game	3D Gaming	Miss Rate	4	0	37	2,250	8,192 3D coordinates	262,144 3D coordinates	18 → 8 → 2	17.32%
laplacian	Image sharpening filter	Nvidia SDK	Image Processing	Image Diff	0	2	1	51	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 → 2 → 1	6.01%
meanfilter	Image smoothing filter	Nvidia SDK	Machine Vision	Image Diff	0	2	1	35	Three 512x512 pixel images	Twenty 2048x2048 pixel images	7 → 4 → 1	7.06%
newton-raph	Newton-Raphson equation solver	Likelihood Estimator	Numerical Analysis	Average Relative Error	2	2	1	44	8,192 cubic equations	262,144 cubic equations	5 → 2 → 1	3.08%
sobel	Edge detection	Nvidia SDK	Image processing	Image Diff	0	2	1	86	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 → 4 → 1	5.45%
srad	Speckle reducing anistropic diffusion	Rodinia	Medical Imaging	Image Diff	0	0	0	110	Three 512x512 pixel images	Twenty 2048x2048 pixel images	5 → 4 → 1	7.43%

rate by examining the output quality loss on a held-out evaluation input dataset. Starting from 100% invocation rate, the compiler gradually reduces the invocation rate until the quality loss is less than the quality target. During runtime, a quality monitor, similar to the one proposed in SAGE [9], stochastically checks the output quality of the application and adjusts the invocation rate. We also investigated a more sophisticated approach that uses another neural network to filter out invocations of the accelerator that significantly degrade quality. The empirical study suggested that the simpler approach of reducing the invocation rate provides similar benefits.

2.7 Evaluation

We evaluate the benefits of the proposed architecture across different bandwidth and accelerator settings. We use a diverse set of applications, cycle-accurate simulation, logic synthesis, and consistent detailed energy modeling.

2.7.1 Applications and Neural Transformation

Applications. As Table 4.2 shows, we use a diverse set of *approximable* GPU applications from the Nvidia SDK [53] and Rodinia [54] benchmark suites to evaluate the integration of neural accelerators within GPU architectures. We added three more applications to the mix

from different sources [55, 56, 57]. As shown, the benchmarks represent workloads from finance, machine learning, image processing, vision, medical imaging, robotics, 3D gaming, and numerical analysis. We did not reject any benchmarks due to their performance, energy, or quality shortcomings.

Annotations. As described in Section 2.3.1, the CUDA source code for each application is annotated using the #pragma directives. We use theses directives to delineate a region within a CUDA kernel that has fixed number of inputs/outputs and is safe to approximate. Although it is possible and may boost the benefits to annotate multiple regions, we only annotate one region that is easy to identify and is frequently executed. We did not make any algorithmic changes to enable neural acceleration.

As illustrated by the numbers of function calls, conditionals, and loops in Table 4.2, these regions exhibit a rich and diverse control flow behavior. For instance, the target region in inversk2j has three loops and five conditionals. Other regions similarly have several loops/conditionals and function calls. Among these applications, the region in jmeint has the most complicated control flow with 37 if/else statements. The regions are also diverse in size and vary from small (binarization with 27 PTX instructions) to large (jmeint with 2,250 PTX instructions).

Evaluation/training datasets. As illustrated in Table 4.2, the datasets that are used for measuring the quality, performance, and energy are completely disjoint from the ones used for training the neural networks. The training inputs are typical representative inputs (such as sample images) that can be found in application test suites. For instance, we use the image of lena, peppers, and mandrill for applications that operate on image data. Since the regions are frequently executed, even a single application input provides large number of training data. For example, in sobel a 512×512 pixel image generates 262,144 training data elements.

Neural networks. The "Neural Network Topology" column shows the topology of the neural network that replaces the region of code. For instance, the topology for blackscholes is

$6 \rightarrow 8 \rightarrow 1$. That is the neural network has 6 inputs, one hidden layer with 8 neurons, and 1 output neuron. These topologies are automatically discovered by our compiler and we use the 10-fold cross validation technique to train the neural networks. As the results suggest, different applications require different topologies. Therefore, the SM architecture should be changed in a way that is reconfigurable and can accommodate different topologies.

Quality. We use application-specific quality metrics, shown in Table 4.2, to assess the quality of each application's output after neural acceleration. In all cases, we compare the output of the original precise application to the output of the neurally accelerated application. For blackscholes, inversek2j, newton-raph, and srad that generate numeric outputs, we measure the average relative error. For jmeint that determines whether two 3D triangles intersect, we report the misclassification rate. The convolution, binarization, laplacian, meanfilter, and sobel that produce image outputs, we use the average root-mean-square image difference. In Table 4.2, the "Quality Loss" column reports the whole-application quality degradation based on the above metrics. This loss includes the accumulated errors due to repeated execution of the approximated region. The quality loss in Table 4.2 represents the case where all of the dynamic threads with the safe-to-approximate region are neurally accelerated.

Even with 100% invocation rate, the quality loss with neural acceleration is less than 10% except in the case of jmeint. The jmeint application's control flow is very complex and the neural network is not able to capture all the corner cases to achieve below 10% quality loss. These results are commensurate with prior work on CPU-based neural acceleration [6, 46]. Prior work on GPU approximation such as SAGE [9] and Paraprox [42] reports similar quality losses in the default setting. EnerJ [26] and Truffle [13] show less than 10% loss for some applications and even 80% loss for others. Green [36] and loop perforation [58] show less than 10% error for some applications and more than 20% for others. Later, we will discuss how to use the invocation rate to control the quality tradeoffs, and achieve even lower quality loss when desired.

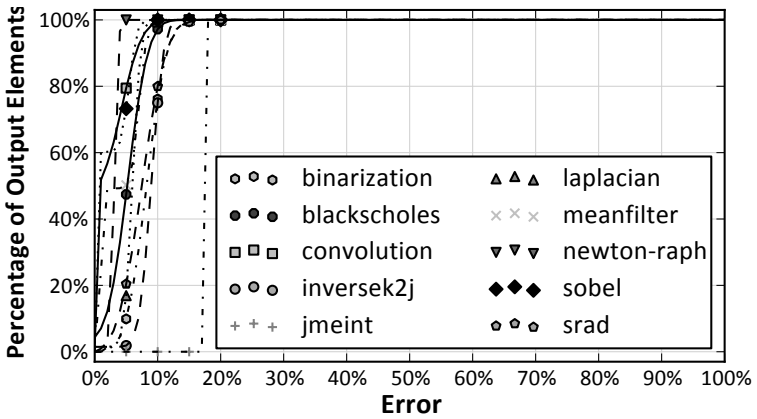


Figure 2.6: Cumulative distribution function (CDF) plot of the applications output quality loss. A point (x,y) indicates that y fraction of the output elements see quality loss less than or equal to x.

To better illustrate the application quality loss, Figure 2.6 shows the Cumulative Distribution Function (CDF) plot of the final quality loss for each element of the output. Each application output is a collection of elements – an image consists of pixels; a vector consists of scalars; etc. The loss CDF shows the distribution of output quality loss among the output elements and shows that very few output elements see a large loss. As shown, the majority of output elements (from 78% to 100%) see a loss less than 10%.

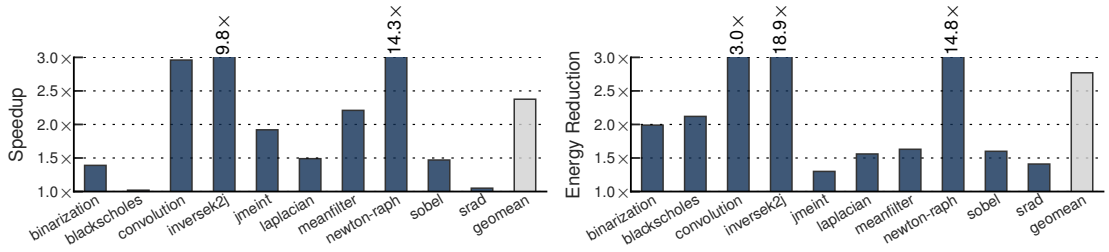
2.7.2 Experimental Setup

Cycle-accurate simulations. We use the GPGPU-Sim cycle-accurate simulator version 3.2.2 [59]. We modified the simulator to include our ISA extensions and include the extra microarchitectural modifications necessary for the integration of neural accelerators within GPUs. The overhead of ISA extensions that communicate with the accelerator are modeled. For baseline simulations that do not include any approximation or acceleration, we use the unmodified GPGPU-Sim. We use one of the GPGPU-Sim's default configurations that closely models the Nvidia GTX 480 chipset with Fermi architecture. Table 3.2 summarizes the microarchitectural parameters of the chipset. We run the applications to completion. We use NVCC 4.2 with -O3 to enable aggressive compiler optimizations. Moreover, we optimize the number of thread blocks and number of threads-per-block of each kernel for the simulated hardware.

Energy modeling and overheads. To measure GPU energy, we use GPUWattch [60],

Table 2.2: GPU microarchitectural parameters.

System Overview: No. of SMs: 15, Warp Size: 32 threads/warp; **Shader Core Config**: 1.4 GHz, GTO scheduler [61], 2 schedulers/SM; **Resources / SM**: No. of Warps: 48 Warps/SM, No. of Registers: 32,768; **Interconnect**: 1 crossbar/direction (15 SMs, 6 MCs), 700 MHz; **L1 Data Cache**: 16KB, 128B line, 4-way, LRU; **Shared Memory**: 48KB, 32 banks; **L2 Unified Cache**: 768KB, 128B line, 16-way, LRU; **Memory**: 6 GDDR5 Memory Controllers, 924 MHz, FR-FCFS [**fcfs**]; **Bandwidth**: 177.4 GB/sec.



(a) Speedup **(b)** Energy Reduction

Figure 2.7: NGPU **whole application speedup and energy reduction.**

which is integrated with GPGPU-Sim. To measure the accelerator energy, we also generate its event log during the cycle-accurate simulations . Our energy evaluations use a 40 nm process node and 1.4 GHz clock frequency. Neural acceleration requires the following changes to the SM and SIMD lanes and are modeled using McPAT [32] and CACTI 6.5 [33]. In each SM, we add a 2 KB weight FIFO. The extra input/output FIFOs are 256 bytes per SIMD lane. The sigmoid LUT which is added to each SIMD lane contains 2048 32-bit entries. Since GPUWattch also uses McPAT and CACTI, our added energy models, which use the same tools, provide a unified and consistent framework for energy measurement.

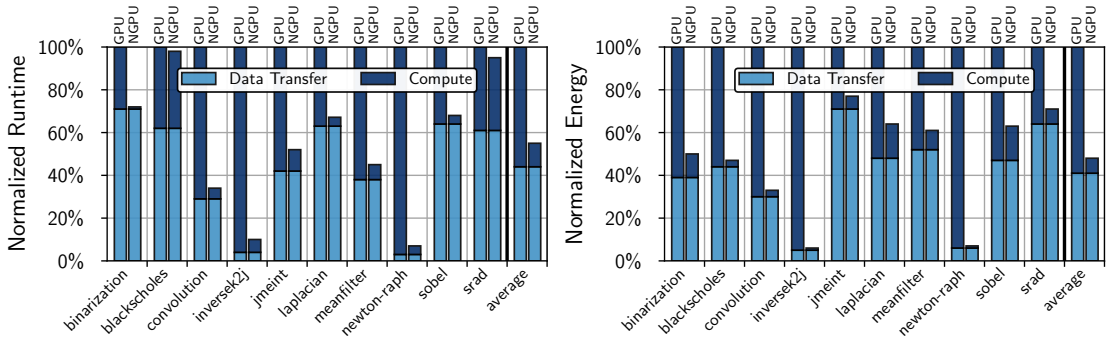
2.7.3 Experimental Results

Performance and energy benefits. Figure 2.7a shows the whole application speedup when all the invocations of the approximable region are accelerated with the neural accelerator. The remaining part (i.e., the non-approximable region) is executed normally. The results are normalized to the baseline where the entire application is executed on the GPU with no acceleration. The highest speedup is observed for newton-raph (14.3×) and inversek2j (9.8×), where the bulk of execution time is spent on approximable parts (see Figure 2.1). The lowest speedup is observed for blackscholes and srad (about 2% and 5%) which are

bandwidth-hungry applications. While a considerable fraction of the execution time in blackscholes and srad is spent in the approximate region (See Figure 2.1), the speedup of accelerating these two applications is modest. That is because these applications use most of the off-chip bandwidth, even when they run on GPU (without acceleration). Due to bandwidth limitation, neural acceleration cannot reduce the execution time. Next, we study the effect of increasing the off-chip bandwidth on these two applications and show that with reasonable improvement in bandwidth, even these benchmarks observe significant benefits. On average, the evaluated applications see a 2.4 $\times$ speedup through neural acceleration.

Figure 2.7b shows the energy reduction for each benchmark as compared to the baseline where the whole benchmark is executed on GPU. Similar to the speedup, the highest energy saving is achieved for inversek2j (18.9$\times$) and newton-raph (14.8$\times$), where bulk of the energy is consumed for the execution of approximable parts (see Figure 2.1). The lowest energy saving is obtained on jmeint (30%) since for this application, the fraction of energy consumed on approximable parts is relatively small (See Figure 2.1). On average, the evaluated applications see a 2.8 $\times$ reduction in energy usage. The quality loss when all the invocations of the approximable region get executed on neural accelerators (i.e., the highest quality loss) is shown in Table 4.2 (labeled Quality Loss). We study the effects of our quality control mechanism for trading off performance and energy savings for better quality later in this section.

Area overhead. To estimate the area overhead, we synthesize the sigmoid unit using Synopsys Design Compiler and NanGate 45 nm Open Cell library, targeting the same frequency as the SMs. We extract the area of the buffers and FIFOs from CACTI. Overall, the added hardware requires about 0.27 mm^2. We estimate the area of the SMs by inspecting the die photo of GTX 480 that implements the Fermi architecture. Each SM is about 22 mm^2 and the die area is 529 mm^2 with 15 SMs. The area overhead per SM is approximately 1.2% and the total area overhead is 0.77%. The low area overhead is because our architecture uses the same ALUs that are already available in each SIMD



(a) Runtime **(b)** Energy

Figure 2.8: Breakdown of (a) runtime and (b) energy consumption between non-approximable and approximable regions normalized to the runtime and energy consumption of the GPU, respectively. For each application, the first (second) bar shows the normalized value when the application is executed on the GPU (NGPU).

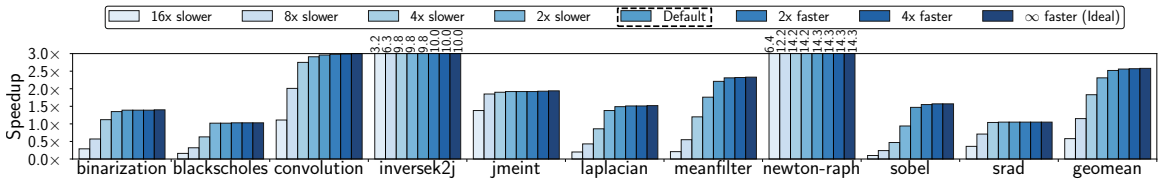


Figure 2.9: Sensitivity of the total application's speedup to the neural accelerator delay. Each bar indicates the total application's speedup when the neural accelerator delay is altered by different factors. The default delay for neural accelerator varies from one application to the other and depends on the neural network topology trained for that application. The ideal case (∞ faster) shows the total application speedup when neural accelerator has zero delay.

lane, shares the weight buffer across the lanes, and implements the sigmoid unit as a read-only lookup table, enabling the synthesis tool to optimize its area. This low area overhead confirms the scalability of our design.

Opportunity for further improvements. To explore the opportunity for further improving the execution time by making the neural accelerator faster, Figure 2.8a shows the time breakdown of approximable and non-approximable parts of applications when applications run on GPU (no acceleration) and NGPU (neurally accelerated GPU), normalized to the case where the application runs on GPU (no acceleration). As Figure 2.8a depicts, NGPU is effective at reducing the time that is spent on approximable parts for all but two applications: blackscholes and srad. These two applications use most of the bandwidth of the GPU, and consequently, do not benefit from the accelerators due to the bandwidth wall. The rest of the applications significantly benefit from accelerators. On some applications (e.g., binarization, laplacian, and sobel), the execution time of approximable parts on NGPU is significantly smaller than the execution time of the non-approximable

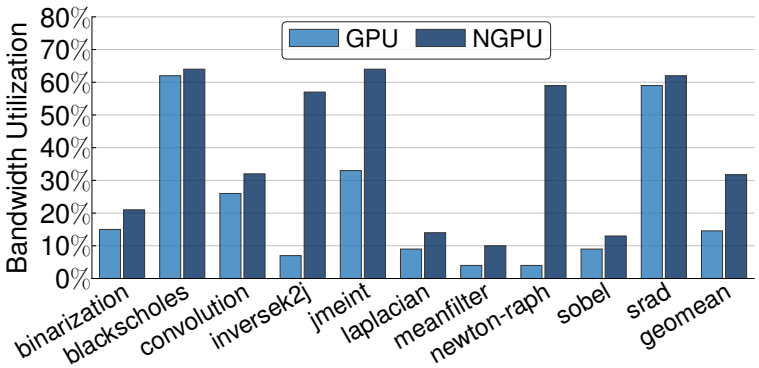


Figure 2.10: Memory bandwidth consumption when the applications are executed on GPU (first bar) and NGPU **(second bar).**

parts. Hence, no further benefits are possible with faster accelerators. For the rest of the applications, the execution time of approximable parts on NGPU, although considerably reduced, is comparable to and sometimes exceeds (e.g., inversek2j) the execution time of non-approximable parts. Thus, there is a potential to further speed these applications up with faster accelerators.

We similarly study the opportunity to further reduce the energy usage with more energy-efficient accelerators. Figure 2.8b shows the energy breakdown between approximable and non-approximable parts when applications run on GPU and NGPU, normalized to the case where the application runs on GPU. These results clearly shows that neural accelerators are effective in reducing the energy usage of applications when executing the approximable parts. For many of the applications, the energy that is consumed for running approximable parts is modest as compared to the energy that is consumed for running the non-approximable parts (e.g., blackscholes, convolution, jmeint, etc.). For these applications, a more energy-efficient neural accelerator may not bring further energy savings. However, there are some applications, such as binarization, laplacian, and sobel, for which the fraction of energy that is consumed on neural accelerators is comparable to the fraction of energy consumed on non-approximable parts. For these applications further energy saving is possible with a more energy-efficient implementation of neural accelerators (e.g., analog neural accelerators [6]).

Sensitivity to accelerator speed. To study the effects of accelerators' speed on performance gains, we vary the latency of neural accelerators and measure the overall speedup as

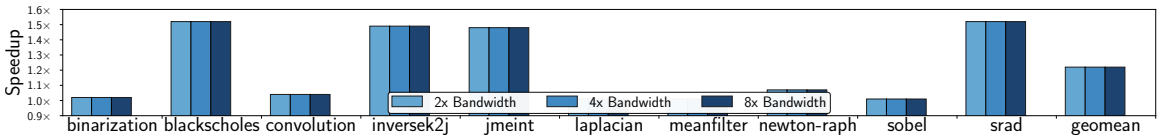


Figure 2.11: The total application speedup with NGPU for different off-chip memory communication bandwidth normalized to the execution with NGPU with default bandwidth. The default bandwidth is 177.4 GB/s.

shown in Figure 2.9. We decrease the delay of the default accelerators by a factor of 2 and 4 and also include an ideal neural accelerator with zero latency. Moreover, we show the speedup numbers when the latency of the default accelerators increases 2×, 4×, 8× and 16×. Unlike Figure 2.8a that suggests performance improvement for some applications by benefiting from faster accelerators, Figure 2.9 shows virtually no speedup benefit by making neural accelerators faster beyond what they offer in the default design. Even making accelerators slower by a factor of two does not considerably change the speedup. Slowing down the accelerators by a factor of four, many applications observe performance loss. (e.g., laplacian). To explain this behavior, Figure 2.10 shows the bandwidth usage of GPU and NGPU across all applications. While on the baseline GPU, only two applications use more than 50% of the off-chip bandwidth (i.e., blackscholes and srad), on NGPU, many applications use more than 50% of their off-chip bandwidth (e.g., inversek2j, jmeint, and newton-raph). As applications run faster with accelerators, the rate at which they access data increases, which puts pressure on off-chip bandwidth. This phenomena shifts the bottleneck of execution time from computation to data delivery. *As computation is no longer the major bottleneck after acceleration, speeding up thread execution beyond a certain point has marg-inal effect on the overall execution time.* Even increasing the accelerator speed by a factor of two (e.g., by adding more multiply-and-add units) has marginal effect on execution time. We leverage this insight to simplify the accelerator design and reuse available ALUs in the SMs as described is Section 2.5.1.

Sensitivity to off-chip bandwidth. To study the effect of off-chip bandwidth on the benefits of NGPU, we increase the off-chip bandwidth up to 8× and report the performance numbers. Figure 2.11 shows the speedup of NGPU with 2×, 4×, and 8× bandwidth over the baseline NGPU (i.e., 1× bandwidth) across all benchmarks. As NGPU

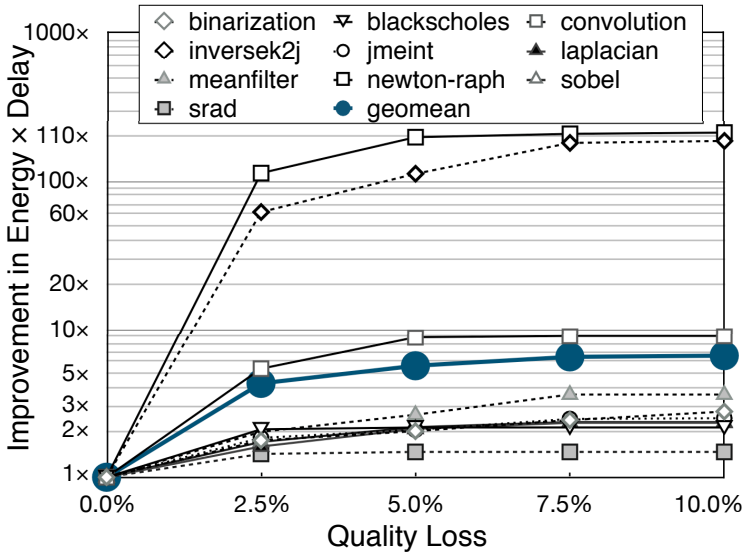


Figure 2.12: Energy×delay benefits vs output quality (log scale).

is bandwidth limited for many applications (See Figure 2.10), we expect a considerable improvement in performance as the off-chip bandwidth increases. Indeed, Figure 2.11 shows that bandwidth-hungry application (i.e., blackscholes, inversek2j, jmeint, and srad) observe speedup of 1.5× when we double the off-chip bandwidth. After doubling the off-chip bandwidth, no application remains bandwidth limited, and therefore, increasing the off-chip bandwidth to 4× and 8× has little effect on performance. It may be possible to achieve, the 2× extra bandwidth by using data compression [62] with little changes to the architecture of existing GPUs. While technologies like 3D DRAM that offer significantly more bandwidth (and lower access latency) can be useful, they are not necessary for providing the off-chip bandwidth requirements of NGPU for the range of applications that we studied. However, even without any of these likely technology advances (compression or 3D stacking), the NGPU provides significant benefits across most of the applications.

Controlling quality tradeoffs. To study the effect of our quality control mechanism, Figure 2.12 shows the energy-delay product of NGPU normalized to the energy-delay product of the baseline GPU (without acceleration) when the output quality loss changes from 0% to 10%. The quality control mechanism enables navigating the tradeoff between the quality loss and the gains. All applications see declines in benefits when invocation rate decreases (i.e., output quality improves). Due to the Amdahl's Law effect, the applications

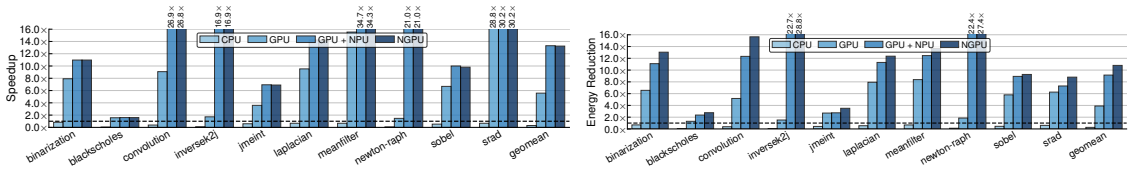
that spend more than 90% of their execution in the approximable segment (inversek2j and newton-raph), see larger declines in benefits when invocation rate decreases. However, even with 2.5% quality loss, the average speedup is 1.9× and the energy savings is 2.1×.

Comparison with prior CPU neural acceleration. Prior work [12] has explored improving CPU performance and efficiency with NPUs. Since NPUs offer considerably higher performance and energy efficiency with CPUs, we compare our NGPU proposal to CPU+NPU and GPU+NPU. For the evaluation, we use MARSSx86 cycle-accurate simulator for the single-core CPU simulations with a configuration that resembles Intel Nehalem (3.4 GHz with 0.9 V at 45 nm) and is the same as the setup used in the most recent NPU work [6].

Figure 2.13 shows the application speedup and energy reduction with CPU, GPU, GPU+NPU, and NGPU over CPU+NPU. Even without using neural acceleration, GPU provides significant performance and efficiency benefits over NPU-accelerated CPU by leveraging data level parallelism. GPU offers 5.6× average speedup and 3.9× average energy reduction compared to CPU+NPU. A GPU enhanced with our proposal (NGPU) increases the average speedup and energy reduction to 13.2× and 10.8×, respectively. Moreover, as GPUs already exploit data-level parallelism, our proposal offers virtually the same speedup as the area-intensive GPU+NPU. However, accelerating GPU with the NPU design imposes 31.2% area overhead while our NGPU imposes 1.2%. GPU with area-intensive NPU (GPU+NPU) offers 17.4% less energy benefits compared to NGPU mostly due to more leakage. In summary, our proposal offers the highest level of performance and energy efficiency across the examined benchmarks with the modest area overhead of approximately 1.2% per SM.

2.8 Conclusion

Many of the emerging applications that can benefit from GPU acceleration are amenable to inexact computation. We exploited this opportunity by integrating an approximate form



(a) Speedup **(b)** Energy Reduction

Figure 2.13: Speedup and energy reduction with CPU, GPU, GPU+NPU, and NGPU**.(The baseline is CPU+NPU, which is a CPU augmented with a NPU accelerator [12]).**

of acceleration within GPU architectures. Our neurally accelerated GPU architecture, provides significant performance and efficiency benefits while providing reasonably low hardware overhead. The quality control knob and mechanism also provided a way to navigate the tradeoff between the quality and the benefits in efficiency and performance. Even with as low as 2.5% quality loss, our neurally accelerated GPU architecture (NGPU) provides average speedup of 1.9× and average energy savings of 2.1×. These benefits are more than 10× in several cases. These results suggest that *hardware* neural acceleration for GPU throughput processors can be a viable approach to significantly improve their performance and efficiency.

CHAPTER 3

IN-DRAM NEAR-DATA NEURO-GENERAL COMPUTING

3.1 Summary

GPUs are bottlenecked by the off-chip communication bandwidth and its energy cost; hence near-data acceleration is particularly attractive for GPUs. Integrating the accelerators within DRAM can mitigate these bottlenecks and additionally expose them to the higher internal bandwidth of DRAM. However, such an integration is challenging, as it requires low-overhead accelerators while supporting a diverse set of applications. To enable the integration, this work leverages the approximability of GPU applications and utilizes the neural transformation, which converts diverse regions of code mainly to Multiply-Accumulate (MAC). Furthermore, to preserve the SIMT execution model of GPUs, we also propose a novel approximate MAC unit with a significantly smaller area overhead. As such, this work introduces AXRAM—a novel DRAM architecture—that integrates several approximate MAC units. AXRAM offers this integration without increasing the memory column pitch or modifying the internal architecture of the DRAM banks. This chapter is based on work presented in Approximate Computing Workshop 2016 [63] and PACT 2018 [64]. This work is a result of collaboration with Choungki Song[1], Jacob Sacks[2], Pejman Lotfi-Kamran[3], Nam Sung Kim[4], and Hadi Esmaeilzadeh[5].

[1]University of Wisconsin-Madison
[2]Georgia Institute of Technology
[3]Institute for Research in Fundamental Sciences
[4]University of Illinois at UrbanaChampaign
[5]University of California-San Diego

3.2 Introduction

GPUs are one of the leading computing platforms for a diverse range of applications—from artificial intelligence to medical prognosis. They are architected to exploit large-scale data-level parallelism in these workloads through simultaneous many-thread execution. However, this processing capability is hindered by the *bandwidth wall* [65, 66, 67] and bottlenecked by overwhelming numbers of concurrent memory requests. Yet, offering higher bandwidth with either conventional DRAM or HBM is challenging due to package pin and/or power constraints. Moreover, raising the pin data transfer rate deteriorates signal integrity and superlinearly increases power [68]. Additionally, the data transfer energy cost is orders of magnitude higher than on-chip data processing [69, 70, 71]. Such a limitation makes near-data acceleration alluring for GPUs. There are two main options for such an integration: (1) 3D/2.5D stacking [72, 73, 74] and (2) integration within DRAM. The former option may incur a significant cost to expose higher internal bandwidth to 3D/2.5D-stacked accelerators than the external bandwidth exposes to the GPU [75], as the TSVs for standard HBM already consume nearly 20% of each 3D-stacked layer [76]. For example, to provide 2× higher bandwidth to the logic layer of High Bandwidth Memory (HBM) compared with the external bandwidth, each 3D-stacked HBM layer requires 2× more TSVs. However, the TSVs for standard HBM already consume nearly 20% of each 3D-stacked layer [76]. Exposing the same internal bandwidth to the HBM logic[6] layer requires adding 6× more TSVs. That is, it may require an additional logic layer to place a sufficient number of accelerators and/or expose higher internal bandwidth by placing more TSVs, which are expensive in terms of many design metrics. By analyzing the die photo of the SK Hynix HBM, which 3D-stacks four DRAM and one logic dies [76], a recent study [75] unveils that there is not enough space to place the accelerators on the logic die. First, the logic die is substantially occupied by *(a) the 1024 PHYs*–which connect to the GPU through 2.5D silicon interposers; *(b) the Through Silicon Vias (TSVs)*–which connect the PHYs

[6] A recent HBM architecture [75, 76] provides 1024 TSVs to the logic layer.

to the 1024 I/O signals of the four 3D-stacked DRAMs; *(c) the decoupling capacitors*–which mitigate the large power fluctuations that may occur when concurrently driving the 1024 PHYs; and *(d) the Memory Built-In Self-Test (MBIST) units and logic/ports*–which enable external testing. Second, the number of TSVs on the logic die needs to be more than doubled to provide higher bandwidth to the accelerators. For instance, an Nvidia GTX 480 architecture [52] has six memory channels, each offering 1024 bits of internal I/O. Exposing the same internal bandwidth to the HBM logic layer requires adding 6× more TSVs. The TSVs for standard HBM already consume nearly 20% of the logic die area [75]. Therefore, the HBM logic layer does not have sufficient area to integrate an effective number of accelerators. Altogether, effectual near-data acceleration for GPUs may require a separate die, which is a non-trivial overhead. In addition, GPUs utilize increasing amounts of memory (Maxwell Titan X has 12GB of GDDR5 memory [77]), while the most recent Samsung HBM2 memory offers 4GB capacity [78] (3× less capacity). As HBM is on-package, it offers limited capacity due to package-level power, thermal, and space constraints in contrast to off-chip DRAM. Due to these limitations, we set out to tightly integrate accelerators within DRAM modules to utilize their higher internal bandwidth and larger capacity. Such a tight integration can be attractive if it incurs *little overhead* while enables the acceleration of a *diverse range of applications*. However, integrating many complex accelerators within DRAM is not practical, since DRAM is under tight area, power, and thermal constraints [79, 80, 81, 82, 83, 84]. Moreover, even the number of metal layers for routing is limited [85, 86, 87], which severely hinders integrating complex accelerators. Finally, it is highly desirable to avoid changing the innards of DRAM banks, as they have been optimized over decades of engineering. This work tackles these challenges by exploiting the approximability of many GPU applications. We leverage the neural transformation [38, 46, 45, 6, 12], which can accelerate diverse applications by approximating regions of GPU code and converting them into a neural representation comprised of only *two* types of operations: Multiply-and-Accumulate (MAC) and Look-Up Table (LUT) ac-

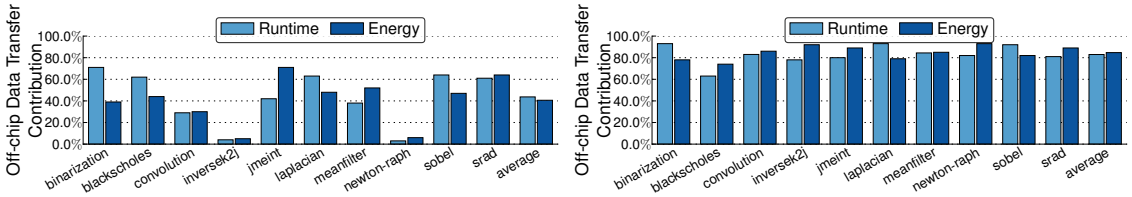
cesses for calculating the nonlinear function. Hence, the accelerator architecture becomes relatively simple. To further minimize the power and area overhead and enable a low-overhead integration of many in-DRAM accelerators, we further approximate the MAC units. Specifically, these approximate MAC units convert the multiplication into *limited* iterations of shift-add and LUT access operations with early termination by exploiting a unique property of neural transformation, i.e., one of the operands for each MAC operation is fixed. While the accelerators merely comprise simple shift, add, and LUT access operations, they are able to support a wide variety of applications. We attach these simplified units to the wide data lines, which connect the DRAM banks to the global I/O, to avoid altering the banks and memory column pitch. Note that our approach, which significantly simplifies the accelerator design, has merits even when accelerators are placed on logic layers of 3D/2.5D-stacked DRAM. Specifically, package-level power/thermal constraints get more stringent with more stacked-DRAM dies while processors powerful enough to fully exploit high-internal bandwidth will consume high power. Also, the challenges of tying DRAM design to accelerators that only cover few applications may be limiting for DRAM manufacturers. AXRAM tackles this dilemma by introducing a significantly simple and power-efficient design while supporting diverse applications as well as neural networks that are being adopted in various domains. As such, this work defines AXRAM, a novel accelerated DRAM architecture with the following contributions.

3.3 Overview

In this section, we first overview the challenges and opportunities of in-DRAM acceleration for GPUs and how approximation plays an enabling role.

3.3.1 Challenges and Opportunities

Opportunity to reduce data transfer cost. Off-chip data transfer imposes a significant energy cost relative to data processing. With a 45 nm process, a 32-bit floating-point addition



(a) Baseline GPU **(b)** Accelerated GPU

Figure 3.1: The fraction of total application runtime and energy spent in off-chip data transfer for (a) a baseline GPU and (b) an accelerated GPU [38].

costs about 0.9 pJ, while a 32-bit DRAM memory access costs about 640 pJ [70, 69]. As such, off-chip data transfer consumes over 700× more energy than on-chip data processing. This cost becomes even more pronounced in GPU applications, since they typically stream data and exhibit low temporal locality (*i.e.,* high cache miss rates) [88, 89, 90, 91, 92, 93]. Moreover, off-chip data transfer assumes a significant portion of application runtime. Near-data processing provides an opportunity to cut down this cost. To concretely examine the potential benefits of near-data processing, we conducted a study which teases apart the fraction of runtime and energy consumption spent on off-chip data transfer[7]. As Figure 3.1a illustrates, on average, applications spend 42% of their runtime and 39% of their energy dissipation on off-chip data transfer on a GPU. In Figure 3.1b, we further examine this trend with a neurally accelerated GPU (NGPU [38]), to speed up the data processing portion of each thread. The acceleration reduces the data processing time of each thread, in turn increasing the rate of accesses to off-chip memory. This increased rate exacerbates the contribution of data transfer to the application runtime and energy. Moreover, accelerating the GPU further compounds the already significant pressure on the off-chip communication bandwidth [38, 66, 65]. On average, applications spend 83% (85%) of their runtime (energy) on off-chip data transfer on neurally accelerated GPU (NGPU). These results indicate a significant opportunity for near-data processing to address the overhead of off-chip data transfer in GPUs.

Challenges of near-data processing on GPUs. GPUs present unique challenges for near-data processing, as they comprise many cores simultaneously running many threads. To

[7]Section 3.9 presents our experimental methodology and settings.

preserve the SIMT execution model of GPUs, we need to integrate many accelerator units near the data. There are two options for where to integrate the accelerator units: (1) the on-chip memory controller or (2) inside the DRAM itself. Option (1) provides the accelerator units with no additional bandwidth, as the on-chip memory controller receives the same bandwidth from memory as the rest of the GPU. Furthermore, placing the accelerator units in the memory controller only circumvents data transfer through the on-chip caches. In addition, integration within the memory controller requires large buffers for holding the accelerators' data, which would impose a significant area overhead. Option (2), which integrates the accelerators in DRAM, reduces the data transfer distance and exploits the high internal bandwidth of the memory. Moreover, integrating the accelerators in DRAM enables us to utilize DRAM as buffers for the accelerators' data. However, this design point can introduce a substantial area and power overhead to the space-limited and power-constrained DRAM. In this work, we integrate the accelerator units in DRAM and leverage the approximability of many GPU applications to significantly simplify the accelerator architecture. These simplifications enable the accelerator to minimize changes to the underlying DRAM architecture and overhead to the DRAM power consumption.

3.3.2 Approximation for Near-Data Processing

Among approximation techniques, the neural transformation [12] is an attractive, yet unexplored, approach for near-data processing in DRAM. The neural transformation converts a code segment into a neural representation comprising only two operations: multiply-and-accumulate (MAC) and sigmoid. Reducing computation to two operations provides an opportunity to significantly simplify the accelerator. This simplified design minimizes changes to the DRAM and can be replicated many times to preserve the GPUs' SIMT execution model.

The neural transformation trains a neural network to replace an approximable region of conventional code [38, 6, 12, 45]. Figure 3.2(a) illustrates the transformation of a code seg-

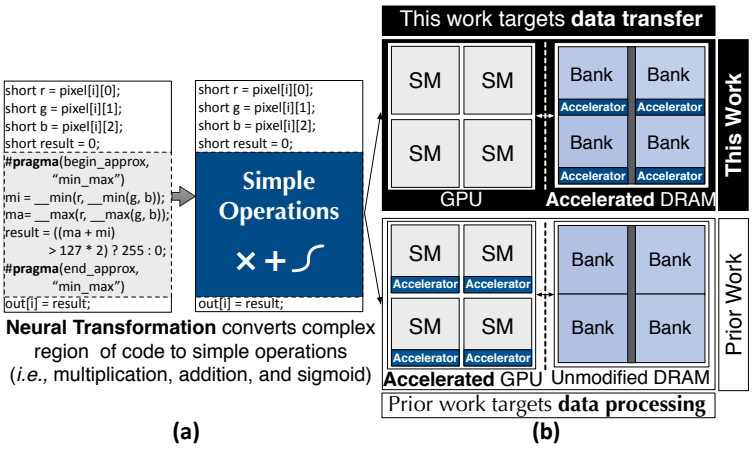


Figure 3.2: (a) Neural transformation of a code segment from the binarization **benchmark. (b) Comparison of prior work (bottom diagram) [38] and this work (top diagram).**

ment, where the approximable region is highlighted in gray. An approximable region is a segment that, if approximated, will not lead to any catastrophic failures (e.g., segmentation fault). Its approximation will only gracefully degrade of the application output quality. As is customary in approximate computing [49, 50, 26, 27], the programmer *only* annotates the code region(s) that can be safely approximated. The compiler then automatically performs the transformation and replaces the code segment with a neural hardware invocation [38]. As shown in Figure 3.2(b), prior work addresses data processing by integrating neural accelerators within the GPU cores and defines a neurally accelerated architecture for GPUs (NGPU) [38]. This work, on the other hand, develops a neurally accelerated architecture for DRAM, dubbed AXRAM, which addresses *off-chip data transfer*. Moving the neural acceleration to DRAM enables AXRAM to reduce the data transfer overhead and supply more bandwidth to the accelerators. Moreover, we leverage the approximability of the GPU applications to further simplify the architecture of the accelerator units (Section 3.7).

3.4 AxRAM Execution Flow and ISA

This section discusses the execution flow and instruction set architecture (ISA) extensions which enable the seamless integration of AXRAM with the GPU's SIMT execution model. Unlike prior work [38, 6, 12, 45, 46], AXRAM is disjoint from the processor core and is

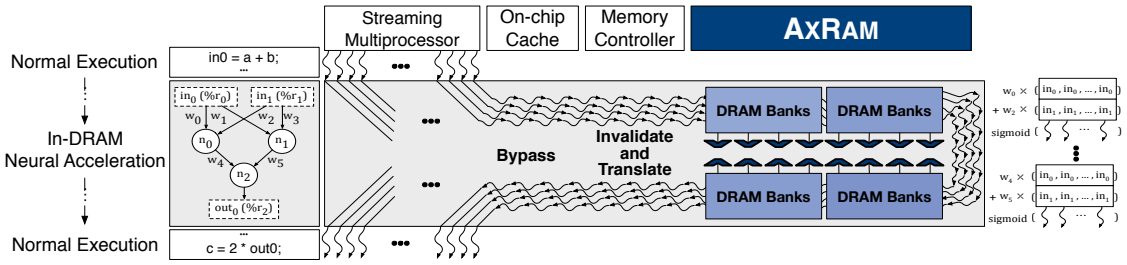


Figure 3.3: Execution flow of the accelerated GPU code on the in-DRAM accelerator.

instead integrated into DRAM. Hence, the ISA extensions must enable the on-chip memory controller to configure and initiate the in-DRAM accelerator.

3.4.1 Neural Acceleration of GPU Warps

GPU applications consist of one or more kernels, which are executed by each of the GPU threads. Threads are executed on GPU processing cores called streaming multiprocessors (SMs), which divide the threads into small groups called warps. A warp executes the same instruction of the same kernel in lock-step but with different input data. The neural transformation approximates segments of the GPU kernels and replaces the original instructions of these segments with the computation of a neural network, as shown in Figure 3.3. A neurally accelerated warp computes the same neural network, one neuron at a time, across all the threads for different inputs. Due to the neural transformation, this computation only consists of MAC and lookup (sigmoid) operations. Specifically, the output y of each neuron is given by $y = sigmoid(\Sigma_i(w_i \times in_i))$, where in_i is the input to the neuron and w_i is the weight of the connection. The neural computation portion of the threads are offloaded to the in-DRAM neural accelerator. Instructions which invoke and configure the in-DRAM neural accelerator are added to the GPU's ISA (Section 3.4.3). These instructions are added by the compiler to the accelerated kernel and are executed by the threads in SIMT mode like other GPU instructions. Thus, the accelerated warp comprises both the normal precise instructions of the unmodified code segments and approximate instructions which communicate with the in-DRAM accelerator. Before explaining these ISA extensions, we provide a high level picture of the execution flow of AXRAM.

3.4.2 Execution Flow with AxRAM

Figure 3.3 illustrates the execution flow of the neurally accelerated warp and communication amongst the GPU, on-chip memory controller, and in-DRAM neural accelerator in one GDDR5 chip. We assume that all data for the neural computation of a given warp is located on one GDDR5 chip. This assumption is enabled by a series of data organization optimizations discussed in Section 3.6. First, the SM fetches the warp and begins the execution of the precise instructions normally without any in-DRAM acceleration. The warp then reaches the approximable region, which instructs the SM to send an initiation request directly to the on-chip memory controller. Once the initiation request has been sent, the issuing warp goes into halting mode. This is *not* an active warp waiting mechanism but is similar to a load miss in the cache. The core may switch to the execution of another warp while the in-DRAM neural computation proceeds, provided the warp does not have any conflicts with the ongoing in-DRAM computation.

Augmented logic in the on-chip memory controller first sends invalidate signals to the on-chip caches and nullifies dirty data to be modified by the neural computation. The invalidate signals are sufficient to prevent GPU cores from using stale data. As most GPU caches use a write-through policy [94], it is guaranteed that in-DRAM accelerators have access to the most up-to-date data. Then, the on-chip memory controller configures and initiates the in-DRAM accelerators (Figure 3.3). Specifically, the on-chip memory controller translates the initiation request and instructs the in-DRAM accelerator where the inputs to the neural network are located in memory and to where the accelerator should store its final outputs. Furthermore, the on-chip memory controller blocks any other memory commands to that particular DRAM chip to ensure the atomicity of the in-DRAM neural computation. The on-chip memory controller also does not assign any other neural computations to a GDDR5 chip with an ongoing neural computation. We added a simple on-chip queue per memory controller to keep track of in-flight requests for in-DRAM approximate acceleration. The area overhead of these queues to the GPU die is modest ($\approx$1%). Similar to [72], the on-chip

memory controller allows critical memory operations such as *refreshing* to be performed during in-DRAM neural computation.

During neural computation, the in-DRAM accelerator takes full control of accessing and issuing commands to the banks. The in-DRAM accelerator performs the MAC and sigmoid operations (Figure 3.3). Neural computation for the threads of the neurally accelerated warp is performed in lock-step by the many integrated arithmetic units. Once neural computation is completed, the in-DRAM accelerator writes its results back to the banks in locations dictated by the memory controller. We consider two options for notifying GPU that in-DRAM computation has completed: waiting a fixed number of cycles and polling. The former approach requires pre-determining the execution time of each invocation and exposing that to the compiler. The memory controller would then wait for this pre-determined number of cycles before notifying the warp to continue precise execution. However, the execution time of an in-DRAM invocation depends on the neural network topology and the accelerator's DRAM accesses patterns. Anticipating the DRAM's accesses patterns necessitates exposing DRAM microarchitectural parameters to the compile. These details are not always readily available, making this design point less desirable. Instead, we choose the polling approach, in which the accelerator sets the DRAM memory-mapped mode register MR0 [95], similar to [72]. The on-chip memory controller periodically polls this register to determine if the computation has finished. Once it detects that the register has been set, the on-chip memory controller notifies the GPU that the neural computation for the specific warp is finished and the warp can continue precise execution. To enable the controller to properly initiate and configure the in-DRAM accelerator, we need to extend the ISA with instructions that communicate the configuration data.

3.4.3 ISA Extensions for AxRAM

We augment the ISA with three instructionswhich bypass the on-chip caches and communicate directly with the memory controller. The proposed ISA extensions are as follows:

1. **config.axram** [%start_addr], [%end_addr]

   reads the preloaded neural network configuration from the memory region [%start_addr] to [%end_addr] and sends it to the in-DRAM accelerator. The configuration includes both the weight values and the topology of the neural network.

2. **initiate.axram** [%start_addr], [%end_addr]

   sends the start ([%start_addr]) and end ([%end_addr]) addresses of a continuous memory region which constitutes the neural network inputs for the warp and then initiates the in-DRAM accelerator.

3. **wrt_res.axram** [%start_addr], [%end_addr]

   informs the in-DRAM accelerator to store the computed value(s) of the neural computation in a continuous memory region defined by the start ([%start_addr]) and end ([%end_addr]) addresses.

Both invoke.axram and wrt_res.axram use virtual addresses like normal CUDA instructions.

The dimensionality of the different neural network layers is statically identified at compile time and used to configure the in-DRAM accelerator. Thus, the in-DRAM accelerator knows how many neurons to expect per layer, and specifying sufficient memory regions to ensure proper execution. However, this means that input order is important and necessitates a series of data organization optimizations to ensure correct execution (See Section 3.6). As with other GPU instructions, these ISA extensions are executed in SIMT mode. That is, each thread in a warp will communicate its input/output data regions to the in-DRAM neural accelerator. Additionally, the weights and the topology of each neural network are embedded by the compiler in the ".data" section of the ELF-formatted CUDA binary code (cubin) [96] during compilation. Along with the CUDA binary code, the weight values and the topology of the trained neural network are copied in a preallocated memory region. Using the **config.axram** instruction, the in-DRAM accelerator pre-loads these weights and topology configuration of the trained neural network from memory before starting the

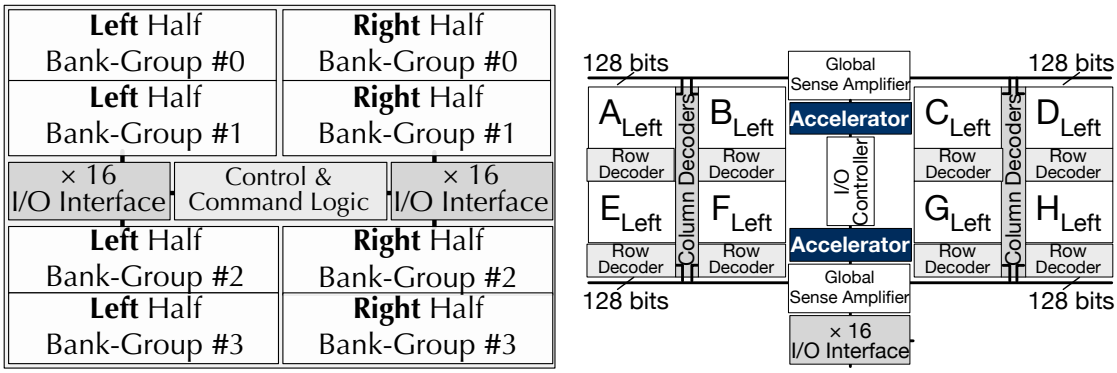
neural computation. These ISA extensions unify the execution flow of AXRAM and the GPU. The microarchitectural modifications to the DRAM need to support such a unified execution flow while minimizing changes to the DRAM architecture.

3.5 AxRAM Microarchitecture

To describe our design, we use a GDDR5 DRAM architecture [86, 87, 97]. High Bandwidth Memory (HBM), a 3D-stacked DRAM architecture, has also recently been employed as the memory for high-end GPUs (*e.g.,* AMD's Fiji architecture [98, 99]). HBM is a 3D-stacked DRAM architecture which is placed side-by-side with the GPU and connects to it through 2.5D interposers. Since HBM generally stacks GDDR5-like DRAM [100], our modifications can potentially be extended to such memory architectures, in which the logic die does not have sufficient space to expose higher internal bandwidth (cf. Section 3.2). Furthermore, AxRAM is appropriate for these 3D-stacked structures, because, as our evaluations show (see Section 3.9), our design does not increase the DRAM power consumption due to data transfer. Our main design objectives are to (1) preserve the SIMT execution model while (2) keeping the modifications to the baseline GDDR5 minimal and (3) leveraging the high internal bandwidth of DRAM. AxRAM achieves these goals by integrating many simple arithmetic and sigmoid units into GDDR5. To describe the microarchitecture of AXRAM, we first give an overview of the GDDR5 architecture.

3.5.1 Background: GDDR5 Architecture

While GDDR5 has a I/O bus width of 32 bits per chip, it has a much higher internal bus width of 256 bits per bank. This provides an 8× higher bitwidth that would significantly benefit GPUs, which already place significant pressure on the off-chip bandwidth [65, 66, 101]. Furthermore, the bank-group organization of GDDR5 provides intrinsic parallelism which can be leveraged to feed data to a large number of arithmetic units. By exploiting the attribute of the bank-group organization, we can further utilize 1024 bits of internal bus



(a) High-level GDDR5 DRAM **(b)** Two Half Bank-Groups

Figure 3.4: (a) High-Level GDDR5 DRAM organization. (b) Layout of two half bank-groups (Left Half Bank-Group #0 and Left Half Bank-Group #1) and the accelerators. The black-shaded boxes show the placement of the accelerators.

width (32× higher bitwidth than the I/O bus).

Figure 3.4a shows the GDDR5 DRAM architecture, which consists of four bank-groups, each with four banks. Each bank-group can operate independently, meaning requests to different bank-groups can be interleaved. The bank-groups are organized into upper and lower pairs partitioned by the I/O interface and control and command logic. Moreover, each bank-group contains four banks, which are subdivided into two half-banks. Subdividing the banks splits each bank-group into a left and right half, each with four half-banks. Two upper-left half bank-groups (*i.e.*, Left Half Bank-Group #0 and Left Half Bank-Group #1) are depicted in Figure 3.4b. In each half bank-group, the four half-banks are split into pairs (*e.g.,* A_{Left} and B_{Left} vs. C_{Left} and D_{Left}) by a global sense amplifier and shared I/O controller. Each half-bank has its own row decoder, while column decoders are shared between the half-bank pairs of the two adjacent half bank-groups. Both the right and left half bank-groups provide a bus width of 128 bits for a total of 256 bits. However, this higher internal bus width is serialized out through the right and left 16-bit I/O interface.

For instance, when the DRAM receives a memory command to access Bank A in Bank-Group #0, both the half-banks, A_{Left} and A_{Right}, process the command in unison to supply the data. For the sake of simplicity, we focus on the left half of Bank-Group #0 shown in Figure 3.4b. The global row decoder of the half-bank decodes the address and accesses the data. The shared column decoder asserts the column select lines, which drives the data

onto a 128-bit global dataline shared between half-banks A_{Left} and B_{Left}. Since the global dataline is shared between the pairs of half-banks, only one may send or receive data at a time. The global sense amplifier then latches the data from the global dataline and drives the data on the bank-group global I/O through the I/O controller. The right and left I/O interfaces then serialize the 256-bit (128-bit each) data on the Bank-Group #0's global I/O before sending them through the 32-bit data I/O pins. By placing the accelerators inside the DRAM, we aim to exploit the higher internal bandwidth instead of relying on the lower bandwidth of the data I/O pins.

We next discuss how AXRAM integrates the accelerators into the GDDR5. Additionally, we describe how the accelerator uses the aforementioned GDDR5 attributes to preserve the SIMT execution model and minimize changes while providing data to all the arithmetic units each cycle.

3.5.2 In-DRAM Accelerator Integration

To minimize DRAM changes yet benefit from its high internal bandwidth, AXRAM integrates a set of arithmetic and sigmoid units within each half-bank group (Figure 6.5). These arithmetic and sigmoid units are connected to the half-bank groups' global sense amplifiers. Below we discuss the design choices, structure, and components of this integration.

Accelerator architecture. As mentioned, the accelerator is a set of arithmetic and sigmoid units. Each pair of arithmetic and sigmoid units is assigned to a thread of the neurally accelerated warp. The sigmoid units are implemented as a read-only LUT synthesized as *combinational logic* to minimize the area overhead. We will further simplify the arithmetic units in Section 3.7. Here, we discuss how we guarantee SIMT execution of the neurally accelerated warp with these units. Each arithmetic unit can execute one MAC operation each clock cycle with a *32-bit input* and *32-bit weight*. The banks need to be able to feed a 32-bit input to each of the integrated arithmetic units at the same time, such that the arithmetic units can perform the neural computation for all the threads within a warp

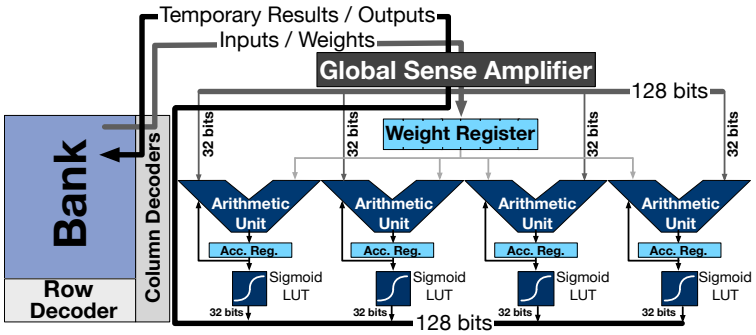


Figure 3.5: Integration of weight register, arithmetic units, accumulation registers, and sigmoid LUTs.

simultaneously. As mentioned before, each bank-group in the baseline GDDR5 has a 256-bit wide global I/O, 128-bit per each half bank-group. Since each bank group can function independently, the DRAM can provide a total of 1024 bits (32×32 bits) of data from the banks at a time. Thus, we integrate 32 pairs of arithmetic and sigmoid units in each GDDR5 chip, 8 pairs per each bank-group. In Section 3.6, we describe a data organization which enables us to read and write 1024 bits of data simultaneously.

Unit placement. There are multiple design points for integrating arithmetic units within a GDDR5 chip. To minimize changes to the DRAM architecture, we aim to avoid modifying the underlying mat[8] and bank design. One option is to add arithmetic units close to each half-bank to utilize their high internal bandwidth. However, this would require cutting the global datalines shared between pairs of half-banks (Figure 3.4b) and adding a separate sense amplifier per half-bank. Therefore, this design point imposes a large area overhead and necessitates significant changes to each GDDR5 chip. Another option is to add arithmetic units in a central manner close to the I/O interface in Figure 3.4a. Although this option does not suffer from the drawbacks of placing the accelerators close to each half-bank, it requires extensive routing. Because the aforementioned options require such extensive modifications, they are infeasible design points. Instead, AXRAM adds four arithmetic units per half bank-group after the shared sense amplifier within the I/O controller boundary, for a total of eight arithmetic units per bank-group. The accelerators'

[8] A mat constitutes an array of 512×512 DRAM cells. Each mat comes with its own row decoder, datalines, and sense amplifiers.

placement is illustrated in Figure 3.4b, while the specific accelerator logic layout, including the arithmetic units, is shown in Figure 6.5. This design choice imposes minimal changes to the DRAM architecture and avoids altering the design of the mats or banks.

Design optimizations. Each of the arithmetic units implements the neural network MAC operations. However, to properly supply and retrieve data from the arithmetic units, we need storage for the (1) inputs, (2) weights, (3) temporary results, and (4) outputs of the network. Generally, neural accelerators use dedicated buffers as storage [38, 12]. However, placing the arithmetic units near the data allows AXRAM to perform a series of design optimizations which minimize the modifications to the baseline GDDR5. As Figure 6.5 shows, AXRAM is able to instead use the GDDR5 banks as buffers. Input data is read directly from the GDDR5 banks and fed to the arithmetic units for processing. AXRAM leverages the large number of sense amplifiers within the DRAM banks to store temporary results in pre-allocated memory regions during in-DRAM computation. Outputs from the arithmetic units are written directly back to the GDDR5 banks. By not using dedicated buffers, we avoid adding large registers to each GDDR5 chip and reduce the area overhead. We only add dedicated weight registers to supply weights to all the arithmetic units. This enables AXRAM to avoid having to read the weights from the memory banks each cycle and instead utilize the internal buses to supply all the arithmetic units with inputs. Thus, we can simultaneously provide each arithmetic unit with an input and weight each cycle.

Weight register. Since all threads within a warp perform the computation of the same neuron in lock-step, the weights are the *same* among all the threads for a given neural network. Therefore, AXRAM can use *one weight* at a time and share it among the arithmetic units within a half-bank group. We add a weight register (shown in Figure 6.5) per half bank-group, or for each group of four arithmetic units. As shown in Figure 6.5, the weights are pre-loaded into the weight register before the computation starts. If the number of weights exceeds the capacity of the register, the next set of weights are loaded after the first set has been depleted. This weight register has 8×32-bit entries per each half bank-group.

Since each half bank-group can provide 128 bits of data at a time, the weight register should have at least four entries to fully utilize the provided bandwidth. We increase the number of weight register entries to allow computation to move forward while the next set of weights are loaded and avoid unnecessary stalls.

GDDR5 timing constraints. Adding arithmetic units to the half bank-groups increases the load to the half bank-groups' global I/Os.The only timing constraint affected by the increased load is the column access latency (tCL). To estimate the timing impact of tCL by HSPICE simulation, we measure the increase in load due to the accelerator on the GIOs after the placement and routing. Based on our evaluation, the extra loading on the half bank-groups' global I/Os increases the tCL by $\approx$ 20 ps. This increase is 0.16% of the typical value for tCL, which is around 12.5 ns to 15 ns [102, 103], and is less than the guardband which accounts for various design variations [104]. Thus, the 20 ps increase has virtually no effect on the timing of GDDR5.

Connection between DRAM banks and arithmetic units. The internal half bank-groups' global I/Os need to support two different modes: (1) normal mode and (2) in-DRAM acceleration mode. When the accelerator performs the computation, the half bank-group's global I/Os are connected to the arithmetic units to transmit input data. Once the computation of a neuron completes, the arithmetic unit inputs arithmetic units are disconnected from the half bank-group's global I/Os. The arithmetic units outputs are then connected to the global datalines through the global I/Os for storing the computed data into the memory banks. We use a series of pass transistors to control the connection between the inputs and outputs of the arithmetic units and the GDDR5 half bank-groups. Supporting a direct connection between the arithmetic units and the GDDR5 banks also requires additional routing paths in the DRAM. To enable the in-DRAM accelerator to gain access of the GDDR5 chip, we also modify the internal address/command bus. In normal mode, the on-chip memory controller has the full access of the address/command bus. However, in in-DRAM acceleration mode, the accelerator gains access to the address/command bus. A set of pass transistors supports

this functionality in memory as well. We evaluate the overhead of pass transistors and routing paths in Section 3.9. To orchestrate the flow of data in the banks to and from the in-DRAM accelerator, we add an in-DRAM controller. Furthermore, we augment the on-chip memory controller with additional logic to translate the ISA extensions and properly initiate and configure the in-DRAM accelerator.

3.5.3 Interfacing the GPU with AxRAM

Memory controller. We extend the on-chip memory controllers to send invalidation signals to the on-chip caches upon receiving AXRAM instructions. Moreover, we extend the on-chip memory controller to translate the AXRAM instructions (Section 3.4) to a sequence of special memory instructions. These memory instructions (1) configure the in-DRAM accelerator and (2) initiate the in-DRAM neural computation. The on-chip memory controller is augmented with customized address mapping logic to perform this translation. Upon receiving AXRAM instructions, the implemented address mapping logic inside each on-chip memory controller sends a series of special memory commands to the in-DRAM accelerator to configure and initiate the in-DRAM acceleration. We also add a one-bit flag inside each memory controller to keep track of the status of its corresponding GDDR5 chip. During in-DRAM neural computation, the flag is set so that the memory controller knows not to issue any further memory commands to the memory chip.

However, the memory controller may regain the ownership of the memory chip for performing mandatory memory operations such as refreshing [105]. Similar to prior work [72], the memory controller sends a `suspend` command to the in-DRAM controller if the GDDR5 chip is in neural computation mode. Upon receiving the `suspend` command, the in-DRAM control unit stores any temporary results in the DRAM and stops computation. Once the refresh period finishes, the memory controller instructs the in-DRAM controller to continue the suspended neural computation.

In-DRAM controller. Previous work [106] has proposed integrating an on-DIMM con-

troller and a handful of specialized microcontrollers in memory to accelerator associative computing. However, since the neural network does not require a complicated controller [38, 12], we instead add a simple control unit inside each GDDR5 chip. This in-DRAM controller (1) marshals data and weights between memory banks and the in-DRAM accelerator and (2) governs the sequence of neural network operations. Specifically, it fetches input data from the banks and sends them to the arithmetic units, reads weights from memory and loads them into the weight buffers, and stores temporary results and neural output(s) into the banks. When the in-DRAM controller receives instructions from the on-chip memory controller, it gains full control of the internal DRAM buses. As discussed, the memory controller only re-gains ownership of the internal DRAM buses when neural computation completes and for performing mandatory memory operations such as random refreshing.

3.6 Data Organization for AXRAM

Our proposed architecture (Section 3.5) leverages bank-group level parallelism to supply all arithmetic units with inputs simultaneously. For this design to comply with the SIMT execution model, we require data to be laid out in a specific order on a single GDDR5 chip. Recent work [107, 108, 109] has shown the benefits of data organization in improving the efficiency of near-data processing for certain applications. A neural network execution has consistent and predictable memory access patterns [110, 111]. Similar to recent work [108], we leverage the predictability of the memory access patterns in neural network execution to perform a series of data organization optimizations to fully utilize the inherent *bank-group* and *bank-level* memory parallelism in memory. Since the weights of the network are shared amongst all the threads and loaded into the weight register before in-DRAM neural computation, we only need to ensure that the input data is properly placed in memory.

Data partitioning. We logically divide a warp into *four* partitions, each with *eight* threads. The data for all the eight threads of each partition is allocated within each bank-group.

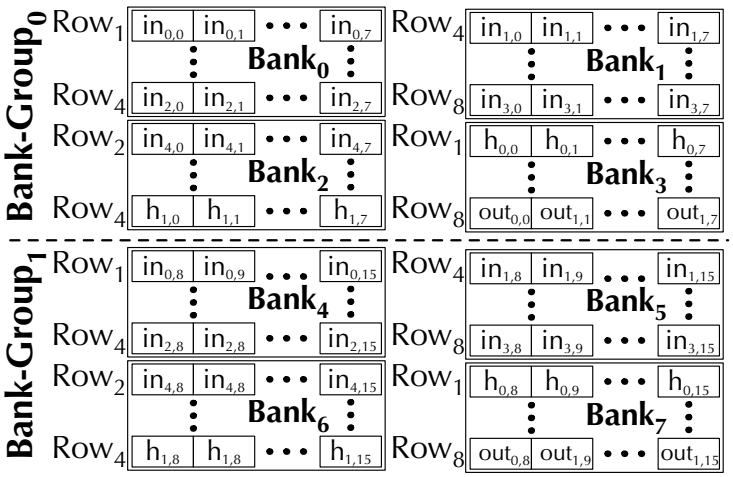


Figure 3.6: The data layout for a neural network with 5→2→1 configuration in bank-group$_0$ and bank-group$_1$ after data shuffling. For simplicity, we assume a row buffer (256 bits).

That is, the data for the first partition of threads (e.g., thread$_{0-7}$) is allocated to the first bank-group. Similarly, the data for *thread*$_{8-15}$, *thread*$_{16-23}$, *thread*$_{24-31}$ is allocated to the *second*, *third*, and *fourth* bank-group, respectively. If there is shared data between warps, we replicate it during the data partitioning. On average, the overhead of duplicated data is ≈2% in terms of storage.

Data shuffling. Within a partition, the data has to be organized in such a way that we can read and write all the data for the 32 arithmetic units at a time and efficiently utilize the bank-level parallelism. Specifically, AXRAM requires two constraints to be met for the data layout: (1) the row and column addresses of a given neuron's inputs for all the 32 threads have to be the same across the bank-groups and (2) the addresses of a neuron's inputs for each thread in a given partition have to be consecutive. Furthermore, similar to address mapping in baseline GPU [112], data for different neurons for a given partition is distributed among the banks to enable interleaving requests to different banks on the chip.

We illustrate this scheme with an example (Figure 3.6), assuming a topology with a 5→2→1 configuration. For simplicity, we assume banks with 256-bit sense amplifiers and only show the layout for thread$_{0-15}$ of a warp. Input neuron i has an input $in_{i,j}$ for the j^{th} thread. Similarly, $h_{i,j}$ and $out_{i,j}$ represent inputs for the i^{th} hidden and output neuron, respectively, for the j^{th} thread. Due to the first constraint, inputs for the 0^{th} input neuron

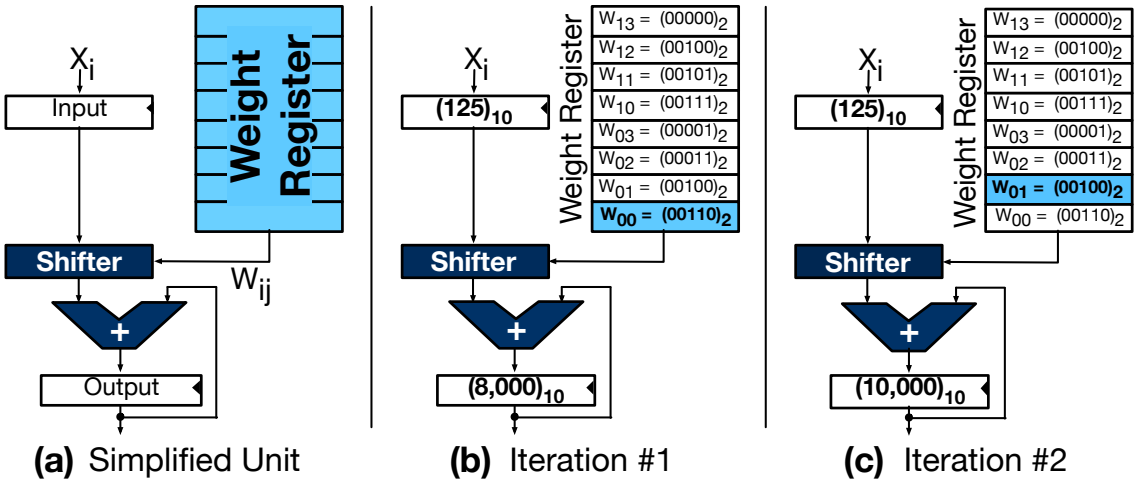


Figure 3.7: (a) Example of the simplified shift-add unit with pre-loaded shift amounts. (b-c) Two iterations of the shift-add unit.

for all threads are located in row_1 for both bank-groups. Following the second constraint, inputs $in_{0,0}$-$in_{0,7}$ and $in_{0,8}$-$in_{0,15}$ are consecutively placed in the same row in $bank_0$ and $bank_1$ respectively. The same constraints are met for the other neurons as well. Due to bank interleaving, inputs $in_{0,0}$-$in_{0,7}$ are stored in $bank_0$, while the inputs $in_{1,0}$-$in_{1,7}$ are in $bank_1$.

Memory management APIs. Similar to AMD's early generation APUs [113], we adopt a memory model which provides a single physical memory space divided into two separate and non-overlapping logical memory spaces for the GPU and in-DRAM neural accelerator respectively. The separation between the GPU and in-DRAM accelerator data and the proposed data partitioning and shuffling schemes are performed on-the-fly when the host transfers the data to the GPU memory during kernel initialization using customized memory management APIs. We use an approach similar to prior work [114, 115] and modify the CUDA driver API (*e.g.* `cuMemCopyHtoD()`, `cuMemCopyDtoH()`) to implement the proposed data organization optimizations (*e.g.* data partitioning and shuffling) for in-DRAM neural acceleration. The overhead of performing the proposed data organization is amortized over the long CUDA kernel execution time and is accounted for in Section 3.9.

3.7 Arithmetic Units Simplification

There exist two options for the arithmetic units. The first option is to use floating-point arithmetic units to perform the neural computation. Another option is to use fixed-point arithmetic units for energy gains and a smaller area overhead. We propose a third option to approximate the arithmetic units to further reduce the area overhead and keep the impact on the overall DRAM system power low. These simplified arithmetic units break down the MAC operations into iterations of add and shift operations. More iterations of this shift-add unit offers higher precision at the expense of the throughput of the unit. Since the weights W_i remain constant after training a neural network, the shift amounts can be pre-determined based off the bit indices of ones within the 32-bit weight value, starting with the most significant one. Figure 3.7a shows an implementation of this simplified shift-add unit. X_i represents the input of a neuron and W_{ij} is the shift amount for the i^{th} weight in its j^{th} iteration. The weight register stores these predetermined shift amounts. Since the shift amounts are indices of bits within a 32-bit weight value, the maximum shift amount is 32, which can be represented by a 5 bit value. Thus, each 32-bit entry in the weight register can hold a total of five shift amounts.

Figure 3.7 shows the design using an example in which $W_i = 01011010_2(90_{10})$ and $X_i = 01111101_2\ (125_{10})$. Multiple iterations of the simplified shift-add unit execution are shown in Figure 3.7b and 3.7c. The W_{ij} shift amount can be pre-determined by obtaining the bit index of the j_{th} leading one of W_i. In this example, the most significant one in W_i is in the *sixth* bit position, meaning X_i is shifted by $W_{00} = 6_{10} = 110_2$. The result is then accumulated to the sum, which is initialized to zero. The *first* iteration (Figure 3.7b) yields 8000_{10}, which achieves 71% accuracy to the actual sum 11250_{10}. More iterations leads to higher accuracy at the cost of higher energy consumption. The *second* (Figure 3.7c), *third*, and *fourth* iterations achieve 89%, 98%, and 100% (*e.g.* zero accuracy loss) accuracy, respectively. We evaluate the trade-offs between different arithmetic units for in-DRAM

neural acceleration in Section 3.9.

3.8 Memory Model

Virtual memory. Current GPUs support simple virtual memory [116, 117, 118, 119, 120, 121]. AXRAM instructions use virtual addresses similar to CUDA instructions. Once a GPU core issues an AXRAM instruction, the virtual addresses are translated to physical addresses through TLBs/page tables placed in the on-chip memory controllers, similar to other CUDA instructions [116, 121]. Then, the physical addresses are sent to the memory for in-DRAM neural computation. Virtual address support in AXRAM instructions expels the need to modify the underlying GPU virtual memory management system.

As mentioned in Section 3.6, to fully utilize the inherent parallelism in memory, AXRAM requires the data to be allocated in a consecutive memory region. Most CUDA-enabled GPUs do not support on-demand paging [122, 123]. Thus, all the virtual memory locations are backed by actual physical memory before the kernel initialization. To guarantee that a contiguous virtual memory is translated to a consecutive physical memory, we use our proposed custom memory management API to copy the allocated data to consecutive physical pages before the kernel execution. Additionally, AXRAM may be extended to HSA-enabled GPUs [124]. One potential solution is to raise a page fault exception if the data for an in-DRAM invocation is not in the memory. The in-DRAM accelerator will then stall until all the demanded pages are loaded into the memory. Exploring the challenges and opportunities for integrating in-memory accelerators to HSA-enabled GPUs is outside the scope of this work.

Cache coherency. We adopt a similar technique as [108] to guarantee the cache coherency in AXRAM. The AXRAM instructions bypasses the on-chip caches and communicate directly with on-chip memory controller. A GPU core always pushes all of its memory update traffic to memory before issuing any of the AXRAM instructions. Sending memory update traffic along with write-through policy used in most GPUs [125] ensure that the

in-DRAM accelerators have access to the most up-to-date data. `wrt_res.axram` is the only AXRAM instruction that updates the data in memory. Upon receiving this instruction and in order to guarantee cache coherency, the on-chip memory controller sends a series of invalidate signals to on-chip caches and nullify any cache block that will be updated by the offloaded in-DRAM computation. The invalidate signals ensure that GPU cores never consume stale data. On average, it takes ten cycles to invalidate all the cache lines related to one neural execution. Based on our evaluation, the overhead of sending the invalidate signals to guarantee cache coherency is, on average, only 1.9%.

Memory consistency. The neural transformation does *not* introduce additional memory accesses to the approximable region. Therefore, there is no need to alter the applications. AXRAM simply maintains the same memory consistency model as the baseline GPU.

3.9 Evaluation and Methodology

We evaluate AXRAM with our simplified shift-add units (AXRAM-SHA), fixed-point arithmetic units (AXRAM-FXP), and floating-point arithmetic units (AXRAM-FP).

3.9.1 Methodology

Applications and datasets. As Table 6.1 shows, we use a diverse set of benchmarks from the AXBENCH suite [126] to evaluate AXRAM. AXBENCH comprises a combination of memory- (blackscholes, jmeint, and srad) and compute-intensive applications and comes with annotated source code, the compiler for neural transformation, separate training and test data sets, and quality measurement toolsets [126]. These benchmarks represent workloads from image processing, finance, machine learning, robotics, 3D gaming, vision, numerical analysis, and medical imaging. Datasets used for measuring quality, performance, and energy are completely disjoint from those used to train the neural networks.

Neural networks. Table 6.1 shows the neural network topology automatically discovered by the AXBENCH compiler [126] which replaces the annotated code region. For instance,

Table 3.1: Applications (from AXBENCH [126]), quality metrics, train and evaluation datasets, and neural network configurations.

Applications			Input Dataset		Neural Network
Name	Domain	Quality Metric	Training	Evaluation	Topology
binarization	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	3→4→2→1
blackscholes	Finance	Avg. Rel. Error	8,192 options	262,144 options	6→8→1
convolution	Machine Learning	Avg. Rel. Error	8,192 data points	262,144 data points	17→2→1
inversek2j	Robotics	Avg. Rel. Error	8,192 2D coordinates	262,144 2D coordinates	2→16→3
jmeint	3D Gaming	Miss Rate	8,192 3D coordinates	262,144 3D coordinates	18→8→2
laplacian	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	9→2→1
meanfilter	Machine Vision	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	7→4→1
newton-raph	Numerical Analysis	Avg. Rel. Error	8,192 cubic equations	262,144 cubic equations	5→2→1
sobel	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	9→4→1
srad	Medical Imaging	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	5→4→1

the topology for blackscholes is $6 \rightarrow 8 \rightarrow 1$ (6 input neurons, 1 hidden layer with 8 neurons, and 1 output neuron). These topologies were automatically discovered by the AXBENCH compiler [126]. As shown by the results, different applications require different topologies to minimize quality loss.

Quality. As shown in Table 6.1, we use application-specific quality metrics provided by AXBENCH [126] to assess the output quality of each application after in-DRAM acceleration (Section 3.9.2). This quality loss is due to accumulated errors from repeated execution of the approximated region.

Cycle-level microarchitectural simulation. We use the `GPGPU-Sim 3.2.2` cycle-level microarchitectural simulator [59] modified with our AXRAM ISA extensions with the latest configuration which closely models an `NVIDIA GTX 480` chipset with a Fermi architecture.[9] For the memory timing, this configuration models the GDDR5 timing from `Hynix` [95]. Additionally, we augmented the simulator to model the microarchitectural modifications in the GPU, the memory controller, and the GDDR5 for in-DRAM neural acceleration. The overheads of the extra instructions and logics in AXRAM, on-chip memory controller invalidate signals, and the data partitioning and shuffling are faithfully modeled in our simulations. For all the baseline simulations that do not include any approximation or acceleration, we use a plain version of `GPGPU-Sim`. Table 3.2 summarizes the microarchitectural parameters of the GPU and GDDR5 DRAM. We use NVCC 4.2 with -O3 to enable aggressive compiler optimizations. Furthermore, we found optimal kernel parameters, such as number of thread blocks and threads per block of each kernel, separately for each

[9]NVIDIA GTX 480 is the latest configuration in GPGPU-Sim as of time of submission.

Table 3.2: Major GPU, GDDR5, and in-DRAM neural accelerator microarchitectural parameters.

System Overview	15 SMs, 32 Threads/Warp, 6 × 32-bit P2P Memory Channels
Shader Core	1.4 GHz, 1,538 Threads (48 Warps), 32,768 registers, GTO Scheduler [84] Two Schedulers / SM
L1 Data Cache	16 KB, 128B Cache Line, 4-Way Associative, LRU Replacement Policy [44] Write Policy: Write-Evict (hit), Write No-Allocate (Miss)
Shared Memory	48 KB, 32 Banks
Interconnect	1 Crossbar/Direction (15 SMs, 6 MCs), 1.4 GHz
L2 Cache	768 KB, 128B Cache Line, 16-Way Associative, LRU Replacement Policy [44] Write Policy: Write-Evict (hit), Write No-Allocate (Miss)
Memory Model	6 × GDDR5 Memory Controllers (MCs), Double Data Rate ×32 mode 64 Columns, 4K Rows, 256 Bits/Column, 16 Banks/MC, 4 Bankgroups 2KB Row Buffer/Bank, Open Row Policy, FR-FCFS Scheduling [81, 82] 177.4 GB/Sec Off-Chip Bandwidth
GDDR5 Timing [40]	t_{WCK} = 3,696 MHz, t_{CK} = 1,848 MHz, t_{CL} = 12, t_{RP} = 12, t_{RC} = 40 t_{RAS} = 28, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12, t_{CCD} = 2, t_{CCDL} = 3 t_{RTPL} = 2, t_{FAW} = 23, t_{32AW} = 184
GDDR5 Energy	RD/WR without I/O = 12.5 pJ/bit [40], Activation = 22.5 pJ/bit [40] DRAM I/O Energy = 2 pJ/bit, Off-Chip I/O Energy = 18 pJ/bit [70, 95]
Arithmetic Unit Energy [26, 34]	32-bit Floating-Point MAC = 0.14 nJ, 32-bit Fixed-Point MAC = 0.030 nJ 32-bit Approximate MAC = 0.0045 nJ, 32-bit Register Access = 0.9 pJ

Table 3.3: Area overhead of the added major hardware components.

Hardware Units	Area (mm^2)	
	8 Metal Layers	3 Metal Layers
AxRAM-SHA (32 × 32-bit Approximate MACs)	0.09	0.15
AxRAM-FxP (32 × 32-bit Fixed-Point MACs)	0.40	0.76
AxRAM-FP (32 × 32-bit Floating-Point MACs)	0.54	0.97
64 × 32-bit Weight Registers	0.03	0.06
32 × Sigmoid LUT ROMs	0.19	0.34
In-DRAM Controller	0.23	0.40

benchmark in our simulated architecture. In all the experiments, we run the applications to completion.

Circuit and synthesis. We use the `Synopsys Design Compiler (J-2014.09-SP3)` with a `NanGate 45nm` library [127] for synthesis and energy analysis of our architecture. Additionally, we use `Cadence SoC Encounter (14.2)` for placement and routing. As DRAM technology has only three metal layers, naïvely taking the area numbers from the `Synopsys Design Compiler` underestimates the area. To account for this, we restrict the number of metal layers to three in `Cadence SoC Encounter` for I/O pins and routing. We measure and report the area overhead of the added hardware components after the placement and routing stage with three metal layers. Similarly, for the added registers, we extract the area after the placement and routing stage while restricting the number of metal layers to three. With this infrastructure, we analyze the proposed arithmetic units, in-DRAM controllers, routing multiplexers, bypass transistors, and sigmoid LUTs.

Energy modeling. To measure the energy numbers, we use `GPUWattch` [60]. We also modified the `GPGP-Sim` to generate an event log of the in-DRAM neural accelerator and all the other added microarchitectural components. We use the collected event logs to measure

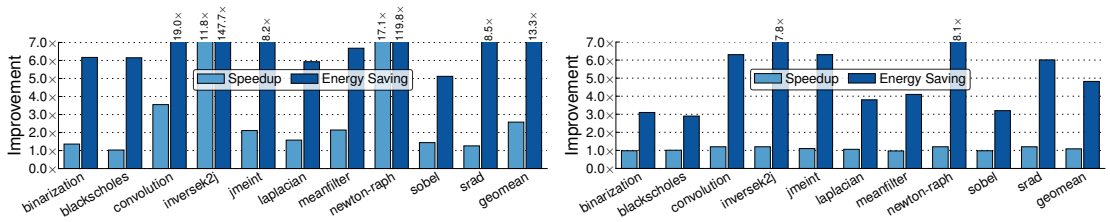
the energy of the in-DRAM neural acceleration. Our energy evaluations use a NanGate 45nm [127] process node and 1.4GHz clock frequency for the shader core (see Table 3.2 for further details). In-DRAM AXRAM changes are modeled using McPAT [32] and CACTI 6.5 [33]. Since GPUWattch uses the results from McPAT and CACTI, our added energy models provide a unified and consistent framework for energy measurement.

3.9.2 Experimental Results

Performance and energy benefits with AXRAM-SHA. Figure 3.8 shows the whole application speedup and energy reduction when all the warps undergo approximation, normalized to a baseline GPU with no acceleration and an accelerated GPU (NGPU) [38], respectively. The highest speedups are in inversek2j and newton-raph, where a large portion of their execution time is spent in the approximable region. The speedup with AXRAM-SHA compared to NGPU is modest, because in AXRAM-SHA, we use up to four iterations of shift and add operations. On average, AXRAM-SHA provides 2.6× (1.1×) speedup compared to GPU (NGPU).

Figure 3.8 also shows the energy reduction benefit of using AXRAM-SHA normalized to a baseline GPU and NGPU, respectively. The maximum energy reduction are in applications–inversek2j and newton-raph–with the highest contribution of off-chip data transfer to the whole application energy (cf. Figure 3.1). The off-chip data transfer contribution in jmeint is also high (90%). However, this application has a large neural network topology (cf. Table 6.1) which leads to a higher number of accesses to the DRAM banks to read and write temporary data, diminishing the energy reduction. On average, the studied benchmarks enjoy 13.3× (4.8×) energy reduction compared to a baseline GPU (NPU).

Energy reduction breakdown. To better understand the source of energy savings, Figure 3.9 shows the energy breakdown of the DRAM system, data transfer, and data computation for AXRAM-SHA, normalized to NGPU [38]. The first bar shows the breakdown of energy consumption in NGPU [38], while the second bar shows the breakdown of energy



(a) Baseline GPU **(b)** Accelerated GPU (NGPU)

Figure 3.8: AXRAM-SHA whole application speedup and energy reduction compared to (a) baseline GPU and (b) an accelerated GPU (NGPU) [38].

consumption in AXRAM-SHA normalized to NGPU [38]. As the first bar shows, the NPGU [38] significantly reduces the contribution of the data computation in the overall system energy. Therefore, the contribution of the other main parts (e.g., data transfer and DRAM system) increases. The second bar illustrates how AXRAM-SHA significantly reduces the contribution of data transfer between the GPU cores and memory to the overall energy consumption of the system. On average, AXRAM-SHA reduces the energy consumption of data transfer by a factor of 18.5 ×. AXRAM-SHA also reduces the average energy consumption of the DRAM system by a factor of 2.5 × due to (1) decreased I/O activity and (2) a higher row-buffer hit rate. Based on our evaluation, the proposed data organization improves the row-buffer hit rate by 2.6×. Finally, the use of simplified shift-add units reduces the average contribution of data computation to the whole application energy consumption by a factor of 1.7 × compared to NGPU. These results elucidate how AXRAM reduces the overall energy consumption compared to a neurally accelerated GPU (NGPU) [38].

Design overheads. Table 3.3 shows the area overhead of the major hardware components added to ea ch DRAM chips. We implement the added hardware units in `Verilog` and synthesize them with `Design Compiler` using the `NanGate 45 nm` library. Similar to other DRAM architecture research work [128, 129], we use two or three generation older logic technology to have conservative estimations. Then, we use `Cadence SoC Encounter` to perform the placement and routing on the synthesized designs using only three metal layers, similar to the baseline DRAM layout, for both routing and I/O pins. We increase the area up to a point

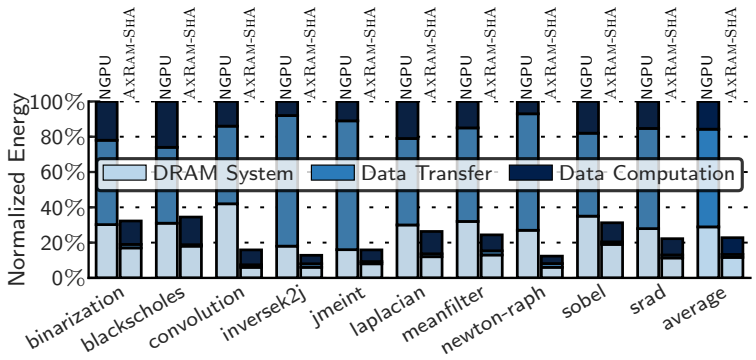


Figure 3.9: Breakdown of AxRam-ShA's energy consumption between DRAM system, data transfer, and data computation normalized to NGPU [38].

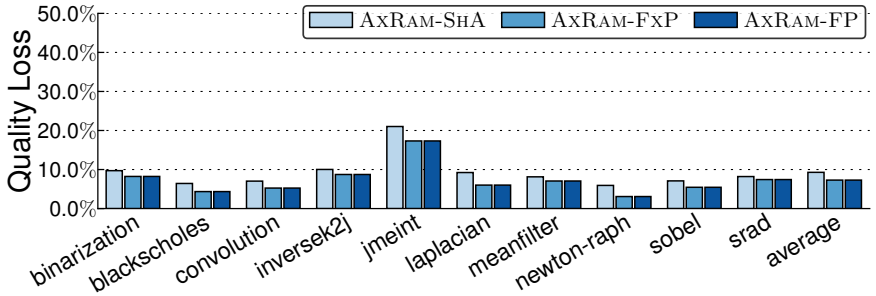


Figure 3.10: Application quality loss with AxRam-ShA, AxRam-FxP, and AxRam-FP compared to a baseline GPU.

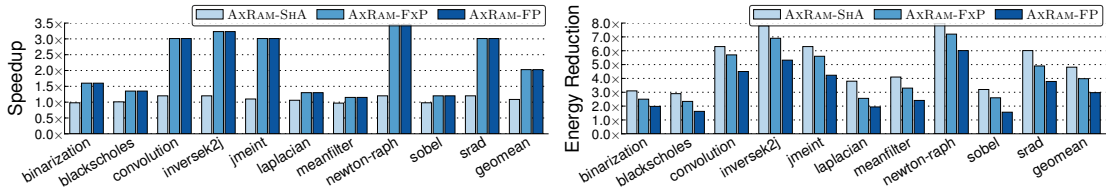
where no placement and routing violations are identified by `Cadence SoC Encounter`. We also obtain the area overhead numbers with 8 metal layers. On average, the area overhead with three metal layers is ≈1.9× higher than with eight metal layers (Table 3.3). In total (including extra routing for power distribution and clock network), AxRam-ShA consumes 1.28mm² (2.1%) per each GDDR5 chip with a 61.6mm² area [86, 87]. AxRam-FxP and AxRam-FP impose 2.0× and 2.4× higher area overhead compared to AxRam-ShA. Recent work [72, 75] has proposed the integration of CGRA-style [130] accelerators atop commodity DRAM, either through TSVs or to the global I/Os. Based on our evaluation, such an integration on each DRAM chip incurs ≈47.8% area overhead. This large area overhead makes such integration an inefficient design point in GPUs. In contrast, our work leverages approximation to integrate many simplified shift-add units inside each GDDR5 chip to enable in-DRAM acceleration.

Quality loss. Figure 3.10 shows the quality loss of AxRam-ShA, AxRam-FxP, and AxRam-FP. The quality loss is compared with that of the original precise application executed on a baseline GPU with no acceleration and an unmodified DRAM. Using fixed-point

arithmetic units in AXRAM-FXP has negligible impact on the quality loss compared to using floating-point arithmetic units in AXRAM-FP. These results are commensurate with other work [46, 6]. Furthermore, the quality loss due to AXRAM-FP and AXRAM-FXP are the same as with NGPU. To achieve an acceptable output quality in AXRAM-SHA, we use up to four iterations of shifts and adds operations. On average, using AXRAM-SHA increases the output quality loss by 2.1% compared to the two other AXRAM microarchitectures.

Sensitivity study of AXRAM with different arithmetic units. We perform a sensitivity study of AXRAM with different arithmetic unit options. Figure 3.11a compares the whole application speedup with AXRAM-SHA, AXRAM-FXP, and AXRAM-FP normalized to the NGPU. Since AXRAM-SHA performs multiple iterations of shifts and adds for each MAC operations its average speedup is less than the other two AXRAM microarchitectures. AXRAM-SHA, with multiple iterations per each multiply-accumulate operation, still provides a 1.1× speedup on average. We see the same speedup across the evaluated applications for AXRAM-FP and AXRAM-FXP, which both take the same number of cycles to compute an in-DRAM neural accelerator invocation. On average, AXRAM-FP and AXRAM-FXP provide 2.0× speedup for the evaluated benchmarks. Figure 3.11b shows the whole application energy reduction of the three AXRAM options normalized to NGPU. On average, AXRAM-SHA achieves 4.8× energy reduction, which is 1.6× and 1.2× more than that of AXRAM-FP and AXRAM-FXP, respectively. AXRAM-SHA achieves a higher energy reduction by simplifying the integrated arithmetic units and trading off the speedup and output quality loss.

Off-chip bandwidth utilization. In Figure 3.12, we compare the off-chip bandwidth of AXRAM-SHA with a baseline GPU with no acceleration and an accelerated GPU (NGPU) [38]. NGPU can accelerate the data processing part of GPU applications, but it increases the off-chip bandwidth utilization by 2.2×. However, AXRAM-SHA significantly can reduce the off-chip bandwidth pressure by performing the neural computation in



(a) Speedup **(b)** Energy Reduction

Figure 3.11: AxRam whole application (a) speedup and (b) energy reduction with the different microarchitectural options compared to a neurally accelerated GPU (NGPU [38]).

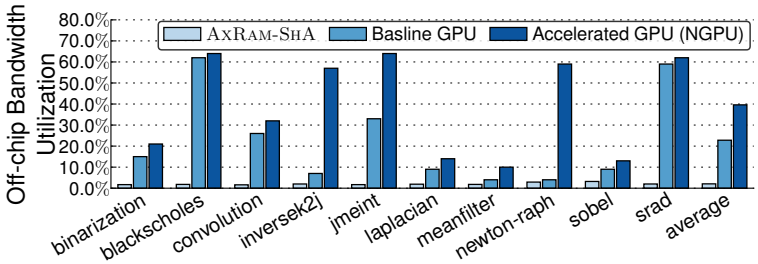


Figure 3.12: Off-chip memory bandwidth consumption for AxRam-ShA, a baseline GPU, and an accelerated GPU (NGPU) [38].

DRAM. This effectively eliminates most of the data transfer of the approximable region between GPU cores and DRAM. Yet, there is still a small amount of communication between the GPU cores and memory for initializing the in-DRAM execution and transferring the control messages. On average, AxRam-ShA can effectively reduce the off-chip bandwidth by a factor of 7.3× (16×) compared to NGPU (baseline GPU).

DRAM power. In this work we aim to offset that the increase in power due to integrating the arithmetic units with the decrease in overall DRAM power due to the reduction in memory I/O activity and increased row-buffer hit rate. To determine if AxRam is able to remain power neutral within DRAM, we analyze DRAM power consumption with three AxRam options in Figure 3.13. The reduction in the data communication and the increase in row-buffer hit rate for all the three AxRam options is the same (see Figure 3.12). How-

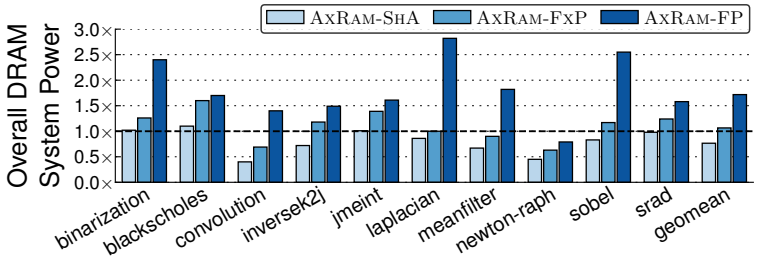


Figure 3.13: AxRam average DRAM system power with the different microarchitectural options normalized to a baseline GPU.

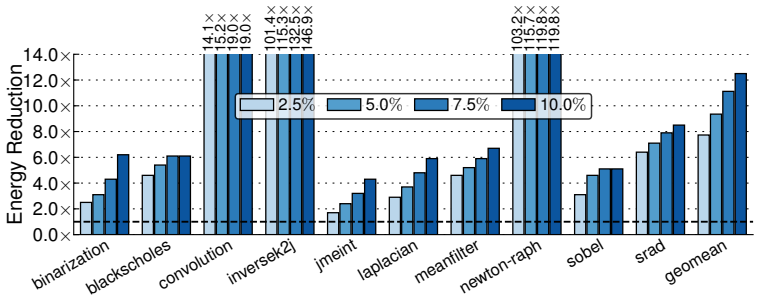


Figure 3.14: The AXRAM-SHA application energy reduction *vs.* different target output quality loss (2.5%, 5%, 7.5%, and 10%), normalized to a baseline GPU with no acceleration.

ever, as we simplify the arithmetic units, the contribution of the in-DRAM accelerators to the overall DRAM power decreases. AXRAM-FP and AXRAM-FXP increase the overall DRAM system power consumption by 70% and 5% on average, respectively. On the other hand, AXRAM-SHA with its simplified shift-add units effectively *decreases* the average overall DRAM power consumption by 26%.

Quality control. Similar to prior work [131, 9, 132, 65, 38, 133], we propose a quality control mechanism that enables the user to trade off the output quality for additional gains. We use the *in-DRAM invocation rate* as our quality knob for controlling the output. The in-DRAM invocation rate (α) indicates the fraction of warps that are offloaded to the memory for in-DRAM neural computation. Given the desired quality requirement for an application, the AXRAM compiler pre-determines the invocation rates. The compiler examines the output quality of the application by executing the application with a user-provided evaluation dataset while varying the invocation rate until the target output quality is met. Then the compiler generates two versions of the kernel. One version contains the unmodified and precise implementation of the kernel without any modifications and the other one contains the neurally transformed version of the kernel. At runtime, the quality control mechanism decides which version of the code to execute based on the pre-determined invocation rate. For the same output quality loss, the invocation rate may be different from one application to another. This behavior is attributed to the characteristic of each application [134].

Figure 3.14 shows the energy reduction of AXRAM when the target output quality loss

varies from 2.5% to 10%, normalized to a baseline GPU with no acceleration. For any given output quality loss, the compiler finds an invocation rate α for each application. A higher output quality translates to a lower invocation rate (*e.g.* fewer warps offloaded to the memory). Thus, the amount of data transfer between the GPU cores and DRAM increases. However, even with a 2.5% output quality loss, the applications experiences 8.0× energy reduction on average over a baseline GPU with no acceleration.

3.10 Conclusion

PIM and approximate computing are two promising approaches for higher performance and energy efficiency. Prior to this work, these techniques were explored disjointly. This work developed AXRAM, a low-overhead accelerated memory architecture that represents the confluence of these two approaches. AXRAM delivers 1.1× speedup and 4.8× higher energy efficiency over even an accelerated GPU with with less than 2.1% added area to each DRAM chip. These results confirm that approximation can play an enabling role for in-DRAM near-data acceleration and pave the way for its further intellectual development and technological adoption.

CHAPTER 4
LANGUAGE SUPPORT FOR ACCELERATION-APPRIXIMATION HARDWARE DESIGN

4.1 Summary

Relaxing the traditional abstraction of "near-perfect" accuracy in hardware design can lead to significant gains in energy efficiency, area, and performance. To exploit this opportunity, there is a need for design abstractions that can systematically incorporate approximation in hardware design. We introduce Axilog, a set of language annotations, that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise. Axilog is coupled with a Relaxability Inference Analysis that automatically infers the relaxable gates and connections from the designer's annotations. The analysis provides formal safety guarantees that approximation will only affect the parts that the designer intended to approximate, referred to as relaxable elements. Finally, this work describes a synthesis flow that approximates only the relaxable elements. Axilog enables applying approximation in the synthesis process while abstracting away the details of approximate synthesis from the designer. We evaluate Axilog, its analysis, and the synthesis flow using a diverse set of benchmark designs. The results show that the intuitive nature of the language extensions coupled with the automated analysis enables safe approximation of designs even with thousands of lines of code. This chapter is based on work presented in DATE 2015 [50] and IEEE Micro 2015 [135]. This work is a result

of collaboration with Divya Mahajan[1], Bradley Thwaites[1], Jongse Park[1], Anandhavel Nagendrakumar[1], Sindhuja Sethuraman[2], Kartik Ramkrishnan[2], Nishanthi Ravindran[2], Rudra Jariwala[2], Abbas Rahimi[3], Hadi Esmaeilzadeh[3], and Kia Bazargan[2].

4.2 Introduction

Emerging applications such as data analytics, machine learning, multimedia, search, and cyber physical systems are inherently approximate and can tolerate imprecision in many parts of their computation. The prevalence of these applications has coincided with diminishing performance and energy returns from traditional CMOS scaling [1, 16]. Several pioneering works have shown significant benefits with approximation at the circuit level [136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 16, 150] Most of these techniques focus on optimization of individual functional units and approximate synthesis algorithms, opening avenues for utilizing approximation at the circuit level. However, there is a lack of abstractions that enable designers to methodically control which parts of the circuit can be synthesized approximately while keeping critical elements, such as the control logic, precise. Thus, there is a need for *approximate hardware description languages* for systematic approximate hardware design.

In this work, we introduce Axilog—a set of concise, intuitive, and high-level annotations—that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog enables designers to reason about and delineate which parts of a hardware system or circuit design are critical and cannot be approximated. A key factor in our language formalism is to abstract away the details of approximation while maintaining the designer's oversight in deciding which circuit elements are synthesized approximately. Axilog is also devised with modular reusability as a first order consideration. In general, hardware systems implementation relies on modular design practices where

[1]Georgia Institute of Technology
[2]University of Minnesota
[3]University of California-San Diego

the engineers build libraries of modules and reuse them to build more complex hardware systems. Axilog provides a specific set of annotations to support reusability. Section 4.3 elaborates on the Axilog annotations for approximate hardware design and reuse.

There are a number of approximate software programming languages including EnerJ [26] and Rely [27]. We do not extend EnerJ or Rely's language constructs to Verilog because they require a large number of manual annotations. Instead, we introduce a new set of annotations and couple them with a Relaxability Inference Analysis that automatically infers which circuit elements are relaxable with respect to the designer's annotations. The Relaxability Inference Analysis formally guarantees that approximation will only affect the circuit elements that the designer intended to approximate. Section 4.4 details this analysis. In Section 4.5, we describe an approximate synthesis flow that leverages a commercial synthesis tool (Synopsys Design Compiler) to apply approximation to the parts of the design that are deemed safe to approximate by the analysis. Section 4.6 evaluates Axilog, its analysis, and the synthesis flow using a set of benchmark designs from domains including arithmetic units, signal processing, robotics, machine learning, and image processing. The evaluations use TSMC 45-nm multi-V_t libraries at the slowest PVT corner and show that by setting the quality loss to 5%, our framework achieves, on average, 45% energy savings and 1.8× area reduction. Allowing a quality loss of 10% results in 54% average energy savings and 1.9× area reduction. Further, we evaluate the robustness of our approach across a wide range of temperature variations (ΔT=125°C). Axilog yields these significant benefits while only requiring between 2 and 12 annotations even with complex designs containing up to 22,407 lines of code. These results confirm the effectiveness of Axilog in incorporating approximation in the hardware design cycle.

4.3 Approximate Hardware Design with Axilog

Our principle objectives for approximate hardware design in Axilog are (1) to carefully craft a small number of Verilog annotations which provide the designer with complete

Table 4.1: Summary of Axilog's language syntax.

Phase	Annotations	Arg	Description
Design	relax	wire, reg, output, inout	Declare an argument as relaxable. Any design element that exclusively affects the argument is safe to approximate.
	relax_local		Similar to relax but the approximation does not cross module boundaries.
	restrict		Any design element that affects the argument is made precise unless explicitly relaxed with another annotation.
	restrict_global		All the design elements affecting the argument are precise.
Reuse	approximate	output, inout	Indicates the output carries relaxed semantics.
	critical	input	Indicates the input is critical and approximate elements cannot drive it.
	bridge	wire, reg	Allows connecting an approximate element to a critical input.

oversight and governance over the approximation; (2) to minimize the number of manual annotations while relying on the Relaxability Inference Analysis to automatically infer the designer's intent for approximation; (3) to relieve the designer from the details of the approximate synthesis process by providing an intuitive separation between approximate design and synthesis and (4) to support the reuse of Axilog modules across different designs without the need for reimplementation. Furthermore, Axilog is a backward-compatible extension of Verilog. That is, an Axilog code with no annotations is a normal Verilog code and the design carries the traditional semantics of strict accuracy. Axilog provides two sets of language extensions, one set for the design and the other for the reuse and interfacing of hardware modules. Table 4.1 summarizes the syntax for the Axilog annotations. The annotations for design dictate which operations and connections are relaxable (safe to approximate) in the module. Henceforth, for brevity, we refer to operations and connections as design elements. The annotations for reuse enable designers to use the annotated approximate modules across various designs without the need for reimplementation. The back-end flow then uses these annotations to determine where in the design to use less costly hardware resources that allow relaxed accuracy (see section 4.4). We provide detailed examples to illustrate how designers are able to appropriately relax or restrict the approximation in hardware modules. Using these examples, we elucidate the interplay

between annotations and language constructs for hardware design, such as instantiation, concurrent assignment, and vector declaration. In the examples, we use **background shading** to highlight the relaxable elements inferred by the analysis.

4.3.1 Design Annotations

Axilog allows each design element to be precise or approximate. The designer's annotations provide the guidelines to identify the design elements that are safe to approximate.

Relaxing accuracy requirements. By default, all design elements (operations and connections) are precise. The designer can use the `relax(arg)` statement to *implicitly* approximate a subset of these elements. The variable `arg` is either a `wire`, `reg`, `output`, or `inout`. Design elements that *exclusively* affect signals designated by the `relax` annotation are safe to approximate. The use of `relax` is illustrated using the following example.

```
module full_adder(a, b, c_in, c_out, s);
    input a, b, c_in; output c_out; approximate output s;
    assign s = a ^ b ^ c_in;
    assign c_out = a & b + b & c_in + a & c_in; relax(s);
endmodule
```

In this `full_adder` module, `s` is the sum of the three inputs, `a`, `b`, and `c_in`. The `relax(s)` statement shows the designer's intent to relax the accuracy requirement of the design elements that exclusively affect `s`, while keeping the unannotated `c_out` (carry out) signal precise. The `relax(s)` statement implies that the analysis can automatically approximate the XOR operations. Adhering to the designer's intent, the unannotated `c_out` signal and the logic generating it will not be approximated. Furthermore, since `s` will carry relaxed semantics, its corresponding output is marked with the `approximate` annotation. In general any output port that carries approximate semantics needs to be marked with the `approximate` annotation. The `approximate` annotation is necessary for reusing modules and will be discussed in Section 4.3.2. With these annotations and the automated analysis, the designer does not need to *individually* declare the inputs (`a, b, c_in`) or any of the XOR (^) operations as approximate. Thus, while designing approximate hardware modules,

this abstraction significantly reduces the burden on the designer to understand and analyze complex data flows within the circuit.

Scope of approximation. Scope of the `relax` annotation crosses the boundaries of instantiated modules. The code on the left side of the following example illustrates this characteristic. The `relax(x)` annotation in the `nand_gate` module implies that the `AND (&)` operation in the `and_gate` module is relaxable. In some cases, the designer might not prefer the approximation to cross the scope of the instantiated modules. For example, the designer might not want the approximation to affect a third-party IP core. Axilog provides the `relax_local` annotation to limit the scope of approximation and its effects on the logic within the same module in which the annotation is declared. The code on the right side shows that the `relax_local` annotation does not affect the semantics of the instantiated `and_gate` module, `a1`. In this case, the `AND(&)` operation in the `and_gate` module is not relaxable. However the NOT(~) operation which shares the scope of the `relax_local` annotation is relaxable. The scope of approximation for both `relax` and `relax_local` is the module in which they are declared. `Relax` penetrates the boundary of the module instantiations but `relax_local` does not. The `relax_local` and `relax` annotations can also be applied selectively to certain bits of a vector.

```
module and_gate(n,a,b);
   input a, b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   relax(x);
endmodule
```

```
module and_gate(n,a,b);
   input a,b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   relax_local(x);
endmodule
```

Restricting approximation. In some cases, the designer might want to *explicitly* restrict approximation in certain parts of the design. Axilog provides the `restrict(arg)` annotation

that ensures that any design element that affects the annotated argument (`arg`) is precise, *unless* a preceding `relax` or `relax_local` annotation has made the driving elements relaxable.

```
module and_gate(n, a, b);
   input a,b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   relax(w0)
   restrict(x);
endmodule
```

```
module and_gate(n, a, b);
   input a,b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   restrict(w0)
   relax(x);
endmodule
```

The above examples show the interplay between the `relax` and `restrict` annotations. On the left side, the designer intends to relax the accuracy of the elements that affect `w0` while keeping the ones that affect `x` precise; hence `relax(w0)` and `restrict(x)`. With these two declarations, the NOT(`~`) operation is not approximated but the AND(`&`) operation will be approximated. Conversely, in the example on the right, the designer relaxes the accuracy of the elements that affect `x` excluding that which affects `w0`. The pair of `restrict(w0)` and `relax(x)` imply that the NOT operation is approximated while the `and_gate` and its AND(`&`) operation remains precise. The `restrict` annotation crosses the boundary of instantiated modules. In both examples, the output `x` carries approximate semantics and needs to be annotated with `approximate`.

Restricting approximation globally. The `restrict` annotation does not have precedence over `relax`. However, there might be cases where the designer intends to override preceding `relax` annotations. For instance, the designer might intend to reuse a third-party approximate IP core in a precise setting. Certain approximate outputs of the IP core might be used to drive critical signals such as the ones that feed to the controller state machine, write enable of registers, address lines of a memory module, or even clock and reset.

These signals are generally critical to the functionality of the circuit and the designers would want to avoid approximating them. To ensure the precision of these signals Axilog provides the `restrict_global` annotation that has precedence over `relax` and `relax_local`. The `restrict_global(arg)` implies that any design element that affects `arg` shall not be subject to any approximation. Note that `restrict_global` penetrates through the boundaries of instantiated modules. The following code snippet illustrates the semantics of the `restrict_global` annotation.

```
module and_gate(n,a,b);
  input a,b;
  approximate output n;
  assign n = a & b;
  relax(n);
endmodule
```

```
module nand_gate(x, a, b);
  input a, b; output x; wire w0;
  and_gate a1(w0, a, b);
  assign x = ~w0;
  restrict_global(x);
endmodule
```

In the code, `restrict_global(x)` precedes the `relax(n)` in the `and_gate` module. The `restrict_global` annotation does not allow any form of relaxation to affect the logic that drives `x` and therefore it is not declared `approximate`. The rest of this section discusses language annotations, similar to the `approximate` annotation, that enable reusability in Axilog.

4.3.2 Reuse Annotations

This section describes the abstractions that are necessary for reusing approximate modules. Our principle idea for these language abstractions is maximizing the reusability of the approximate modules across designs that may have different accuracy requirements. Axilog's reuse annotations concisely modify the module interface. These annotations declares which outputs carry approximate semantics and which inputs cannot be driven by relaxed wires without explicit annotations.

Outputs carrying approximate semantics. As mentioned, the designers can use annotations to selectively approximate the design elements in a module. These design elements

might have a direct or indirect effect on the accuracy of some of the output ports. An approximate module could be given to a different vendor as an IP core. In this case the reusing designer needs to be aware of the accuracy semantics of the input/output ports without delving into the details of the module. To enable the reusing designer to view the port semantics, Axilog requires that all output ports that might be influenced by approximation to be marked as `approximate`. Below, the code snippets illustrate the necessity of the `approximate` annotation. On the left side, output `n` carries relaxed semantics due to the `relax` annotation and is therefore declared as an `approximate` output. Consequently, the `a1` instance in the `nand_gate` module will cause its `x` output to be relaxed. Therefore, the `x` marked as an `approximate` output. On the right side, the `x` output is explicitly relaxed and `x` is marked as an `approximate` output. Relaxing `x` also implies that the AND operation is relaxable in the `a1` instance. However, the `and_gate` module here does not carry approximate semantics by default. Therefore, the output of the `and_gate` is not marked as `approximate` and the approximation is only specific to the `a1` instance.

```
module and_gate(n,a,b);
    input a,b;
    approximate output n;
    assign n = a & b;
    relax(n);
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
endmodule
```

```
module and_gate(n,a,b);
    input a, b;
    output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(x);
endmodule
```

Critical inputs. At design time, the designer of a module may have no knowledge of the circumstances in which the module will be used. The designer may want to prevent approximation to affect certain inputs, which are critical to the functionality of the circuit. To

mark these input ports, Axilog provides `critical` annotation. Wires that carry approximate semantics cannot drive the `critical` inputs without designer's explicit permission at the time of reuse.

```
module multiplexer(select, x0, x1, z);
    critical input select;
    input x0, x1; output z;
    assign z = (s == 1) ? x1 : x0;
endmodule
```

In this example, the `select` input of the multiplexer is declared as `critical` to prevent approximation to affect it.

Bridging approximate modules to critical inputs. As of yet, Axilog does not allow any wire that is affected by approximation to drive a `critical` input. However, we recognize that there may be cases when the reusing designer entrusts critical input with an approximate driver. For such situations, Axilog provides an annotation called `bridge`, which shows designer's explicit intent to drive a critical input by an approximate signal and certifies this connectivity. The example below shows the use of the `bridge` annotation.In this code, the designer annotation relaxes the logic driving `s` that is connected to a critical input `select` of `multiplexer`. This connectivity therefore requires designer's consent. The `bridge(s)` annotation certifies the connectivity of approximated signal `s` to the `select critical` input of the `m1` instance of the `multiplexer` module.

```
module top(x0, x1, z);
    input x0, x1;
    approximate output z; wire s;
    and a1(s, x0, x1);
    relax(s); bridge(s);
    multiplexer m1(s, x0, x1, z);
endmodule
```

In summary, the semantics of the `relax` and `restrict` annotations provides abstractions for designing approximate hardware modules while enabling Axilog to provide *formal guarantees* of safety that the approximation will only be restricted to the design elements

that are specifically selected by the designer. Moreover, the `approximate` output, `critical` input, and `bridge` annotations enable reusability of the modules across different designs. In addition to the modularity, the design and reuse annotations altogether enable *approximation polymorphism* in hardware design. That is, with Axilog, the modules with approximate semantics can be used in a precise manner without reimplementation and conversely precise modules can be instantiated with approximate semantics. These abstractions provide a natural extension to the current practices of hardware design and enable the designer to apply approximation with full control without adding substantial overhead to the conventional hardware design and verification cycle.

4.4 Relaxability Inference Analysis

After the designer provides annotations, the compiler needs to perform a static analysis to find the approximate and precise design elements in accordance with these annotations. This section presents the *Relaxability Inference Analysis*, a static analysis that identifies these relaxable gates and connections. To simplify the implementation, we first translate the RTL Verilog design to primitive gates, while maintaining the module boundaries. We then apply the *Relaxability Inference Analysis* at the gate level. The *Relaxability Inference Analysis* is a backward slicing algorithm that starts from the annotated wires and iteratively traverses the circuit to identify which wires must carry precise semantics. Subtracting the set of precise wires from all the wires in the circuit yields the relaxable set of wires. The gates that immediately drive these relaxable wires are the ones that the synthesis can potentially approximate. Algorithm 1 illustrates the procedure that identifies the precise wires.

This procedure is a *backward-flow* analysis that operates in three phases: (1) **The first phase** starts by identifying a set of *sink* wires. The sink wires are either unannotated outputs or wires that are *explicitly* annotated with `restrict`. The procedure identifies the gates that are driving the sink wires and adds their input wires to the precise set if

Algorithm 1 **Backward flow analysis for finding precise wires.**

Inputs:
$\mathbb{K}$: Circuit-under analysis
$\mathbb{M}$: Set of all the modules within the circuit
$\mathbb{R}$: Set of all the globally restricted wires
Output:
$\mathbb{P}$: Set of precise wires
Initialize $\mathbb{P} \leftarrow \emptyset$
for each $m_i \in \mathbb{M}$ **do**
I: Set of all the inputs ports in m_i
A: Set of all the relaxed wires in m_i
LA: Set of all the locally relaxed wires in m_i
$Sink$: Set of all the restricted wires in $m_i \cup$ Set of unannotated output ports
UW: Set of wires driven by modules that are instantiated within m_i
//Phase1: This loop identifies the m_i module's local precise wires (w_i)
Initialize $N \leftarrow \emptyset$
while ($Sink \neq \emptyset$) **do**
$w_i \leftarrow$ dequeue($Sink$)
if ($w_i \notin I$ & $w_i \notin (A \cup LA)$) **then**
if ($w_i \in UW$) **then**
N.append(w_i)
else
$\mathbb{P}$.append(w_i)
end if
enqueue($Sink$, for all the input wires of the gate that w_i in m_i)
end if
end while
//Phase2: This loop identifies the relaxed wires (w_j) that are driven by the m_j submodules; the m_j submodules are the instantiated modules in m_i
for ($w_j \in UW$) **do**
if ($w_j \notin N$ & w_j drives wire $\in A$) **then**
m_j = module driving the wire w_j
$m_j.A$.append(w_j)
end if
end for
end for
//Phase3: This loop identifies the precise wires (w_k) that are globally restricted
while ($\mathbb{R} \neq \emptyset$) **do**
$w_k \leftarrow$ dequeue($\mathbb{R}$)
$\mathbb{P}$.append(w_k)
$\mathbb{R}$.append(Set of all the input wires of the gate that is driving w_k)
end while

they are not explicitly annotated as relaxed. The algorithm repeats this step for the newly added wires until it reaches an input or an explicitly relaxed wire. However, this phase is only limited to the scope of the module-under-analysis; (2) **In the second phase**, the algorithm identifies the relaxed outputs of the instantiated submodules. Due to the semantic differences between `relax` and `relax_local`, the output of a submodule will be considered relaxed if the following two conditions are satisfied. (a) The output drives another explicitly relaxed wire, which is not inferred due to a `relax_local` annotation; and (b) the output is not driving a wire already identified as precise. The algorithm automatically annotates these qualifying outputs as relaxed. The analysis repeats these two phases for all the instantiated submodules. For correct functionality of this analysis, all the module instantiations are distinct entities in the set $\mathbb{M}$ and are ordered hierarchically; (3) **In the final phase**, the algorithm marks any wire that affects a globally restricted wire as precise. This final phase allows the `restrict_global` to override any other annotations in the design.

Finally, the Relaxability Inference Analysis–part of which is presented in Algorithm 1–identifies the safe-to-approximate subset of the gates and wires with regards to the designer annotations. An approximation-aware synthesis tool can then generate an optimized netlist, with the approximation applied to only the safe-to-approximate circuit elements.

Axilog's language semantics and the Relaxability Inference Analysis are independent of the approximate synthesis. That is, Axilog abstracts away the details of the approximate synthesis and relieves the designer from its specifics. Axilog can be used with virtually any approximate synthesis tool.

4.5 Approximate Synthesis

In our framework, the synthesis tool first takes in the annotated Verilog source code and produces a gate-level netlist without employing any approximate optimizations. However, the synthesis tool preserves the approximate annotations. Then, the Relaxability Inference Analysis identifies the safe-to-approximate subset of the gates and wires with regards to the

designer annotations. In the next step, the synthesis tool applies approximate synthesis and optimization techniques *only* to the safe-to-approximate circuit elements. The tool has the liberty to apply any approximate optimization technique including gate substitution, gate elimination, logic restructuring, voltage over-scaling, and timing speculation as it deems prudent. The objective is to minimize a combination of error, delay, energy, and area considering final quality requirements. Figure 4.1 shows one such approximate synthesis technique. Our synthesis technique uses commercial tools to selectively relax timing requirements on safe-to-approximate paths of the circuit. As shown in Figure 4.1a, we first use Synopsys Design Compiler to synthesize the design with no approximation. We perform a multi-objective optimization targeting the highest frequency while minimizing power and area. We will refer to the resulting netlist as the baseline netlist and its frequency as the baseline frequency. We account for variability by using Synopsys PrimeTimeVX which, given timing constraints, provides the probability of timing violations due to variations. In case of violation, the synthesis process is repeated by adjusting timing constraints until PrimeTimeVX confirms no violations.

Second, as shown in Figure 4.1b, we selectively relax the timing constraints and provide more slack on the safe-to-approximate paths. For the precise paths, the timing constraints are set to the most strict level (the baseline frequency). We then extract the post-synthesis gate delay information in Standard Delay Format (SDF) and perform gate-level timing simulations with a set of input datasets. We use the baseline frequency for the timing simulations even though some of the safe-to-approximate paths are synthesized with more timing slack. The timing simulations yield a set of output values that may incur quality loss since the approximated paths in the circuit may not generate the correct output at the baseline frequency. We then measure the quality loss and if the quality loss is more than designer's requirements, we tighten the timing constraints on the safe-to-approximate paths. We repeat this step until the designer quality requirements are satisfied. This methodology has a potential to reduce energy and area by utilizing slower and smaller gates in the safe

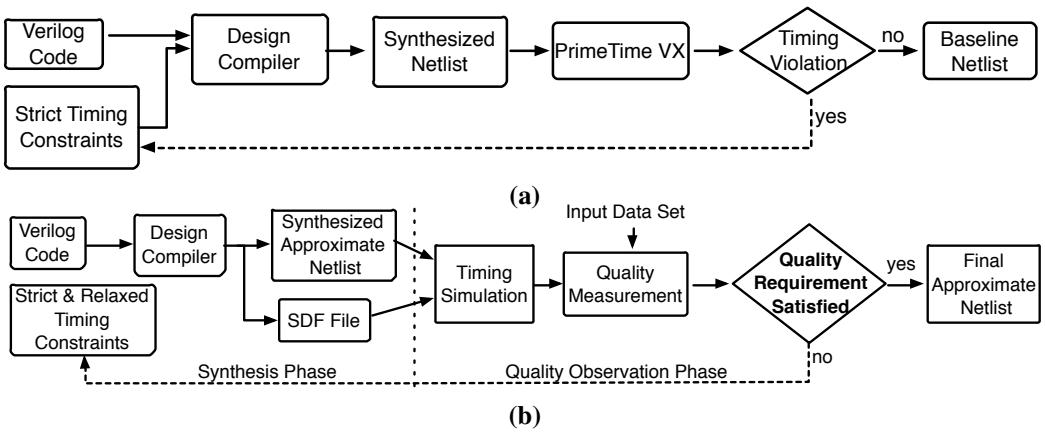


Figure 4.1: Synthesis flow for (a) baseline and (b) approximate circuits.

to approximate paths in which we use relaxed timing constraints.

4.6 Evaluation

To evaluate the effectiveness of Axilog, we annotate several benchmark designs and apply our Relaxability Inference Analysis and synthesis flow.

Benchmarks and Code Annotation. Table 4.2 lists the design benchmarks implemented in Verilog. We use Axilog annotations to judiciously relax some of the circuit elements. The benchmarks span a wide range of domains including arithmetic units, signal processing, robotics, machine learning, and image processing. Table 4.2 also includes the input datasets, application-specific quality metrics, number of lines, and number of Axilog annotations for design and reuse.

Axilog annotations. We annotated the benchmarks with the Axilog extensions. The designs were either downloaded from open-source IP providers or developed without any initial annotations. After development, we analyzed the source Verilog codes to identify relaxable parts. The last two columns of Table 4.2 show the number of design and reuse annotations for each benchmark. The number of annotations range from 2 for Brent-Kung with 352 lines to 12 for InverseK with 22,407 lines. The Axilog annotation coupled with the Relaxability Inference Analysis has enabled us to only use a handful of annotations to effectively approximate designs that are implemented with thousands of lines of Verilog.

Table 4.2: Benchmarks, input datasets, and error metrics.

Benchmark Name	Description	Domain	Input Data Set	Quality Metric	# Lines	Annotations	
						Design	Reuse
Brent-Kung	32-bit adder	Arithmetic Computation	1,000,000 32-bit integers	Average Relative Error	352	1	1
FIR	8-bit FIR filter	Signal Processing	1,000,000 8-bit integers	Average Relative Error	113	6	5
ForwardK	Forward kinematics for 2-joint arm	Robotics	1,000,000 32-bit fixed-point values	Average Relative Error	18,282	5	4
InverseK	Inverse kinematics for 2-joint arm	Robotics	1,000,000 32-bit fixed-point values	Average Relative Error	22,407	8	4
K-means	K-means clustering	Machine Learning	1024x1024-pixel color image	Image Diff	10,985	7	3
Kogge-Stone	32-bit adder	Arithmetic Computation	1,000,000 32-bit integers	Average Relative Error	353	1	1
Wallace Tree	32-bit multiplier	Arithmetic Compuation	1,000,000 32-bit integers	Average Relative Error	13,928	5	3
Neural Network	Feedforward neural network	Machine Learning	1024x1024-pixel color image	Image Diff	21,053	4	3
sobel	Edge detection	Image processing	1024x1024-pixel color image	Image Diff	143	6	3

The relaxable parts are more common in datapath of the benchmarks designs rather than their control logic. For example, K-means involves a significant number of multiplies and additions before the calculated result can be written in a memory module. We used the `relax` annotations to declare these arithmetic operations approximable; however, we used `restrict` to ensure the precision of all the control signals. For smaller benchmarks, such as Brent-Kung, Kogge-Stone and Wallace Tree, only a subset of the least significant output bits were annotated to limit the quality loss. To be able to reuse some of the design, we also annotated the benchmarks with reuse annotations. The number of this type of annotation are listed in the last column of Table 4.2. For example, the add_sub signal that selects the addition and subtraction operation for an ALU is annotated with the `critical` reuse annotation. Overall, one graduate student was able to annotate all the benchmarks within two days without being involved in their design. The intuitive nature of the Axilog extensions makes annotating straightforward.

Application-specific quality metrics. Table 4.2 shows the application-specific error metrics to evaluate the quality loss due to approximation. Using application-specific quality

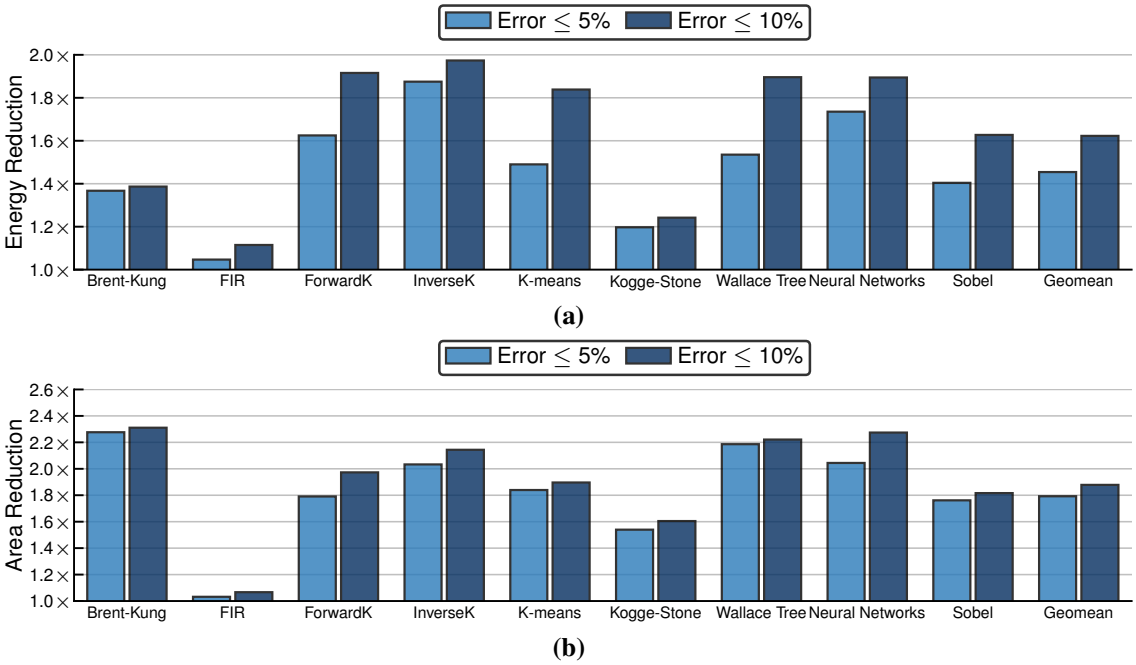


Figure 4.2: Reductions in (a) energy and (b) area when the quality degradation limit is set to 5% and 10% in the synthesis flow.

metrics is commensurate with prior work on approximate computing and language design [26, 27]. In all cases, we compare the output of the original baseline application to the output of the approximated design. For the benchmarks which generate numeric outputs, including brent-kung adder, FIR filter, forward kinematics, inverse kinematics, kogge-stone adder, and wallace tree multiplier, we measure the average relative error. For the neural network, kmeans clustering, and sobel edge detection applications, which produce images, we use the average root-mean-square image difference.

Tools and experimental setup. We use Synopsys Design Compiler (G-2012.06-SP5) and Synopsys PrimeTime (F-2011.06-SP3-2) for synthesis and energy analysis, respectively. We use Cadence NC-Verilog (11.10-s062) for timing simulation with SDF back annotations extracted from various operating corners. We use the TSMC 45-nm multi-V_t standard cells libraries and the primary results are reported for the slowest PVT corner (SS, 0.81V, 0° C).

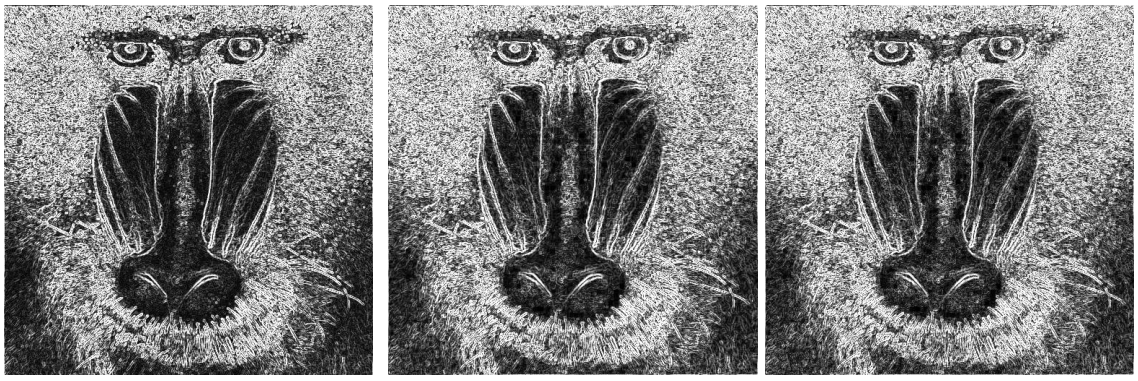
Experimental results. Figure 4.2 illustrates the energy savings (4.2a) and area reduction (4.2b) when the quality loss limit is set to 5% and 10% in our synthesis flow. The baseline is synthesis with no approximation. With the 5% limit, our framework achieves, on average,

Table 4.3: The energy reduction when the quality degradation limit is set to 10% for two different PVT corners. Here, we consider temperature variations.

PVT Corners	Brent-Kung	FIR	ForwarK	InverseK	K-means	Kogge-Stone	Wallace Tree	Neural Network	Sobel	Geomean
(SS, 0.81V, 0°C)	34%	11%	78%	87%	69%	24%	65%	83%	57%	54%
(SS, 0.81V, 125°C)	32%	7%	72%	79%	65%	21%	63%	72%	41%	48%

45% energy and 1.8× area reduction, respectively. When the quality loss limit is set to 10%, the average gains grow to 54% energy reduction and 1.9× area reduction. The Axilog annotations force the control logic in these benchmarks to be precise. Therefore, the benchmarks such as InverseK, Wallace Tree, Neural Network, and Sobel–that have a larger datapath–provide a larger scope for approximation and are usually the ones that see larger benefits. The structure of the circuit also affects the potential benefits. For instance, Brent-Kung and Kogge-Stone adders benefit differently from approximation due to the structural differences in their logic trees. The FIR benchmark shows the smallest energy savings since it is a relatively small design which does not provide many opportunities for approximation. Nevertheless, FIR still achieves 11% energy savings and 7% area reduction with 10% quality loss. This result suggests that even designs with limited opportunities for approximation can benefit significantly from the precisely targeted relaxation that Axilogprovides. We evaluate the effectiveness of our technique in the presence of temperature variations for a full industrial range of 0° C to 125° C. We measured the impact of temperature fluctuations on the energy benefits for the same relaxed designs. Table 4.3 compares the energy benefits at the lower and higher temperatures (the quality loss limit is set to 10%). In this range of temperature variations, the average energy benefits ranges from 54% (at 0° C) to 48% (at 125° C). These results confirm the robustness of our framework that yields significant benefits even when temperature varies.

We visually examine the output of the Sobel application,which generates an image. Figure 4.3 displays the output with 0% (no approximation), 5%, and 10% quality degradation. Interestingly, even 10% quality loss is nearly indiscernible to the eye. Nevertheless, for the 10% error level approximate synthesis provides 57% energy saving and 1.82× area



(a) 0% Quality Loss **(b)** 5% Quality Loss **(c)** 10% Quality Loss

Figure 4.3: Visual depiction of the output quality degradation with approximate synthesis for the Sobel application.

reduction. These results suggest that Axilog can achieve significant savings while preserving the application functionality. This tradeoff is attainable because the high-level language annotations and design abstractions allow the designer to target approximation where it is most effective without compromising the critical parts of the computation. Furthermore, the synthesis tunes the approximate parts of the circuit within the quality constraints specified by the designer. Axilog thereby achieves a balance between quality and efficiency which is advantageous for the specific application.

4.7 Conclusion

Axilog provides a less arduous framework compared to a mere extension of existing approximate programming models for hardware design. Axilog's automated analysis enables the designers to approximate hardware without delving deeper into the intricacies of synthesis and optimization. Furthermore, all the abstractions presented in this thesis are concrete extensions to the mainstream Verilog HDL providing designers with backward compatibility. We evaluated Axilog, its automated Relaxability Inference Analysis, and the presented approximate synthesis and demonstrate 54% average energy savings and 1.9× area reduction with merely 2 to 12 annotations per benchmark. These results confirm that Axilog is a methodical step toward practical approximate hardware design and reuse.

CHAPTER 5

ACCELERATION-APPROXIMATION IN DEEP NEURAL NETWORKS

5.1 Summary

Deep Convolutional Neural Networks (CNNs) perform billions of operations for classifying a single input. To reduce these computations, this work offers a solution that leverages a combination of runtime information and the algorithmic structure of CNNs. Specifically, in numerous modern CNNs, the outputs of compute-heavy convolution operations are fed to activation units that output zero if their input is negative. By exploiting this unique algorithmic property, we propose a predictive early activation technique, dubbed SNAPEA. This technique cuts the computation of convolution operations short if it determines that the output will be negative. SNAPEA can operate in two distinct modes, exact and predictive. In the exact mode, with no loss in classification accuracy, SNAPEA statically re-orders the weights based on their signs and periodically performs a single-bit sign check on the partial sum. Once the partial sum drops below zero, the rest of computations can simply be ignored, since the output value will be zero in any case. In the predictive mode, which trades the classification accuracy for larger savings, SNAPEA speculatively cuts the computation short even earlier than the exact mode. To control the accuracy, we develop a multi-variable optimization algorithm that thresholds the degree of speculation. As such, the proposed algorithm exposes a knob to gracefully navigate the trade-offs between the classification accuracy and computation reduction. This chapter is based on work presented in ISCA 2018 [151]. This work is a result of collaboration with Vahideh Akhlaghi[1], Kambiz

[1] University of California-San Diego

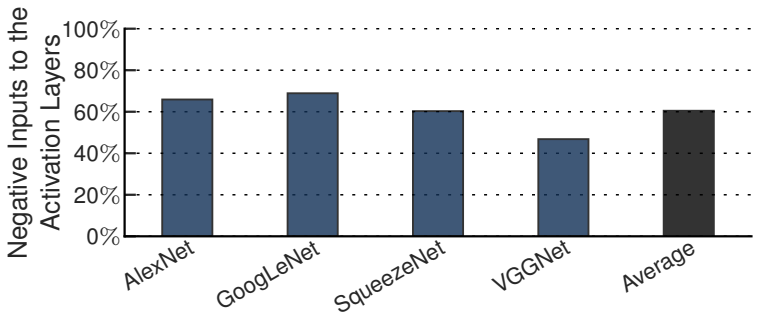


Figure 5.1: Fraction of activation input values that are negative.

Samadi[2], Rajesh K. Gupta[1], and Hadi Esmaeilzadeh[1].

5.2 Introduction

Deep Convolutional Neural Networks (CNNs) are among the most widely used family of machine learning methods that have had a transformative effect on a wide range of applications. CNNs require ample amounts of computation even for a single input query. For instance, assigning a label to a relatively small RGB image ($224\times224\times3$) from the ImageNet database [152] requires billions of multiply-and-accumulate operations [111, 153, 70]. This work aims to reduce these copious amount of computation by exploiting both their runtime information and algorithmic structure. In convolutional layers of many modern CNNs, each convolution operation is commonly followed by an activation function called a Rectifying Linear Unit (ReLU) that returns zero for negative inputs and yields the input itself for the positive ones.

We observe that a large fraction of ReLU outputs are zero, indicating a large number of negative convolution outputs. Figure 5.1 illustrates this trend among several modern CNNs where ReLU nullifies 42%-68% of inputs. In addition, comparing the outputs of intermediate convolutional layers for different input images shows the zero values vary spatially across the images. Figure 5.2 illustrates this insight across two images passing through GoogLeNet [154]. The highlighted differences in the output of the intermediate convolutional layer attest to the varying spatial distribution of zeros. Harnessing these

[2]Qualcomm Technologies, Inc.

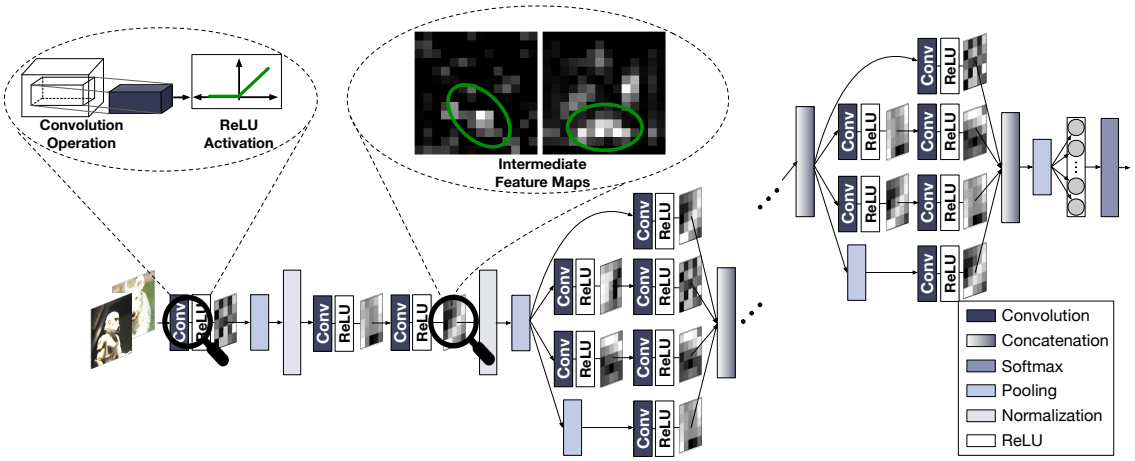


Figure 5.2: GoogLeNet [154], in which the intermediate feature maps for two input images are magnified. The ellipses on the intermediate feature maps highlight the varying spatial distribution of non-zero values for distinct input images.

insights, we devise SNAPEA[3], a holistic software-hardware solution, that cuts a large fraction of the computations short by identifying the zero intermediate values earlier during the runtime.

SNAPEA operates in two distinct modes, namely exact and predictive. In the exact mode, in which the classification accuracy remains unchanged, SNAPEA detects the zero values by static re-ordering of weights along with a low-overhead sign-bit monitoring of partial sums. A negative partial sum triggers early termination of convolution operations. SNAPEA, in the predictive mode, trades off the classification accuracy for larger computation savings by predicting the zero values. Predictive mode results in earlier termination of the convolution operations compared to the exact mode, further reducing the amount of computation. Notwithstanding the higher benefits of predictive mode, an undisciplined prediction of zero values leads to significant loss compared to the nominal CNN classification accuracy. To minimize this loss while maximizing the reduction in computation, we propose a co-designed hardware-software solution that (1) statically pre-arranges the weights, (2) determines a threshold for triggering predictive early activation, and (3) uses a low-overhead runtime monitoring mechanism to apply the early activation. As such, SNAPEA makes the following contributions:

1. **SNAPEA leverages the algorithmic structure of CNNs to to reduce their compu-**

[3]**SnaPEA**: **Sna**ppy **P**redictive **E**arly **A**ctivation

tation. This work provides an insight that the amount of computation in CNNs can be significantly reduced by using a combination of runtime information along with the algorithmic structure of CNNs, which feeds many negative inputs to the activation function.

2. **SNAPEA is a runtime technique that cuts the CNN computations short.** Exploiting the aforementioned insight, this thesis devises an exact runtime approach that relies on a single-bit sign-check to cut the computation short without losing any accuracy. In addition, SNAPEA comes with a predictive mode that speculates on the outcome of sign-check and terminates the computation even earlier, trading off accuracy for less computation.
3. **SNAPEA provides hardware-software solution to control the accuracy trade-offs.** We develop a multi-variable optimization algorithm that systematically thresholds the degree of speculation based on the sensitivity of the CNN output to each layer. The threshold becomes a knob for controlling the accuracy-computation tradeoff.

To evaluate the effectiveness of the proposed technique, we evaluate it on a number of modern CNNs. In the exact mode, which has no effect on the classification accuracy, SNAPEA, on average, delivers 28% (maximum of 74%) speedup and 16% (maximum of 51%) energy reduction over EYERISS [111], a state-of-the-art CNN accelerators. With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers can operate in the predictive mode. The average speedup and energy saving of the layers in the predictive mode over EYERISS are 2.02× and 1.89×, respectively. GoogLeNet sees the maximum benefit of 3.59× speedup and 3.14× energy reduction. Finally, we evaluate the benefits of SNAPEA along with static pruning techniques using the already pruned SqueezeNet CNN [155]. In the exact mode, SqueezeNet achieves 30% speedup and 15% energy reductions with no loss of accuracy, demonstrating the complimentary nature of SNAPEA's dynamic approach to the static pruning techniques. Overall, these benefits suggests that coalescing runtime information with algorithmic insights can lead to new

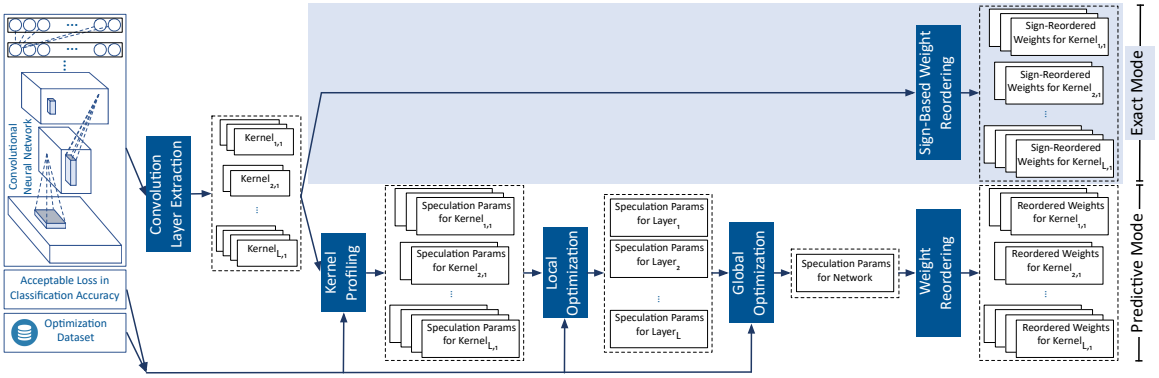


Figure 5.3: Software workflow for SnaPEA.

avenues for reducing the heavy computations of CNNs.

5.3 SnaPEA Hardware-Software Solution

SNAPEA provides a hardware-software solution to reduce the computation in a given CNN. The software part of SNAPEA, illustrated in Figure 5.3, is comprised of two distinct passes: one for the exact mode, and the other for the predictive mode. In the latter pass, the solution finds the thresholds for speculation while considering the acceptable loss in accuracy. In both cases, the task is to reorder weights of the convolution kernels, depending on the operating mode. To utilize these transformations, the SNAPEA comes with an accelerator design that can efficiently execute the CNN with reordered convolution weights with support for early termination of convolution. This section overviews the hardware and software components of SNAPEA.

5.3.1 SnaPEA Software Workflow

Figure 5.3 depicts the software workflow of SNAPEA which takes a CNN model, an acceptable accuracy loss, and an optimization dataset as its inputs. The CNN goes through the multiple passes of this workflow. The first pass, called Convolution Layer Extraction, elicits the convolution kernels of the CNN. Then, the weights of each kernel are re-ordered through the remaining passes, depending on the operating mode, exact or predictive.

Software workflow in the exact mode. To develop this flow, we leverage the observation that in the CNNs with ReLU activation layers, the inputs to the convolution layers are

positive. Consequently, in these layers, the convolution output remains positive by performing Multiply-Accumulate (MAC) operations with the positive subset of the weights. Only performing the remaining MAC operations with the negative subset of the weights can turn the convolution output negative. Given this insight, in the exact mode, Sign-Based Weight Reordering pass reorders the weights of convolution kernels based on their sign such that the positive subset are followed by the negative subset. The reordering enables SNAPEA to first perform MAC with the positive subset and then cut the computation and apply activation function earlier in the case of observing a negative partial output during the computation with negative weights.

Software workflow in the Predictive mode. To reduce the computations further, SNAPEA in the predictive mode, speculates on the sign of the convolution outputs before starting to go through the negative weights. A thresholding mechanisms controls the aggressiveness of the speculation. The intuition is that if the partial output of a convolution after a certain number of MAC operations is less than a threshold, the final convolution output will likely be negative. In this mode, since SNAPEA may misspeculate a positive convolution output as negative, the final classification accuracy may decline. Therefore, to utilize this intuition effectively, the software part of SNAPEA needs to deliberately determine: (1) a threshold value and (2) its associated number of MAC operations, such that the loss in the classification accuracy remains below the acceptable level while the computation reduction is maximized. These two speculation parameters need to be determined for as many layers as possible to maximize the benefits. To determine a proper set of parameters, SNAPEA formulates the problem as a multi-variable constrained optimization problem, and provides a greedy algorithm to solve it (See Section **??** for more details). The algorithm is run by the software part on the Optimization Dataset through the following three passes. This triad of passes is to mange the complexity of accounting for the combined effects of the layers without an exponential explosion of the search space. First, the software *statically* runs a characterization pass, named Kernel Profiling, that measures the sensitivity of the accuracy

to the imprecision introduced in each kernel in isolation. According to this sensitivity, the Kernel Profiling pass determines a set of speculation parameters for each kernel. Then, the next pass (Local Optimization) consolidates the kernel parameters of each layer and identifies a set of speculation parameters for the layer. This pass also considers the effects of speculation in each layer in isolation. Finally, the Global Optimization pass iteratively adjusts the speculation parameters of all layers such that the cross-layer effect yields an acceptable accuracy with the maximal computation reduction. The optimization algorithm runs *once* offline and does not impose additional runtime overhead during the execution of CNNs. Based on the obtained speculation parameters for the entire network, the weights of each kernel are reordered by the Weight Reordering pass. This pass reorders the kernel weights by placing the ones determined by the speculation parameters ahead of the others. Then, the remaining weights are reordered based on the same procedure used for the Sign-Based Weight Reordering pass, which puts the negative weights after the positive ones. Finally, these reordered weights determine the execution of the CNN on the SNAPEA hardware.

5.4 Computation Reduction in SnaPEA

Figure 5.4 demonstrates how SNAPEA reduces the computation by an example of 1×3 convolution. Figure 5.4a performs the unaltered convolution in which all of the MAC operations are performed and yields "-9" as the output. Figure 5.4b illustrates convolution in the exact mode. In this mode, SNAPEA reorders the weights based on their sign, and starts the computation with the positive weights. The computation is terminated after performing only two MAC operations as the results is already negative, "-3". The simple sign check stops the computation. Although the partial sum after two MAC operations ("-3") has not reached the final convolution output ("-9"), it will be converted to zero by the following ReLU operation. As such, the results is the same as the unaltered convolution. Therefore, the exact SNAPEA does not change the final output after ReLU and does not

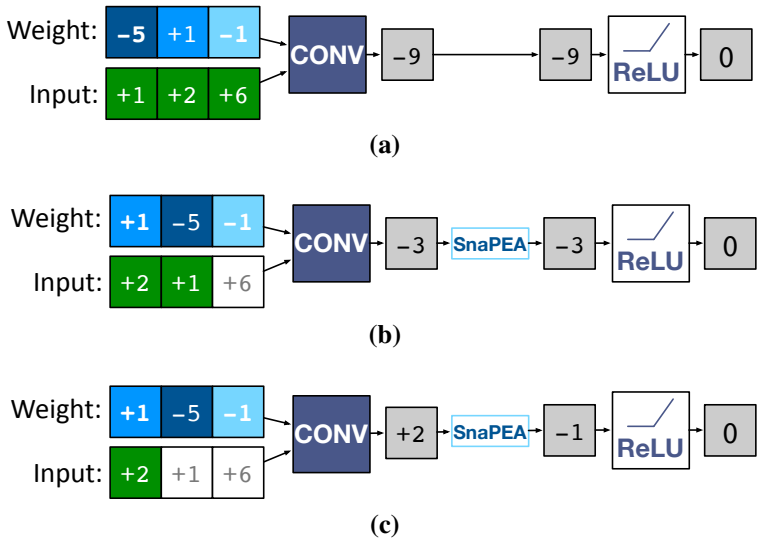


Figure 5.4: Example of a 1×3 convolution in (a) unaltered (b) exact, and (c) predictive modes. In the latter two, the weights and their corresponding inputs are reordered. The white boxes highlight the operations that are cut.

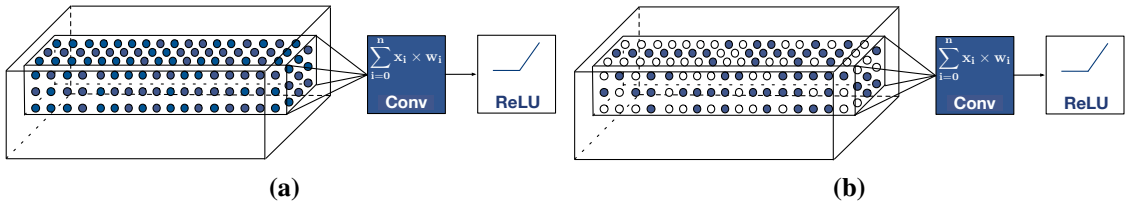


Figure 5.5: (a) The unaltered 3D convolution where all the MAC operations (bubbles) are carried out. (b) The same convolution with SnaPEA, where a significant number of operations are eliminated, delineated by the white bubbles.

lead to accuracy degradation.

Figure 5.4c illustrates how predictive mode cuts the operations earlier than the exact mode. As shown, after performing the MAC operations on only one weight, SnaPEA predicts that the convolution value will eventually be negative. Even though the corresponding partial sum value is positive ("+2"), SnaPEA speculatively triggers the ReLU function early with a negative value (e.g., "-1") and puts out zero. This speculation reduces the computation from two in the exact mode to one. In real-world CNNs, convolution is most often 3D and requires a relatively large number of MAC operations as depicted in Figure 5.5a. Using these methods, SnaPEA can forgo a significant number of the MAC operations as illustrated in 5.5b.

5.4.1 Problem Formulation

The problem of finding the speculation parameters (i.e., $(\mathrm{Th}, \mathrm{N})$) to maximize the computation reduction with an acceptable loss can be formulated as an optimization problem. In order to formulate the problem, we measure the computation reduction by subtracting the number of MAC operations that are performed by SNAPEA from the one performed by an unaltered CNN. However, since the number of MAC operations in the unaltered CNN is constant across various inputs, maximizing the computation reduction becomes equivalent to minimizing the number of MAC operations performed by SNAPEA. Accordingly, we define a function that calculates the number of MAC operations in SNAPEA as follows.

Let $o^d_{l,k}$ be the result of a single convolution window obtained by kernel k in layer l with the speculation parameters Th^k_l and N^k_l for the input image d. The number of MAC operations to compute $o^d_{l,k}$ can be calculated by the function Op shown in (5.1). Let assume that the reordered weights are stored in a 1D array such that the N^k_l speculation weights are placed at the beginning of the array while the remaining positive weights followed by the remaining negative weights are placed at the end.

The function in (5.1) returns N^k_l if the value of partial sum after performing N^k_l operations (i.e., $\mathrm{PartialSum}_{\mathrm{N}^k_l}$) is less than the threshold value Th^k_l. Otherwise, the number of operations is determined by checking the sign of the partial sum value obtained by performing operations with the negative weights (i.e., $\mathrm{PartialSum}_{w^-}$). If a negative partial sum is observed, the function returns the index of the corresponding negative weight in the array (i.e., Idx_{w^-}). If none of the above cases occurs (last part in 5.1), the number of operations is set to the total number of weights in the kernel. Total number of weights of the kernel is $\mathrm{C}_{in,l} \times \mathrm{D}^k_l \times \mathrm{D}^k_l$, in which $\mathrm{C}_{in,l}$ is the number of input channels of the layer l,

and D_l^k is the kernel width.

$$\mathrm{Op}(o_{l,k}^d, \mathrm{Th}_l^k, \mathrm{N}_l^k) = \begin{cases} \mathrm{N}_l^k, & \text{if PartialSum}_{\mathrm{N}_l^k} \leq \mathrm{Th}_l^k, \\ \mathrm{Idx}_{w^-}, & \text{if PartialSum}_{\mathrm{N}_l^k} > \mathrm{Th}_l^k \textit{ and } \text{PartialSum}_{w^-} \leq 0, \\ \mathrm{C}_{in,l} \times \mathrm{D}_l^k \times \mathrm{D}_l^k, & \text{otherwise} \end{cases} \tag{5.1}$$

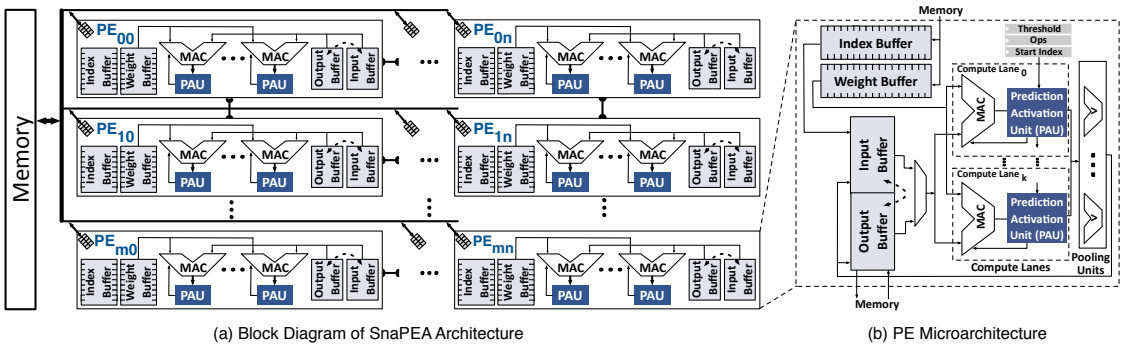
The amount of computation to produce all the convolution outputs is the sum of the number of MAC operations required to produce each individual output. Based on this definition, the problem is translated into finding the speculation parameters that minimize total number of MAC operations and meet the constraint on the accuracy loss, which can be formulated as the following constrained optimization problem.

Let L be a set of all the layers in a given CNN, K_l a set of all the kernels in layer l, $\mathscr{D}$ an optimization dataset, ε an acceptable accuracy loss, Th_l^k and N_l^k the speculation parameters of kernel k of layer l, $\mathrm{O}_{l,k}^d$ the outputs of the convolution generated by kernel k in layer l for the input image d from $\mathscr{D}$, and Accuracy_{CNN} and $\text{Accuracy}_{\text{SNAPEA}}$ the classification accuracy of the CNN and the classification accuracy obtained by SNAPEA, respectively. Now, $(\mathrm{Th}, \mathrm{N})$ can be determined by solving the following problem:

$$\begin{aligned} &\min_{\mathrm{Th,N}} \quad \sum_{d \in \mathscr{D}} \sum_{l \in L} \sum_{k \in K_l} \sum_{o \in O_{l,k}^d} \mathrm{Op}(o, \mathrm{Th}_l^k, \mathrm{N}_l^k) \\ &\text{Subject to} \quad \text{Accuracy}_{CNN} - \text{Accuracy}_{\text{SNAPEA}} \leq \varepsilon \end{aligned} \tag{5.2}$$

5.4.2 Finding the Speculation Parameters

In order to solve the optimization problem formulated as (5.2), we devise a greedy algorithm (i.e., Algorithm 2), which is run by the software part. The algorithm takes a CNN, an optimization dataset $\mathscr{D}$, and an acceptable accuracy loss ε and returns a list named ParamCNN that stores the value of the speculation parameters $(\mathrm{Th,N})$. The algorithm first characterizes the sensitivity of the CNN to the speculation performed in each kernel in isolation. Then, it adjusts the speculation parameters for all the kernels through a greedy



(a) Block Diagram of SnaPEA Architecture (b) PE Microarchitecture

Figure 5.6: (a) The overall structure of the SNAPEA architecture and its multilevel memory hierarchy, containing an off-chip memory and a distributed on-chip buffer for input and outputs. (b) The microarchitecture of each PE. The weights are shared across the compute lanes.

search such that they cooperatively minimize the computation while keeping the loss less than ε. Accordingly, we break the algorithm into two main stages (i.e., the profiling and the optimization stage) as follows:

Profiling stage. Function KernelProfilingPass in Algorithm 2 profiles the number of operations (op) and the accuracy loss (err) corresponding to various values of $(\mathrm{Th}_l^k, \mathrm{N}_l^k)$ for the kernel k in layer l. The process is repeated for all the kernels in the CNN. The acceptable profiling results in terms of the accuracy loss, are accumulated in a list called ParamK. Each sub-list ParamK$[l][k]$ in the list ParamK is sorted in ascending order based on the value of op.

Optimization stage. The optimization stage evaluates the combined effects of kernels and determines the proper speculation parameters for them. To avoid the complexity of evaluating the combined effects, the optimization stage consists of two functions: LocalOptimizationPass and GlobalOptimizationPass. The function LocalOptimizationPass in Algorithm (2), aims to evaluate the combined effects of kernels in each layer when the speculation is performed in the layer in isolation. Then, the function identifies a set of speculation parameters for each individual layer separately that leads to acceptable accuracy with minimum operations. To do this, the function LocalOptimizationPass generates T configurations for layer l such that in the t-th configuration, the speculation parameters of kernel k is set to t-th profiled parameters from the sorted list ParamK$[l][k]$. The configurations yielding an acceptable accuracy are selected as the set of configurations

for the layer l. The acceptable configurations of all layers are populated in a list called ParamL, and passed to the next function.

The second function, GlobalOptimizationPass, evaluates the effect of speculation performed in all the layers simultaneously and adjusts their speculation parameters with respect to the cross-layer effect on the classification accuracy and computation reduction. The output of the function is the final speculation parameters for all the kernels in the CNN which is stored in the list ParamCNN. To find the final parameters, the function first initializes the ParamCNN by setting the speculation parameters of each layer l to ParamL[l][0]. This initialization leads to the maximum computation reduction given the configurations stored in ParamL. However, the accuracy loss obtained by the initial setting may not be acceptable. In case of meeting the desired accuracy, the current parameters in ParamCNN is returned. Otherwise, the parameters are adjusted iteratively until the accuracy loss becomes less than ε. For adjusting the parameters, in the next iteration, those parameters are of interest that lead to small increase in the number of operations while large improvement in the classification accuracy. Hence, we define a merit value as $-\Delta_{err}/\Delta_{op}$, where the larger the Δ_{err} and the smaller the Δ_{op} are, the larger the merit is. Accordingly, the function GlobalOptimizationPass selects the configuration with the maximum merit value among all the configuration in ParamL and updates the corresponding speculation parameters in the list ParamCNN.

5.5 Architecture Design for SnaPEA

SnaPEA provides an accelerator architecture in order to efficiently execute the CNN with the transformed convolution operations. Modern CNNs consist of several back-to-back layers including convolution, ReLU activation, pooling, and fully-connected. To provide an end-to-end solution, the accelerator architecture consists of several units to execute the computation of all layers in the CNN. In order to efficient execution of CNNs, the architecture, specifically, targets to optimize the hardware of the convolution layers because of

the following reasons. The first reason is that the computation of the convolution layers dominates the overall runtime of modern CNNs [110, 111, 156, 157, 153, 158]. The second reason is to execute the convolutions with the reordered weights and to support the predictive early activation at the hardware level. To perform the computations of the fully-connected layers, the same hardware unit designed for the convolution layers is employed. The fully-connected layers are mainly used to perform the actual classification. CNNs usually have much smaller number (i.e. one or two) of fully-connected layers compared to the convolution layers at the final stage of the network. For example, GoogleNet has 57 convolution layers and only *one* fully-connected layer. On average, the computation of fully-connected layers accounts for ≈1% of the total number of computations performed in CNNs [110, 111, 153]. Therefore, using the same hardware unit for the fully-connected layers has virtually no impact on the total runtime of the CNNs. Finally, the SNAPEA architecture consists of dedicated units to support the computations of ReLU activation and pooling layers as well. Figure 5.6 (a) illustrates the high-level block diagram of the proposed accelerator architecture. The accelerator consists of a 2D array of identical Processing Engines (PEs). Each PE is equipped with an input and output buffer that communicates with the off-chip memory. The weights of kernels and the inputs—coming from an off-chip memory—are stored in the dedicated buffers within each PE. In the following, we explain each unit of the accelerator architecture in more details.

Processing Engine (PE). Figure 5.6 (b) depicts the microarchitecture of one PE in the SNAPEA architecture. Each PE comprises multiple compute lanes, a weight and index buffer, an input/output buffer, and multiple Predictive Activation Units. Each compute lane consists of one dedicated Multiply-and-Accumulate (MAC) unit and one Predictive Activation Unit (PAU). The weight, index, and input/output buffers are shared across all the compute lanes within each PE. The computation of a convolution layer in each PE starts upon receiving a block of input features, their corresponding weights, and the weight indices from the off-chip memory. In every cycle, the PE controller reads one weight value

from the weight buffer and broadcasts it to all the compute (MAC units) lanes. The PE controller also reads one weight index from the index buffer and sends the fetched index to the input buffer. Upon receiving the index, the input buffer reads a set of values (one value per each MAC unit) and sends them to the MAC unit for processing. Each compute lane is dedicated to perform all the computations of *one* convolution window. That is, each MAC unit performs the multiplication of one input and weight for each convolution window and sends the results to the accumulation register. The accumulation register accumulates the partial sums for each convolution window. At the same time, the Predictive Activation Unit (PAU) checks the values of the partial sums to determine whether further computations for each convolution window is required. If the PAU determines that no further computations for a convolution window is required, it data gates the corresponding multiplier and accumulator to save energy. This process continues until either all the computations for the current convolution window are performed or the PAU determines to apply the activation early.

Weight and index buffers. The weight buffer contains the weight values of the convolution kernels in the pre-determined order. The weights are ordered offline and loaded into the memory with the proper ordering. Since the ordering of the weights are changed, we also need to add an index buffer to properly index the input buffer. This index is used to load a value from the index buffer. In every cycle, the controller fetches one weight from the weight buffer and broadcasts it to all the compute lanes. Simultaneously, the controller reads an index and sends it to the input buffer to read the corresponding input value. The input buffer delivers the inputs to each compute lane to perform one multiplication for adjacent convolution windows.

Input/Output Buffers. The input buffer holds a portion of input data for each convolution layer. Upon completion of all the computations, the results are written into the output buffer. We use one physical buffer for inputs and outputs. However, the buffer is logically divided into two sub-buffers for holding the input and output data of each layer. The logical

partitioning allows us to use each of the sub-buffers as an input or an output buffer. The results of a layer l stored in the output buffer may be used by the next layer $l+1$ in . In this case, the data of each sub-buffers are logically swapped without wasting additional cycles for data transfers.

Predictive Activation Unit (PAU). Figure 5.7 illustrates the microarchitecture of the Predictive Activation Unit (PAU). One PAU unit is added to each compute lane to support the convolution operations in the exact and predictive mode. Performing the convolution operations in the exact mode only requires to check the sign of the partial sum value during the MAC operations with the negative weights. Accordingly, in the exact mode, the signal `Predict` is set to zero which allows the sign-bit of the partial sum stored in the register `Acc Reg` to determine the termination of the convolution operations. Once the sign-bit becomes one, the signal `terminate` is asserted and notifies the controller to terminate the rest of computations for the underlying convolution window.

In the predictive mode, the sign of the convolution output is speculated through the threshold value (th) and its associated number of operations (n) which are statically determined through the software part (See Algorithm 2). To perform speculation, PAU first checks the partial sum value, coming from the accumulator register, after a pre-determined number of MAC operations against a threshold value. Here, the controller set the signal `Predict` to one. If the partial sum value is less than the pre-determined threshold value, PAU predicts that the final value of this convolution window will eventually become negative. In this case, the PAU performs the following tasks: (1) notifies the controller that no further computations are required for this convolution window and (2) performs the early ReLU activation and sends zero to the output buffer. If the partial sum value is larger than the pre-determined threshold, the compute lane continues the computations for the convolution window normally until it reaches the negative weights. The next check on the partial sum starts upon starting the MAC operations with the negative weights. Here, the signal `Predict` is de-asserted, and PAU periodically performs a simple one-bit sign check on the

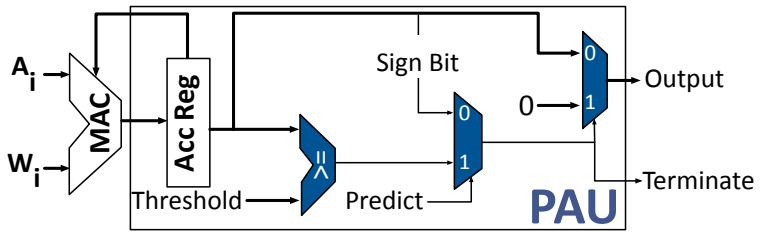


Figure 5.7: Prediction Activation Unit (PAU). The Predict signal determines the PAU operation mode (exact or predictive). The Terminate signal, once asserted, terminates the computation early.

partial sum values after each MAC operations, similar to the process mentioned in the exact mode. Once the sign-bit becomes one, the PAU terminates the convolution operations of the current window and sends a zero value to the output.

The mechanism of dynamically checking the partial sum values might lead to idle computation lanes. These computation lanes remain idle until the rest of the lanes finish the computations of their assigned convolution window. On the other hand, increasing the computation lanes provides higher parallelism between the convolution windows. In Section 5.6, we evaluate the effect of increasing computation lanes on the idle cycles and how it affects the performance and energy savings.

Pooling unit. Once the computations of a group of convolution windows complete, the PE performs the pooling operation on the results. Once done, the PE writes the results back into the output buffer. These results are either used in the computations of the next layers of CNNs or written back to the off-chip memory, if no further computations is required.

Organization of PEs. As shown in Figure 5.12, the SNAPEA architecture contains multiple identical PEs organized in a 2D array. The PEs are logically grouped both *vertically* and *horizontally*. The input data are partitioned between the horizontal PEs and the kernels are partitioned between the vertical PEs. The PEs in the same horizontal and vertical groups work on the same portion of the input data and kernels, respectively. Before the computation starts, a portion of input data are broadcasted to all the PEs within the same horizontal group. Similarly, one or more kernels are broadcasted to the PEs within the same vertical group. After the input and kernel data distribution, the PEs start and proceed their computations independent from other PEs. Once the computations for all the PEs within

Network	Year	Model Size (MB)	# of Layers		Classification Accuracy
			Conv.	FC	
AlexNet	2012	224	5	3	72.6%
GoogLeNet	2015	54	57	1	84.4%
SqueezeNet	2016	6	26	1	74.1%
VGGNet	2014	554	13	3	83.0%

Table 5.1: Workloads, their released year, model size, number of convolution (`Conv.`) and fully-connected (`FC`) layers, and baseline classification accuracy. The model size shows the size of weights in Megabytes.

the same horizontal group end, the on-chip buffer delivers the next portion of input data. In this partitioning, some of the PEs may finish their computations earlier than other PEs within the same horizontal group. These PEs remain idle until all the other PEs complete their computations for all the assigned kernels and input data portion. This synchronization mechanism reduces the cost of multiple data broadcasting among the PEs while having a small impact on the performance. We evaluate the impact of this synchronization mechanism in Section 5.6.2 by analyzing the sensitivity of performance to the number of compute lanes per each PE.

5.6 Evaluation

5.6.1 Methodology

Workloads. To evaluate SNAPEA, we use several popular medium to large scale dense CNN workloads. We also include SqueezeNet [155] that maintains AlexNet-level accuracy with 50× fewer parameters through a static pruning approach. The fewer parameters in SqueezeNet are attained using an iterative pruning and re-training of the convolution weights. Including SqueezeNet in our set of evaluated workloads shows the effectiveness of our approach in further reducing the number of computations of the convolution layers in statically-pruned CNNs. Table 5.1 summarizes the evaluated networks and some of the most pertinent parameters such as model size, number of convolution layers (Conv.), number of fully-connected layers (FC), and the baseline classification accuracy. In all of the evaluations, we use `ILSVRC-2012` [159] validation dataset.

System setup. We use `Caffe v1.0` [160] to run the pre-trained networks on a GPU. We

Table 5.2: SNAPEA and EYERISS [111] design parameters and area breakdown.

		SnaPEA		EYERISS	
		Size	Area (mm^2)	Size	Area (mm^2)
PE	# Compute Lanes / PE	4	0.012	1	0.003
	Partial Sum Register	N/A	0	48 B	0.002
	Input Register	N/A	0	24 B	0.001
	Weight Buffer	0.5 KB	0.014	0.5 KB	0.014
	Index Buffer	0.5 KB	0.007	N/A	0
	Input / Output RAM	20 KB	0.250	N/A	0
	Predictive Activation Units	4	0.008	N/A	0
Accl.	Number of PEs	64	18.62	256	4.94
	Global Buffer	N/A	0	1.25 MB	12.9
Total Area		18.6 mm^2		17.8 mm^2	

compile `Caffe` using `NVCC v8.0.62` and `GCC v4.8.4` with maximum architecture-specific and compiler optimizations enabled. We configure Caffe to use `Nvidia cuDNN v6.0`, a highly tuned GPU-accelerated deep neural network library.

Training/testing datasets. To learn the threshold values and their associated set of operations for each kernel, we implement Algorithm 2 through updating the data of convolutional layers in `Caffe v1.0`. We uniformly sample a subset of images from each of the 1,000 classes in `ImageNet` [159] to obtain the training and testing datasets for the proposed algorithm. The uniform sampling among all the classes enables us to cover images from distinct classes during the training and testing phases of Algorithm 2.

Architecture design and synthesis. We implement the microarchitectural units of the proposed architecture including the controllers, PEs, predictive activation unit (PAU), and registers in Verilog. We use `Synopsys Design Compiler (L-2016.03-SP5)` and a `TSMC 45-nm` standard-cell library to synthesize the proposed architecture and obtain the area, delay, and energy numbers of the logic hardware units.

SNAPEA and baseline architecture configurations. In this thesis, we explore an 8×8 array of PEs in SNAPEA, each with four compute lanes, with a total of 256 MAC units. However, the SNAPEA architecture can be scaled up to larger numbers of PEs. Table 5.2 lists the major architectural parameters of the SNAPEA design. We add a weight buffer and an index buffer, each 0.5 `KB` per each PE. Both weight and index buffers are shared across all the compute lanes within each PE. Each PE is also equipped with a 20 `KB` buffer, that

is evenly divided between input and output. The total capacity of the buffers therefore is 1.25 MB. Similar to the weight and index buffers, both input and output buffers are shared across all the compute lanes within a PE. Sharing the on-chip memories across multiple PEs enables us to reduce the overhead of index buffers. We size the input and output buffer so that the activations of all the CNN models, except VGGNet, fit within these on-chip buffer. This sizing eliminates the need of draining and filling the on-chip buffers during the execution. For VGGNet, which has deeper and larger layers, however, SNAPEA has to spill the activations to memory during the accelerations. We consider the overhead of spilling the data to the off-chip memory in our experiments. For the baseline architecture, we use the EYERISS [111] accelerator. Table 5.2 shows the major architectural components for EYERISS. To have the same peak throughput in both accelerators, we configure EYERISS to have the same number of MAC units (256) as ours. In addition, we allocate the same on-chip memory size (1.25 MB) to both accelerators. The frequency of both accelerators are fixed to 500 MHz. Table 5.2 summarizes the area of the major microarchitectural components in SNAPEA and EYERISS. Overall, the SNAPEA accelerator needs ≈4.5% more area compared to the EYERISS architecture with the specified configurations (Table 5.2). This increase in the area is mainly attributed to the added predictive activation units (PAUs) in the PEs and the controllers.

Energy measurements. Table 6.2 lists the energy consumption of SNAPEA microarchitectural units. For hardware units, we use the synthesis results with TSMC 45-nm and reported numbers in TETRIS [110], which uses the same technology node and has a similar PE architecture as EYERISS. We include the energy overhead of the predictive activation unit in the energy cost of PE (second row in Table 6.2). However, for the baseline architecture (EYERISS), we exclude the energy consumption of the predictive activation unit and use a relative cost of 1.0 in the evaluations. We use the publicly available Micron's DDR4 system power calculator [161] to estimate the energy cost of accesses to the off-chip memory.

Cycle-level microarchitecture simulation. We develop a cycle-level microarchitectural

Operation	Energy (pJ/Bit)	Relative Cost
Register File Access	0.20	1.0
16-bit Fixed Point PE	0.30	1.5
Inter-PE Communication	0.40	2.0
Global Buffer Access	1.20	6.0
DDR4 Memory Access	15.00	75.0

Table 5.3: Absolute and relative energy comparison for different components of SnaPEA architecture along with off-chip memory access energy cost. PE energy includes the cost of Predictive Activation Unit (PAU).

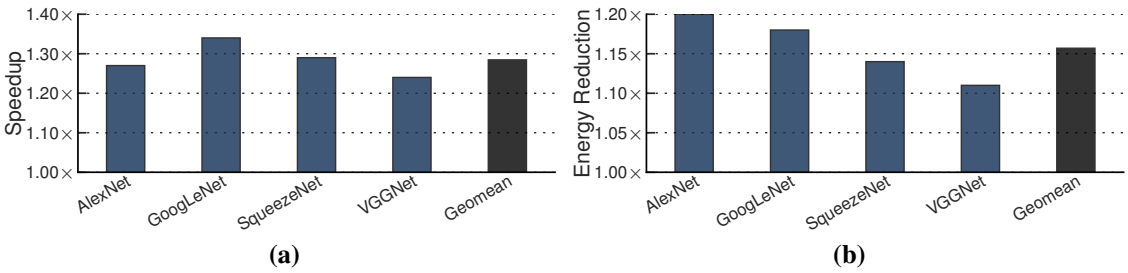


Figure 5.8: Overall (a) speedup and (b) energy reduction with exact mode.

simulator that closely model the architecture of Eyeriss and SnaPEA hardware to measure the performance and energy savings of both hardware. We integrate the microarchitectural components explained in Section 5.5 into the simulator in a cycle-level manner. To measure the energy savings, we use the synthesis results and the reported energy numbers from some of the recent works [110, 111, 162]. Furthermore, we use CACTI-P [163] to calculate the area and power of the register files and on-chip buffers. In the case of any inconsistency in terms of technology node, we properly scaled the area, delay, and energy numbers to make them consistent with our synthesis flow. We integrate the delay and energy numbers collected from the aforementioned sources into our cycle-level simulator. The simulator takes the configuration of a CNN architecture as input and generates an event log for each hardware component. Finally, using the generated event log along the integrated delay and energy numbers, the simulator reports the number of cycles and energy numbers for the whole network.

5.6.2 Experimental Results

Overall benefits in the exact mode. Figure 5.8 illustrates the speedup and energy re-

ductions when the predictive activation is disabled (i.e. exact mode). In this approach, SNAPEA hardware only applies the early activation when the value of partial sum drops below zero. As there is no prediction, the CNN classification accuracy will *not* be deteriorated. That is, the classification accuracy is maintained the same as that in the baseline (Table 5.1). In this setting, SNAPEA, on average, delivers 1.3× speedup and 1.16× energy reductions over EYERISS, respectively. Even for SqueezeNet [164]—a statically pruned convolutional neural network—SNAPEA yields 1.3× and 1.14×. These savings for SqueezeNet show that static pruning techniques are complimentary to the dynamic approach of SNAPEA. Overall, the results in the exact mode show the practicality of SNAPEA in delivering speedup and energy reductions even in the pure exact mode, in which the CNN classification accuracy remains untampered (Table 5.1).

Overall benefits in predictive mode. Figure 5.9a illustrates the performance of SNAPEA over EYERISS in the predictive mode while maintaining the classification accuracy within 3% range of its baseline value (See Table 5.1). In this configuration, the predictive activation units (PAUs) might mis-predict a positive activation value as negative and squashes its value to zero; hence, imposing inaccuracy into the convolutional layers. The injected error in the convolutional layers may lead to a drop in the final classification accuracy. The highest speedup (2.08×) is observed in GoogLeNet, in which a large fraction of the features are negative, and hence the saving is larger.

Figure 5.9b illustrates the energy reduction with SNAPEA in predictive mode over EYERISS [111]. Similar to the simulation settings for speedup, the degradation in classification accuracy is maintained within 3%. Among all the CNN models, GoogLeNet enjoys the highest energy reductions (1.63×). Also, in SqueezeNet [164], a statically pruned CNN model, our technique yields 1.80× and 1.42× speedup and energy reductions, respectively. On average, SNAPEA delivers 1.9× speedup and 1.4× energy reductions across the studied CNN models. This result endorses the effectiveness of SNAPEA, even compared to static pruning techniques [164], in exploiting the runtime information to provide significant

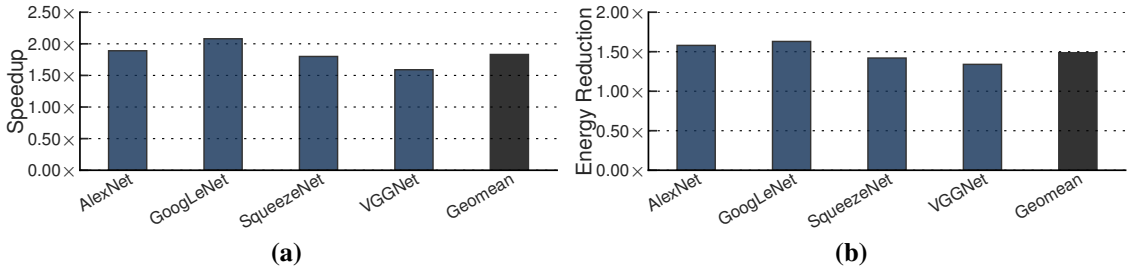


Figure 5.9: Overall (a) speedup and (b) energy reduction with SNAPEA over EYERISS [111] in the predictive mode. The acceptable classification accuracy drop is maintained within ≤3% range of its baseline value.

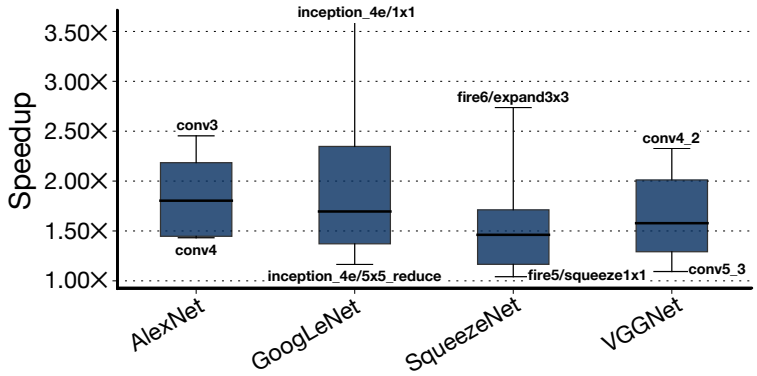


Figure 5.10: Speedup of convolutional layers in each network for the predictive mode when the degradation in classification accuracy is set to ≤ 3%.

savings.

Figure 5.10 illustrates the speedup of convolutional layers in different networks when accuracy drop is set to 3%. The maximum range of speedup is observed in GoogLeNet, in which the maximum speedup is 3.59× achieved by convolution layer `inception_4e/1x1`, and the minimum speedup is 17% achieved by the layer `inception_4e/5x5_reduce`.

Moreover, in the predictive mode, to achieve acceptable accuracy drop, a fraction of the convolutional layers can operate in the predictive mode, which are specified by the software part. Table 5.4 summarizes the percentage of convolutional layers that operate in the predictive mode in each network when the accuracy drop is set to 3%. The average speedup and energy saving across those layers are also brought in the table. The results show that, on average, 67.8% of the convolutional layers operate in the predictive mode, and the average speedup and energy saving across these layers are 2.02× and 1.89×, respectively.

Prediction accuracy. We study how effective the predictive mode is in predicting the negative values. Table 5.5 shows the average true negative and false negative rate across

Table 5.4: The percentage of convolution layers that operates in the predictive mode, when classification accuracy drop is set to $\leq 3\%$. The second and third column illustrates the average speedup and energy reduction across these convolution layers.

Network	% of Convolution Layers	Average Speedup	Average Energy Reduction
AlexNet	60.0%	2.11×	1.97×
GoogLeNet	84.21%	2.17×	2.04×
SqueezeNet	65.38%	1.94×	1.84×
VGGNet	61.50%	1.87×	1.73×

Table 5.5: True negative and false negative rate in predictive mode when classification accuracy drop is set to $\leq 3\%$.

Network	True Negative Rate	False Negative Rate
AlexNet	61.84%	21.39%
GoogLeNet	66.36%	28.37%
SqueezeNet	49.32%	16.69%
VGGNet	47.54%	15.21%

all the convolutional layers in the studied CNN models. The true negative rate measures the proportion of negative values that are correctly identified as negative. Applying early activation on these values does not have any effect on final classification accuracy. The false negative rate measures the proportion of the positive values that are mis-predicted as negative and squashed to zero; hence, *might* lead to degradation in the final classification accuracy. On average, the true (false) negative rate of our proposed prediction mechanism is 56.26% (20.41%). Due to our optimization technique (See Algorithm 2), on average, more than 86% of the error occurs on the small positive values. The small positive values in the activations generally have slight effect on the final classification accuracy. The main reason for this is attributed to the fact that each convolutional layer is commonly accompanied by a max-pooling layer, in which the small values are filtered out. The high true negative rate enables us to apply the activation on the negative values early and significantly reduce the ineffectual operations. Furthermore, the high true negative rate along the modest false negative rate exhibits the capability of SNAPEA in utilizing the runtime information to predict the negative values while meticulously injecting errors mainly on small positive values.

Sensitivity to the degree of speculation. To study the effect of our proposed predictive early activation technique, Figure 5.11 illustrates the speedup with SNAPEA over EYE-

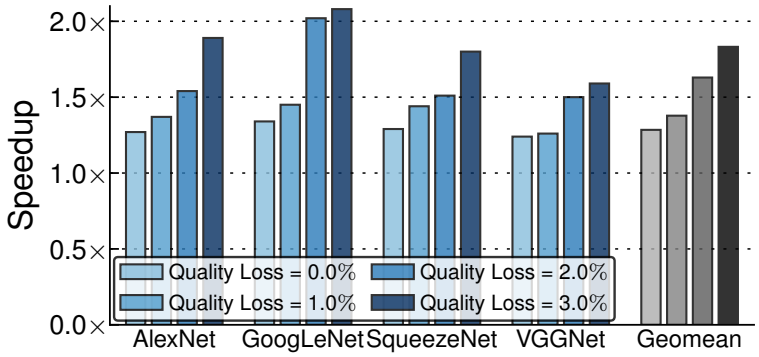


Figure 5.11: Speedup vs. loss in the CNN classification accuracy. Each bar indicates the speedup when the acceptable degradation in the classification accuracy is 0% (pure exact mode), 1% (predictive mode), 2.0% (predictive mode), and 3.0% (predictive mode), respectively.

RISS [111] when the classification accuracy loss varies from 0% to 3%. The 0% classification accuracy loss is when we do *not* use any prediction mechanism (exact mode). The remaining classification accuracy loss levels (e.g., 1.0%, 2.0%, 3.0%) is when we use the predictive early activation mechanism (predictive mode). In fact, supporting distinct levels of loss in the classification accuracy is one of the contributions of our work. The proposed predictive early activations technique exposes a knob for the user to gracefully navigate the trade-offs between CNN classification accuracy and performance and efficiency gains. On average, SNAPEA delivers 1.3×, 1.38×, 1.63×, and 1.9× speedup when we relax the constraint on the acceptable degradation of classification accuracy to 0.0%, 1.0%, 2.0%, and 3.0%, respectively. As we increase the acceptable degradation in the classification accuracy all the evaluated CNNs enjoy a boost in the speedup and energy reductions.

Sensitivity to the number of compute lanes. Figure 5.12 illustrates the impact of varying the number of compute lanes within each PE on speedup with SNAPEA over EYERISS. We present the results for the predictive mode when the maximum loss in the CNN classification accuracy is set to 3%. The second bar (Default) shows the speedup in the baseline SNAPEA system (i.e., four compute lanes) over EYERISS with the same number of compute elements. The rest of the bars (first, third, and fourth bar) show the speedup of SNAPEA when the number of compute lanes per each PE is altered uniformly across all the PEs by a factor of half, two, and four, respectively. Increasing the number of compute lanes potentially increases the parallelization level between different convolutional windows.

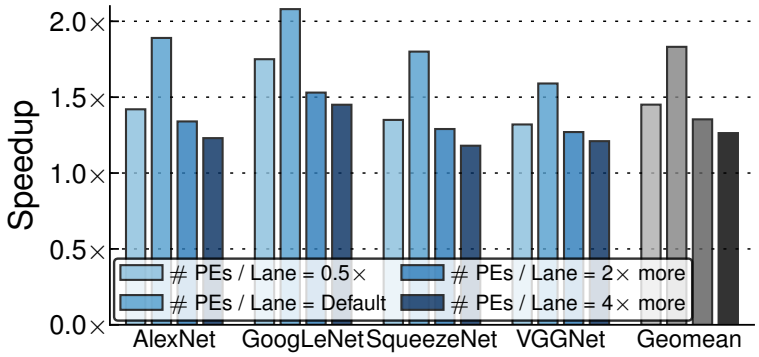


Figure 5.12: Sensitivity of speedup with SNAPEA over EYERISS to the number of compute lanes per each PEs. Each bar indicates the speedup when the number of compute lanes per each PEs is altered by different factors. The acceptable classification accuracy drop is maintained within ≤3% range of its baseline value.

However, due to the synchronization overhead between the compute lanes per each PE (See Section 5.5, Organization of PEs), the improvements diminish. The results show that increasing the number of lanes two times and four times hurts the performance by $\approx 36\%$ and $\approx 45\%$, respectively. Also, if we reduce the number of lanes by $0.5\times$, the performance decreases by $\approx 26\%$. The reason for this behavior is mostly because of an uneven amount of computations performed by each compute lane. In contrast to EYERISS [111], in SNAPEA the number of operations in each convolution window varies due to its runtime early activation. Therefore, increasing the number of arithmetic units reduces the utilization of the compute lanes and diminishes the benefit of higher parallelization.

5.7 Conclusion

Traditionally, layers of deep neural networks have been thought to work in separation while handing each other their results. However, our work took a disparate approach in considering the most common sequence of layers in emerging deep networks to reduce the amount of computation. As such, SNAPEA has devised a predictive early activation that operates in two distinct modes, namely exact and predictive mode. In the exact mode, in which the nominal classification accuracy remains untampered, SNAPEA uses a combination of static re-ordering of the weights and low-overhead sign check to determine when to terminate the computation. SNAPEA further improves the performance and efficiency of convolution operations in the predictive mode by speculatively cutting the computation of convolution

operations if it predicts its output is negative, immediately applying activation. Compared to a recent CNN accelerator, SNAPEA in the exact mode yields 28% speedup (maximum of 74%) and 16% (maximum of 51%) energy reductions across various modern CNNs without affecting their classification accuracy. With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers operate in the predictive mode, and the average speedup and energy saving across these layers are 2.02× and 1.89×, respectively. The significant gains due to the computation and memory access reduction across several modern CNNs show the effectiveness of our approach that conjoins runtime information and algorithmic insights into a unified accelerator.

Algorithm 2 Finding the threshold value and its associated number of operations for all kernels in a CNN

```
Inputs:  CNN: a CNN model, 𝒟: an optimization dataset,
         ε: Acceptable loss in classification accuracy
Outputs: ParamCNN: Speculation parameters (Th,N) for the CNN
// Analyze each kernel individually
function KernelProfilingPass(CNN, 𝒟, ε)
    Initialize ParamK[l][k]→ ∅
    for ∀ layer l in CNN do
        for ∀ kernel k in layer l do
            for a set of values (th,n) do
                op, err = Simulate(CNN, 𝒟, k, th, n)
                if err≤ ε then
                    ParamK[l][k].append((th,n,op))
                end if
            end for
            Sort ParamK[l][k] based on op
        end for
    end for
    return ParamK
end function
// Local Optimizer to find a set of params for each layer individually
function LocalOptimizationPass(CNN, 𝒟, ε, ParamK)
    for layer l in CNN do
        for t in range(0,T) do
            for k in layer l do
                param = ParamK[l][k][t]
            end for
            op, err = Simulate(CNN, 𝒟, ε,param)
            if err ≤ ε then
                ParamL[l].append((param,op,err))
            end if
        end for
    end for
    return ParamL
end function
// Parameter tuning to accommodate for cross-kernel effect
function AdjustParam(CNN,ParamCNN,ParamL)
    for ∀ layer l in CNN do
        for ∀ t in range(len(ParamL[l])) do
            meritL[l][t] = (-(ParamL[l][2]-ParamCNN[l][2])) / (ParamL[l][1] − ParamCNN[l][1])
        end for
    end for
    l,t = Argmax(meritL)
    remove ParamL[l][t] from ParamL[l]
    return (l,t)
end function
// Global Optimizer to find the parameters for the entire network
function GlobalOptimizationPass(CNN,𝒟, ε,ParamL)
    for ∀ layer l in CNN do ParamCNN[l] = ParamL[l][0]
    end for
    err = Simulate(CNN,𝒟,ParamCNN)
    while err> ε do
        l,t=AdjustParam(CNN,ParamCNN,ParamL)
        ParamCNN[l] = ParamL[l][t]
        err = Simulate(CNN, 𝒟, ε,ParamCNN)
    end while
    return ParamCNN
end function
Initialize ParamCNN[l]→ ∅
ParamK = KernelProfilingPass(CNN, 𝒟, ε)
ParamL = LocalOptimizationPass(CNN, 𝒟, ε,ParamK)
ParamCNN =GlobalOptimizationPass(CNN, 𝒟, ε,ParamL)
```

CHAPTER 6

UNSUPERVISED LEARNING ACCELERATION

6.1 Summary

Generative Adversarial Networks (GANs) are one of the most recent deep learning models that generate synthetic data from limited genuine datasets. GANs are on the frontier as further extension of deep learning into many domains (e.g., medicine, robotics, content synthesis) requires massive sets of labeled data that is generally either unavailable or prohibitively costly to collect. Although GANs are gaining prominence in various fields, there are no accelerators for these new models. In fact, GANs leverage a new operator, called transposed convolution, that exposes unique challenges for hardware acceleration. This operator first inserts zeros within the multidimensional input, then convolves a kernel over this expanded array to augment information to the embedded zeros. Even though there is a convolution stage in this operator, the inserted zeros lead to underutilization of the compute resources when a conventional convolution accelerator is employed. We propose the GANAX architecture to alleviate the sources of inefficiencies associated with the acceleration of GANs using conventional convolution accelerators, making the first GAN accelerator design possible. We propose a reorganization of the output computations to allocate compute rows with similar patterns of zeros to adjacent processing engines, which also avoids inconsequential multiply-adds on the zeros. This compulsory adjacency reclaims data reuse across these neighboring processing engines, which had otherwise diminished due to the inserted zeros. The reordering breaks the full SIMD execution model, which

is prominent in convolution accelerators. Therefore, we develop and introduce a unified MIMD-SIMD design for GANAX that leverages repeated patterns in the computation to create distinct microprograms that execute concurrently in SIMD mode. The interleaving of MIMD and SIMD modes is performed at the granularity of single microprogrammed operation. To amortize the cost of MIMD execution, we propose a decoupling of data access from data execute in GANAX. This decoupling leads to a new design that breaks each processing engine to an access micro-engine and an execute micro-engine. The proposed architecture extends the concept of access-execute architectures to the finest granularity of computation for each individual operand. This chapter is based on work presented in ISCA 2018 [165] and FCCM 2018 [166]. This work is a result of collaboration with Michael Brzozowski[1], Behnam Khaleghi[2], Philip J. Wolfe[1], Soroush Ghodrati[1], Hajar Falahati[3], Kambiz Samadi[4], Nam Sung Kim[5], and Hadi Esmaeilzadeh[2].

6.2 Introduction

Deep Neural Networks (DNNs) have been widely used to deliver unprecedented levels of accuracy and performance in various applications. However, they rely on the availability of copious amount of labeled training data, which can be costly to obtain as it requires human effort to label. To address this challenge, a new class of deep networks, called Generative Adversarial Networks (GANs), have been developed with the intention of automatically generating larger and richer datasets from a small initial labeled training dataset. GANs combine a *generative model*, which attempts to create synthetic data similar to the original training dataset, with a *discriminative model*, a conventional DNN that attempts to discern if the data produced by the generative model is synthetic, or belongs to the original training dataset [167]. The generative and discriminative models compete with each other in a

[1]Georgia Institute of Technology
[2]University of California-San Diego
[3]Institute for Research in Fundamental Sciences
[4]Qualcomm Technologies, Inc.
[5]University of Illinois at Urbana-Champaign

minimax situation, resulting in a stronger generator and discriminator. As such, GANs can create new impressive datasets that are hardly discernible from the original training datasets. With this power, GANs have gained popularity in numerous domains, such as medicine, where overtly costly human-centric studies need to be conducted to collect relatively small labeled datasets [168, 169]. Furthermore, the ability to expand the training datasets has gained considerable popularity in robotics [170], autonomous driving [171], and media synthesis [172, 173, 174, 175, 176, 177, 178] as well.

Currently, advances in acceleration for conventional DNNs are breaking the barriers to adoption [179, 180, 181, 111, 153, 182]. However, while GANs are set to push the frontiers in deep learning, there is a lack of hardware accelerators that address their computational needs. This work sets out to explore this state-of-the-art dimension in deep learning from the hardware acceleration perspective. Given the abundance of the accelerators for conventional DNNs [183, 184, 156, 110, 185, 186, 187, 188, 181, 189, 190, 111, 191, 153, 157, 192, 193, 194, 195, 196, 46, 197, 38, 198, 6, 182, 199, 12, 200], designing an accelerator for GANs will only be attractive if they pose new challenges in architecture design. By studying the structure of emerging GAN models [172, 173, 174, 175, 176, 177, 178], we observe that they use a fundamentally different type of mathematical operator in their generative model, called *transpose convolution*, that operates on multidimensional input feature maps.

The transposed convolution operator aims to extrapolate information from input feature maps, in contrast to the conventional convolution operator which aims to interpolate the most relevant information from input feature maps. As such, the transposed convolution operator first inserts zeros within multidimensional input feature maps and then convolves a kernel over this expanded input to augment information to the inserted zeros. The transposed convolution in GANs fundamentally differs from the operators in the backward pass of training conventional DNNs, as these do not insert zeros. Moreover, although there is a convolution stage in the transposed convolution operator, the inserted zeros lead to

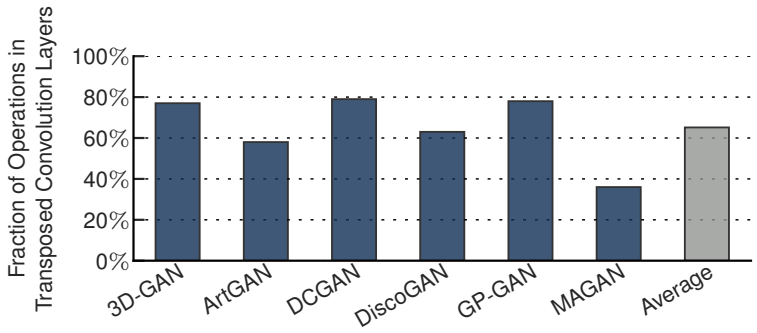


Figure 6.1: The fraction of multiply-add operations in transposed convolution layers that are inconsequential due to the inserted zeros in the inputs.

underutilization of the compute resources if a conventional convolution accelerator were to be used. The following highlights the sources of underutilization and outlines the contributions of this work, making the first GAN accelerator design possible.

1. **Performing multiply-add on the inserted zeros is inconsequential.** Unlike conventional convolution, the accelerator should skip over the zeros as they constitute more than 60% of all the multiply-add operations as Figure 6.1 illustrates. Skipping the zeros creates an irregular dataflow and diminishes data reuse if not handled adequately in the microarchitecture. To address this challenge, we propose a reorganization of the output computations that allocates computing rows with similar patterns of zeros to adjacent processing engines. This forced adjacency reclaims data reuse across these neighboring compute units.

2. **Reorganizing the output computations is imperative but breaks the SIMD execution model.** The inserted zeroes, even with the output computation reorganization, create distinct patterns of computation when sliding the convolution window. As such, the same sequence of operations cannot be repeated across all the processing engines, breaking the full SIMD execution model. Therefore, we propose a unified MIMD-SIMD accelerator architecture that exploits repeated patterns in the computations to create different microprograms that can execute concurrently in SIMD mode. To maximize the benefits from both levels of parallelism, we propose an architecture, called GANAX, that supports interleaving MIMD and SIMD operations at the finest granularity of a single microprogrammed operation.

3. **MIMD is inevitable but its overhead needs to be amortized.** Changes in the dataflow and the computation order necessitate irregular accesses to multiple different memory structures while the operations are still the same. That is, the data processing part can be SIMD but the irregular data access patterns prevent using this execution model. For GANAX, we propose the decoupling of data accesses from data processing. This decoupling leads to breaking each processing engine into an access micro-engine and an execute micro-engine. The proposed architecture extends the concept of access-execute architecture [201, 202, 203, 204] to the finest granularity of computation for each individual operation.

Although GANAX addresses these challenges to enable efficient execution of the transposed convolution operator, it does not impose extra overhead, but instead offers the same level of performance and efficiency. To establish the effectiveness of our architectural innovation, we evaluate GANAX using six recent GAN models, on distinct applications. On average, GANAX delivers 3.6× speedup and 3.1× energy savings over a conventional convolution accelerator. These results indicate GANAX is an effective step towards designing accelerators for the next generation of deep networks.

Generative Adversarial Networks (GANs) have revolutionized modern machine learning by significantly improving generative models while using only limited number of labeled training data. Figure 6.2 shows an overall visualization of a GAN, consisting of two deep neural network models, a generative model and a discriminative model. These two neural network models oppose each other in a minimax situation. Specifically, the generative model tries to generate data that will trick the discriminative model to believing the data is from the original training dataset. Meanwhile, the discriminative model is handed data from either the generative model or the training data and tries to discern between the two. After these networks compete with each other, they refine their abilities to generate and discriminate, respectively. This process creates a stronger generative model and discriminative model than could be obtained otherwise [167]. This arrangement of

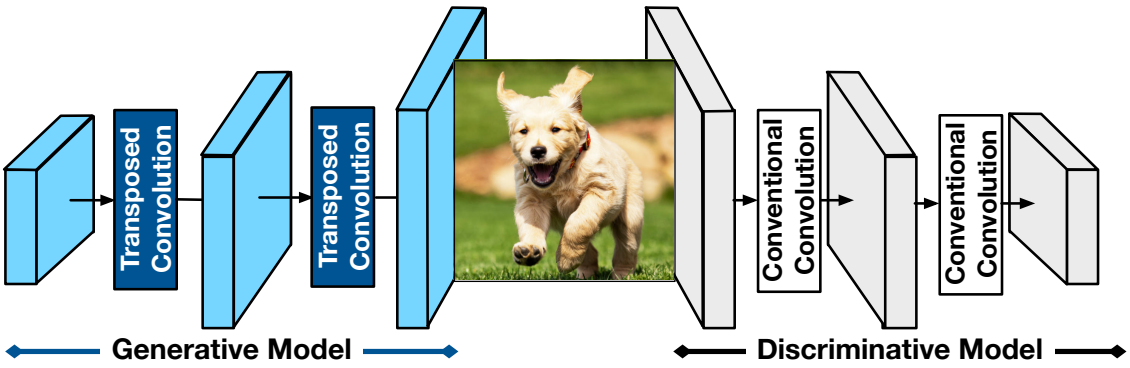
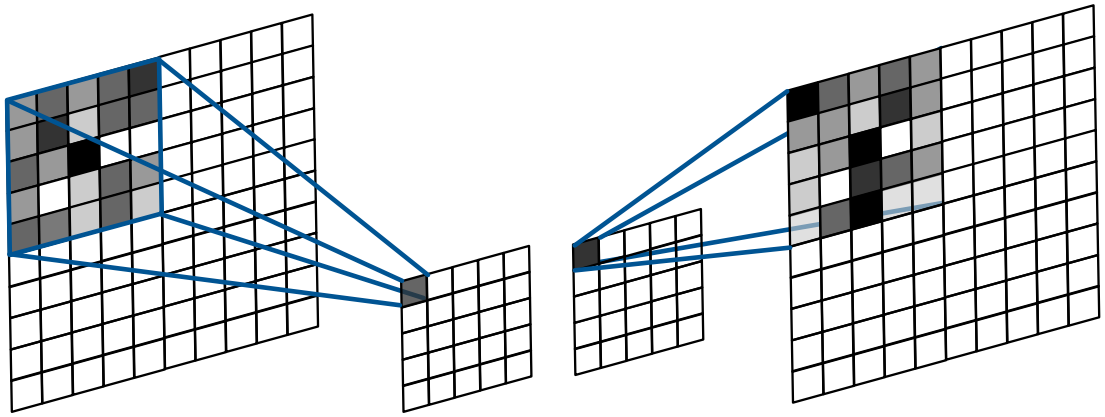


Figure 6.2: High-level visualization of a Generative Adversarial Network (GAN).



(a) Conventional Convolution (Data Reduction) **(b)** Transposed Convolution (Data Expansion)

Figure 6.3: (a) Convolution operations decreases the size of data (data reduction). (b) Transposed convolution increases the size of data (data expansion).

neural networks has opened up many applications, some of which include music generation with accompaniment [175] and the discovery new drugs to cure diseases [205]. GANs are enabling our future by pushing forward development in autonomous vehicles, allowing us to imitate human drivers [206] and simulate driving scenarios to save testing and training costs [171]. GANs enable imagination [207], a major advancement for machine learning and a key step towards true general artificial intelligence. Here, we overview the challenges and opportunities that were encountered while designing hardware accelerators for GANs.

6.3 Flow of Data in Generative Models

Challenges and opportunities for GAN acceleration. The generative models in GANs

are fundamentally different from the discriminative models. As Figure 6.2 illustrates, while the discriminative model mostly consists of convolution operations, the generative model uses transposed convolution operations. Accelerating convolution operations has been the focus of a handful of studies [183, 184, 156, 110, 185, 186, 187, 188, 181, 189, 190, 111, 191, 153, 157, 192, 193, 194, 195, 196, 46]; however, accelerating transposed convolution operations has remained unexplored. Figure 6.3 depicts the fundamental difference between the conventional convolution and transposed convolution operations. The convolution operation performs *data reduction* and generally transforms the input data to a smaller representation. On the other hand, the transposed convolution implements a *data expansion* and transforms the input data to a larger representation. The transposed convolution operation expands the data by first transforming the input data through inserting zeros between the input rows and columns and then performing the computations by sliding a convolution window over the transformed input data. Due to this fundamental difference between convolution and transposed convolution operations, using the same conventional convolution dataflow for generative model may lead to inefficiency. The main reason for such inefficiency can be attributed to the variable number of operations per each convolution window in the transposed convolution. The variable number of operations per each convolution window is the main result of zero insertion step in transposed convolution. Because of this zero-insertion step, distinct convolution windows may have a different number of consequential multiplications between inputs and weights.[6] This discrepancy in the number of operations is the root cause for inefficiency in the computations of generative models, if the same convolution dataflow is used. As such, we aim to design an efficient flow of data for GANs by focusing on: (1) managing the discrepancy in the number of operations per each convolution window in order to mitigate the inefficiencies in the execution of generative models, (2) leveraging the similarities between convolution and transposed convolution operations in order to accelerate both discriminative and generative models

[6] A consequential multiplication is a multiplication in which none of the source operands are zero and contributes to the final value of the convolution operation.

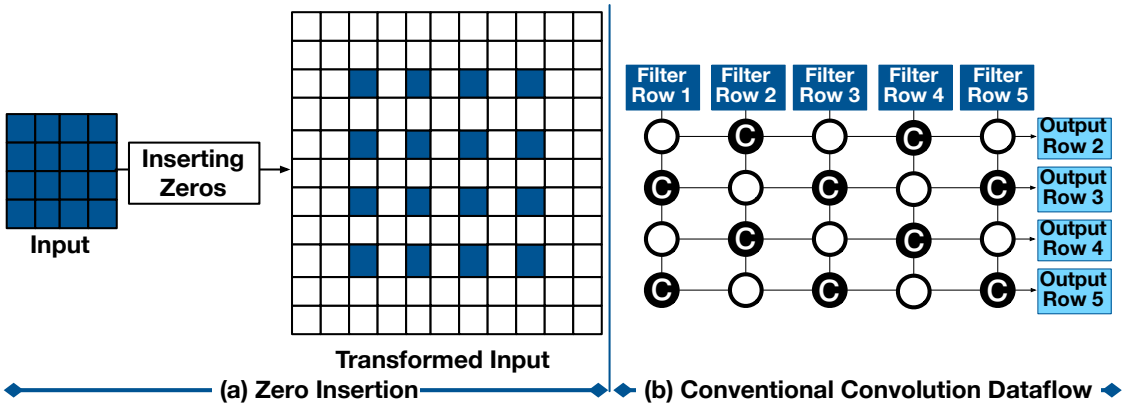


Figure 6.4: (a) Zero-insertion step in a transposed convolution operation for a 4×4 input and the transformed input. The light-colored squares display zero values in the transformed input. (b) Using conventional dataflow for performing a transposed convolution operation.

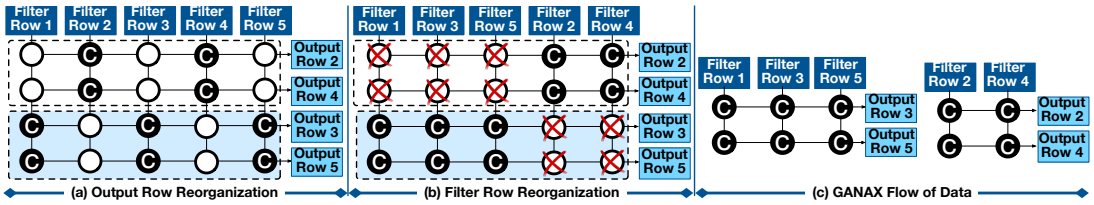


Figure 6.5: The GANAX flow of data after applying (a) output row reorganization and (b) filter row reorganization. (c) The GANAX flow of data after applying both output and filter row reorganization and eliminating the idle compute nodes. The combination of these flow optimizations reduces the idle (white) compute nodes and improves the resource utilization.

on the same hardware platform, and (3) improving the data reuse in discriminative and generative models.

Why using a conventional convolution dataflow is not efficient for transposed convolution? Going through a simple example of a 2-D transposed convolution, we illustrate the main sources of inefficiencies in performing transposed convolution, if a conventional convolution dataflow is used. Figure 6.4(a) illustrates an example of performing a transposed convolution operation using a conventional convolution dataflow. In this transposed convolution operation, a 5×5 filter with stride of one and padding of two is applied on a 4×4 2D input. In the initial step, the transposed convolution operation inserts one row and one column of zeros between successive rows and columns (white squares). Performing this zero-insertion step, the input is expanded from a 4×4 matrix to a 11×11 one. The number of zeros to be inserted for each transposed convolution layer in the generative models may vary from one layer to another and is a parameter of the network. After performing the

zero-insertion, the next step is to slide a convolution window over the transformed input and perform the multiply-add operations. Figure 6.4(b) illustrates performing this convolution operation using a conventional convolution dataflow [184, 110, 111]. To avoid clutter in Figure 6.4(b), we only show the dataflow for generating the output rows 2-5.

Each circle in Figure 6.4(b) represents a compute node that can perform vector-vector multiplications between a row of the filter and a row of the zero-inserted input. The filter rows are spatially reused across each of the computation nodes in a vertical manner. Once a vector-vector multiplications finish, the partial sums are aggregated horizontally to yield the results of performing transposed convolution operation for each output row. The black circles represent the compute nodes that are performing consequential operations, whereas the white circles which represent the compute nodes performing inconsequential operations. As depicted in Figure 6.5(b), there will be inconsequential operation (white circles) if a conventional convolution dataflow is used for the execution of transposed convolution operations. Because of the inserted zeros, some of the filter rows are not used to compute the value of an output row. For example, since the 1^{st}, 3^{rd}, and 5^{th} rows of the input are zero, the 2^{nd} output row only needs to perform the operations for non-zero elements; hence using only the 2^{nd} and 4^{th} filter rows, leaving three compute nodes idle. Overall, in this example, 50% of the compute nodes remain idle during the execution of this transposed convolution operation. Analyzing this transposed convolution operation reveals three main sources of inefficiency when a conventional convolution dataflow is used.

(1) **Coarse-grain resource underutilization:** Since the consequential filter rows vary from one output row to another, a significant number of compute nodes remain idle. In the aforementioned example, this underutilization applies to 50% of the compute nodes, which perform vector-vector multiplications.

(2) **Fine-grain resource underutilization:** Even within a compute node a large fraction of the multiply-add operations are inconsequential due to the columnar zero insertion.

(3) **Reuse reduction:** While the compute units pass along the filter rows for data reuse,

the inserted zeros render this data transfer futile.

We address the first two sources of inefficiencies with a series of optimizations on the flow of data in GANs. Then, to address the last source of inefficiency that arises because of the inconsequential multiply-add operations within each compute node, we introduce an architectural solution (Section 6.4).

Flow of data for generative models in GANAX. Figure 6.5 illustrates the proposed flow of data optimizations for generative models in GANAX. To mitigate the challenges of using conventional convolution dataflow for transposed convolution operations in generative models, we leverage the insight that even though the patterns of computation may vary from one output row to another, they are still structured. Taking a closer look at Figure 6.4, we learn that there are only two distinct patterns[7] in the output row computations. In this example, the even output rows (i.e., 2nd and 4th) use one pattern of computation, whereas the odd output rows (i.e., 3rd and 5th) use a different pattern for their computations. Building upon this observation, we introduce a series of flow of data optimizations to mitigate the aforementioned inefficiencies in the computation of transposed convolution operation, if a conventional convolution dataflow used.

The first optimization maximizes the data reuse by reorganizing the computation of the output rows in a way that the rows with the same pattern in their computations become adjacent. Figure 6.5(a) illustrates the flow of data after applying this output row reorganization. Applying the output row reorganization in this example, make the even-indexed (2nd and 4th output rows) output rows adjacent. Similar adjacency is established for odd-indexed (3rd and 5th output rows) output rows. Although this optimization addresses the data reuse problem, it does not deal with the resource underutilization (i.e., idle compute nodes (white circles) still exist). To mitigate this resource underutilization, we introduce the second optimization that reorganizes the filter rows. As shown in Figure 6.5(b), applying the filter row reorganization establishes an adjacency for the 1st, 3rd, and 5th filter rows.

[7]The location of white and black circles (compute nodes) defines each pattern.

Similarly, the 2nd and 4th filter rows become adjacent. After applying output and filter row reorganization, as shown in Figure 6.5(b), the idle compute nodes can be simply eliminated from the dataflow. Figure 6.5(c) illustrates the GANAX flow of data after performing both optimizations, which improves the resource utilization for transposed convolution operation from 50% to 100%.

The proposed GANAX flow of data also addresses the inefficiency in performing the horizontal accumulation of partial sums. As shown in Figure 6.4(b), the conventional convolution dataflow requires five cycles to perform the horizontal accumulation for each output row, regardless of their locations. However, comparing Figure 6.4(b) and Figure 6.5(c), we observe that after applying output and filter row reorganization optimizations, the number of required cycles for performing the horizontal accumulation reduces from five to two for even-indexed output rows and from five to three for odd-indexed output rows. While the proposed flow of data optimizations effectively improve the resource utilization for transposed convolution, there arises an interesting architectural challenge: *how to fully utilize the parallelism between the computations of the output rows that require different number of cycles for horizontal accumulation (two cycles for even-indexed and three cycles for odd-indexed output rows)?* If a SIMD execution model is used, some of the compute nodes have to remain idle until the accumulations for the output rows that require more cycles for horizontal accumulation, finish. The next section elaborates on the GANAX architecture that exploits the introduced flow of data for transposed convolution and fully utilize the parallelism between distinct output rows by conjoining the MIMD and SIMD execution models.

6.4 Architecture Design for GANAX

The execution flow of the generative model (i.e., zero-insertion and variable number of operations per each convolution window) in GANs poses unique architectural challenges that the traditional convolution accelerators [110, 184, 111, 153, 182] can not adequately

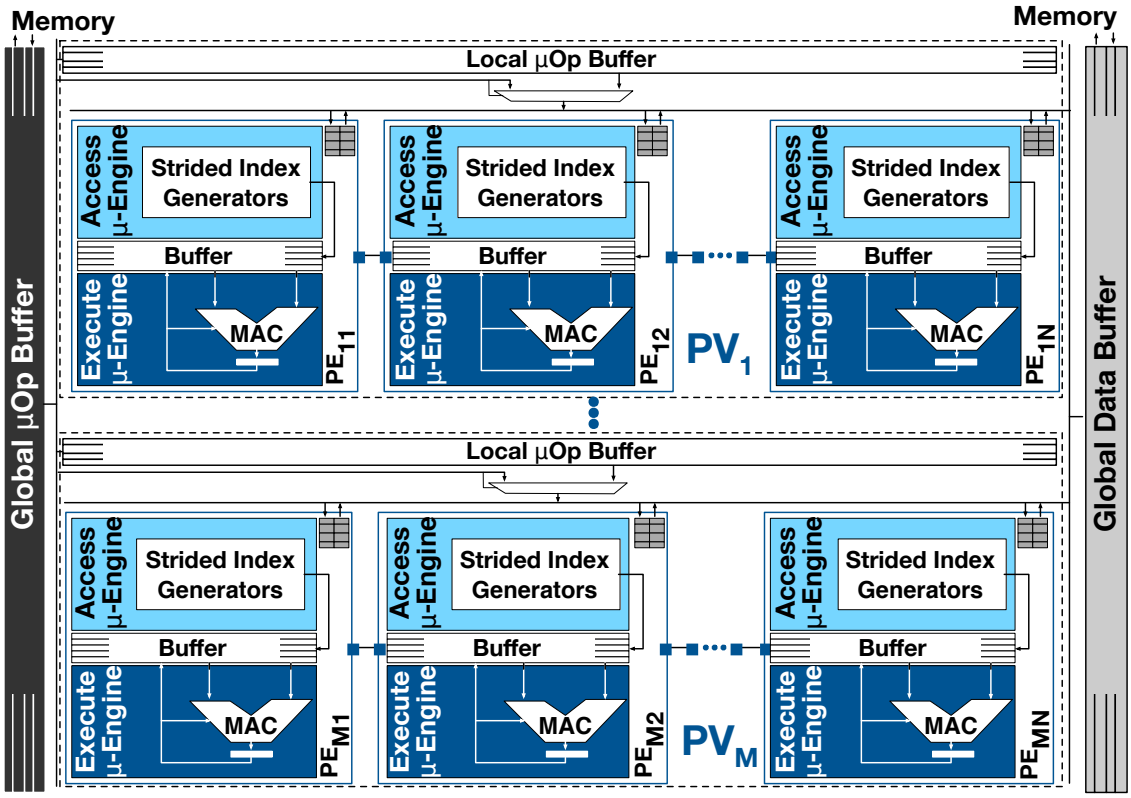


Figure 6.6: Top-level block diagram of GANAX architecture.

address. There are two fundamental architectural challenges for GAN acceleration as follows:

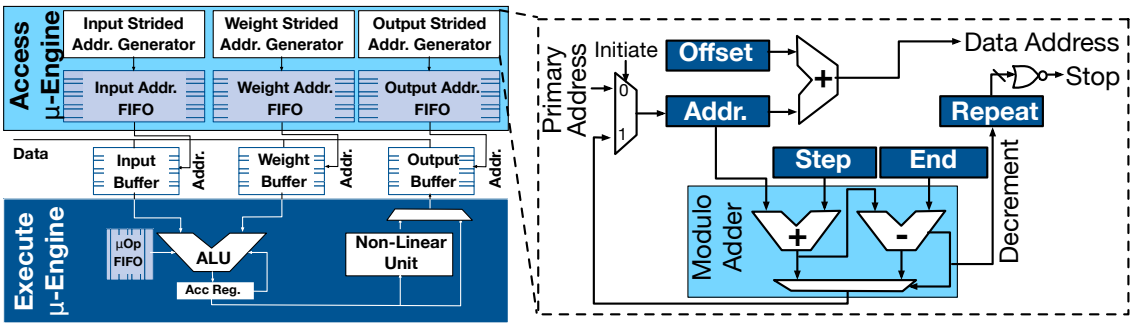
Resource underutilization. The first challenge arises due to the variable number of operations per each convolution window in transposed convolution operation. In most of recent accelerators [184, 110, 153, 182], which mainly target conventional convolution operation, the processing engines generally work in a SIMD manner. The convolution windows in conventional convolution operation follow a regular pattern and the number of operations per each of these windows remains invariable. Due to these algorithmic characteristics of conventional convolution operation, a SIMD execution model is an efficient and practical model. However, since the convolution windows in transposed convolution operations exhibit a variable number of operations, a SIMD execution model is not an adequate design choice for these operations. While using a SIMD model utilizes the data parallelism between the convolution windows with the same number of operations, its

efficiency is limited in exploiting this execution model for the windows with a different number of operations. That is, if one uses a convolution accelerator with a SIMD execution model for transposed convolution operations, the processing engines that are performing the operations for a convolution window with fewer number of operations have to remain idle until the operations for other convolution windows finish. To address this challenge, we introduce a unified MIMD-SIMD architecture to accelerate the transposed convolution operation without compromising the efficiency of conventional convolution accelerators for convolution operations. This unified MIMD-SIMD architecture effectively maximizes the utilization of accelerator compute resources while effectively utilizing the parallelism between the convolution windows with different number of operations.

Inconsequential computations. The second challenge emanates from the large number of zeros inserted in the multidimensional input feature map for transposed convolution operations. Performing MAC operations on these zeros is inconsequential and wastes accelerator resources (See Figure 6.1), if not skipped. We address this challenge by leveraging an observation that even though the data access patterns in transposed convolution operations are irregular, they are still structured. Furthermore, these structured patterns are repetitive across the execution of transposed convolutional operations. Building upon these observations, the GANAX architecture decouples the operand access and execution. Each processing engine in this architecture consists of a simple access engine that repetitively generates the addresses for operand accesses without interrupting the execute engine. In the next sections, we examine these architectural challenges in details for GAN acceleration and expound the proposed microarchitectural solutions.

6.4.1 Unified MIMD-SIMD Architecture

In order to mitigate the resource underutilization, we devise a unified SIMD-MIMD architecture that reaps the benefits of SIMD and MIMD execution models at the same time. That is, while our architecture executes the operations for convolution windows with distinct



(a) GANAX Decoupled Access-Execute Architecture (b) Strided μIndex Generator

Figure 6.7: Organization of decoupled Access-Execute architecture.

computation patterns in a MIMD manner, it performs the operations of the convolution windows with the same computation pattern in a SIMD manner. Figure 6.6 illustrates the high-level diagram of the GANAX architecture, which is comprised of a set of identical processing engines (PE). The PEs are organized in a 2D array and connected through a dedicated network. Each PE consists of two μ-engines, namely the access μ-engine and the execute μ-engine. The access μ-engine generates the addresses for source and destination operands, whereas execute μ-engine merely performs simple operations such as multiplication, addition, and multiply-add. The memory hierarchy is composed of an off-chip memory and two separate on-chip global buffers, one for data and one for μops. These global on-chip buffers are shared across all the PEs. Each PE operates on one row of filter and one row of input and generates one row of partial sum values. The partial sum values are further accumulated horizontally across the PEs to generate the final output value.

Using a SIMD model for transposed convolution operations leads to resource under-utilization. The PEs that perform the computation for convolution windows with fewer number of operations remains idle, wasting computational resources. The simple solution is to replace the SIMD model with a fully MIMD computing model and utilize the parallelism between the convolution windows with different number of operations. However, a MIMD execution model requires augmenting each processing engine with a dedicated operation buffer. While this design resolves the underutilization of resources, it imposes

a large area overhead, more than 3.0× area overhead. Furthermore, fetching and decoding instructions from each of these dedicated operation buffers significantly increases the von Neumann overhead of instruction fetch and decode. To address these challenges, we design the GANAX architecture upon this observation that PEs in the same row perform same operations for a large period of time. As such, the proposed architecture leverages this observation and develop a middle ground between a fully SIMD and a fully MIMD execution model. The goal of designing the GANAX architecture is multi-faceted: (1) improve the PE underutilization by combining MIMD/SIMD model of computation for transposed convolution operations (2) without compromising the efficiency of SIMD model for conventional convolution operations. Next, we explain the two novel microarchitectural components that enable an efficient MIMD-SIMD accelerator design for GAN acceleration.

Hierarchical μop buffers. To enable a unified MIMD and SIMD model of execution, we introduce a two-level μop buffer. Figure 6.6 illustrates the high-level structure of the two-level μop buffer. The two-level μop buffer consists of a global and a local μop buffer. The local and global μop buffers work cooperatively to perform the computations for GANs. Each horizontal group of PEs, called processing vector (PV), shares a local μop buffer, whereas, the global μop buffer that is shared across all the PVs. The GANAX accelerator can operate in two distinct modes: SIMD mode and MIMD-SIMD mode. Since all the convolution windows in the convolution operation have the same number of multiply-adds, the SIMD execution model is a best fit. As such for this case, the global μop buffer bypasses the local μops and broadcasts the fetched μop to all the PEs. On the other hand, since the number of operations varies from one convolution window to another in transposed convolution operation, the accelerator works in MIMD-SIMD mode. In this mode, the global μop buffer sends distinct indices to each local μop buffer. Upon receiving the index, each local μop buffer broadcasts a μop, at the location pointed by the received index, to all the underlying PEs. Using MIMD-SIMD mode enables the GANAX accelerator to not only utilize the parallelism between the convolution windows with the same number

of operations, but also utilize the parallelism across the windows with distinct number of operations.

Global μop buffer. Before starting the computations of a layer, a sequence of high-level instructions, which defines the structure of each GAN layer, are statically translated into a series of μops. These μops are pre-loaded into the global μop buffer, and then the execution starts. Each of the μops either performs an operation across all the PEs (SIMD) or initiates an μop in each PV (MIMD-SIMD). The initiated operation in the MIMD-SIMD mode may vary from one PV to another. The SIMD and MIMD μops can be stored in the global μop buffer in any order. A 1-bit field in the global μop identifies the type of μop: SIMD or MIMD-SIMD.

In the SIMD mode—all the PEs share the same μop globally but execute it on distinct data—the global μop defines the intended operation to be performed by all the PEs. In this mode, the local μop buffer is bypassed and the global μop are broadcasted to all the PEs at the same time. Upon receiving the μop, all the PEs perform the same operation, but on distinct data. In the MIMD-SIMD mode—all the PEs within the same PV share the same μop but different PVs may execute different μops—the global μop is partitioned into multiple fields (one filed per each PV), each of which defines an index for accessing an entry in the local μop buffer. Upon receiving the index, each local μop buffer retrieves the corresponding μop stored at the given index and broadcasts it to all the PEs which it controls. The global μop buffer is double-buffered so that the next set of μops for performing the computations of GAN layer$_{i+1}$ can be loaded into the buffer while the μops for GAN layer$_i$ are being executed.

Local μop buffer. In the GANAX architecture, each PV has a dedicated local μop buffer. In the SIMD mode, the local μop buffers are completely bypassed and all the PEs perform the same operation that are sent from global μop buffer. In the MIMD-SIMD mode, each local μop buffer is accessed at the location specified by a dedicated field in the global μop. This location may vary from one local μop buffer to another. Then, the fetched μop

is broadcasted to all the PEs within a PV to perform the same operation but on distinct data. Each GAN layer may require a distinct sequence of μops both globally and locally. Furthermore, each PE may need to access millions of operands at different locations to perform the computations of a GAN layer. Therefore, we may need not to only add large μop buffers to each PE, but also drain and refill the μop buffers multiple times. Adding large buffers to the PEs adds a large area overhead, which could have been utilized to improve the computing power of the accelerator. Also, the process of draining and refilling the μop buffers imposes a significant overhead in terms of both performance and energy. To mitigate these overheads, we introduce decoupled access-execute microarchitecture that enables us to significantly reduce the size of μop buffers and eliminate the need to drain and refill the local μop buffers for each GAN layer.

6.4.2 Decoupled Access-Execute μEngines

Though the data access patterns in transposed convolution operation are irregular they are still structured. Furthermore, the data access patterns are repetitive across the convolution windows. Building upon this observation, we devise a microarchitecture that decouples the data accesses from from the data processing. Figure 6.7 illustrates the organization of the proposed decoupled access-execute architecture. The GANAX decoupled access-execute architecture consists of two major microarchitectural units, one for address generation (access μ-engine) and one for performing the operations (execute μ-engine).

The access μ-engine generates the addresses for the input, weight, and output buffers. The input, weight, and output buffers consume the generated addresses for each data read-/write. The execute μ-engine, on the other hand, receives the data from the input and weight buffers, performs an operation, and stores the result in the output buffer. The μops of these two engines are entirely segregated. However, the access and execute μ-engines work cooperatively to perform an operation. The μops for access μ-engine handle the configuration of index generator units. The μops for execute μengine *only* specify the type

of operation to be performed on data. As such, the execute μops do *not* need to include any fields for specifying the source/destination operands. Every cycle, the access μengine sends out the addresses for source and destination operands based on its preconfigured parameters. Then, the execute μengine performs an operation on the source operands. The result of the operation is, then, stored in the location that is defined by the access μengine. Having decoupled μ-engines for accessing the data and executing the operations has a paramount benefit of reusing execute μops. Since there is no address field in the execute μops, we can reuse the same execute μop on distinct data over and over again without the need to change any fields in the μops. Reusing the same μop on distinct data helps to significantly reduce the size of μop buffers.

Access μ-engine. Figure 6.7 illustrates the microarchitectural units of access μ-engine. The main function of access μ-engine is to generate the addresses for source and destination operands based on a preloaded configuration. While designing a full-fledged access μ-engine that is capable of generating various patterns of data addresses establishes flexibility for the GANAX accelerator, but it is an overkill for our target application (i.e., GANs). As mentioned in the dataflow section (Section 6.3), the data access patterns for transposed convolution operations are irregular, yet structured. Based on our analysis over the evaluated GANs, we observe that the data accesses in the GANAX dataflow are either *strided* or *sequential*. The stride value for a strided data access pattern depends on the number of inserted zeros in the multidimensional input activation. Furthermore, these data access patterns are repetitive across a large number of convolution windows and for large number of cycles. We leverage these observations to simplify the design of the access μ-engine. Figure 6.7(a) depicts the block diagram of the access μengine in GANAX. The access engine mainly consists of one or more strided μindex generators. The μindex generator can generate one address every cycle, following a pattern governed by a preloaded configuration. Since the data access patterns may vary from one layer to another, we design a reconfigurable μindex generator.

Figure 6.7(b) depicts the block diagram of the proposed reconfigurable μindex generator. There are five configuration registers that govern the pattern for data address generation.

The Addr. configuration register specifies the initial address from which the data address generation starts, while the Offset configuration register can be used to offset the range of generated addresses as needed. The Step configuration register specifies the step size between two consecutive addresses, while the End configuration register specifies the final value up to which the addresses should be generated. Finally, the Repeat configuration register indicates the number of times that a configured data access pattern should be replayed. The modulo adder, which consists of an adder and a subtractor, is used to enable data address generation in a rotating manner. The modulo adder performs a modulo addition on the values stored in the Addr. and Step registers. If the result of this modulo addition is fewer than the value in End register, the calculated result is sent to the output. This means that the next address to be generated is still within the range of Addr. and End register values. Otherwise, the result of the modulo addition minus the value of End register is sent to the output. That is, the next address to be generated is beyond the End register value and the address generation process must start over from the beginning. In this scenario, the Decrement signal is also asserted which cause the value of the Repeat register to be decreased by one, indicated one round of address generation is finished. Once the Repeat register reaches zero, the Stop signal is asserted and no more addresses are generated. After configuring the parameters, the strided μindex generator can yield one address per cycle without any further interventions from the controller. Using this configurable μindex generator along the observation that the data address patterns in GANs are structured, the GANAX architecture can bypass the inconsequential computations and save both cycles and energy.

Execute μ-engine. Figure 6.7(b) depicts the microarchitectural units of execute μ-engine. The execute μ-engine consists of an ALU, which can perform simple operations such

as addition, multiplication, comparison, and multiply-add. The main job of execute μ-engine is *just* to perform an operation on the received data. At each cycle the execute μ-engine consumes one μop from the μop FIFO and performs the operation on the source operands and store the result back into the destination operand. If the μOp FIFO becomes empty, the execute μop halts and no further operation is performed. In this case, all the input/weight/output buffers are notified to stop their reads/writes. The decoupling between access and execute μengines enables us to remove the address field from the execute μops. Removing the address field from the execute μops allow us to reuse the same μops over and over again on different data. Furthermore, we leverage this μop reuse and the fact that the computation of the CNN requires a small set of μops ($\approx$ 16) to simplify the design of the μop buffers. Instead of draining and refilling the μop buffers, we preload all the necessary μops for convolution and transposed convolution operations in the μop buffers. For the local μop buffer, we load *all* the μops before starting the computation of a GAN.

Synchronization between μengines. In the GANAX architecture (Figure 6.7), there is one address FIFO for each strided μindex generator. The address FIFOs perform the synchronization between access μ-engine and execute μ-engine. Once an address is generated by a strided μindex generator, the generated address is pushed into the corresponding address FIFO. The addresses in the address FIFOs are later consumed to read/write data from/into the data buffers (i.e., input/weight/output buffers). If any of the address FIFOs are full, the corresponding strided μindex generator stops generating new addresses. In the case that any of the address FIFOs are empty, no data is read/written from/into its corresponding address FIFO.

6.5 Instruction Set Architecture Design (μOps)

The GANAX ISA should provide a set of μops to efficiently map the proposed flow of data for both generative and discriminative models onto the accelerator. Furthermore, these μops should be sufficiently *flexible* to serve distinct patterns in the computation for both

convolution and transposed convolution operations. Finally, to keep the size of μop buffers modest, the set of μops should be *succinct*. To achieve these multifaceted goals, we first introduce a set of algorithmic observations that are associated with GAN models. Then, we introduce the major μops that enable the execution of GAN models on GANAX.

6.5.1 Algorithmic Observations

The following elaborates a set of algorithmic observations that are the foundation of the GANAX μops.

(1) MIMD/SIMD execution model. Due to the regular and structured patterns in the computation across the convolution windows in conventional DNNs, they are best suited for SIMD processing. However, the patterns in the computation of GANs are inherently different between generative and discriminative models. Due to the inserted zeros in the generative models, their patterns in the computation vary from one convolutional window to another. We observe that exploiting a combination of SIMD and MIMD execution model can be more efficient in accelerating GAN models than solely relying on SIMD. Therefore, the focus of the GANAX μops is to include the operations that enable GANAX to fully utilize the SIMD and MIMD execution models.

(2) Repetitive computation patterns. We observe that even though GANs require a large number of computations, most of these computations are similar between generative and discriminative models. In addition, these computations are repetitive over a long period of time. Building upon this observation, we introduce a customized `repeat` μop that significant reduces the μop footprints. In addition, the commonality between the operations in generative and discriminative models allows us to design a succinct, yet representative, set of μops. To further reduce the μop footprints, we introduce a dedicated set of execute μops that only define the type of operations. These μops are reused for distinct data during the execution of generative and discriminative models on the GANAX architecture.

(3) Structured and repetitive memory access patterns. We observe that despite the

irregularity of memory access patterns in generative models, they are still structured and repetitive. Analyzing the data access patterns of various GANs reveals that their memory access patterns are either sequential or strided. Building upon this observation and our decoupled access-execute architecture, we introduce a set of access μops that are used merely to configure the access μengines and initiate the address generation process. Once initiated, the access μengines generate the configured access patterns over and over until they are intervened.

6.5.2 Access μOps

GANAX access μops are used to configure the access μengine and initiate/stop the process of address generation. These μops are executed across all the PEs within a PV whose index is indicated by `pv_index` field in the μops. Furthermore, in all of these μops, `%addrgen_idx` specifies the index of the targeted address generator in the access μengine. The supported μops in the access μengines are as follows:

1. **`access.cfg`** `%pv_idx, %addrgen_idx, %dst, imm`: This μop loads a 16-bit `imm` value into one of the five `%dst` configuration registers (*i.e.,* as shown in Figure 6.7(b), these configuration registers are Addr., Offset, Step, End, and Repeat) of one of the address generators in the access μengine.
2. **`access.start`** `%pv_idx, %addrgen_idx`: This μop initiates the address generation in one of the address generators in the access μengine. The process of address generation continues until an **`acceess.stop`** μop is executed or the iteration register reaches zero.
3. **`access.stop`** `%pv_idx, %addrgen_idx`: This μop intervenes the address generation of one of the address generators in the access μengine. The address generation can be re-initiated again by executing an `access.start` μop.

6.5.3 Execute μOps

Execute μops are categorized into two groups: (1) SIMD μops are fetched from each PE's local μop buffer and executed locally within each PE and (2) the MIMD μops are fetched from the global μop buffer and executed across all PEs. The SIMD μops can be executed in the MIMD manner as well. That is, the MIMD μops are a superset of the SIMD μops. We first introduce the SIMD μops, then explain the extra μops that belong to the MIMD group.

SIMD μops. SIMD group only comprises a succinct, yet representative set of μops for performing convolution and transposed convolution operations. The combination of SIMD μops and the decoupled access-execute architecture in GANAX helps to reduce the size of local μop buffers. The SIMD μops do not have source or destination fields and only specify the type of operation to be executed. Once executed, depending on the type of operation, a given PE consumes the generated addresses by the μindex generators and delivers the data to the execute μengine. Since these μops do not have any source or destination register, they are pre-loaded into the local μop buffers before starting the execution. Then, they are re-used over and over, on distinct data whose addresses are generated by the access μengines. The SIMD μops are as follows:

1. `add`, `mul`, `mac`, `pool`, and `act`: Depending on the type, these μops consume one or more addresses from the μindex generators for source and destination operands. For example, `add` consumes two addresses for the source operands and one address for the destination operand, but `act` uses one address for the source operand and one address for the destination operand.
2. `repeat`: This μop causes the next fetched μop to be repeated a specified number of times. This number is specified in a microarchitectural register in each PE. This register is pre-loaded with a MIMD μop before the execution starts.

MIMD μops. The MIMD μops are loaded into the global μop buffers and executed

Table 6.1: The evaluated GAN models, their released year, and the number of convolution (`Conv`) and transposed convolution (`TConv`) layers per generative and discriminative models.

			Generative		Discriminative	
Name	**Year**	**Description**	**# Conv**	**# TConv**	**# Conv**	**# TConv**
3D-GAN	2016	3D objects generation	-	4	5	-
ArtGAN	2017	Complex artworks generation	-	5	6	-
DCGAN	2015	Unsupervised representation learning	-	4	5	-
DiscoGAN	2017	Style transfer from one domain to another	5	4	5	-
GP-GAN	2017	High-resolution image generation	-	4	5	-
MAGAN	2017	Stable training procedure for GANs	-	6	6	6

globally across all the PEs. In addition to all the SIMD μops, the following μops execute in a MIMD manner:

1. **`mimd.ld`** `%pv_idx, %dst, imm`: This μop loads the immediate value (`imm`) into one of the microarchitectural registers (`%dst`) of all the PEs with a PV. The `%pv_idx,` specifies the index of the target PV. This μop is mainly used to load an immediate value into the repeat register.
2. **`mimd.exe`** `%`μ`op_index`$_1$`,..., %`μ`op_index`$_i$: Upon receiving this μop, the ith PV fetches a μop located at location `%`μ`op_index`$_i$ from its local μop buffer and executes it across all the PEs horizontally. Since the value of the `%`μ`op_index` may vary from one PV to another, this μop causes GANAX to operate in a MIMD manner.

6.6 Methodology

Workloads. We use several state-of-the-art GANs to evaluate the GANAX architecture. Table 6.1, shows the evaluated GANs, a brief description of their applications, and the number of convolution (`Conv`) and transposed convolution (`TConv`) layers per generative and discriminative models.

Hardware design and synthesis. We implement the GANAX microarchitectural units including the strided μindex generator, the arithmetic logic of the PEs, controllers, non-linear function, and other logic hardware units in Verilog. We use `TSMC 45 nm` standard-cell library and `Synopsys Design Compiler (L-2016.03-SP5)` to synthesize these units and obtain the area, delay, and energy numbers.

Energy measurements. Table 6.2 shows the energy numbers for major micro-architectural

Operation	Energy (pJ/Bit)	Relative Cost
Register File Access	0.20	1.0
16-bit Fixed Point CE	0.36	1.8
Inter-PE Communication	0.40	2.0
Global Buffer Access	1.20	6.0
DDR4 Memory Access	15.00	75.0

Table 6.2: Energy comparison between GANAX microarchitectural units and memory. PE energy includes the energy consumption of an arithmetic operation and the strided μindex generators.

units, memory operations, and buffer accesses in TSMC 45nm technology. To measure the area and read/write access energy of the register files, SRAMs, and local/global buffers, we use CACTI-P [163]. To have a fair comparison, we use energy numbers reported in TETRIS [110], which has a similar PE architecture as EYERISS. In Table 6.2, the energy overhead of strided μindex generators is included in the normalized energy cost of PE. For DRAM accesses, we use the Micron's DDR4 system power calculator [161]. The same frequency (500 MHz) is used for both EYERISS and GANAX in all the experiments.

Architecture configurations. In this thesis, we study a configuration of GANAX with 16 Processing Vectors (PVs) each with 16 Processing Engines (PEs). We use the default EYERISS configurations for on-chip memories such as the size of input and partial sum registers, weight SRAM, and global data buffer. The same on-chip memory sizes are used for GANAX. Each local μop buffer has 16 entries. The number of entries is sufficient to encompass all the execute μops. The global μop buffer has 32 entries each with 64 bits, four bits per each PV. Each local μop uses these four bits to index its local μop buffer. An extra one bit in the global μops determines the execution model of the accelerator for the current operation (i.e., SIMD or MIMD-SIMD).

Area analysis. Table 6.3 shows the major architectural components for the baseline architecture (EYERISS [184, 111]) and GANAX in 45 nm technology node. For logic of the microarchitectural units, we use the reported area from the synthesis. For the memory elements, we use CACTI-P [163] and the reported numbers in EYERISS [184]. In order to be consistent in the results, we scaled down the reported area numbers in EYERISS from 65 nm to 45 nm. To have a fair comparison between EYERISS and GANAX, the same number of PEs and on-chip memory are used for both accelerators. Under this setting, GANAX has

Table 6.3: Area measurement of the major hardware units with TSMC 45nm.

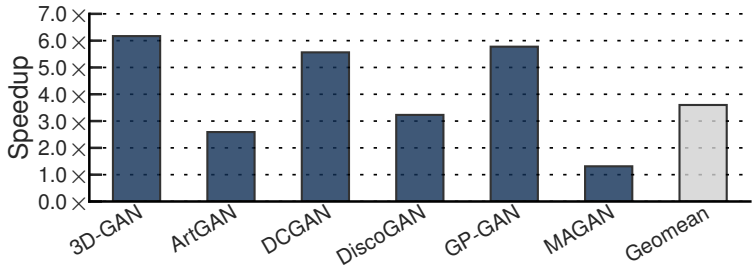
	GANAX Hardware Units	Configuration	Area	
			um^2	%
Processing Engine (PE)	Input Register	12 × 16 Bits	766.9	2.6%
	Partial Sum Register	24 × 16 Bits	1533.7	5.2%
	Weight SRAM	224 × 16 Bits	14378.7	48.8%
	Multiply-and-Accumulate	16-bit Fixed Point	2875.7	9.8%
	Non-Linear Function	Lookup Table	95.9	0.3%
	Strided µIndex Generator	3	479.3	1.6%
	Local µOP Buffer	16 × 16 Bits	958.6	3.3%
	I/O FIFOs	8 × 32 Bits	5026.8	17.1%
	PE Controller	N/A	3356.0	11.4%
Total Area / PE			29471.6	100.0%
GANAX	Total PE Array	16 × 16	7544466.2	83.2%
	Global µOP Buffer	32 × 64 Bits	9585.8	0.1%
	Global Data Buffer	108 KBytes	1102366.9	12.2%
	Global Instruction Buffer	27 KBytes	275591.7	3.0%
	Others (NoC, Config Buffers)	N/A	115029.6	1.3%
	Global Controller	N/A	19171.6	0.2%
GANAX Total Area			9066211.8	100.0%

an area overhead of ≈7.8 % compared to EYERISS.

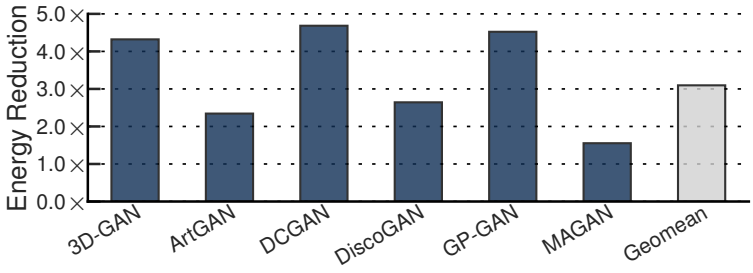
Microarchitectural simulation. Table 6.3 shows the major microarchictural parameters of GANAX. We implement a microarchitectural simulator on top of the EYERISS simulator [110]. The extracted energy numbers from logic synthesis and CACTI-P are integrated into the simulator to measure the energy consumption of the evaluated network models on GANAX. To evaluate our proposed accelerator, we extend the EYERISS simulator with the proposed ISA extensions and the GANAX flow of data. For all the baseline numbers, we use the plain version of the simulator.

6.7 Evaluation

Overall performance and energy consumption comparison. Figure 6.8a depicts the speedup of the generative models with GANAX over EYERISS [111]. On average, GANAX yields 3.6× speedup improvement over EYERISS. The generative models with a larger fraction of inserted zeros in the input data and larger number of inconsequential operations in transposed convolution layers enjoy a higher speedup with GANAX. Across all the evaluated models, 3D-GAN achieves the highest speedup (6.1 ×). This higher speedup is mainly attributed to its larger number of inserted zeros in its transposed convolution layers. On average, the number of inserted zeros for 3D-GAN is around 80% (See Figure 6.1). On the other extreme, MAGAN enjoys a speedup of merely 1.3×, which



(a) Speedup



(b) Energy Reduction

Figure 6.8: Speedup and energy reduction of generative models compared to EYERISS [111].

is attributed to the lowest number of inserted zeros in its transposed convolution layers compared to other GANs.

Figure 6.8b shows the energy reduction achieved by GANAX over EYERISS. On average, GANAX effectively reduces the energy consumption by 3.1× over the EYERISS accelerator. The GANs (3D-GAN, DCGAN, and GP-GAN) with the highest fraction of zeros and inconsequential operations in the transposed convolution layers enjoy an energy reduction of more than 4.0×. These results reveal that our proposed architecture is efficient in addressing the main sources of inefficiencies in the generative models. Figure 6.10 shows the normalized runtime and energy breakdown between the discriminative and generative models. The first (second) bar shows the normalized runtime (energy) for EYERISS (GANAX). As the results show, while GANAX significantly reduces both the runtime and energy consumption of generative models, it delivers the same level of efficiency as EYERISS for the discriminative models.

Energy breakdown of the microarchitectural units. Figure 6.9 illustrates the overall normalized energy breakdown of the generative models between distinct microarchitectural components of the GANAX architecture.

The first and second bars show the normalized energy consumed by EYERISS and

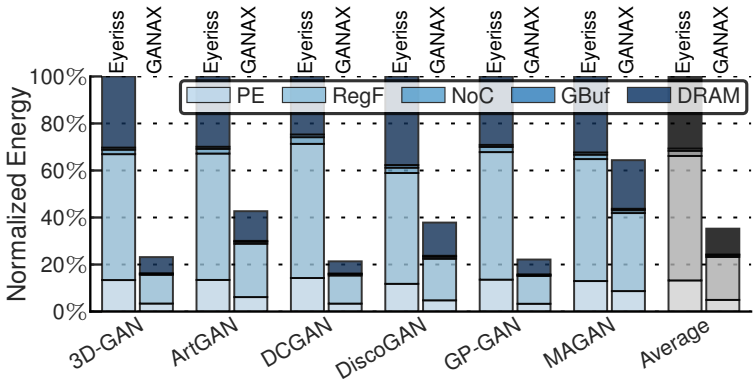
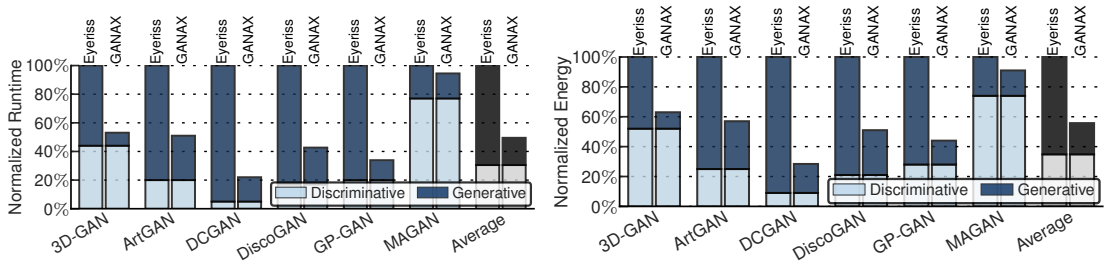


Figure 6.9: Breakdown of energy consumption of the generative models between different microarchitectural units. The first bar shows the normalized energy breakdown for EYERISS. The second bar show the energy breakdown for GANAX normalized to EYERISS.



(a) Runtime **(b)** Energy

Figure 6.10: Breakdown of (a) runtime and (b) energy consumption between discriminative and generative models normalized to the runtime and energy consumption of EYERISS. For each network, the first (second) bar show the normalized value when the application is executed on EYERISS (GANAX).

GANAX, respectively. As the results show, GANAX reduces the energy consumption of all the microarchitectural units. This reduction is mainly attributed to the efficient flow of data in GANAX and the decoupled access-execute architecture that cooperatively diminishes the sources of inefficiencies in the execution of transposed convolution operations.

Processing elements utilization. To show the effectiveness of GANAX dataflow in improving the resource utilization, we measure what percentage of the total runtime, the PEs are actively performing a consequential operation. Figure 6.11 depicts the utilization of PEs for EYERISS and GANAX. GANAX exhibits a high percentage of PE utilization, around 90% across all the evaluated GANs.

This high resource utilizations in GANAX is mainly attributed to the proposed dataflow that can effectively force the computation of the rows with similar computation pattern adjacent to each other. This adjacency eliminates the inconsequential operations, which leads to a significant improvement in the utilization of the processing engines.

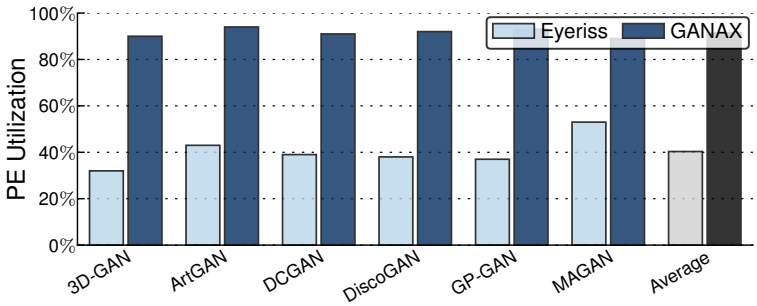


Figure 6.11: Average PE utilization for the generative models in EYERISS and GANAX.

6.8 Conclusion

Generative adversarial networks harness both generative and discriminative deep models in a game theoretical framework to generate close-to-real synthetic data. The generative model uses a fundamentally different mathematical operator, called transposed convolution, as opposed to the conventional convolution operator. Transposed convolution extrapolates information by first inserting zeros and then applying convolution that needs to cope with irregular placement of none-zero data. To address the associated challenges for executing generative models without sacrificing accelerator performance for conventional DNNs, this work devised the GANAX accelerator. This accelerator introduced a unified architecture that conjoins SIMD and MIMD execution models to maximize the efficiency of the accelerator for both generative and discriminative models. On the one hand, to conform to the irregularities in the generative models, which are formed due to the zero-insertion step, GANAX supports selective execution of only the required computations by switching to a MIMD-SIMD mode. To support this mixed execution mode, GANAX offers a decoupled micro access-execute paradigm at the finest granularity of its processing engines. On the other hand, for the conventional discriminator DNNs, it sets the architecture in a purely SIMD mode. The evaluation results across a variety of generative adversarial networks reveal that the GANAX accelerator delivers, on average, 3.6× speedup and 3.1× energy reduction for the generative models. These significant benefits are attained without sacrificing the execution efficiency of the conventional discriminator DNNs.

CHAPTER 7

RELATED WORK

7.1 Limited Precision Neuro-General Computing

This research lies at the intersection of (a) general-purpose approximate computing, (b) accelerators, (c) analog and digital neural hardware, (d) neural-based code acceleration, (e) and limited-precision learning. This work combines techniques in all these areas to provide a compilation workflow and the architecture/circuit design that enables code acceleration with limited-precision mixed-signal neural hardware. In each area, we discuss the key related work that inspired our work.

General-purpose approximate computing. Several studies have shown that diverse classes of applications are tolerant to imprecise execution [208, 209, 210, 211, 26]. A growing body of work has explored relaxing the abstraction of full accuracy at the circuit and architecture level for gains in performance, energy, and resource utilization [9, 11, 10, 12, 13, 14, 19, 15, 16, 17, 18]. These circuit and architecture studies, although proven successful, are limited to purely digital techniques. We explore how a mixed-signal, analog-digital approach can go beyond what digital approximate techniques offer.

Accelerators. Research on accelerators seeks to synthesize efficient circuits or FPGA configurations to accelerate general-purpose code [212, 213, 214, 215, 130]. Similarly, static specialization has shown significant efficiency gains for irregular and legacy code [39, 216]. More recently, configurable accelerators have been proposed that allow the main CPU to offload certain code to a small, efficient structure [217, 218]. This paper extends the prior

work on digital accelerators with a new class of mixed-signal, analog-digital accelerators.

Analog and digital neural hardware. There is an extensive body of work on hardware implementations of neural networks both in digital [219, 220, 221] and analog [222, 223, 219, 224, 22]. Recent work has proposed higher-level abstractions for implementation of neural networks [225]. Other work has examined fault-tolerant hardware neural networks [226, 227]. In particular, Temam [226] uses datasets from the UCI machine learning repository [228] to explore fault tolerance of a hardware neural network design. In contrast, our compilation, neural-network selection/training framework, and architecture design aim at applying neural networks to general-purpose code written in familiar programming models and languages, not explicitly written to utilize neural networks directly.

Neural-based code acceleration. A recent study [229] shows that a number of applications can be manually reimplemented with explicit use of various kinds of neural networks. That study did not prescribe a programming workflow, nor a preferred hardware architecture. More recent work exposes analog spiking neurons as primitive operators [199]. This work devises a new programming model that allows programmers to express digital signal-processing applications as a graph of analog neurons and automatically maps the expressed graph to a tiled analog, spiking-neural hardware. The work in [199] is restricted to the domain of applications whose inputs are real-world signals that should be encoded as pulses. Our approach addresses the long-standing challenges of using analog computation (programmability and generality) by not imposing domain-specific limitations, and by providing analog circuitry that is integrated with a conventional digital processor in a way that does not require a new programming paradigm.

Limited-precision learning. The work in [230] provides a complete survey of learning algorithms that consider limited precision neural hardware implementation. We tried various algorithms, but we found that CDLM [24] was the most effective. More sophisticated

limited-precision learning techniques can improve the reported quality results in this thesis and further confirm the feasibility and effectiveness of the mixed-signal, approach for neural-based code acceleration.

7.2 Neuro-General Computing for GPU Throughput Processors

Recent work has explored a variety of approximation techniques that include: (a) approximate storage designs [231, 14] that longer lifetime [231] and trades quality of data for reduced energy [14]. (b) voltage over-scaling [13, 148, 16], (c) loop perforation [58, 37, 232], (d) loop early termination [36], (e) computation substitution [9, 233, 36, 234], (f) memoization [17, 42, 43], (g) limited fault recovery [15, 235, 235, 211, 37, 208], (h) precision scaling [26, 236], (i) approximate circuit synthesis [50, 237, 142, 238, 137, 239, 240], and (j) neural acceleration [12, 6, 48, 45, 46, 47].

This work falls in the last category; yet, we exclusively focus on the integration of neural accelerators within GPU throughput processors. The prior work on neural acceleration mostly focuses on single-threaded CPU code acceleration by either loosely coupled neural accelerators [46, 47, 45, 199, 241] or tightly-coupled ones [12, 6]. Grigorian et al. study the effects of eliminating control-flow divergence by converting SIMD code to software neural networks with no hardware support [48]. However, prior work does not explore tight integration of neural hardware in throughput processors; and does not study the interplay of data parallel execution and hardware neural acceleration. Prior to this work, the benefits, limits, and challenges of integrating hardware neural acceleration within GPUs for many-thread data-parallel applications was unexplored.

There are several other approximation techniques in the literature that can or have been applied to GPU architectures. Loop perforation [58] periodically skips loop iteration for gains in performance and efficiency. Green [36] terminates loops early or substitute compute intensive functions with simpler, lower quality versions that are provided by the programmer. Relax [15] is compiler/architecture system for suppressing hardware fault

recovery in approximable regions of code, exposing these errors to the application. Fuzzy memoization forgoes invoking a floating point unit if the inputs are in the neighborhood of previously seen inputs. The results of the previous calculation is reused as an approximate result. Arnau et al. use hardware memoization to reduce redundant computation in GPUs [43]. Sartori et al. propose a technique that mitigates branch divergence by forcing the divergent threads to execute the most popular path [233]. In case of memory divergence, they force all the threads to access the most commonly demanded memory block. SAGE [9] and Paraprox [42] perform compile-time static code transformations on GPU kernels that include data compression, profile-directed memoization, thread fusion, and atomic operation optimization. Our quality control mechanism takes inspiration from the quality control in these two works.

In contrast, we describe a hardware approximation technique that integrates neural accelerators within the pipeline of the GPU cores. In our design, we aim at minimizing the pipeline modifications and utilizing existing hardware components. Distinctively, our work explores the interplay between data parallelism and neural acceleration and studies its limits, challenges, and benefits.

7.3 Acceleration-Approximation in Deep Neural Networks

SNAPEA is fundamentally different from the prior studies on designing accelerators for CNNs in three major ways: (1) we exploit the inherent algorithmic structure of CNNs and runtime information to judiciously perform early activation and save ineffectual computations , (2) we expose a knob that enables the user to gracefully navigate the trade-offs between the classification accuracy, performance, and energy efficiency , and finally (3) we study the rich and unexplored area of task skipping in the domain of deep convolutional neural networks and conjoin these two disjoint lines of research in SNAPEA. Below, we discuss the most related works.

CNN accelerators. Several accelerators for convolutional neural networks has been pro-

posed [110, 156, 111, 242, 243, 244, 188, 181, 157, 195, 245, 196, 158, 200]. In some of the most recent works [111, 195, 200], 2D spatial architectures have been proposed to match with the convolution dataflow and maximize the data reuse. TETRIS [110] and Neurocube [242] have almost the same compute engines as the previous CNN accelerators. However, these works studied the challenges and opportunities for designing efficient CNN accelerators in a 3D-stacked memory setting. Neither of these accelerators evaluated the benefits of performing early activation in the convolution operation.

Pruning techniques. A handful of research [246, 247, 248, 70, 164] proposed various static pruning techniques to reduce the overhead of computation in deep convolutional neural networks. These static pruning techniques are agnostic to the dynamically-generated zeros whose locations in the activation layer vary from one image to another. As our results show, SNAPEA is complementary to these techniques and further improve the benefits over the static pruning techniques. Furthermore, several architectures also have been proposed [156, 157, 244, 188, 181] for exploiting the sparsity in the input activations and/or weights to improve the efficiency of the accelerator. In one of the most recent work, SCNN [156] designs an accelerator that exploits the sparsity in both the activations and weights. The proposed novel dataflow in SCNN maximizes the data reuse in the sparse activations and weights. This work is orthogonal to the previous efforts that focused on exploiting the sparsity in CNN accelerators. SNAPEA takes on a distinct approach than prior designs by judiciously re-ordering the MAC operations in a sliding window and performing the early activation in convolutional windows.

Task skipping. A handful of research efforts [249, 65, 250, 58, 37, 251, 252, 253] have looked into task skipping in various domains. In one of the most recent efforts [58], Sidiroglou et al. proposed loop perforation in which the accuracy is traded in return for improvement in performance. In their proposal, they algorithmically transform the critical loops in the program and *only* execute a subset of their iterations. The rate of loop perforation is determined statically before executing the program and is agnostic about the partial

values after each iteration. PredictiveNet [249] proposes a skipping mechanism for CNNs. They first perform the computations on the most-significant bits and then speculatively decide whether to perform the computation on the least-significant bits. However, SNAPEA completely skips the computations of the significant fraction of the operations. As such, SNAPEA not only reduces the computation cost, but also reduces the number of accesses to the on-chip buffers. Whereas PredictiveNet only reduces the computation cost and does *not* change the number of memory accesses, which significantly contribute to the overall energy consumption [110, 111, 153, 183]. In contrast to PredictiveNet, which inherently imposes inaccuracy in the final classification, SNAPEA's basic approach (i.e. exact mode), which only relies on a simple sign check, does not impose any classification inaccuracies. Although SNAPEA takes inspiration from the prior proposals in task skipping, it uniquely applies the task skipping mechanism in the domain of deep convolutional neural networks in order to effectively eliminate the ineffectual data transfers and computations.

7.4 Unsupervised Learning Acceleration

GANAX has fundamentally a different accelerator architecture than the prior proposals for deep network acceleration. In contrast to prior work that mostly focus on convolution operation, GANAX accelerates transposed convolution operation, a fundamentally different operation than conventional convolution. Below, we overview the most relevant work to ours along two dimensions: neural network acceleration and MIMD-SIMD acceleration.

Neural network acceleration. Accelerator design for neural networks has become a major line of computer architecture research in recent years. A handful of prior work explored the design space of neural network acceleration, which can be categorized into ASICs [183, 110, 184, 156, 111, 181, 188, 157, 189, 195, 38, 197, 182, 199, 12], FPGA implementations [190, 153, 46, 196, 200], using unconventional devices for acceleration [194, 191, 6], and dataflow optimizations [185, 187, 186, 193, 111, 192, 198]. Most of these studies have focused on accelerator design and optimization of merely one specific type of convolutional

as the most compute-intensive operation in deep convolutional neural networks.

EYERISS [111] proposes a row stationary dataflow that yields high energy efficiency for convolutional operation. EYERISS exploits data gating to skip zero inputs and further improves the energy efficiency of the accelerator. However, EYERISS still wastes cycles for detecting the zero-valued inputs. Cnvlutin [157] can save compute cycle and energy for zero-values inputs but still wastes resources for zero-valued weights. In contrast, Cambricon-X [188] can skip zero-valued weights but still wastes compute cycles and energy for zero-input values. SCNN [156] proposes an accelerator that can skip both zero-valued inputs and weights and efficiently performs convolution on highly sparse data. However, not only can SCNN handle dynamic zero-insertion in input feature maps, but also is not efficient for non-sparse vector-vector multiplications, which are the dominant operation in discriminative models of GANs. None of these works can perform zero-insertion into the input feature maps, which is fundamentally a requisite for transposed convolution operation in the generative models. In contrast to these successful prior work in neural network acceleration, GANAX proposes a unified architecture for efficient acceleration of both conventional convolution and transposed convolution operations. As such, GANAX encompasses the acceleration of a wider range of neural network models.

MIMD-SIMD accelerators. While the idea of access-execute is not brand-new, GANAX extends the concept of access-execute architecture [201, 202, 203, 204] to the finest granularity of computation for each individual operand for deep network acceleration. A wealth of research has studied the benefits of MIMD-SIMD architecture in accelerating specific applications [254, 255, 256, 257, 258, 259, 260, 261, 262]. Most of these works have focuses on accelerating computer vision applications. For example, PRECISION [255] proposes a reconfigurable hybrid MIMD-SIMD architecture for embedded computer vision. In the same line of research, a recent work [262] proposes a multicore architecture for real-time processing of augmented reality applications. The proposed architecture leverages SIMD and MIMD for data- and task-level parallelism, respectively. While these

works have studied the benefits of MIMD-SIMD acceleration mostly for computer vision applications, they did not study the potential gains of using MIMD and SIMD accelerators for modern machine learning applications. Prior to this work, the benefits, limits, and challenges of MIMD-SIMD architectures for modern deep model acceleration was unexplored. Conclusively, the GANAX architecture is the first to explore this uncharted territory of MIMD-SIMD acceleration for the next generation of deep networks.

7.5 In-DRAM Near-Data Neuro-General Computing

There have been several proposed architectures and accelerators for processing in memory. However, AxRAM is fundamentally different from the prior studies on PIM in two major ways: (1) instead of using 3D/2.5D-stacked technology, we build on conventional graphics DRAM devices and (2) we study the unexplored area of tightly integrating approximate accelerators in memory. AxRAM represents a convergence of two main bodies of research, approximate computing and processing in memory. Below, we discuss the most related work in these two domains.

Neural acceleration. Several architectures have been proposed for neural acceleration [38, 45, 46, 47, 197, 6, 241, 199, 48, 12]. For example, prior work tightly integrated such neural accelerators with GPUs for significant improvement in performance and energy efficiency [38], but the improvement quickly diminishes due to limited off-chip DRAM bandwidth. In contrast, we leverage the simplicity of the neural accelerator architecture to tightly integrate them with conventional DRAM devices. This makes in-DRAM acceleration more practical and improves the gains from neural acceleration by overcoming the off-chip memory bandwidth wall. Prior to this work, the benefits, limits, and challenges of tight integration of neural accelerators in the conventional graphics DRAM devices was unexplored.

Processing in memory. Traditional PIM systems [263, 264, 265, 266, 267, 268, 269] integrate logic and memory onto a single die to enable lower data access latency and higher

memory bandwidth, but they suffer from high manufacturing cost and low yield. Recently, a wealth of architectures for PIM have been proposed, ranging from fully programmable to fixed-function, using 3D/2.5D stacking technologies [110, 270, 271, 75, 272, 242, 273, 72, 274, 275, 276, 277, 278, 279, 280] A handful of recent work [271, 281, 282] also studied the effect of data organization/mapping to improve the performance of near-data processing. For instance, TOP-PIM [277] uses GPGPU compute-units for scientific and data analytics processing, Tesseract [274] uses in-order cores for graph processing, NDC [278] uses in-order cores for map-reduce workloads, IBM's Active Memory Cube [276] uses vector units for scientific applications, and others [275, 279, 72] use fixed-function or programmable accelerators for application acceleration. Ahn et al. [274] propose a scalable PIM accelerator for large-scale graph processing using 3D integration technology. To accelerate the processing of sparse matrix data, Zhu et al. [280] propose a 3D-stacked logic-in-memory (LiM) system. In this work, the customized LiM layers are integrated between DRAM dies to efficiently perform sparse matrix operations. Gao et al. [272] propose a heterogenous reconfigurable logic for near-data processing. There have also been proposals to enable near-memory processing for commodity 2D DRAM devices [72, 273]. Compared with the prior work, our work makes in-DRAM processing more practical and efficient in heavily-threaded GPU systems by exploiting approximate computing, while reaping the full benefit of in-DRAM processing. By leveraging techniques from approximate computing, we are able to integrate approximate accelerators in commodity 2D DRAM. This allows us to not only accelerate data processing but reduce the energy cost and overhead of data transfer.

CHAPTER 8

FUTURE WORK

I conclude this dissertation with a discussion of future research avenue that I am interested to pursue. For my future research, I am interested in exploring the interplay between machine learning and architecture/system design.

Machine learning for architecture and system design. After devoting my graduate studies to exploring architectural techniques to improve the efficiency of machine learning applications, I have since become interested in the inverse relationship of these disciplines. That is, I have gravitated towards exploring the rich and relatively less-studies area of leveraging the recent advances in machine learning algorithms to mitigate the current challenges in computer architecture and design more efficient systems. As such, I have defined two research problems that shape my research passion for upcoming years. First, I aim to answer this research question for spatial accelerators: ”Under a fixed area budget, how can the on-chip resources among memory and compute units be efficiently allocated in order to maximize the joint performance and energy efficiency?”. The main challenge is that the potential design space for this problem is prohibitively large for an exhaustive search. In response, I propose to formulate on-chip resource allocation as an optimization problem. I am interested to use machine learning techniques, particularly reinforcement learning, to find an optimal policy for designing efficient spatial accelerators under a given area constraint. Building upon this project, I aim to explore an optimal mapping and runtime scheduling of a given computational graph across heterogeneous compute devices in a distributed cloud-based system. The end goal is to strike a balance between the commu-

nication and computation loads across the heterogeneous compute devices and maximally utilize the available resources. Similar to the first project, reinforcement learning is well suited to solve such an optimization problem. However, in contrast to the previous research challenge, in which the reinforcement learning finds a fixed optimal policy for the entire duration of the application lifecycle, the main challenge here is that the reinforcement learning must exploit the runtime feedbacks from the system and dynamically refine its learned policy in order to balance the overall computation and communication load in the system. I am very excited to embark on these projects and explore the mutually beneficial interplay between machine learning and system/architecture design.

CHAPTER 9

OTHER WORK FROM THIS AUTHOR

Over the course of my PhD journey, I have conducted research projects outside the scope of this thesis. Also, I had this unique opportunity to collaborate with leading scholars on many exciting projects. Below, I summarize these projects which are categorized into three main research topics: (1) approximate computing, (2) FPGA acceleration, and (3) heterogeneous computing.

9.1 Approximate Computing

Online and operand-aware detection of failures utilizing false alarm vectors. Collaborating with David Palframan[1], Azadeh Davoodi[1], Nam Sung Kim[11], and Mikko Lipasti[1], we present a framework [283] which detects online and at operand level of granularity all the vectors which excite already-diagnosed failures in combinational modules. Our framework is flexible with the ability to update vectors in the future. Moreover, the ability to detect failures at operand level of granularity can be useful to improve yield, for example by not discarding those chips containing failing and redundant computational units (e.g., two failing ALUs) as long as they are not failing at the same time. The main challenge in realization of such a framework is the ability for on-chip storage of all the (test) cubes which excite the set of diagnosed failures, e.g., all vectors that excite one or more slow paths or defective gates. The number of such test cubes can be enormous after applying various minimization techniques, thereby making it impossible for on-chip storage and

[1] University of Wisconsin-Madison

online detection. A major contribution of this work is to significantly minimize the number of stored test cubes by inserting only a few but carefully-selected false alarm vectors. As a result, a computational unit may be mis-diagnosed as failing for a given operand however we show such cases are rare while the chip can safely be continued to be used, i.e., our approach ensures that none of the true-positive failures are missed.

A compilation workflow for neural acceleration for general-purpose approximate programs. In this work, we introduce the very first neural processing unit compilation workflow, called NPiler [38, 6, 12][2], which automatically converts annotated regions of imperative code to a neural network representation. First, the programmer annotates the regions of imperative code which he/she wants to transform to a neural representation. NPiler accepts inputs from the programmer to train the network. During this step, NPiler automatically observes the input and output pairs to the annotated regions to collect training and testing data. Then, NPiler trains each possible NPU topology given constraints provide by the programmer. The outcome of this exploration provides the best possible NPU topology in terms of minimum root mean square error (RMSE) on test data. Finally, our compiler replaces the annotated regions with the final neural network representation. We use FANN library to execute the neural network representation. This work was done with the collaboration of Hadi Esmaeilzadeh[7], Adrian Sampson[3], Luis Ceze[3], and Doug Burger[4].

Rollback-Free Value Prediction. In this work [284, 65, 285], we tackle two fundamental memory bottlenecks: limited off-chip bandwidth (bandwidth wall) and long access latency (memory wall). To achieve this goal, our approach exploits the inherent error resilience of a wide range of applications. We introduce an approximation technique, called Rollback-Free Value Prediction (RFVP). When certain safe-to-approximate load operations miss in the cache, RFVP predicts the requested values. However, RFVP does not check for or

[2] `http://act-lab.org/artifacts/npiler/`
[3] University of Washington
[4] Microsoft Research

recover from load value mispredictions, hence, avoiding the high cost of pipeline flushes and re-executions. RFVP mitigates the memory wall by enabling the execution to continue without stalling for long-latency memory accesses. To mitigate the bandwidth wall, RFVP drops some fraction of load requests which miss in the cache after predicting their values. Dropping requests reduces memory bandwidth contention by removing them from the system. The drop rate is a knob to control the tradeoff between performance/energy efficiency and output quality. This work is done with the collaboration of Gennady Pekhimenko[5], Bradley Thwaites[8], Hadi Esmaeilzadeh [7], Onur Multu[6], and Todd C. Mowry[6].

An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration. In collaboration with Atieh Lotfi[7], Abbas Rahimi[7], Rajesh K. Gupta[7], and Hadi Esmaeilzadeh[7], we devised GRATER [286], an automated design workflow for FPGA accelerators, that leverages imprecise computation to improve data-level parallelism and computational throughput. By selectively reducing the precision of the data and operation, the required area to synthesize the kernels on the FPGA decreases allowing to integrate a larger number of operations and parallel kernels in the fixed area of the FPGA. The larger number of integrated kernels provides more hardware context to better exploit data- level parallelism in the target applications.

Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. Together with Divya Mahajan[8], Jongse Park[8], Bradley Thwaites[8], and Hadi Esmailezadeh[7], we introduce MITHRA [131], a co-designed hardware-software solution, that navigates these tradeoffs to deliver high performance and efficiency while lowering the final quality loss. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss and, if so, directs the processor to run the original precise code.

A multi-platform benchmark suite for approximate computing. As approximate com-

[5]University of Toronto
[6]Carnegie Mellon University
[7]University of California, San Diego
[8]Georgia Institute of Technology

puting gains popularity as a viable alternative technique to prolong the traditional scaling of performance and energy-efficiency improvements, it has become imperative to have a representative set of benchmarks for a fair evaluation of different approximation techniques. Collaborating with Divya Mahajan[8], Pejman Lotfi-Kamran[9], and Hadi Esmaeilzadeh[7], we introduce AXBENCH [126][10], a diverse and representative set of benchmarks for evaluating various approximation techniques in CPUs, GPUs, and hardware design. AXBENCH covers diverse application domains such as machine learning, robotics, arithmetic computation, multimedia, and signal processing. Moreover, AXBENCH comes with approximable region of benchmarks marked to facilitate evaluation of approximation techniques. Each benchmark is accompanied with three different sized input data sets (e.g., small, medium, and large) and an application-specific quality metric. AXBENCH enables researchers to study, evaluate, and compare a wider range of approximation techniques on a diverse set of benchmarks in a straightforward manner.

Hardware-software co-design for approximate code memoization. In collaboration with Zhenhong Liu[11], Dong Kai Wang[11], Hadi Esmaeilzadeh[7], and Nam Sung-Kim[11], we introduce AXMEMO an approximate memoization technique for general-purpose applications, exploiting the computational redundancy and fault tolerance of various general-purpose applications. Instead of focusing on expensive special arithmetic operations such as sin/cos and exp for traditional memoization, AXMEMO aims to replace long sequences of instructions with few lookup-table accesses to potentially eliminate a large number of dynamic instructions that would otherwise dominate execution time and energy consumption. AXMEMO takes advantage of synergies between memoization and approximation with simple hardware support.

[9] Institute for Research in Fundamental Sciences
[10] `http://axbench.org`
[11] University of Illinoise at Urbana-Champaign

9.2 FPGA Acceleration

A unified template-based framework for accelerating statistical machine learning. Collaborating with Divya Mahajan[8], Jongse Park[8], Emmanuel Amaro[8], Hardik Sharma[8], Joon Kim[8], and Hadi Esmaeilzadeh[7], we develop TABLA [190][12], a framework that generates accelerators for a class of machine learning algorithms. The key is to identify the commonalities across a wide range of machine learning algorithms and utilize this commonality to provide a high-level abstraction for programmers. TABLA leverages the insight that many learning algorithms can be expressed as stochastic optimization problems. Therefore, a learning task becomes solving an optimization problem using stochastic gradient descent that minimizes an objective function. The gradient solver is fixed while the objective function changes for different learning algorithms.

An end-to-end solution for FPGA acceleration of generative adversarial networks. Generative Adversarial Networks (GANs) are a frontier in deep learning. GANs consist of two models: generative and discriminative. While the discriminative model uses the conventional convolution, the generative model depends on a fundamentally different operator, called transposed convolution. This operator initially inserts a large number of zeros in its input and then slides a window over this expanded input. This zero-insertion step leads to a large number of ineffectual operations and creates distinct patterns of computation across the sliding windows. The ineffectual operations along with the variation in computation patterns lead to significant resource underutilization when using conventional convolution hardware. To alleviate these sources of inefficiency, we devise FlexiGAN [166], an end-to-end solution, that generates an optimized synthesizable FPGA accelerator from a high-level specification of generative adversarial networks. FlexiGAN is coupled with a novel template architecture that aims to harness the benefits of both MIMD and SIMD execution models to avoid ineffectual operations. This work was done with the collaboration of

[12] `http://act-lab.org/artifacts/tabla`

Michael Brzozowski[8], Behnam Khaleghi[7], Soroush Ghodrati[7], Kambiz Samadi[13], Nam Sung Kim[11], and Hadi Esmaeilzadeh[7].

9.3 Heterogeneous Computing

A heterogeneous split architecture for in-memory acceleration of learning. In collaboration with Hajar Falahati[9], Pejman Lotfi-Kamran[9], Michael Brzozowski[8], Fatemehsadat Mireshghallah[7], Hardik Sharma[8], and Hadi Esmaeilzadeh[7], we introduce ORIGAMI, a heterogeneous design for in-memory acceleration of the learning across a range of machine learning algorithms. ORIGAMI provides a unique opportunity to utilize off-the-shelf FPGA accelerators in coalescence with the in-memory acceleration. To deliver such capabilities, we devise a pattern matching technique to identify the similar patterns of computation across a set of machine learning algorithms. Given these patterns, ORIGAMI extracts heterogenous compute engines that offer a high-level of fine-grained parallelism for each of the patterns. These heterogenous compute engines constitute the accelerators that are integrated as in-memory units on the logic die of the 3D stacked memory. To utilize these accelerators along with the FPGA, ORIGAMI comes with a computation splitting compiler that divides the learning across the in-memory and out of memory FPGA. The combination of pattern matching and split execution offers a new design point for the acceleration of learning.

[13] Qualcomm Technologies, Inc.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *ISCA*, 2011.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Power Challenges May End the Multicore Era," *Commun. ACM*, vol. 56, no. 2, pp. 93–102, 2013.

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012.

[4] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward Dark Silicon in Servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[5] S. Y. Borkar and A. A. Chien, "The future of microprocessors," *CACM*, vol. 54, 5 May 2011.

[6] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," in *ISCA*, 2014.

[7] ——, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," 2015.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[9] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning Approximation for Graphics Engines," in *MICRO*, 2013.

[10] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-State Memories," in *MICRO*, 2013.

[11] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality-Programmable Vector Processors for Approximate Computing," in *MICRO*, 2013.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO*, 2012.

[13] ——, "Architecture Support for Disciplined Approximate Programming," in *ASPLOS*, 2012.

[14] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning," in *ASPLOS*, 2011.

[15] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *ISCA*, 2010.

[16] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (Embedded) SOC Architectures Based on Probabilistic CMOS (PCMOS) Technology," in *DATE*, 2006.

[17] C. Alvarez, J. Corbal, and M. Valero, *Fuzzy Memoization for Floating-Point Multimedia Applications*, 2005.

[18] R. Hegde and N. R. Shanbhag, "Energy-efficient Signal Processing via Algorithmic Noise-tolerance," in *ISLPED*, 1999.

[19] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable Stochastic Processors," in *DATE*, 2010.

[20] B. E. Boser, E. Säckinger, J. Bromley, Y. L. Cun, L. D. Jackel, and S. Member, "An Analog Neural Network Processor with Programmable Topology," *JSSC*, 1991.

[21] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale Integration of Analog Neural Networks," in *IJCNN*, 2008.

[22] S. M. Tam, B. Gupta, H. A. Castro, and M. Holler, "Learning on an Analog VLSI Neural Network Chip," in *SMC*, 1990.

[23] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware Spiking Neurons Design: Analog or Digital?" In *IJCNN*, 2012.

[24] F. Choudry, E. Fiesler, A. Choudry, and H. J. Caulfield, "A Weight Discretization Paradigm for Optical Neural Networks," in *ICOE*, 1990.

[25] C. Igel and M. Hüsken, "Improving the RPROP Learning Algorithm," in *NC*, 2000.

[26] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *PLDI*, 2011.

[27] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware," in *OOPSLA*, 2013.

[28] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design*. Oxford University Press, 2002.

[29] D. A. Johns and K. Martin, *Analog Integrated Circuit Design*. John Wiley and Sons, Inc., 1997.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *PDP*, 1986.

[31] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[33] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[34] S Galal and M Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, 2011.

[35] Y. Cao, *Predictive technology models*, 2013.

[36] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation," in *PLDI*, 2010.

[37] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of Service Profiling," in *ICSE*, 2010.

[38] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural Acceleration for GPU Throughput Processors," in *MICRO*, 2015.

[39] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[40] J. Gantz and D. Reinsel, *Extracting value from chaos*.

[41] *GeForce 400 series*, `http://en.wikipedia.org/wiki/GeForce_400_series`, 2015.

[42] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based Approximation for Data Parallel Applications," in *ASPLOS*, 2014.

[43] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ISCA*, 2014.

[44] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, 2013.

[45] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing Reliable Accuracy Into Neurally-Implemented Approximate Computing," in *HPCA*, 2015.

[46] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.

[47] L. McAfee and K. Olukotun, "EMEURO: A Framework for Generating Multi-Purpose Accelerators via Deep Learning," in *CGO*, 2015.

[48] B. Grigorian and G. Reinman, "Accelerating Divergent Applications on SIMD Architectures using Neural Networks," in *ICCD*, 2014.

[49] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "FlexJava: Language Support for Safe and Modular Approximate Programming," in *FSE*, 2015.

[50] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language Support for Approximate Hardware Design," in *DATE*, 2015.

[51] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, 1979, pp. 29–41.

[52] (). Whitepaper: NVIDIA Fermi.

[53] (). NVIDIA corporation. NVIDIA CUDA SDK code samples.

[54] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, 2009.

[55] jMonkeyEngine, 2015.

[56] O. A. Aguilar and J. C. Huegel, "Inverse kinematics solution for robotic manipulators using a cuda-based parallel genetic algorithm," *Advances in Artificial Intelligence*, 2011.

[57] M. Creel and M. Zubair, "A high performance implementation of likelihood estimators on gpus," in *CES*, 2013.

[58] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *FSE*, 2011.

[59] A Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *ISPASS*, 2009.

[60] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[61] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[62] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, O. Mutlu, C. Das, M. Kandemir, and T. C. Mowry, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Efficient Data Compression," in *ISCA*, 2015.

[63] A. Yazdanbakhsh, J. Sacks, S. Choungki, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. Sung-Kim, "Nax: Near-data approximate computing," 2016.

[64] A. Yazdanbakhsh, J. Sacks, C. Song, P. Lotfi-Kamran, N. S. Kim, and H. Esmaeilzadeh, "In-dram near-data approximate acceleration for gpus," in *PACT*, 2018.

[65] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-Free Value Prediction with Safe to Approximate Loads," in *TACO*, 2015.

[66] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, O. Mutlu, C. Das, M. Kandemir, and T. C. Mowry, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Efficient Data Compression," in *ISCA*, 2015.

[67] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," in *ISCA*, 2009.

[68] B. Casper, "Energy Efficient Multi-Gb/s I/O: Circuit and System Design Techniques," in *IEEE Workshop on Microelectronics and Electron Devices*, 2011.

[69] M. Horowitz, *Energy Table for 45nm Process*.

[70] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding," in *ICLR*, 2016.

[71] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Network," in *NIPS*, 2015.

[72] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.

[73] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, "Sonic Millip3De: A Massively Parallel 3D-stacked Accelerator for 3D Ultrasound," in *HPCA*, 2013.

[74] R. Hou, L. Zhang, M. C. Huang, K. Wang, H. Franke, Y. Ge, and X. Chang, "Efficient Data Streaming with On-chip Accelerators: Opportunities and Challenges," in *HPCA*, 2011.

[75] H. Asghari-Moghaddam, Y. Hoon Son, J. Ho Ahn, and N. Sung Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016.

[76] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 A 1.2V 8Gb 8-Channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods using 29nm Process and TSV," in *ISSCC*, 2014.

[77] (). Whitepaper: NVIDIA Maxwell.

[78] *Samsung HBM2 Memory*, `https://news.samsung.com/global/samsung-begins-mass-producing-worlds-fastest-dram-based-on-newest-high-bandwidth-memory-hbm-interface`.

[79] S. Liu, B. Leung, A. Neckar, S. O. Memik, G. Memik, and N. Hardavellas, "Hardware/Software Techniques for DRAM Thermal Management," in *HPCA*, 2011.

[80] K Man, "Bensley FB-DIMM Performance/Thermal Management," in *Intel Developer Forum*.

[81] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang, "Thermal Modeling and Management of DRAM Memory Systems," in *ISCA*, 2007.

[82] J. Iyer, C. L. Hall, J. Shi, and Y. Huang, "System Memory Power and Thermal Management in Platforms Build on Intel Centrino Duo Technology," *Intel Technology Journal*, vol. 10, no. 2, 2006.

[83] J. Lin, H. Zheng, Z. Zhu, E. Gorbatov, H. David, and Z. Zhang, "Software Thermal Management of DRAM Memory for Multicore Systems," *SIGMETRICS*, vol. 36, no. 1, pp. 337–348, 2008.

[84] J. Lin, H. Zheng, Z. Zhu, and Z. Zhang, "Thermal Modeling and Management of DRAM Systems," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2069–2082, 2013.

[85] K. Koo, S. Ok, Y. Kang, S. Kim, C. Song, H. Lee, H. Kim, Y. Kim, J. Lee, S. Oak, Y. Lee, J. Lee, J. Lee, H. Lee, J. Jang, J. Jung, B. Choi, Y. Kim, Y. Hur, Y. Kim, B. Chung, and Y. Kim, "A 1.2V 38nm 2.4Gb/s/pin 2Gb DDR4 SDRAM with Bank Group and x4 Half-Page Architecture," in *ISSCC*, 2012, pp. 40–41.

[86] T. Y. Oh, Y. S. Sohn, S. J. Bae, M. S. Park, J. H. Lim, Y. K. Cho, D. H. Kim, D. M. Kim, H. R. Kim, H. J. Kim, J. H. Kim, J. K. Kim, Y. S. Kim, B. C. Kim, S. H. Kwak, J. H. Lee, J. Y. Lee, C. H. Shin, Y. Yang, B. S. Cho, S. Y. Bang, H. J. Yang, Y. R. Choi, G. S. Moon, C. G. Park, S. W. Hwang, J. D. Lim, K. I. Park, J. S. Choi, and Y. H. Jun, "A 7 Gb/s/pin 1 Gbit GDDR5 SDRAM With 2.5 ns Bank to Bank Active Time and No Bank Group Restriction," *JSSC*, vol. 46, no. 1, pp. 107–118, 2011.

[87] T. Y. Oh, Y. S. Sohn, S. J. Bae, M. S. Park, J. H. Lim, Y. K. Cho, D. H. Kim, D. M. Kim, H. R. Kim, H. J. Kim, J. H. Kim, J. K. Kim, Y. S. Kim, B. C. Kim, S. H. Kwak, J. H. Lee, J. Y. Lee, C. H. Shin, Y. S. Yang, B. S. Cho, S. Y. Bang, H. J. Yang, Y. R. Choi, G. S. Moon, C. G. Park, S. W. Hwang, J. D. Lim, K. I. Park, J. S. Choi, and Y. H. Jun, "A 7Gb/s/pin GDDR5 SDRAM with 2.5ns Bank-to-Bank Active Time and no Bank-group Restriction," in *ISSCC*, 2010.

[88] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive GPU Cache Bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, 2015.

[89] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *MICRO*, 2014.

[90] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ACM SIGARCH Computer Architecture News*, vol. 42, 2014, pp. 743–758.

[91] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt, "GPUDet: A Deterministic GPU Architecture," in *ASPLOS*, 2013.

[92] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[93] A Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[94] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *HPCA*, 2013.

[95] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0., 2015.

[96] *NVIDIA Corporation. CUDA Programming Guide*, `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, 2015.

[97] K. Kim, *Apparatus for Pipe Latch Control Circuit in Synchronous Memory Device*, US6724684 B2, 2004.

[98] J. Macri, "AMD's Next Generation GPU and High Bandwidth Memory Architecture: FURY," in *HCS*, 2015.

[99] (). AMD Radeon Rx 300 Series.

[100] JEDEC, *High Bandwidth Memory DRAM*, `http://www.jedec.org/standards-documents/docs/jesd235`, October 2013.

[101] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[102] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0., 2015.

[103] T. Y. Oh, Y. S. Sohn, S. J. Bae, M. S. Park, J. H. Lim, Y. K. Cho, D. H. Kim, D. M. Kim, H. R. Kim, H. J. Kim, J. H. Kim, J. K. Kim, Y. S. Kim, B. C. Kim, S. H. Kwak, J. H. Lee, J. Y. Lee, C. H. Shin, Y. S. Yang, B. S. Cho, S. Y. Bang, H. J. Yang, Y. R. Choi, G. S. Moon, C. G. Park, S. W. Hwang, J. D. Lim, K. I. Park, J. S. Choi, and Y. H. Jun, "A 7Gb/s/pin GDDR5 SDRAM with 2.5ns Bank-to-Bank Active Time and no Bank-group Restriction," in *ISSCC'10*.

[104] K. Iniewski, *CMOS Processors and Memories*. Springer Science & Business Media, 2010.

[105] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems," in *ISCA*, 2013.

[106] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "AC-DIMM: Associative Computing with STT-MRAM," in *ISCA*, 2013.

[107] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, "Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore," in *MEMSYS*, 2015.

[108] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[109] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory using 3D-stacked DRAM," in *ISCA*, 2015.

[110] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[111] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[112] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.

[113] P. Boudier and G. Sellers, "Memory System on Fusion APUs," *AMD Fusion developer summit*, 2011.

[114] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System," in *USENIX*, 2012.

[115] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data Transfer Matters for GPU Computing," in *ICPADS*, 2013.

[116] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014.

[117] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," in *ISPASS*, 2010.

[118] R. Danilak, "System and Method for Hardware-based GPU Paging to System Memory," US7623134 B1, 2009.

[119] P. C. Tong, S. S. Yeoh, K. J. Kranzusch, G. D. Lorensen, K. L. Woo, A. K. Kaul, C. S. Case, S. A. Gottschalk, and D. K. Ma, "Dedicated Mechanism for Page Mapping in a GPU," US20080028181 A1, 2008.

[120] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy through Microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, no. 99, 2016.

[121] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.

[122] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[123] T. Beri, S. Bansal, and S. Kumar, "A Scheduling and Runtime Framework for a Cluster of Heterogeneous Machines with Multiple Accelerators," in *IPDPS*, 2015.

[124] M. Harris, *Inside Pascal: Nvidia's Newest Computing Platform*, `https://devblogs.nvidia.com/parallelforall/inside-pascal/`, 2016.

[125] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *HPCA*, 2013.

[126] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "AxBench: A Multi-Platform Benchmark Suite for Approximate Computing: Acceleration for GPU Throughput Processors," *IEEE Design and Test*, 2016.

[127] *NanGate FreePDK45 Open Cell Library*, `http://www.nangate.com`, 2015.

[128] Y. H. Son, O. Seongil, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.

[129] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM," in *HPCA*, 2016.

[130] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[131] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *ISCA*, 2016.

[132] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel, "The Bunker Cache for Spatio-Value Approximation," in *MICRO*, 2016.

[133] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *MICRO*, 2014.

[134] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing," in *ASPLOS*, 2016.

[135] D. Mahajan, K. Ramkrishnan, R. Jariwala, A. Yazdanbakhsh, J. Park, B. Thwaites, A. Nagendrakumar, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Abstractions for approximate hardware design and reuse," *IEEE MICRO Special issue on Alternative Computing Designs and Technologies*, 2015.

[136] M. Kamal, A. Ghasemazar, A. Afzali-Kusha, and M. Pedram, "Improving efficiency of extensible processors by using approximate custom instructions," in *DATE*, Dresden, Germany, 2014, ISBN: 978-3-9815370-2-4.

[137] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits," in *DATE*, 2014.

[138] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *ICCAD*, 2013.

[139] S. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan, "Relax-and-retime: A methodology for energy-efficient recovery based design," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, 2013, pp. 1–6.

[140] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.

[141] Y. Liu, R. Ye, F. Yuan, R. Kumar, and Q. Xu, "On logic synthesis for timing speculation," in *ICCAD*, 2012, pp. 591–596.

[142] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic Logic Synthesis of Approximate Circuits," in *DAC*, 2012.

[143] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.

[144] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic methodologies for ultra-efficient inexact architectures for sus-

taining technology scaling," in *Proceedings of the 9th Conference on Computing Frontiers*, 2012.

[145] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI*, 2011.

[146] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, IEEE, 2011, pp. 1–6.

[147] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, IEEE Press, 2011, pp. 409–414.

[148] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," in *DATE*, 2010.

[149] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE*, 2010.

[150] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, 2004.

[151] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. K. Gupte, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *ISCA*, 2018.

[152] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, 2015.

[153] 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 *MICRO*, 2016.

[154] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *CVPR*, 2015.

[155] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and¡0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[156] 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 *ISCA*, 2017.

[157] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *ISCA*, 2016.

[158] 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 *ASPLOS*, 2014.

[159] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, 2015.

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

[161] *DDR4 Spec - Micron Technology, Inc*, https://goo.gl/9Xo51F.

[162] S. Galal, "Energy efficient floating-point unit design," PhD thesis, The Department of Electrical Engineering of Stanford University, 2012.

[163] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *ICCAD*, 2011.

[164] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and¡0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[165] A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and N. S. Kim, "Ganax: A unified simd-mimd acceleration for generative adversarial network," in *ISCA*, 2018.

[166] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, H. Esmaeilzadeh, and N. S. Kim, "FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks," in *FCCM*, 2018.

[167] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *NIPS*, 2014.

[168] D. Nie, R. Trullo, J. Lian, C. Petitjean, S. Ruan, Q. Wang, and D. Shen, "Medical Image Synthesis with Context-aware Generative Adversarial Networks," in *MICCAI*, 2017.

[169] P. Costa, A. Galdran, M. I. Meyer, M. Niemeijer, M. Abràmoff, A. M. Mendonça, and A. Campilho, "End-to-end Adversarial Retinal Image Synthesis," *T-MI*, 2017.

[170] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning," in *NIPS*, 2016.

[171] A. Ghosh, B. Bhattacharya, and S. B. R. Chowdhury, "SAD-GAN: Synthetic Autonomous Driving using Generative Adversarial Networks," *arXiv*, 2016.

[172] W. R. Tan, C. S. Chan, H. Aguirre, and K. Tanaka, "ArtGAN: Artwork Synthesis with Conditional Categorial GANs," *arXiv*, 2017.

[173] H. Wu, S. Zheng, J. Zhang, and K. Huang, "GP-GAN: Towards Realistic High-Resolution Image Blending," *arXiv*, 2017.

[174] T. Kim, M. Cha, H. Kim, J. K. Lee, and J. Kim, "Learning to Discover Cross-Domain Relations with Generative Adversarial Networks," *ArXiv*, 2017.

[175] L.-C.Y.Y.-H. Y. Hao-Wen Dong Wen-Yi Hsiao, "MuseGAN: Symbolic-domain Music Generation and Accompaniment with Multi-track Sequential Generative Adversarial Networks," *arXiv*, 2017.

[176] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang, "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions," *arXiv*, 2017.

[177] J. Wu, C. Zhang, T. Xue, W. T. Freeman, and J. B. Tenenbaum, "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," in *NIPS*, 2016.

[178] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv*, 2015.

[179] Microsoft, *Microsoft unveils Project Brainwave for real-time AI*, `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`, 2017.

[180] 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 *ISCA*, 2017.

[181] 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," in *ISCA*, 2016.

[182] 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 *ASPLOS*, 2014.

[183] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in *ISCA*, 2018.

[184] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *JSSC*, 2017.

[185] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017.

[186] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in *HPCA*, 2017.

[187] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-based Accelerator for Deep Learning," in *HPCA*, 2017.

[188] 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 *MICRO*, 2016.

[189] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing," *ArXiv*, 2016.

[190] 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 *HPCA*, 2016.

[191] 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 *ISCA*, 2016.

[192] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A Systematic Approach to Blocking Convolutional Neural Networks," *ArXiv*, 2016.

[193] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding," in *ICLR*, 2016.

[194] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.

[195] 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 *ISCA*, 2015.

[196] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[197] S. Eldridge, A. Waterland, M. Seltzer, J. Appavoo, and A. Joshi, "Towards General-Purpose Neural Network Computing," in *PACT*, 2015.

[198] B. Grigorian and G. Reinman, "Accelerating Divergent Applications on SIMD Architectures Using Neural Networks," *TACO*, 2015.

[199] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous Real-World Inputs Can Open Up Alternative Accelerator Designs," in *ISCA*, 2013.

[200] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," in *CVPR Workshops*, 2011.

[201] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," in *ISCA*, 2017.

[202] K. Wang and C. Lin, "Decoupled Affine Computation for SIMT GPUs," in *ISCA*, 2017.

[203] T. Chen and G. E. Suh, "Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling," in *MICRO*, 2016.

[204] J. E. Smith, "Decoupled Access/Execute Computer Architectures," in *ACM SIGARCH Computer Architecture News*, 1982.

[205] M. Benhenda, "ChemGAN challenge for drug discovery: can AI reproduce natural chemical diversity?" *arXiv*, 2017.

[206] Y. Li, J. Song, and S. Ermon, "Inferring The Latent Structure of Human Decision-Making from Raw Visual Inputs," *ArXiv*, 2017.

[207] H. Che, B. Hu, B. Ding, and H. Wang, "Enabling Imagination: Generative Adversarial Network-Based Object Finding in Robotic Tasks," in *NIPS*, 2017.

[208] Y. Fang, H. Li, and X. Li, "A fault criticality evaluation framework of digital systems for error tolerant video applications," in *ATS*, 2011.

[209] V. Wong and M. Horowitz, "Soft error resilience of probabilistic inference applications," in *SELSE*, 2006.

[210] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *ASGI*, 2006.

[211] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.

[212] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*, 2009.

[213] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[214] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO*, 1994.

[215] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[216] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, S. Swanson, and M. Taylor, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO*, 2011.

[217] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," in *HPCA*, 2011.

[218] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[219] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *ISCAS*, 2006.

[220] J. Zhu and P. Sutton, "FPGA implementations of neural networks: A survey of a decade of progress," in *FPL*, 2003.

[221] K. Przytula and V. P. Kumar, Eds., *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1993.

[222] A. Joubert, B. Belhadj, O. Temam, and R. Heliot, "Hardware spiking neurons design: Analog or digital?" In *IJCNN*, 2012.

[223] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *IJCNN*, 2008.

[224] B. E. Boser, E. Säckinger, J. Bromley, Y. Lecun, L. D. Jackel, and S. Member, "An analog neural network processor with programmable topology," *J. Solid-State Circuits*, vol. 26, pp. 2017–2025, 1991.

[225] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic ISAs," in *ASPLOS*, 2011.

[226] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ISCA*, 2012.

[227] A. Hashmi, H. Berry, O. Temam, and M. H. Lipasti, "Automatic abstraction and fault tolerance in cortical microarchitectures," in *ISCA*, 2011.

[228] A. Frank and A. Asuncion, *UCI machine learning repository*, 2010.

[229] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators?" In *IISWC*, 2012.

[230] S. Draghici, "On the capabilities of neural networks using limited precision weights," *Elsevier NN*, 2002.

[231] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO*, 2013.

[232] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, "Patterns and Statistical Analysis for Understanding Reduced Resource Computing," in *Onward!*, 2010.

[233] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications," *Multimedia, IEEE Transactions on*, 2013.

[234] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *PLDI*, 2009, ISBN: 978-1-60558-392-1.

[235] X. Li and D. Yeung, "Exploiting application-level correctness for low-cost fault tolerance," *J. Instruction-Level Parallelism*, 2008.

[236] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[237] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of Approximate Sequential Circuits," in *DATE*, 2014.

[238] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.

[239] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing Parsimonious Inexact Circuits Through Probabilistic Design Techniques," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, 2013.

[240] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic Methodologies for Ultra-efficient Inexact Architectures for Sustaining Technology Scaling," in *CF*, 2012.

[241] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the Error Resilience of Machine-Learning Applications for Designing Highly Energy Efficient Accelerators," in *ASP-DAC*, 2014.

[242] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.

[243] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing," in *MICRO*, 2016.

[244] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernndez-Lobato, G. Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *ISCA*, 2016.

[245] 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 *ASPLOS*, 2015.

[246] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks," *arXiv*, 2017.

[247] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary Neural Networks for Resource-efficient AI Applications," in *IJCNN*, 2017.

[248] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," *arXiv preprint arXiv:1707.06168*, 2017.

[249] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "PredictiveNet: An Energy-efficient Convolutional Neural Network via Zero Prediction," in *ISCAS*, 2017.

[250] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically Accurate Program Transformations," in *SAS*, 2011.

[251] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," MIT, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[252] M. Rinard, "Probabilistic Accuracy Bounds for Fault-tolerant Computations that Discard Tasks," in *ICS*, 2006.

[253] M. C. Rinard, "Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points," in *OOPSLA*, 2007.

[254] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, M. PT Jr, S. HE Jr, and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE TC*, 1981.

[255] A. Nieto, D. L. Vilarino, and V. M. Brea, "PRECISION: A reconfigurable SIMD/MIMD coprocessor for Computer Vision Systems-on-Chip," *IEEE TC*, 2016.

[256] A. N. Choudhary, J. H. Patel, and N. Ahuja, "NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems," *IEEE TPDS*, 1993.

[257] H. P. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Computing*, 1988.

[258] P. P. Jonker, "An SIMD-MIMD architecture for Image Processing and Pattern Recognition," in *Computer Architectures for Machine Perception*, 1993.

[259] A. Nieto, D. L. Vilariño, and V. M. Brea, "SIMD/MIMD Dynamically-reconfigurable Architecture for High-performance Embedded Vision Systems," in *ASAP*, 2012.

[260] H. M. Waidyasooriya, Y. Takei, M. Hariyama, and M. Kameyama, "FPGA Implementation of Heterogeneous Multicore Platform with SIMD/MIMD Custom Accelerators," in *ISCAS*, 2012.

[261] X. Wang and S. G. Ziavras, "Performance-energy Tradeoffs for Matrix Multiplication on FPGA-based Mixed-mode Chip Multiprocessors," in *ISQED*, 2007.

[262] G. Kim, K. Lee, Y. Kim, S. Park, I. Hong, K. Bong, and H.-J. Yoo, "A 1.22 TOPS and 1.52 mW/MHz Augmented Reality Multicore Processor with Neural Network NoC for HMD Applications," *JSSC*, vol. 50, no. 1, 2015.

[263] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 2012.

[264] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, *et al.*, "The Architecture of the DIVA Processing-in-Memory Chip," in *Supercomputing*, 2002.

[265] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.

[266] M. Oskin, F. Chong, and T. Sherwood, "Active Pages: a Computation Model for Intelligent Memory," in *ISCA*, 1998.

[267] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *Micro, IEEE*, vol. 17, no. 2, 1997.

[268] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, "FBRAM: A New Form of Memory Optimized for 3D Graphics," in *SIGGRAPH*, 1994.

[269] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A Memory-SIMD Hybrid and its Application to DSP," in *Custom Integrated Circuits Conference*, vol. 30, 1992.

[270] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.

[271] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[272] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.

[273] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "DRAMA: An Architecture for Accelerated Processing Near Memory," *CAL*, vol. 14, no. 1, 2015.

[274] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[275] C. Shelor, K. Kavi, and A. S., "Dataflow based Near Data Processing using Coarse Grain Reconfigurable Logic," in *WoNDP*, 2015.

[276] R. Nair, S. Antao, C. Bertolli, P. Bose, J. Brunheroto, T. Chen, C. Cher, C. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, D. Gallo, L. Grinberg, J. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, J. O'Brien, M. Ohmacht, Y. Park, D. Prener, B. Rosenburg, K. Ryu, O. Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Z. Sura, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, 2015.

[277] D. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

[278] S. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads," *Micro, IEEE*, vol. 34, no. 4, 2014.

[279] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T.-M. Low, L. Pileggi, J. Hoe, and F. Franchetti, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.

[280] Q. Zhu, T. Graf, H. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.

[281] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, "Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore," in *MEMSYS*, 2015.

[282] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory using 3D-stacked DRAM," in *ISCA*, 2015.

[283] A. Yazdanbakhsh, D. Palframan, A. Davoodi, N. S. Kim, and M. Lipasti, "Online and Operand-Aware Detection of Failures Utilizing False Alarm Vectors," in *GLSVLSI*, 2015.

[284] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "Rfvp: Rollback-free value prediction with safe to approximate loads," in *High Performance Embedded Architectures and Compilers (HiPEAC)*, ACM, 2016.

[285] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free Value Prediction with Approximate Loads," in *TACO*, ACM, 2014, pp. 493–494.

[286] A. Lotfi, A. Rahimi, A. Yazdanbakhsh, H. Esmaeilzadeh, and R. K. Gupta, "GRATER: An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1279–1284.

VITA

Amir Yazdanbakhsh was born in Shiraz and raised in Bandar Abbas, Iran. He has received his Ph.D. from the School of Computer Science at Georgia Institute of Technology. Amir has a master's degree in Electrical and Computer Engineering from University of Wisconsin-Madison and a master's degree in Electrical and Computer Engineering from University of Tehran. Amir is interested in designing efficient specialized hardware for machine learning applications. He is also interested in exploring the interplay between machine learning techniques and efficient computing system design. He has published his work in multiple well-recognized pee-reviewed conferences and journals. His research has been recognized by multiple prestigious fellowships and awards including honorable mention in IEEE Micro Top Picks, Qualcomm Innovation Fellowship, and Microsoft Graduate Research Fellowship. Amir is also a gold medal winner of ACM Student Research Competition. Currently, Amir is an AI resident at Google.