UNIVERSITY OF CALIFORNIA SAN DIEGO

Approximate Computing for GPGPU Acceleration

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

in

Computer Science (Computer Engineering)

by

Daniel Peroni

Committee in charge:

Professor Tajana Šimunić Rosing, Chair
Professor Ryan Kastner
Professor Patrick Mercier
Professor Steve Swanson
Professor Jishen Zhao

2019

The dissertation of Daniel Peroni is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego
2019

DEDICATION

To my family, my friends, and my late Uncle Tom.
You don't know what you have until its gone.

EPIGRAPH

Men must fumble awhile with error to separate it from truth,
as long as they don't seize the error hungrily
because it has a pleasanter taste.

—Walter M. Miller Jr., *A Canticle for Leibowitz*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Tajana Šimunić Rosing for everything she has helped me accomplish throughout my PhD. She has always offered invaluable insight and provided exemplary mentorship. I am thankful she let me join her research team and guided me forward. I want to thank my PhD committee members, Prof. Ryan Kastner, Professor Patrick Mercier, Prof. Steve Swanson and Prof. Jishen Zhao for their valuable insight and contributions to my research.

I want to thank Prof. Ben Ochoa for offering his knowledge and vision. My time working in industry greatly influenced me and I am thankful for my internships with Intel and AMD. Special thanks to Iwen Chao at Intel and Martin Sarov and Pramod Argade at AMD.

Thank you to each and every one of my friends and labmates in SEELab. They are always willing to offer their feedback and hold valuable discussions. Most importantly, I want to thank my friend and colleague Mohsen Imani for teaching me and being a role model to aspire towards. I also want to thank Anthony Thomas, Michael Ostertag, Saransh Gupta, Justin Morris, Sahand Salamat, Joonseop Sim, Minxuan Zhou, Yeseong Kim, and Behnam Khaleghi for everything they have offered.

Thanks to all my friends in San Diego and beyond for always encouraging me. I especially want to thank my friend Ben Gladstone for always making me laugh when I needed it.

Thanks to my parents Ron and Melissa for teaching me the importance of learning and hard work. I want to thank my sister Chelsea for being always supporting me. Thanks to my Grandmother Victoria, my Aunt Mary Ann and my cousins Jessica and Jennifer. Thanks my grandparents Bill and Nina, and my Aunt Leslie and Uncle Tom and my cousin Taylor.

Finally, I thank Elynn for all the love and encouragement she showed me along the way. You helped me through many challenges and I will be forever grateful.

My work was supported by the National Science Foundation (NSF) grants 1527034, 1730158, and 152703. This work was also supported by CRISP, one of six centers in JUMP, an Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapters 1 and 3 contain material from ”ALook: Adaptive Lookup for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapters 1 and 2 contain material from ”Runtime Efficiency-Accuracy Trade-off Using Configurable Floating Point Multiplier”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018. The dissertation author was the primary instigator and author of this paper.

Chapter 3 contains material from ”Resistive CAM Acceleration for Tunable Approximate Computing”, by Mohsen Imani, Daniel Peroni, and Tajana Rosing, which appears in IEEE Transactions on Emerging Topics in Computing (TETC), 2016. The dissertation author was a primary instigator and the second author of this paper.

Chapters 1 and 5 contain material from ”CANNA: Neural Network Acceleration using Configurable Approximation on GPGPU”, by Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2018. The dissertation author was one of the primary instigators and a secondary author of this paper.

Chapters 1, 3, and 4 contain material from ”ARGA: Approximate Reuse for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which appears in IEEE Design Automation Conference (DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapters 1, 4, and 5 contain material from ”Data Reuse for Accelerated Approximate Warps”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which is currently being prepared for submission for publication. The dissertation author was the primary instigator and author of this paper.

Chapter 4 contains material from ”Warp Level Approximation for GPU Acceleration”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which was submitted for publication. The

dissertation author was the primary instigator and author of this paper.

VITA

2015	Bachelor of Science in Computer Engineering, California Polytechnic State University, San Luis Obispo
2015-2018	Graduate Student Researcher, University of California, San Diego
2018	M.S. in Computer Science (Computer Engineering), University of California, San Diego
2019	Ph.D. in Computer Science (Computer Engineering), University of California, San Diego

PUBLICATIONS

Daniel Peroni, Mohsen Imani, Hamid Nejatollahi, Nikil Dutt, and Tajana Rosing "ARGA: Approximate Reuse for GPGPU Acceleration, IEEE/ACM Design Automation Conference (DAC) 2019

Daniel Peroni, Mohsen Imani, and Tajana Rosing "ALook: Adaptive Lookup for GPGPU Acceleration," 24th Asia and South Pacific Design Automation Conference (ASP-DAC) 2019

Daniel Peroni, Mohsen Imani, and Tajana Rosing "Runtime Efficiency-Accuracy Trade-off Using Configurable Floating Point Multiplier, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 2018

Mohsen Imani, Daniel Peroni, and Tajana Rosing "Program Acceleration Using Nearest Distance Associative Search," IEEE International Symposium on Quality Electronic Design (ISQED) 2018

Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang and Tajana Rosing "CANNA: Neural network acceleration using configurable approximation on GPGPU," 23rd Asia and South Pacific Design Automation Conference (ASP-DAC) 2018

Mohsen Imani, Daniel Peroni, and Tajana Rosing, Resistive CAM Acceleration for Tunable Approximate Computing, Government Microcircuit Applications Critical Technology Conference (GOMACTech), 2018

Mohsen Imani, Daniel Peroni, and Tajana Rosing "NVALT: Approximate Lookup Table for GPU Acceleration, IEEE Embedded System Letter (ESL) 2017

Mohsen Imani, Daniel Peroni, and Tajana Rosing "CFPU: Configurable Floating Point Multiplier for Energy-Efficient Computing, IEEE/ACM Design Automation Conference (DAC) 2017

Mohsen Imani, Daniel Peroni, and Tajana Rosing "Non-volatile Content Addressable Memory for Computing Acceleration Non-Volatile Memory Workshop (NVMW) 2017

Mohsen Imani, Daniel Peroni, Yeseong Kim, Abbas Rahimi, and Tajana Rosing "Efficient Neural Network Acceleration on GPGPU using Content Addressable Memory," IEEE/ACM Design Automation and Test in Europe Conference (DATE) 2017

Mohsen Imani, Daniel Peroni, Abbas Rahimi, and Tajana Rosing Resistive CAM Acceleration for Tunable Approximate Computing IEEE Transactions on Emerging Topics in Computing (TETC) 2016

Mohsen Imani, Daniel Peroni, Abbas Rahimi, and Tajana Rosing Resistive CAM Acceleration for Tunable Approximate Computing in IEEE International Conference on Computer Design (ICCD) 2016

ABSTRACT OF THE DISSERTATION

Approximate Computing for GPGPU Acceleration

by

Daniel Peroni

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Tajana Šimunić Rosing, Chair

Faster and more efficient hardware is needed to handle the rapid growth of big data processing. Applications such as multimedia, computer vision, and machine learning can be parallelized and accelerated using General-Purpose Computing on Graphics Processing Units (GPGPUs). GPUs are power intensive, so novel approaches are needed to improve their efficiency. Many applications that run on GPUs are tolerant to error. Approximate computing is a design strategy in which improvements in performance and energy savings can be achieved at the expense of accuracy. This thesis proposes a number of methods to enable approximate computing GPUs.

We first examine a number of approaches for approximating operations at the instruction

level. Floating point arithmetic, specifically multiplies, make up the majority of instructions computed on GPUs. In this dissertation we propose a configurable floating point unit (CFPU) which eliminates the costly manitassa multiply by copying one of the input mantissa directly to the output. For applications with a higher amount of temporal similiarity we propose adaptive lookup (ALook) to use small dynamic tables to store recently computed operations. This low power look up table provides nearest distance match for obtaining results rather than computing on the exact hardware, enabling significant energy savings.

GPUs issue threads to cores in groups called warps. Cores in a warp run the same instructions in lock-step. Every instruction within a warp must be accelerated to provide performance improvement. To ensure sufficient accuracy, some instructions run on the exact hardware. Bottlenecks can arise as some threads in a warp spend time computing exact results while others use approximate solutions. We propose AWARP to handle this problem. First, we use warp pass through to target warps in which a very small fraction of threads must be computed exactly. To handle warps with a larger percentage of exact computations, we utilize warp value trading (WVT). Under WVT, operations are traded between warps prior to running on the same multiprocessor to create uniform groups of either exact or approximate operations, providing significant speedup.

Finally, we focus on application specific approximation. We show approximation can be used to accelerate neural networks during training and inference using DRAAW. Early stages of training tolerate more error than later ones, so we adjust the level of approximation per epoch. To accelerate inference, we approximate larger operations less than smaller ones to increase ALOOK hit rate. DRAAW increases speedup of neural network training by 3.2× and EDP by 4.8× with less than 1% decrease in classification accuracy. For inference we automatically predict error control parameters from user accuracy requirements. DRAAW improves inference speedup by 2.9× speedup and EDP by 6.2× of inference with less than 1% decrease in prediction accuracy.

Chapter 1

Introduction

A little over a decade ago computing was revolutionized by the introduction of GPGPU (General-Purpose Computing on Graphics Processing Units) support by Nvidia. The ability to parallelize applications across thousands of cores opened avenues of research previously unattainable outside of the realm of super computing. GPGPU computing allows advances in applications such as computational mechanics, design automation, image processing, computer vision, medial analysis, language recognition and possibly most importantly the rapid exploration and deployment of machine learning. With the rise of big data, these applications have grown to utilize the available computing power while demanding more cores and faster computation. However, with Dennard scaling [6] long past and Moore's law [7] slowing, improving performance is no longer trivial.

New computing paradigms and novel architectures are needed to further advance computing. One such paradigm is approximate computing, in which application accuracy is traded for speedup and energy savings. Many applications do not need highly accurate computation, so accepting slight inaccuracy, instead of doing all computation precisely, results in significant energy and performance improvements [8–15]. Researchers have proposed different strategies for approximate computing, but these approach fail to address several key problems in order to work on GPUs. First, they target integer arithmetic instead of floating point. Floating point units (FPU) are the cornerstone of GPU processing. New strategies must be employed to optimize

for them. Second, prior approximate hardware is tied to the FPU pipeline and does not provide acceleration. They can achieve energy savings by clock gating operations, but otherwise offer no speedup. Finally, they are not designed around a major aspect of GPU architecture, warps, also known as wave fronts. Instructions on GPUs are issued in groups of threads which remain in lockstep. In most applications, approximating every operation and providing low error is not feasible, so some portion of the operations must be run on exact hardware. In warps, this can lead to a single thread bottlenecking the entire warp which 10s of threads, and remove any speedup approximation may provide.

In this dissertation, we propose an approximate computing architecture for GPUs which allows high energy savings, computation speedup, and fine grained error control. To improve performance of applications with high levels of temporal redundancy, we develop novel computational reuse methods to provide energy savings and speedup computation. We target floating arithmetic, which make up the bulk of many GPU workloads, and design an approximate FPU to simplify multiply and multiply add operations by removing the energy intensive mantissa computation. GPUs issue instructions in group of threads known as warps which must remain in lockstep. To control accuracy, we split computation between exact and approximate hardware. Our design is capable of accelerating applications at a warp level by avoiding bottlenecks and preemptively reordering operations. We propose a software framework for utilizing our approximate hardware to accelerate machine learning tasks. Neural networks can tolerate a significant amount of well selected approximation while still providing sufficient accuracy. Our approximation provides 4.8× EDP improvement and 3.2× speedup over the unmodified GPU for training neural networks with only a 1% decrease in prediction error. Our work automatically select error control parameters based on user prediction requirements for neural networks inference quality and improves speedup by 2.9× and EDP by 6.2× with only a 1% drop in classification accuracy.

1.1 Approximate Computing

Approximate computing is one method to improve performance and reduce energy consumption at the expense of output accuracy. There are many applications where the accuracy is less important than efficiency including multimedia, data analysis, and machine learning [16–18]. Media applications benefit from limitations in human senses, allowing audio and visual renderings to have small amounts of error without being noticeable to a viewer. Approximating portions of media can potentially result in major energy savings on phones or other mobile devices, while GPU approximation can increase framerate in video games. In data sensing, many sensors, such as those found in embedded devices, have error tolerances up to or exceeding 10% [19–22]. Processing the sensor data with extreme precision does not improve output accuracy compared to faster and more energy efficient methods. Devices such as these weather sensors can be placed in areas with limited sunlight and expected to function on batteries for long periods at a time, so it is critical to avoid unnecessary energy waste.

While approximate computing at first appears to be a promising method for improving performance it should be noted that it is not a catchall optimization. First, not all applications can make use of approximation. Cryptography and other highly exact applications are not viable candidates for approximation [23]. Even in approximate applications, not all code regions can safely be approximated. Errors may lead to critical failures such as segmentation faults, invalid jump addresses, or memory out of bounds [24]. These concerns require careful planning and solutions to ensure systems are getting the most out of approximate computing.

1.2 Floating Point Arithmetic

Floating point multiplication is a major component of many GPU workloads. In floating point notation, a number consists of three parts: a sign bit, an exponent, and a fractional value. In *IEEE 754* floating point representation, the sign bit is the most significant bit, bits 31 to 24 hold the exponent value, and the remaining bits contain the fractional value, also known as the

mantissa. The exponent bits represent a power of two ranging from -127 to 128. The mantissa bits store a value between 1 and 2, which is multiplied by 2^{exp} to give the decimal value.

Floating point multiply follows the steps shown in Figure 1.1. First, the sign bit of $A \times B = C$ is calculated by XORing the sign bit of the A and B operands. Second, the effective value of the exponential terms are added together. Finally, the two mantissa values are multiplied to provide the result's mantissa. Because the mantissa ranges from 1 to 2, the output of the multiplication always falls between 1 and 4. If the output mantissa is greater than 2, it is normalized by dividing by 2 and increasing the exponent by 1. The multiplication of the mantissas is the most costly operation, taking over 80% of the total energy of the multiply operation [25].

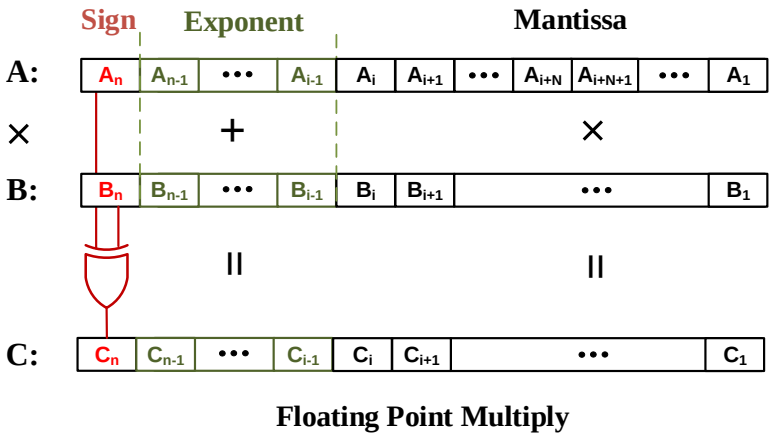


Figure 1.1. Floating Point Multiplication

Multiplication is one of the most common and costly floating point operations, slowing down the computation in many applications such as signal processing, neural networks, and stream processing [26–30]. Approximate arithmetic can improve application efficiency at the computational level. Utilizing the existing structure of floating point arithmetic units can allow for fast efficient approximation, while still allowing exact computation when needed.

1.3 GPU Architecture

GPUs use hundreds to thousands of cores to process data in parallel. Certain tasks, such as vector or matrix operations, can readily be mapped across many cores to compute orders of magnitude faster than CPUs [31]. Figure 1.4 diagrams an Nvidia Titan GPU based on the Kepler GK110 architecture [32]. The GPU has 14 streaming multiprocessors (SMs), each with 192 single precision and 64 double precision floating point units. Cores within SMs have access to shared resources such as memory. Each SM contains a register file and shared L1 cache between the cores. The GPU shares the L2 cache between all streaming multiprocessors. Threads are grouped together and issued as warps across cores with the same SM. Each SM has has four warp schedulers which allows multiple warps to be issued simultaneously. Warps are dispatched to cores which contain both floating point and integer ALU units. Once cores are assigned instructions to the dispatch port, they collect operands for the instruction needed to execute. GPUs cores are optimized perform graphics operations such as vertex and geometry shading, rasterization, and tessellation. These operations require computation of floating point operations such as add, multiply, multiply-accumulate, and square-root [33]. GPGPU applications use these FPU units to accelerate their performance, but often FP32 and FP64 precision provided is unnecessary and requires too much power. Approximate computing offers flexible trading of accuracy for energy savings.

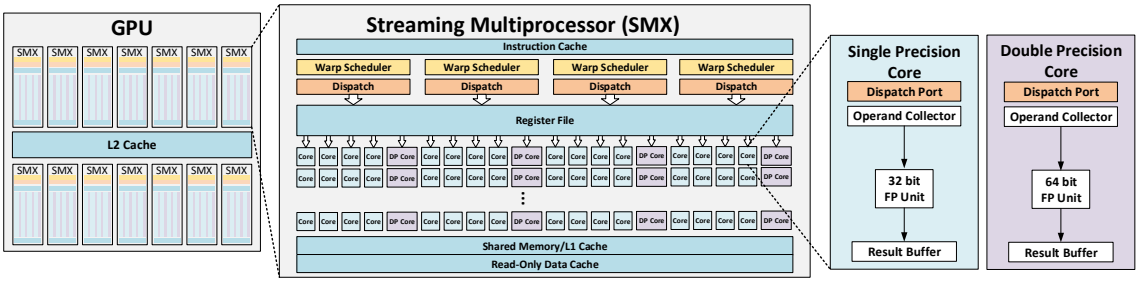


Figure 1.2. Overview Nvidia Kepler GPU Architecture

Figure 1.3 shows the architecture of a traditional floating point multiply accumulate

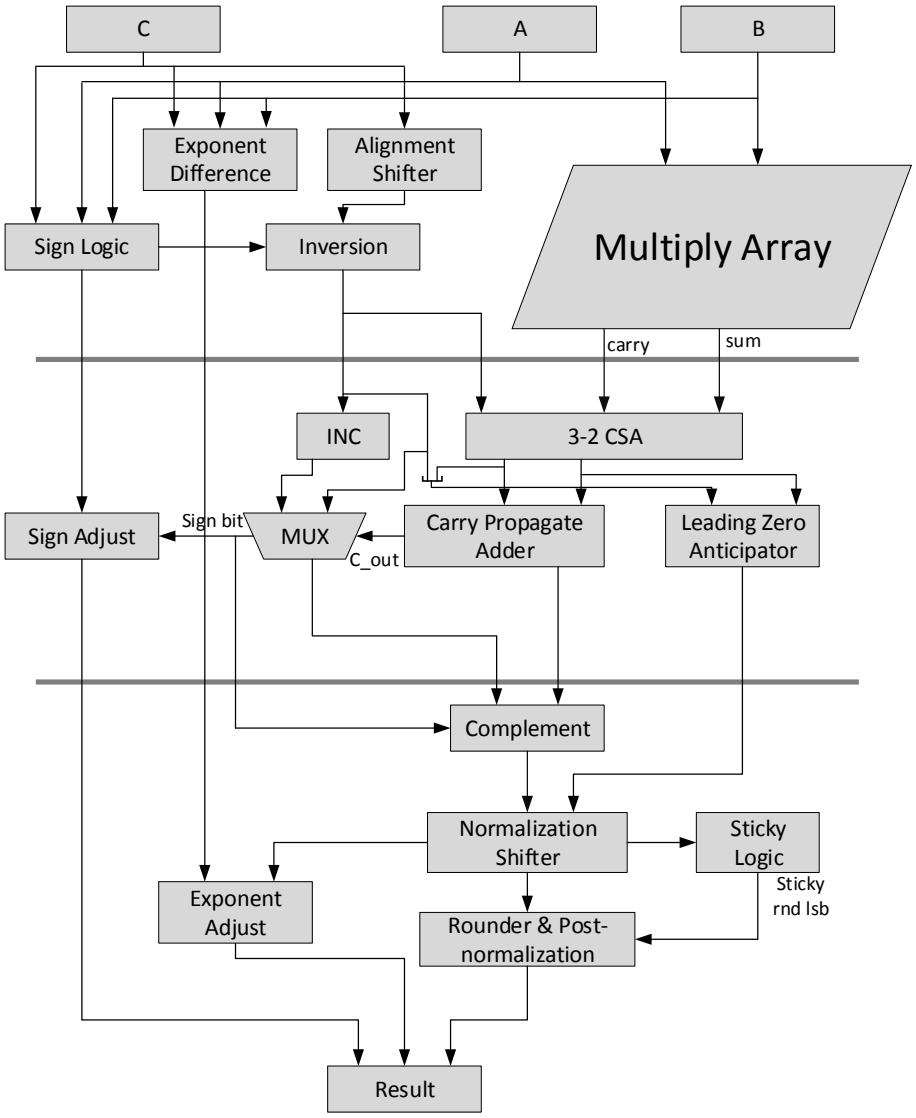


Figure 1.3. Overview floating point multiply accumulate unit [1]

unit [1]. Over 80% of the FPU power is dedicated to the multiplication of the mantissa value [25]. Floating point multiply requires 4× more energy than floating point addition [34] which makes it a good candidate for optimization. We focus on techniques which simplify multiplication or bypass it altogether in order to save power.

As shown in Figure 1.4, in GPUs, instruction threads run in groups, traditionally of 32, called warps or wavefronts. The threads within a warp are issued to an identical number of cores with shared memory. Each core is assigned the same instruction and all operations must

complete before processing the next instruction. The threads are tied in lockstep computation and all must be accelerated or none can be.

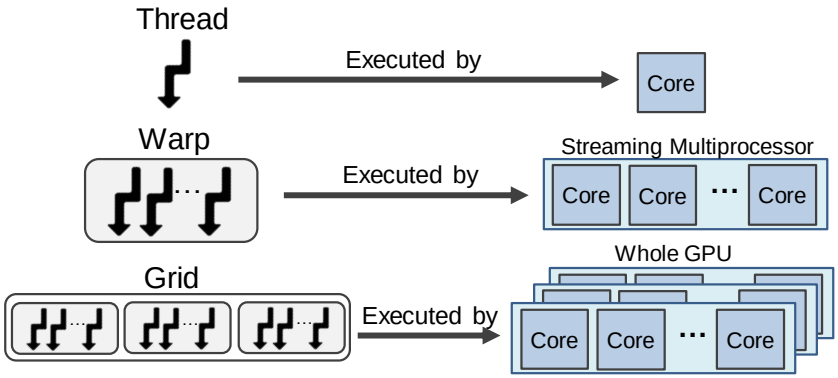


Figure 1.4. Overview of how instructions are issued on GPUs

This creates a problem for approximate computing where cores running in exact mode may prevent the entire warp from accelerating that instruction. A single thread will bottleneck the other 31. In many workloads we find a substantial number of warps are throttled by this issue. This must be addressed in order to maximize the benefits of approximate computing for GPUs.

1.4 Machine Learning

Machine learning is another area where approximation can provide significant benefits to energy and performance. Many machine learning applications are stochastic in nature and exhibit inherent error. They are error-tolerant and can be improved by simplifying some computations without having a significant impact on the final results. As machine learning algorithms continue to gain traction, approximate computing promises to optimize their computation without sacrificing output accuracy.

A pillar of modern machine learning is the neural network (NN). Loosely inspired by the neural structure of the biological brains, NNs utilize groups of neurons to solve complex non-linear problems. These tasks include image classification tasks, language translation, market predictions and more [13, 35, 36]. NNs exploit learned knowledge to deal with data which they

have not previously encountered. Although NNs can outperform many other machine learning models, they require enormous resources to be executed. The development of ResNets [37] opened new possibilities for networks dozens or hundreds of layers deep and training times grew with the increase in network size. The combination of speed and programming flexibility makes GPUs strong candidates for neural network prototyping and training. Many applications require NNs to be executed on embedded devices. NN applications need to update their model at run-time in order to adapt to the environment or enable a personalization. For instance, in speech recognition, NNs personalize as a function of the user's context or accent [38]. Energy saving optimizations must be made to hardware, such as GPUs, on embedded systems to allow training and testing despite limited power budgets. These same energy saving techniques can be applied to larger discrete GPUs.

Most current computing systems deliver only exact solutions at high energy cost, while neural networks do not require exact answers, due to their stochastic nature [14, 39, 40]. Slight inaccuracy due to enabled HW approximation in neural networks often results in little to no quality loss. Neural networks tolerate significant noise to their computation before prediction accuracy degrades significantly. However, once accuracy begins to decrease, it falls sharply. Inputs to neurons are summed together so approximation error in larger values impacts the output more drastically. Large portions of neural networks can accelerated with little change in prediction accuracy by approximating larger inputs less than smaller inputs.

1.5 Dissertation Contributions

This dissertation enhances a GPU architecture with the ability to approximate many applications. We explore a variety of GPGPU workloads to identify data and computation trends which can be exploited for approximation. We present an approximate floating point unit which uses computational reuse and approximate arithmetic techniques to accelerate GPU warps. We also evaluate the the impact and propose enhancements for neural network training and inference

on GPUs. Figure 1.5 shows the design to support approximate computing on a GPU. This dissertation will cover the following contributions:

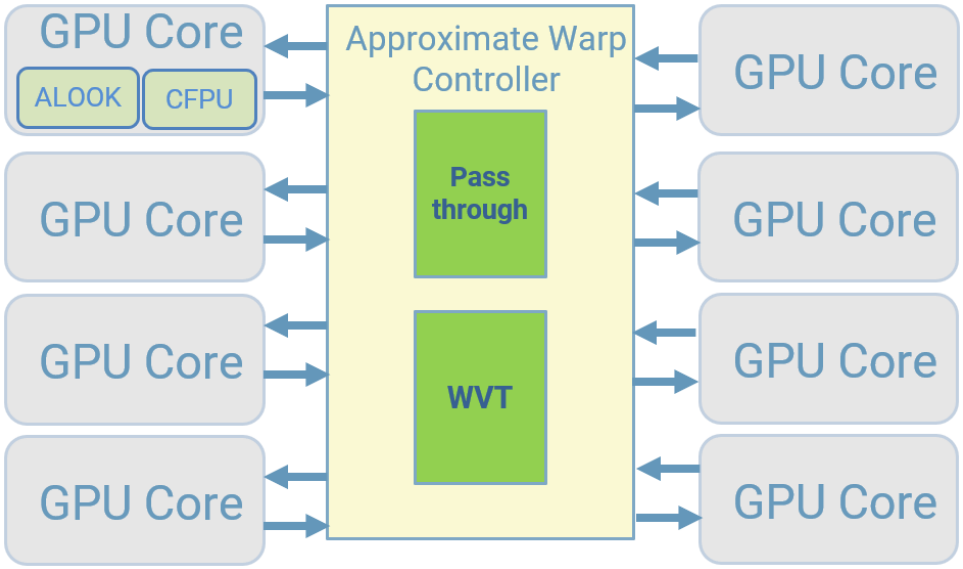


Figure 1.5. Overview of approximate hardware on GPU

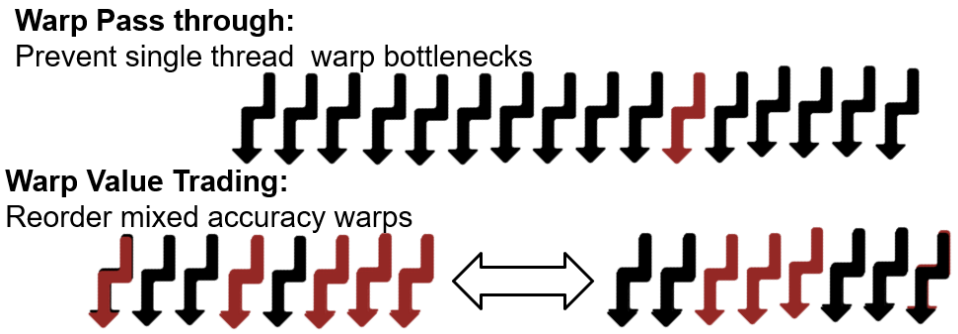


Figure 1.6. Warp acceleration overview

- We propose a tiered approximate floating point multiplier, called CFPU, which significantly reduces energy consumption and improves the performance of multiplication at a slight cost in accuracy. The floating point multiplication is approximated by replacing the costly mantissa multiplication step of the operation with lower energy alternatives. We process the data by using one of the three modes: a basic approximate mode, an intermediate approximate mode, or on the exact hardware, depending on the accuracy requirements.

CFPU avoids mantissa multiplication in one of two ways. First CFPU finds and discards the mantissa which results in the lowest error, then uses the other directly as the output. When the mantissa discarding cannot produce results with the error below a user-specified requirement, we run the operation in a more accurate approximate mode. We shift the saved mantissa based on the discarded one and add it saved value to the create the new mantissa for the result. If neither of these approaches produces an output with an acceptable error, our design can control the level of output accuracy by identifying the inputs that result in the highest output error and assigning them to compute precisely. Our results show that CFPU can offer 4.1× EDP improvement, compared to an unmodified FPU, for less than 10% error. In addition, our results show that the proposed CFPU can achieve 2.8× EDP improvement for multiply operations as compared to state-of-the-art approximate multipliers. CFPU is detailed in Chapter 2.

- We develop an adaptive look-up based approximate computing approach, called ALOOK, which uses a dynamic update policy to maintain a set of recently used operations in associative memory. ALOOK updates with values computed by floating point units at runtime to adapt to the workload and matches the stored results to avoid recomputing similar operations. To accelerate applications, ALOOK duplicates the first stage of the floating point unit (FPU) pipeline enabling processing and computational reuse of two inputs simultaneously. Our evaluation shows that ALOOK provides 3.6× EDP (Energy Delay Product) and 2.7× performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. ALOOK does not account for warps and many become bottlenecked by exact computation. The design and implementation of ALOOK is presented in Chapter 3.

- We present AWARP, an approximate computing technique capable of accelerating GPGPU warps. A single lookup miss may negate any possible acceleration of the accompanying threads by bottlenecking the operation. We examine warps as a whole to prevent a small

number of threads from bottlenecking speedup. Figure 1.6 shows our two techniques to avoid warp bottlenecks. *Warp passthrough* allows groups of threads with a large majority of hits to be accelerated with a minor penalty to output accuracy. To handed warps with a more even mix of operations, we apply *warp value trading* to ensure approximate operations are preemptively reordered into uniform approximate warps which can then be sped accelerated. We show our design improves performance throughput by up to 32.8% and improves EDP by 5.3× compared to the unmodified GPU while maintaining less than 5% output error. Details of AWARP are presented in Chapter 4.

- We propose DRAAW, a method for accelerating neural networks in both training and inference phases. DRAAW is able to approximate over 90% of operations within neural networks by using our proposed hardware. For training, we propose a Gradual Training Approximation (GTA) which significantly accelerates neural network computation, while providing a desirable quality of service. GTA starts training from deep approximation, and gradually reduces the level of approximation as a function of NN internal error, until the accuracy is sufficient. For inference, we provided a methodology to automatically select approximation controls for neural networks. We profile the layers of each neural network to identify output distributions which can then be used to maximize the operations approximated with minimal impact on prediction accuracy. We use this to predict safe approximation values for neural networks which can automatically be set at run time based on user selected maximum prediction accuracy loss. Our experimental evaluation shows that GTA achieves up to 4.84× energy savings and 3.22× speedup when running four different neural network applications with 1% quality loss as compared to baseline GPU. Our work accelerates inference of tested neural networks by 2.1× and improves energy savings by 4.6× for a 1% decrease prediction accuracy. Our method for accelerating neural networks is described in Chapter 5.

Chapter 1 contains material from ”ALook: Adaptive Lookup for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapter 1 contains material from ”Runtime Efficiency-Accuracy Trade-off Using Configurable Floating Point Multiplier”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018. The dissertation author was the primary instigator and author of this paper.

Chapter 1 contains material from ”CANNA: Neural Network Acceleration using Configurable Approximation on GPGPU”, by Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2018. The dissertation author was one of the primary instigators and a secondary author of this paper.

Chapter 1 contains material from ”ARGA: Approximate Reuse for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which appears in IEEE Design Automation Conference (DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapters 1 contains material from ”Data Reuse for Accelerated Approximate Warps”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which currently is being prepared for submission for publication. The dissertation author was the primary instigator and author of this paper.

Chapter 2

Approximate Floating Point Arithmetic

2.1 Introduction

Running machine learning algorithms or multimedia applications on general purpose processors, e.g. GPUs, requires high energy consumption. Many applications do not need highly accurate computation, so accepting slight inaccuracy, instead of doing all computation precisely, results in significant energy and performance improvements [4, 5, 8–15, 41]. While there are a number of proposed approximate solutions, they are limited to a small range of applications because they cannot control the error rate of their output.

Several data processing applications use a large range of values and require high precision. Therefore, computations in many traditional and state-of-the-art computing systems use floating point units (FPUs) [13, 36, 42, 43]. For example, on GPUs, frame rendering or high-performance scientific computations require many FPU operations and use a large amount of power. We observed over 85% of floating point arithmetic involved multiplication in the general OpenCL applications we tested. To cover the same dynamic range as with floating point, the fixed point unit must be five times larger and 40% slower than a corresponding floating point [44, 45].

Multiplication is one of the most common and costly FP operations, slowing down the computation in many applications such as signal processing, neural networks, and stream processing [26–30]. There are a number of approximate multiplication units designed to save power through different techniques. Several prior publications truncate the operands of the

multiplication or use different sized blocks to enable approximate multiplication [4, 5, 46]. However, a lack of accuracy controls and large area overhead reduce the advantages provided by these approximate designs.

In this chapter, we propose a configurable floating point multiplication, called CFPU, which significantly reduces the floating point multiplication energy consumption by trading off accuracy. CFPU avoids the costly multiplication when calculating the fractional part of a floating point number in one of two ways:

1) *Mantissa Discarding* - In floating point multiplies, the bottom 23 bits represent the mantissa. To calculate the result of a multiply operation, the mantissa from each operand are multiplied together, a step which consumes the majority of the total power and bottlenecks the operation. A substantially faster and lower energy approach is to discard one of the input mantissa and use the second one directly. Using one mantissa directly has the potential to generate high error rates for individual multiplies, so we provide two modifications that increase the final output accuracy.

- *Adaptive operand selection* finds and discards the mantissa which results in the lowest error. Minimizing individual operation error allows us to increase the number of calculations performed on the CFPU while maintaining the same output error.

- *Tuning* examines the first N bits of the discarded mantissa to predict error in the output result.

2) *Shift and Add* - When the mantissa discarding cannot produce results with the error below a user-specified requirement, we run the operation in a more accurate approximate mode. We examine the discarded mantissa to find the location of the first '1' bit. We shift the non-discarded mantissa based on this bit's position and add it to itself to create the new mantissa for the result value. This approach requires more energy and computation time compared to the first stage but is more accurate. An operation run in the second stage has at least 50% lower error than one run in the first stage for the same input operands. If neither of these approaches produces an output with an acceptable error, our design can control the level of output accuracy by identifying the

inputs that result in the highest output error and assigning them to compute precisely on the CFPU.

We evaluate the efficiency of the proposed technique on AMD Southern Island GPU architecture by replacing the traditional FPUs with the proposed CFPU. We test OpenCL workloads and our results show that using first stage CFPU approximation results in 3.5× energy-delay product (EDP) improvement, compared to an unmodified GPU, while ensuring less than 10% average relative error. Adding the second stage further increases the EDP improvement to 4.1× for the same level of accuracy. Comparing the proposed CFPU with previous state-of-the-art multipliers [5, 39, 46, 47] shows that our design can achieve 2.8× higher energy-delay product with lower error.

We also examine the impact of CFPU on several machine learning algorithms. With these algorithms becoming increasingly popular, there is a strong need to improve their performance without sacrificing output error. They are often naturally stochastic, allowing them to accept an error in their output. We use our design to run three Rodinia [48] machine learning benchmarks: *K-Nearest Neighbor (KNN), Back Propagation* and *K-means*. *K-means* and *K-nearest neighbor* are used in data mining applications and involve dense linear algebra computation, while *Back Propagation* is used for training weights in neural networks. For machine learning algorithms CFPU design can achieve 2.4× energy saving and 2.0× speedup compared to an unmodified GPU while ensuring less than 1% average relative error. These benchmarks have 50% energy savings and 40% speedup when running on CFPU with two stages rather than the previously proposed one stage design [3].

2.2 Related Work

There are several commonly examined approaches to approximate computing: voltage over scaling (VOS), use of approximate hardware blocks, and use of approximate memory units. VOS involves dynamically reducing the voltage supplied to a hardware component to save energy,

but at the expense of accuracy. Error rates for VOS can be modeled to determine the trade-off between energy and accuracy for applications, allowing voltage to be lowered until an error threshold is reached [43, 49–53]. The circuit is sensitive to any variations, and if the operating voltage of a circuit is decreased too far, timing errors begin to appear which are too large to correct. A configurable associative memory was proposed by [9] which can relax computation by using VOS on the TCAM rows/bitlines to trade between energy and output accuracy. These techniques suffer because they are bound by GPU pipeline stages and therefore cannot improve computation performance. Because they make use of Hamming distance, a metric which does not consider bit position's impact on error, output accuracy is difficult to predict.

Another recently emerged strategy is the application of non-volatile memories (NVM) to create approximate memory units, for energy efficient storage and computing purposes [54–57]. In computing, the goal of this approach is to store common inputs and their corresponding outputs. This style of associative memory can retrieve the closest output for given inputs in order to reduce power consumption [58, 59]. This approach does not work well in applications without a large number of redundant calculations. Associative memory can be integrated into FPUs to reduce these redundancies. Resistive CAMs have been used in order to accelerate application level computation as shown in [60]. This work uses the bit position insensitive metric of hamming distance resulting in poor accuracy in GPU usage.

Finally, approximate hardware involves redesigning basic component blocks to save energy, at the cost of accurate output [5, 39, 40, 61]. Liu *et al.* utilize approximate adders to create an energy efficient approximate multiplier [39]. Hashemi *et al.* designed a multiplier that selects a reduced number of bits used in the multiplication to conserve power [5]. Speculative designs are a recently explored route. Work in [62] proposed a speculative adder with error recognition to perform approximate operations. These types of adders can be utilized in approximate multipliers. Camus *et al.* propose a speculative approximate multiplier combines gate-level pruning and inexact speculative adders to lower power consumption and shrink FPU area [61]. Deep neural networks can tolerate approximate hardware, as shown in [63], which examines the use of

variable fixed point in deep neural networks.

Compared to the previous work [5, 46, 47], we focus on optimizing floating point multiplication by eliminating mantissa multiplication. Our design computes common power of 2 multiplies exactly. Our configurable approximate floating point multiplier predicts accuracy based on the incoming inputs and runs high error operations on exact hardware rather than correcting results after computation. We extend our previous design [3] to offer two levels, instead of just one, of approximation to a tradeoff between approximation error and energy benefits. The first stage approximately multiplies two values by using the mantissa from one operand directly in the output. If the first stage error is too high, in the second stage the kept mantissa is shifted based on the position of the first '1' bit of the discarded mantissa and added to itself to produce an approximate result. If the second stage error is also too high, operations can be computed on exact hardware. For the same level of application accuracy, our modified multiplier reduces application energy by up to 45% compared with our work in [3] and shows a 2.8× EDP improvement for multiply operations when compared to state of the art approximate multipliers [5, 46, 47].

2.3 Approximate FPU Multiplier

Compared to integer computing units, FPUs are usually costly and energy-hungry components, due to the complex way floating point numbers are stored. Multiplication based components are inefficient and slow down many current applications including multimedia, streaming, neural networks, and other machine learning applications [5, 8]. GPU applications show a high prevalence of multiply and multiply-add (muladd) operations compared to additions. Horowitz et al [34] estimates in 45nm a floating point multiply consumes 3.7pJ of energy compared to a floating point add which consumes 0.9pJ. A floating point multiply consumes only 20% more energy than an integer multiply, so performing all operations as fixed point does provide significant energy savings. Based on these power estimates, the floating point multiply

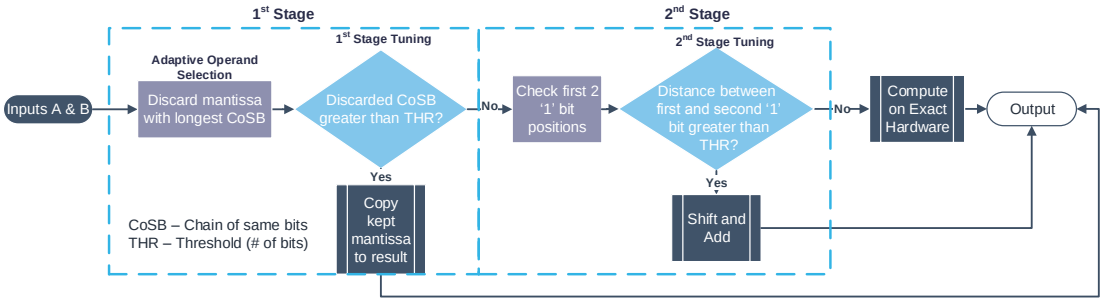


Figure 2.1. CFPU operating flow including 1st and 2nd stage approximation

and muladd operations consume over 90% of the ALU energy for the Sobel application, making these operations good targets for energy optimization based on these values.

In order to make multiplication more efficient, we propose a two-stage floating point multiplier. The first stage optimizes mantissa multiplication by reusing one of the input mantissa directly in the output. The second stage seeks to reduce error further by shifting and adding the retained mantissa to itself.

2.3.1 IEEE 754 Floating Point Multiply

In floating point notation, a number consists of three parts: a sign bit, an exponent, and a fractional value. In *IEEE 754* floating point representation, the sign bit is the most significant bit, bits 31 to 24 hold the exponent value, and the remaining bits contain the fractional value, also known as the mantissa. The exponent bits represent a power of two ranging from -127 to 128. The mantissa bits store a value between 1 and 2, which is multiplied by 2^{exp} to give the decimal value.

FPU multiply follows the steps shown in Figure 2.2. First, the sign bit of $A \times B = C$ is calculated by `XOR`ing the sign bit of the A and B operands. Second, the effective value of the exponential terms are added together. Finally, the two mantissa values are multiplied to provide the result's mantissa. Because the mantissa ranges from 1 to 2, the output of the multiplication always falls between 1 and 4. If the output mantissa is greater than 2, it is normalized by dividing by 2 and increasing the exponent by 1.

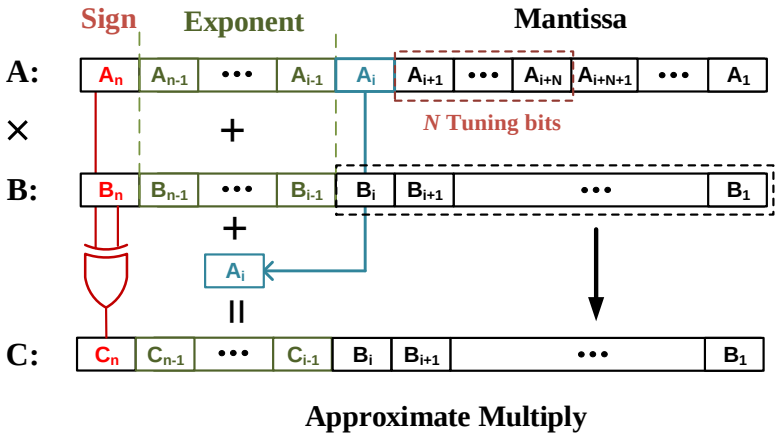


Figure 2.2. Floating Point Multiplication

2.3.2 CFPU Usage

CFPU is highly transparent to the application. A user running an error-tolerant application specifies the maximum error for any given multiply operation, $Error_{max}$, prior to execution. CFPU uses this value to ensure operations producing error greater than the value are sent to either a more accurate approximation mode or the exact hardware.

The flow chart for the proposed design is shown in Figure 2.1. Adaptive selection checks for a mantissa which, when discarded, produces an exact output when possible. The selector controls a multiplexer (MUX) between the two inputs *A* and *B*, and copies the selected mantissa to the output while discarding the other. Tuning utilizes the first N bits from the discarded mantissa to check against a threshold value and, if the threshold is not exceeded, the computation is complete. If the threshold is exceeded, the operation is run in the second stage which uses shift and add to increase accuracy. If the error of the second stage still exceeds the specified $Error_{max}$, the output is computed on the exact FPU hardware. We further explain the modifications in the following sections.

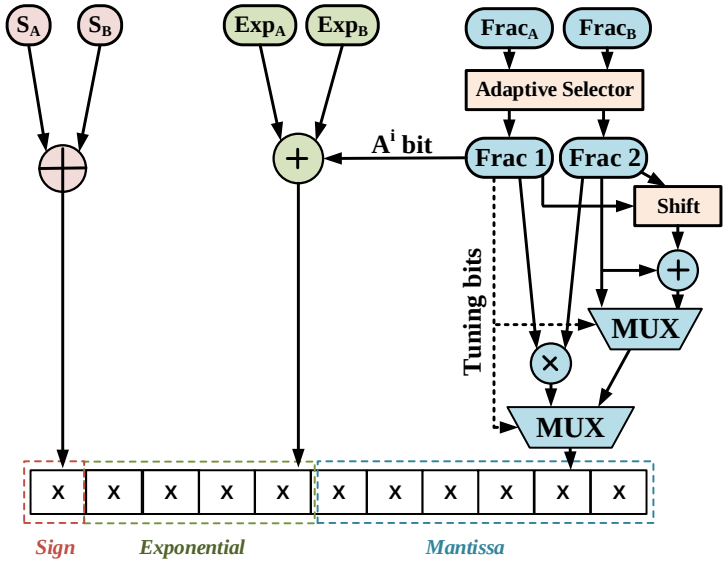


Figure 2.3. CFPU Integration with adaptive selector and N tuning bits

2.3.3 First stage Approximation

Mantissa Discarding

The multiplication of the mantissas is the most costly operation, taking over 80% of the total energy of the multiply operation [25], so the first stage approximation removes it entirely. Rather than multiplying the two mantissas, the unmodified mantissa from one of the input operands (e.g. input B) is used for the output value.

The error of any approximate multiply is $Mantissa_{discarded} - 1$. In the case where an operand is a power of 2, the mantissa is 1 and there is no error in the result.

$$Error = \sum_{i=0}^{n-1} 2^{-((n-i)A_{n-i-1})}. \tag{2.1a}$$

$$MaxError = \sum_{i=0}^{n-1} 2^{-(n-i)} = 0.999..9. \tag{2.1b}$$

Because the largest value a mantissa can be multiplied by is 2, the deviation from the kept mantissa and the correct answer is at most 100%. However, the maximum error can be reduced down to 50% by adding the first bit of the discarded mantissa to the sum of the exponent values.

When the discarded mantissa is greater than 1.5 (the first mantissa bit is 1), the error is less than 50% if the kept mantissa is multiplied by 2 instead of 1. This is the functional equivalent of increasing the exponent by 1. By doing this, the error range is shifted to be -50% to 50% instead of 0% to 100% as shown in Eq 2.

$$MaxError = abs(\sum_{i=0}^{n-1} 2^{-((n-i)A_{n-i-1})} - 0.5). \tag{2.2}$$

The additional logic needed to perform approximate floating point multiplication is shown in Figure 2.2. The *B* mantissa is used directly as the output mantissa, and the first bit of the discarded mantissa *A* is added with the two exponent values.

Figure 2.3 shows the flow for approximation. The sign bit for the result is computed by XORing the two input sign bits. The exponent for the result is computed by adding the two input exponents and the MSB of the discarded mantissa. CFPU ensures all operations compute their results below the user specified $Error_{max}$ through the use of adaptive selection and tuning. In the first level of approximation, Adaptive selection identifies the best mantissa to use in the output and discards the other. The upper N bits of the discarded mantissa are used to predict error and select between copying the other mantissa directly to the output, using the second level of approximation, or multiplying the two mantissas together.

Adaptive Operand Selection

Choosing which mantissa should be used for the result can significantly impact the accuracy of the CFPU output. For example, if the values 2.0 and 3.0 are multiplied, the result will be either 6.0 (exact) or 8.0 (33% error) depending on which mantissa is discarded.

In [3], we only use adaptive operand selection to identify mantissa values of zero. We improve adaptive operand selection to find the best mantissa to discard for all operations. This approach increases hit rate as more operations can be approximated, and reduces error for individual operations.

We compare the two mantissa values to determine the value which produces the lowest error when discarded. The output result uses the best mantissa. The benefits of this approach are twofold: 1) error for individual operations decreases and 2) the percentage of operations run in the first stage approximate mode increases because more operations are below the $Error_{max}$. The worst case error occurs when the discarded mantissa is closest to 1.5, so the preferred mantissa can be identified by detecting the longest continuous series of identical bits starting from the MSB. The mantissa with the longest series of either '1's or '0's gives a lower error when discarded. If all mantissa bits are '0', the error is 0. If both mantissas have an identical length series of bits, mantissa A is discarded.

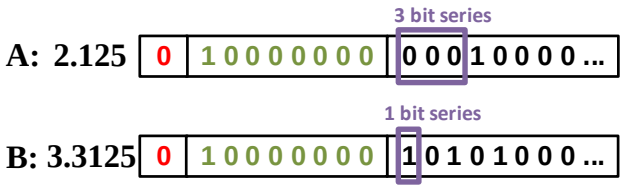


Figure 2.4. Comparison of first mismatch position. Using adaptive selection mantissa A is discarded as it results in lower error. Exact answer: 7.039, Approx discarding A mantissa: 6.625, Approx result discarding B mantissa: 8.5

Figure 2.4 illustrates the distance calculation. If the mantissa from A is discarded, the resulting approximate multiply produces a value of 6.625 with an error of 5.9%. By comparison, if the mantissa from B is discarded instead, the output value is 8.5 with an error of 20.7%. Mantissa A has a series of 3 '0's compared to the single '1' of B, so discarding A results in a lower error.

Tuning Control

It is possible that neither mantissa produces output lower than $Error_{max}$ when discarded. CFPU automatically detects cases where the error exceeds the specified requirement and computes the result in the more accurate second stage. For example, if the maximum desired error is 5%, then the multiplication of A and B in Figure 2.4 cannot be computed using the first stage

mantissa drop approximation.

The N bits after the MSB of the discarded mantissa are checked to tune the level of approximation. The case of maximum error occurs when the discarded mantissa is furthest from a power of 2, which occurs when its value is '1' followed by all '0's. When tuning, the goal is to ensure values over $Error_{max}$ are selected to run in the second stage, so the hardware must change its selection depending on the value of the first bit of the discarded mantissa. If the first bit of the discarded mantissa is '1', the first N tuning bits are checked for '0s', where N is selected based on $Error_{max}$.

$$N = log_2(\frac{1}{Error_{Max}}) - 1. \tag{2.3}$$

If a '0' is found, the hardware runs in exact mode. Similarly, when the first bit is '0', the first N tuning bits are checked for '1's instead. For each guaranteed bit in the A_{i-1} to 1^{st} indexes, the maximum error is reduced by half. Checking only one bit corresponds to a maximum error of 25%, two bits is 12.5%, etc.

$$MaxError = \left| \sum_{i=N}^{n-1} 2^{-((n-i)A_{n-i-1})} - 0.5 \right|. \tag{2.4}$$

An example of CFPU multiplication is shown in Figure 2.5 for two 32-bit floating point numbers in precise FPU and proposed CFPU with $Error_{max}$ set to 12.5%. An $Error_{max}$ of 12.5% requires CFPU to check N=2 tuning bits. The conventional FPU finds the correct solution of -510 by adding the exponents and multiplying the two mantissa, while XORing the sign bit to find three parts of the output data. Our design first compares the mantissas, which both contain a series of 3 identical bits, so operand A is selected for discarding. Next CFPU checks the first

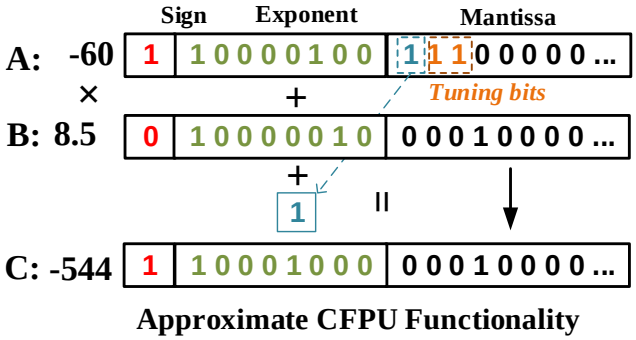


Figure 2.5. An example of 32-bit multiplication running on proposed CFPU first level approximation using N=2 tuning bits.

mantissa bit and the N tuning bits after that. In this case, the first mantissa bit is '1', so the next two bits are checked for '0' to determine if the value is below the desired error rate. When two tuning bits are checked, the maximum error is 12.5%. In this example, both tuning bits are '1', so the calculation continues in approximate mode and the mantissa from the value 8.5 is copied to the output value. The resulting output is -544, which deviates 6.67% from the correct value of -510 and falls below the desired $Error_{max}$.

2.3.4 Second stage Shift and Add

If the first stage produces a result which has the error greater than $Error_{max}$, CFPU activates the second stage instead. It has greater accuracy than the first stage but higher energy cost. The energy draw of the second stage is less than that of the exact hardware.

Some applications when running with only a single level of approximation, they require a high percentage of multiplies to be run in exact mode. FFT, MersenneTwister, and DwtHaar1D require almost 50% of their operations to be run on the CFPU exact mode to maintain output error below 10% as shown in Table 5.5. Running this high a percentage of operation in exact mode limits potential energy savings, so the addition of the second level of approximation to CFPU allows more of the multiply operations to be approximated and while maintaining acceptable output error.

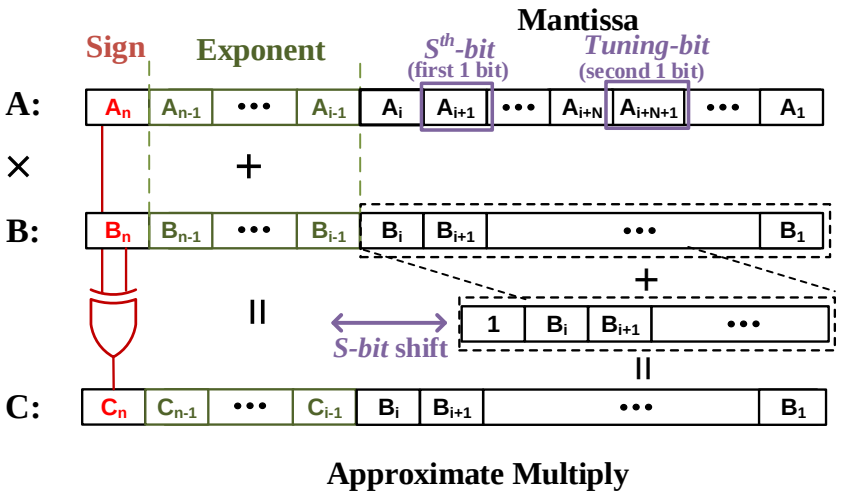


Figure 2.6. Second stage shift and add in CFPU

Our 2nd stage approximation shifts and adds the kept mantissa with itself to produce a value closer to the exact output. Our hardware detects the positions of the first two '1' bits in the discarded mantissa. Figure 2.3 shows the overall flow of CFPU. The first level uses the fractional bits directly, Frac 2, while the second level uses the shifted value of Frac 2 summed with the original value of Frac 2. Frac 1 provides tuning bits to predict if the error for the first level is too large. If it is, then the second level error is predicted. If this predicted error is also too great, the two mantissa values are multiplied together to produce an exact result.

Figure 2.6 demonstrates the shift and add design. The first '1' bit position, P1, is detected by hardware and determines S, the shift amount, where S is the number of mantissa bits minus the bit position. The mantissa used in the output is shifted S positions right and added its unshifted value. The second '1' bit is used for tuning and helps determine the maximum error between the calculated result and the approximate result. The maximum error decreases the further the first '1' bit is from the MSB because of the shift and added mantissa decreases relative to the original mantissa. The closer the second '1' bit, P2, is relative to the first, the higher the error in the result. If the two '1' bits are adjacent, the shift and add value will be up to 50% smaller than necessary to reach an exact result.

$$MaxError = \sum_{i=0}^{P1} 2^{-(B_{P1-i-1})}. \tag{2.5}$$

The maximum error for the second level is based on the P1 within the discarded mantissa. The further P1 is from the MSB, the lower the maximum error will be.

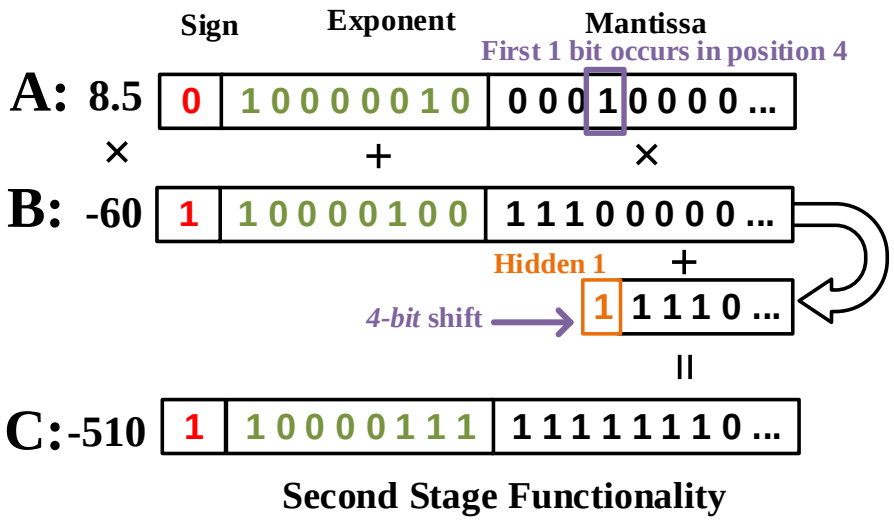


Figure 2.7. Example of shift and add. Produces exact result, while first stage does not.

Figure 2.7 provides an example of the computation which the first stage design computed with 6.67% error as shown in Figure 2.5. In this example, P1 is in the fourth MSB, so the shift value is 4. Each mantissa contains a hidden '1' bit left of the MSB which must be accounted for and shifted in. The mantissa from B is shifted left by 4 bits to create the value '0001111...' and added the unshifted value '1110000...' to produce the approximate output. In this example, the second stage design calculates the output exactly. The error decreases from 6.67% calculated by the first stage to 0% from the second stage while consuming less energy than the exact multiply operation.

In cases where there is only one '1' bit in the mantissa, the shift and add approach produces exact results. In this case, the discarded mantissa is effectively a power of 2 and as

such resolves to an integer value by which to shift the kept mantissa.

Running on Exact Hardware

The output error produced by the second stage approximation can still exceed $Error_{max}$, so some of the computations must be run on the exact hardware. Similar to the mantissa discarding approach, we use a threshold value to determine which values are run on the exact hardware. We search the discarded mantissa for the second occurrence of a '1' bit, $P2$. The position of $P2$ relative to the first bit, and $P1$ relative to the MSB determines the threshold value, V. The threshold is calculated as $V = S + (P1 - P2)$, where S is the shift amount. The minimum threshold occurs when $P1 = 22$ and $P2 = 21$, the upper MSBs, giving $S = 1$ and $V = 2$. $V = 2$ corresponds to a maximum output error of 25% on an individual operation. As V increases, the maximum output error becomes $50\% \times 2^{-V}$.

$$V > N = log_2(\frac{1}{Error_{Max}} - 1). \tag{2.6}$$

V must be greater than N, the number of tuning bits, in order to guarantee the result with produce an error below $Error_{max}$ If the output error still exceeds $Error_{max}$, the operation runs on exact FPU hardware instead.

The shift and add approach is more complex and power intensive when compared to the basic mantissa discard approach. The additional logical overhead and extra power draw make it viable as an intermediate option to reduce output error without requiring the full power consumption of the exact hardware. Shift and add is more accurate than mantissa drop, but if a user requires even greater output accuracy, operands producing highly erroneous results are sent to the exact hardware. The user can configure output accuracy by adjusting $Error_{max}$ value for both stages.

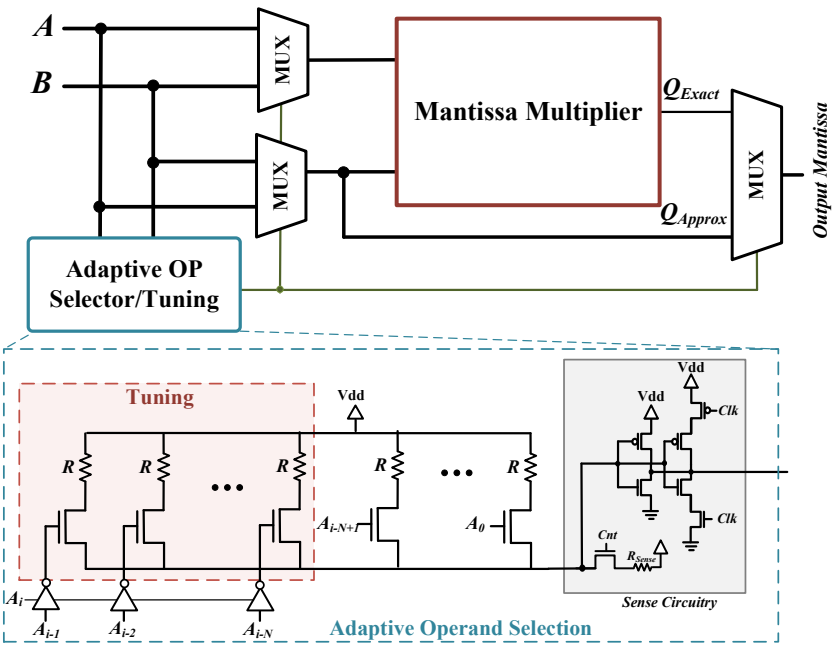


Figure 2.8. Circuitry to support adaptive operand selector and tuning the level of approximation in CFPU.

2.4 CFPU Hardware Support

Figure 2.8 shows the circuitry to enable CFPU adaptive operand selection and accurate tuning. We implement adaptive operand selection by checking the mantissa bits in both input operands. Our design compares the mantissas to determine which produces the lowest output error. If one mantissa has bits which are all zero, the second mantissa is copied to the output to produce an exact result. To ensure the mantissa which produces the greatest error is discarded, the hardware must locate and discard the mantissa furthest from 1.5. The further the discarded mantissa is from 1.5, the lower the output error. The circuit examines the two input mantissas and detects the one with the longest chain of continuous 1s or 0s starting from the first bit. A chain of zeros represents a mantissa close to 1, and similarly, a chain of ones is closer to a mantissa of 2. The mantissa with the longest consecutive chain of either zeros or ones is copied to the output and the other discarded.

As Figure 2.8 shows, the detector circuitry is a simple transistor-resistor circuitry which

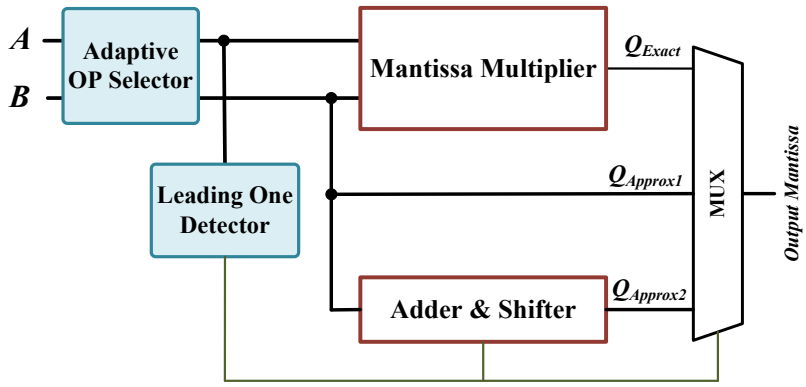


Figure 2.9. Circuitry to support CFPU with two level approximation.

samples the match-line (*ML*) voltage to detect the $A_{i-1}, A_{i-2}, ..., A_0$ input operand. In case of any '1' bit in a mantissa, the sense amplifier detects changes in the *ML* voltage (*ML*=1). However, if all mantissa bits are zero, no current passes through R_{sense} and the *B* operand mantissa is selected as the output mantissa. To detect the '1' bit on A^{th}_{i-1}, ..., A^{th}_0 indices on CFPU, the sense amplifier *Clk* needs to be set to 250ps. Based on the results, we can dynamically change the sampling time to balance the ratio of the running input workload on the approximate CFPU core. The operand selection happens by using two multiplexers which are controlled with our detector hardware signal. Similarly, to tune the level of approximation, our design uses *N* bits (after the first mantissa bit) of the selected mantissa to decide when to perform mantissa multiplication or approximate it. The number of tuning bits sets the level of approximation, with each additional bit reducing the maximum error by half. The goal is to check the value of the $A_{i-1}, ..., A_{i-N}$ to make sure they are same as the A_i. For this purpose, the circuitry selects the original value or inverted values of the tuning bits for the circuitry to search. To make the design area efficient, we use the same circuitry for adaptive operand selection and tuning approximation. For each application, sampling time can be individually set in order to provide target accuracy.

CFPU with two level approximation requires similar hardware to perform adaptive operand section. The two leading '1' bits in selected input operand are detected and their position used to calculate the maximum error. If the estimated error is less than $Error_{max}$, shift and add

is used, otherwise the result is computed on exact hardware. Figure 2.9 shows the proposed hardware supporting two level approximating. The adaptive operand selector identifies the best mantissa for use in the result and the tuning bits of the discarded mantissa are examined. If the error is low, the best mantissa is copied directly to the output. Otherwise, a leading '1' but detector identifies the bit positions of the discarded mantissa to predict the 2nd stage error. If the 2nd stage error meets the accuracy requirement, a shift block and an adder block compute the result mantissa. If the error of the 2nd stage is too great, the standard mantissa multiplier hardware computes the result.

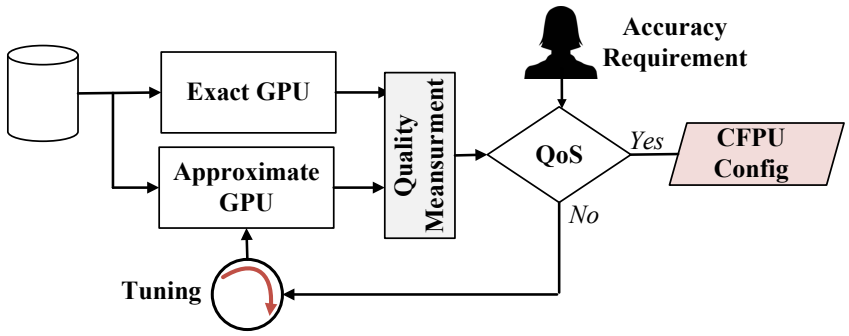


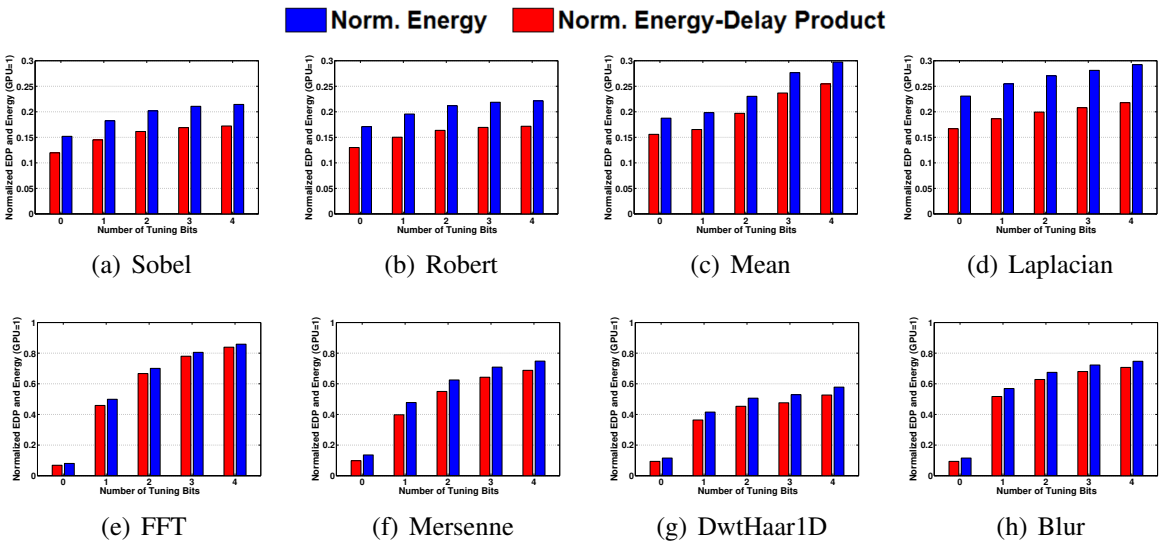
Figure 2.10. Framework to support tunable CFPU approximation.

Table 2.1. Energy and performance improvement and average relative error replacing GPU with proposed floating point multiplications.

Applications	Sobel	Robert	Mean	Lapla-cian	Sharpen	Prewit	QuasiR	Blacksc-holes	Average Improv.
Energy savings	84%	83%	81%	76%	75%	69%	72%	53%	**72%**
Speed up	21%	24%	27%	16%	20%	22%	12%	10%	**19%**
EDP improvement	8.3×	7.7×	6.4×	4.7×	5.3×	4.2×	4.1×	2.7×	**5.4**×
Error (%)	9.02%	1.19%	1.36%	1.96%	0%	0%	0%	6.79%	**2.54%**

2.4.1 Experimental Setup

We integrated the proposed approximate CFPU in the floating point units of an AMD Southern Island Radeon HD 7970 GPU. We modified Multi2sim, a cycle accurate CPU-GPU simulator [64], to model the CFPU functionality in three main floating point operations in GPU architecture: multiplier, multiplier-accumulator (MAC) and multiply-add (MAD). We evaluated



(a) Sobel (b) Robert (c) Mean (d) Laplacian

(e) FFT (f) Mersenne (g) DwtHaar1D (h) Blur

Figure 2.11. Normalized energy consumption of enhanced GPU with a tunable single stage checking N tuning bits for application run.

energy of traditional FPUs using Synopsys Design Compiler and optimized for power using Synopsys Prime Time for 1 ns delay in 45-nm ASIC flow [65]. The circuit level simulation of the CFPU design, such as the adaptive operand selector, has been performed using HSPICE simulator in 45-nm TSMC technology. We first test the efficiency of enhanced GPU on twelve general Open*CL* applications from AMD OpenCL SDK [66]: *Sobel, Robert, Mean, Laplacian, Sharpen, Prewit, QuasiRandom, FFT, Mersenne, DwHaar1D, Blur* and *Blackscholes*. In these applications, roughly 85% of the floating point operations involve multiplication.

Our design is then tested using machine learning applications. Machine learning algorithms are often error-tolerant allowing them to be run more efficiently on approximate hardware. We examine three OpenCL benchmarks from the Rodinia 3.1 machine learning suite [48]. These benchmarks are *K-Nearest Neighbor(KNN), Back Propagation* and *K-means*.

- K-means - Highly parallelizable clustering algorithm used in many data mining applications.
- K-nearest neighbor - Calculates the K nearest neighbors from given data. Calculates euclidean distance from many data points in parallel.

- Backpropagation - Used to train the weights in a neural network. Error values are propagated through the network to retrain nodes and reduce output accuracy.

We propose an automated framework to fine-tune the level of approximation and satisfy required accuracy while providing the maximum energy savings. Figure 2.10 shows the proposed framework, consisting of the accuracy tuning and accuracy measurement blocks. The framework starts by putting the CFPU in the maximum level of approximation when no tuning bits are checked. Then, based on the user accuracy requirement, it dynamically decreases the level of approximation 1 tuning bit at a time until computation accuracy satisfies the user quality of service. The tuning is adjusted using a custom assembly instruction to set the approximation level of the CFPU. For each application, this framework returns the optimal number of CFPU tuning bits checked, providing maximum energy and performance efficiency. In future runs, the detected optimal configuration is set using the custom assembly instruction prior to running approximable code and disabled using the same instruction after completion of the code.

2.4.2 First stage CFPU

We first look at approximate multiplication. The proposed modified FPU can run entirely in approximate mode while providing a level of accuracy that is still acceptable for many applications. Table 2.1 shows the computation accuracy, energy savings, and speedup of running eight general OpenCL applications on the approximate GPU. These applications achieve error below 10% while using only first stage approximation. The energy and performance of proposed hardware are normalized to the energy and performance of a GPU using conventional floating-point units. Our experimental evaluation shows that our approximate hardware can achieve to 72% energy savings, 19% speedup, and 5.4× energy-delay product for these applications compared to the traditional AMD GPU, while providing an acceptable output quality less than 10% average relative error.

Table 2.2. Ratio of approximate to total CFPU operations and average relative error running applications in CFPU with single level approximation.

Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	Approx/total	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*
0 bit	100%	9.02%	100%	1.08%	100%	1.36%	100%	1.96%	100%	73%	100%	38%	100%	94%	100%	11.1%
1 bit	96%	2.70%	97%	0.35%	98%	1.04%	96%	0.50%	54%	9.8%	60%	13%	66%	31%	82%	3.7%
2 bits	94%	0.74%	95%	0.10%	85%	0.03%	94%	0.11%	32%	8.3%	43%	8%	55%	12%	76%	0.92%
3 bits	93%	0.07%	94%	0.03%	85%	0.01%	93%	0.02%	21%	4.1%	33%	5.2%	53%	8.3%	62%	0.36%
4 bits	92%	0.01%	94%	¯0%	84%	¯0%	92%	¯0%	15%	2.3%	29%	3.2%	47%	0.7%	53%	0.21%
Exact	92%	0%	93%	0%	84%	0%	92%	0%	10%	0%	23%	0%	45%	0%	53%	0%

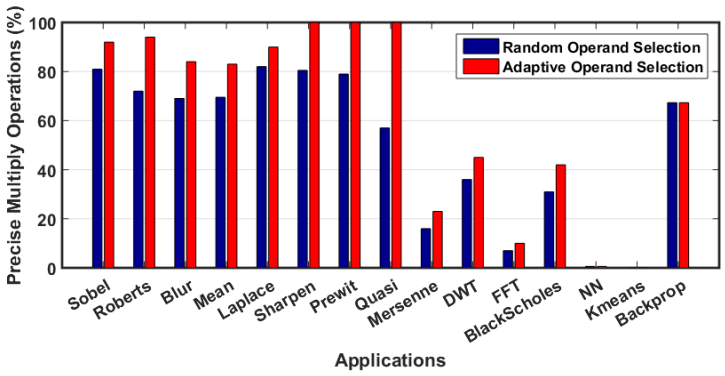
Table 2.3. Ratio of approximate to total CFPU operations and average relative error running applications on CFPU with two level approximation.

Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	Approx/total	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*
0 bit	100%	6.02%	100%	0.9%	100%	1.1%	100%	1.2%	100%	27.4%	100%	26%	100%	35.3%	100%	7.4%
1 bit	98%	2.3%	99%	0.12%	99%	0.3%	96%	0.50%	72%	7.3%	78%	5.4%	81%	18%	91%	3.5%
2 bits	96%	0.34%	97%	0.03%	93%	0.01%	97%	0.06%	64%	3.5%	66%	4.7%	77%	8.4%	79%	1.3%
3 bits	96%	0.34%	96%	0%	95%	¯0%	96%	¯0%	50%	3.2%	54%	2.6%	75%	7.9%	71%	0.59%
4 bits	95%	0%	96%	0%	95%	¯0%	96%	¯0%	41%	2.8%	47%	1.8%	59%	2.5%	66%	0.71%
Exact	95%	0%	96%	0%	93%	0%	95%	0%	28%	0%	38%	0%	45%	0%	60%	0%

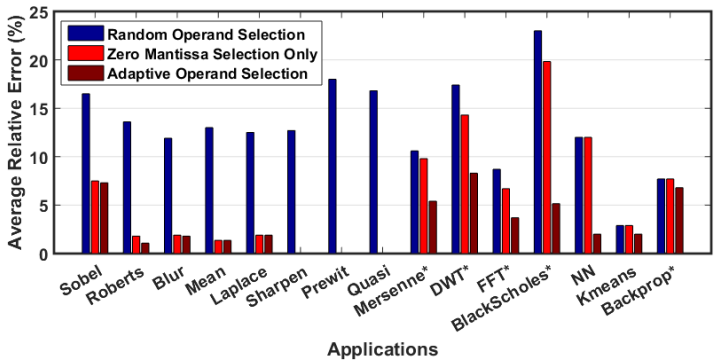
Adaptive Operand Selection

The approximate multiply uses both exponents in its calculation, but discards one of the mantissas, making an operation effectively a multiply by a power of 2. Therefore, a multiplication by a power of 2 always results in an exact answer on our hardware. It is possible to reduce error by ensuring the value of the discarded mantissa is equal to 1. This occurs when all the mantissa bits are 0. In the 11 OpenCL applications we tested an average of 52% of multiplies involved at least one power of 2. Hardware intelligently checking both inputs and adaptively discarding mantissas results in more exact computations and greatly reduced overall output error.

Figure 2.12a compares the portion of multiplications which runs precisely on the proposed CFPU with and without adaptive operand selection for the evaluated applications. Figure 2.12b shows the impact of the adaptive operand selection on the computation accuracy of the proposed CFPU. In random operand selection, the mantissa of the first input is always selected for the output without comparing the potential error of each mantissa. The result shows that adaptive operand selection significantly improves computation accuracy such that for all shown applications, the average relative error decreases to less than 7%. This improvement is due to increasing the portion of multiplications which are run precisely on the CFPU. We verify this by looking at the percentage of precise CFPU operations using the adaptive selection technique. For



(a)

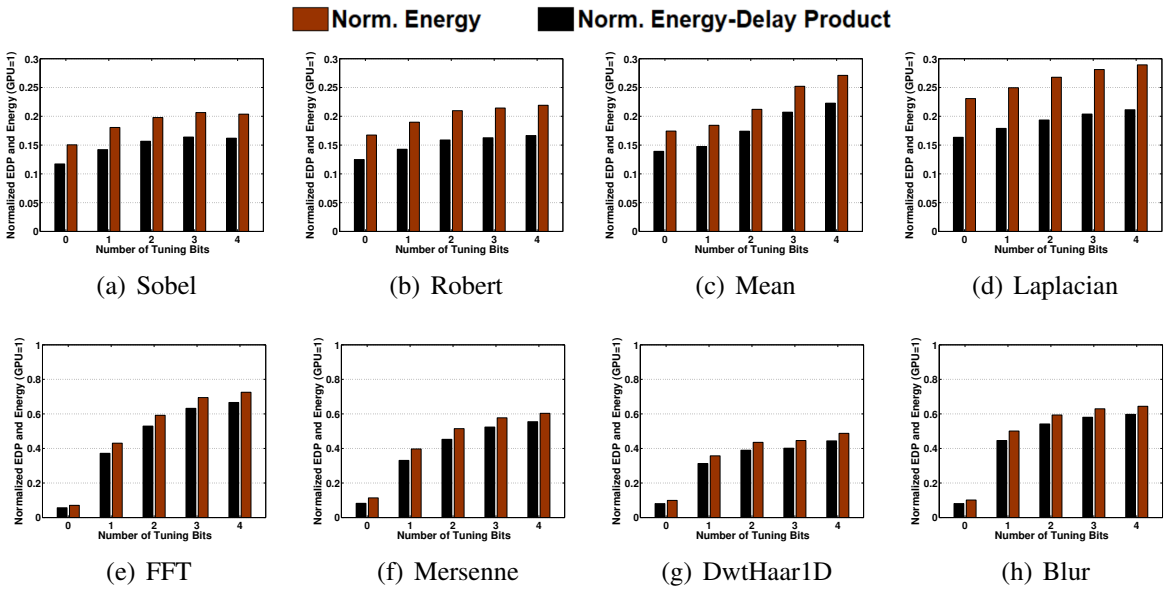


(b)

Figure 2.12. a) The portion of precise CFPU computation in different applications and b) the impact of adaptive operand selection on the computation accuracy. (Applications run partially on exact hardware).

example, in *Sobel* application, 82% of the outputs are calculated exactly, with an overall relative error of 16% when using random operand selection, while adaptive selection shows 92% of the outputs calculated exactly, at an overall relative error of 9%. All operations in Sharpen, *Prewit*, and *QuasiRandom* contain a mantissa of zero, so CFPU computes all results exactly. The image processing applications *Sobel*, *Roberts*, *Blur*, *Mean*, and *Laplace* also involve many operations that can be computed exactly with CFPU. Using adaptive operand selection to only select and discard zero mantissas improves application accuracy by up to 8×.

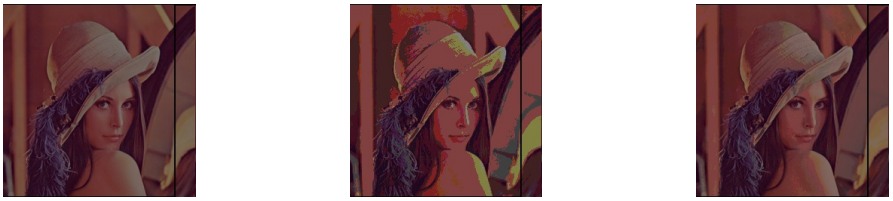
Our work in [3] only utilizes a zero mantissa selection policy, which does not reduce error significantly for many applications. the computation accuracy by up to 13× (8×) compared to random operand selection (zero mantissa only).



(a) Sobel (b) Robert (c) Mean (d) Laplacian

(e) FFT (f) Mersenne (g) DwtHaar1D (h) Blur

Figure 2.13. Normalized energy and energy-delay product of enhanced GPU with tunable two stage CFPU.



(a) Exact (b) Approx (c) Tuned

Figure 2.14. Output quality comparison for *Blur* application running on (a) exact computing, (b) approximate mode ($PSNR = 25dB$), and (c) tuned computing with $PSNR = 34dB$ and 13% run on precise CFPU.

The application showing the best accuracy improvement, Roberts, decreases from 13.6% application error using random operand selection to 1.85% using zero mantissa only selection [3]. The improved adaptive operand selection further decreases the output error to 1.08%. Excluding the three applications that only contain zero mantissa operations and get the best possible results, our evaluation for 12 different applications shows adaptive selection reduces average error by 2.2× more than our previous work [3].

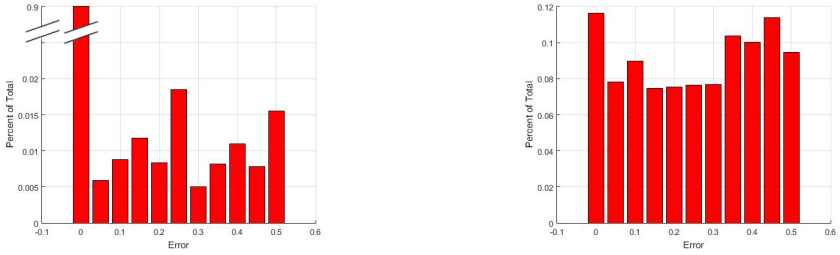
Tuning

We show the efficiency of the proposed CFPU by running different multimedia and general streaming applications on the enhanced GPU architecture. We consider 10% average relative error as an acceptable accuracy metric for all applications, verified by [67]. We tune the level of approximation by checking the N bits of mantissa in one of the input operands. If all N tuning bits match with the first mantissa bit, the multiplication runs in approximate mode, otherwise, it runs precisely by multiplying the mantissa of input operands. For each application, Table 5.5 shows the average relative error and portion of running multiplications in each application on exact and on approximate CFPU, when the number of tuning bit changes from 0 to 4 bits. Increasing the number of tuning bits improves the computation accuracy by processing the far and inaccurate multiplications in precise CFPU mode. Increasing the number of tuning bits slows down the computation because a larger portion of data is processed on precise CFPU. Figure 5.8 shows the energy consumption and energy-delay product of a GPU enhanced with tunable CFPU using different numbers of tuning bits. Our experimental evaluation shows that running applications on proposed CFPU provides respectively 3.5× and 2.7× energy-delay product improvement compared to a GPU using traditional FPUs, while ensuring less than 10% (1%) average relative error.

2.4.3 Second stage CFPU

Although proposed first stage approximate multiplication provides high energy savings, the accuracy of computation depends on the application. For some applications, with quantized inputs, e.g., *Sharpen filter*, the proposed design can work precisely with no average relative error. Other applications, such as recognition algorithms like motion tracking and detection applications, quantify changes in the input data allowing them to tolerate small amounts of error. An approximate multiplier must be able to control the level of output error to ensure close to exact results are calculated for these applications. Figure 2.15 shows the distribution

of error rates for each multiply operation of two applications. In the case of *Sobel*, almost 90% of the multiplies are by a power of 2 and are handled exactly by our approximate solution. The remaining 10% of operations have incorrect values with error rates ranging up to 50%. The Mersenne Twister application, on the other hand, has a evener distribution of error rates. While about 12% of the computations have no error, the error rates are too randomly distributed to provide acceptable overall error without additional optimization. For this application, the first stage approximation does not provide sufficient accuracy on its own, so over 50% of operations must be run on exact hardware to keep error below 1%.



(a) Sobel (b) MersenneTwister

Figure 2.15. Error distribution for applications.

Table 2.3 lists the hit rate of approximate hardware and average relative error for different applications running on hardware using two levels of approximation. The result shows that CFPU using two-level approximation can provide significantly higher accuracy compared to single level approximation. This efficiency comes from the ability of CFPU to assign input data to an approximation hardware which better classifies input data. Figure 2.4.2 shows the energy consumption and energy-delay product of CFPU using a two-level approximation. The result shows that accepting 10% (1%) average relative error, CFPU can provide 4.1× (3.2×) energy-delay product improvement as compared to a GPU using traditional FPUs.To ensure the quality of computation, Figure 2.14 compares the visual results of *Blur* running on precise and approximate hardware. Our result shows that approximate computing creates no noticeable difference between the precise and approximate result images.

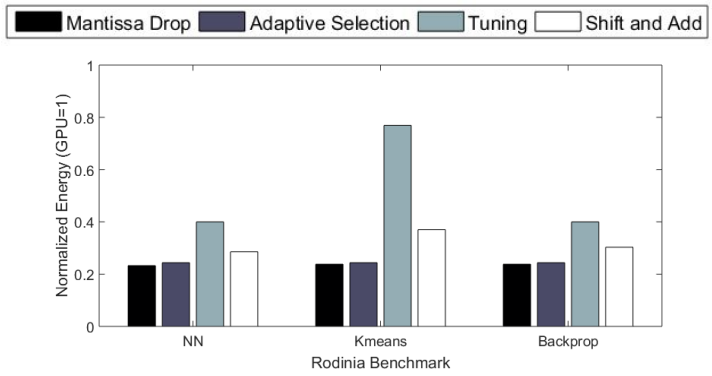
Table 2.4. Impact of one level CFPU approximation on accelerating Rodinia applications.

Tuning bits	**Backpro**				**K-nearest neighbor**				***K-means***			
	Approx/ Exact	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	Speedup
0-bit	100	25.3%	3.25	2.25	100.00	2.5%	2.91	2.21	100.00	2.1%	2.70	1.80
1-bit	80.3	22.7%	2.65	1.84	55.0	2.1%	1.59	1.43	51.6	0.9%	1.68	1.31
2-bits	73.6	22.5%	2.32	1.70	32.0	0.6%	1.32	1.23	26.1	0.3%	1.19	1.14
3-bits	71.6	16.9%	2.16	1.67	17.4	0.3%	1.15	1.11	13.8	0.1%	1.33	1.21
Exact	67.3	0%	1.87	1.60	0.6	0%	1.09	1.06	0.01	0%	1.00	1.00

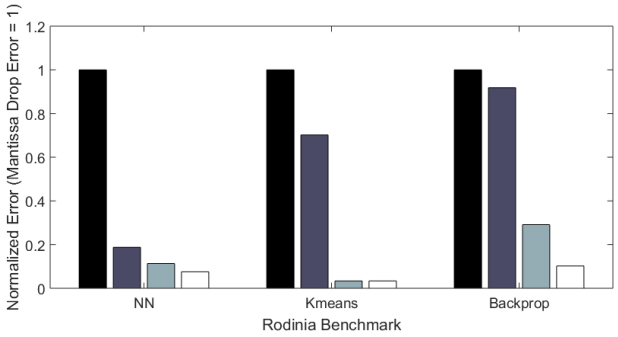
Table 2.5. Impact of two level CFPU approximation on accelerating Rodinia applications.

Tuning bits	**Backpro**				**K-nearest neighbor**				***K-means***			
	Approx/ Exact	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	Speedup
0-bit	100	25.3%	3.25	2.25	100.00	2.5%	2.91	2.21	100.00	2.1%	2.70	1.80
1-bit	98.7	7.89%	3.16	2.22	87.8	1.0%	2.36	1.93	90.2	0.7%	2.31	1.67
2-bits	93.5	3.35%	2.84	2.08	59.7	0.6%	1.65	1.49	61.9	0.1%	1.64	1.38
3-bits	87.7	0.1%	2.54	1.95	40.2	0.2%	1.35	1.28	40.0	0.1%	1.33	1.21
Exact	67.3	0%	1.87	1.60	3.8	0%	1.03	1.02	0.01	0%	1.00	1.00

Table 2.4 and Table 2.5 list the energy efficiency improvement, speedup, and average relative error of running Rodinia applications on CFPU with one and two levels of approximation. The results are listed when the number of tuning bits changes from 0-bits to 4-bits. For machine learning algorithms, CFPU with a single stage achieves 1.6× energy savings and 1.4× speedup while ensuring less than 1% average relative error. Enabling 2nd stage approximation increases energy savings to 2.4× and speedup to 2.0×, 50% and 40% improvements respectively. Figure 2.16 shows the energy and accuracy improvements for each optimization to the CFPU. In the first case, mantissa discarding is used for every operation, resulting in the highest energy savings, but poor accuracy. Adaptive selection reduces error but adds a small additional overhead. Tuning is used to reaching accuracy requirements, but energy savings is decreased drastically because a large portion of operations run on exact hardware. Finally, the second stage shift and add are used to reduce energy, while still maintaining accuracy. Our evaluation shows that the proposed CFPU design can achieve 4.1× (3.2×) EDP improvement while ensuring less than 10% (1%) average relative error. The tested algorithms performed well when coupled with approximate hardware.



(a)



(b)

Figure 2.16. Improvements from CFPU optimizations for (a) energy and (b) output error.

2.4.4 Overhead & Comparison

The first stage adds a 3.4% area overhead to the FPU, while the second stage adds an extra 6.2% area overhead. The energy overhead of a multiply operation when running on CFPU in exact mode is 2.7%, which is negligible compared to efficiency and tuning capability that CFPU can provide. In order to outperform the standard FPU, our design needs to run at least 4% of the data in the 1st or 2nd stage. We observed significantly higher percentages in all of the applications tested on the proposed CFPU.

To understand the advantage of the proposed design, we compare the energy consumption and delay of the proposed CFPU with the state-of-the-art approximate multipliers proposed in [5, 46, 47]. Previous designs are limited to a small range of robust and error-tolerant applications, as they are not able to tune the level of accuracy at runtime. In contrast, our CFPU dynamically predicts the inaccurate results and processes them in precise mode. CFPU tunes the level of

Table 2.6. Comparing the energy, and performance of the CFPU using 3 tuning bits and previous designs ensuring acceptable level of accuracy.

	Power(mW)	**Delay(ns)**	**EDP (pJs)**	**Max Multiply Error**	**Tunable**
CFPU (3 Bits)	0.17	1.6	0.44	6.3%	***Yes***
DRUM6 [5]	0.29	1.9	1.04	6.3%	*No*
ESSM8 [46]	0.28	2.1	1.2	11.1%	*No*
Kulkarni [47]	0.82	3.5	10.0	22.2%	*No*

accuracy at runtime based on the user accuracy requirement. The ability to run CFPU in exact mode and save power increases the range of applications that benefit from it. Table 2.6 lists the power consumption, critical path delay, and energy-delay product of CFPU alongside previous work in [5], [46] and [47] in their best configurations. We set CFPU to use 3 tuning bits, so the maximum output error per operation is the same or less than the multipliers we compare against. Tuning requires bit checks which increase energy consumption, so a CFPU configured to check 3 bits has slightly fewer energy savings than one configured to predict error. Our evaluation shows that at the same level of accuracy, the proposed design can achieve 2.8× EDP improvement compared to the state-of-the-art approximate multipliers for a multiply operation.

2.5 Conclusion

In this chapter, we proposed a configurable floating point multiplier which can approximately perform the computation with significantly lower energy and performance cost. CFPU controls the level of approximation by processing the data in one of the three tiers: basic approximate mode, intermediate approximate mode, and on the exact hardware. The first stage approximate mode discards one input's mantissa and uses the second's directly in the output to save energy. Accuracy is tuned by examining the discarded mantissa to estimate output error. When error exceeds a user-specified maximum, CFPU uses a 2^{nd}level of approximation. This mode uses a shift and add to increase accuracy. If the approximate output error is too high, the multiply is run on exact hardware. Our results show that using first stage CFPU approximation results in 3.5× energy-delay product (EDP) improvement compared to an unmodified FPU, while

ensuring less than 10% average relative error [3]. Adding the second stage further increases the EDP improvement, compared to the base FPU, to 4.1× for that same level of accuracy. In addition, our results show the proposed CFPU achieves 2.8× EDP improvement for multiply operations as compared to the state-of-the-art approximate multipliers.

CFPU is able to provide energy savings for many applications, but those with few powers of two do not approximate well. Computational similarity within applications can be utilized to save power. In the next chapter we discuss ALOOK, which exploits locality within applications to avoid recomputing similar values.

Chapter 2 contains material from "Runtime Efficiency-Accuracy Trade-off Using Configurable Floating Point Multiplier", by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018. The dissertation author was the primary instigator and author of this paper.

Chapter 3

Approximate Computational Reuse

CFPU, described in the previous chapter, functions best in applications with many zero bit mantissa values. Applications with randomly distributed mantissa values in the inputs, such as neural networks, require an alternate approach. Many of these applications involve a large number of redundant computations which can be exploited to save energy by using look-up tables composed of associative memory [9, 28, 51, 59]. Commonly computed values are stored in memory rather than recomputing the same result repeatedly. Computational look-up approaches suffer from two main drawbacks. First, static tables can only cover a limited portion of small applications that do not deviate significantly from the profiling dataset. Based on our testing, in the *Sobel* application up to 75% of operations are redundant, however, a large static table with can only avoid recomputing 40% of them. Second, existing computational reuse architectures perform lookup sequentially and are tied to the GPU pipeline. These designs only improve power efficiency, not performance.

In this chapter we propose an adaptive computational reuse approach, called ALOOK, which dynamically updates values stored in associative memory at runtime to improve the energy efficiency of GPGPU applications. To control the energy cost of ALOOK updates, we reduce the number of writes by only replacing values in ALOOK intermittently. In addition, to the best of our knowledge, we propose the first GPU-based computational reuse approach which significantly exploits associative search to improve the GPU performance. ALOOK duplicates

the first stage of the floating point unit (FPU) pipeline enabling processing and computational reuse of two inputs simultaneously. Doubling the first stage adds less than 4.5% area overhead to a conventional FPU. Using this architecture, ALOOK can parallelize the FPU computation, unless ALOOK cannot match a stored value with either operation.

We examine the proposed techniques for a range of OpenCL GPGPU applications, as well as several machine learning benchmarks from the Rodinia benchmark suite. We test the efficiency of ALOOK by integrating it beside FPUs in an AMD Southern Island GPU. Our evaluation shows that ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. The proposed ALOOK architecture improves the GPU performance by 2.0× as compared to state-of-the-art computational reuse methods [2, 58] for the same level of output error.

3.1 Related Work

Approximate memory based accelerators store common inputs and output pairs to implement computational reuse. The associative memory searches for the nearest distance value in the table to return as the result for the given inputs [9, 53, 58, 59]. This method is effective at saving power in applications with many identical or similar computations. Associative memory is placed adjacent to FPUs to store commonly occurring data. In these designs, developers must identify and profile key regions of approximable code for common inputs and outputs to load into a static table. In [9], the authors propose a configurable associative memory which relaxes computation by applying VOS to the non-volatile associative memory to trade accuracy for energy savings. Work in [2] designed a novel associative memory based on non-volatile memory which supports searches for nearest absolute distance, rather than Hamming distance similarity. All existing work provide low hit rate on practical applications because their stored values are static and cannot change to adapt to a running application. Work in [58] proposed a

method which is capable of updating associative memory through online training. However, this approach suffers from high overhead and energy cost.

We propose ALOOK, a dynamic lookup table capable of improving GPGPU performance and reducing power use. ALOOK does not require code sampling and profiling prior to runtime and reduces energy consumption by searching associative memory for previously computed results. Unlike existing approaches in which GPU performance is bound by pipeline stages, ALOOK is the first design which exploits the redundant GPU computation to improve the computation performance.

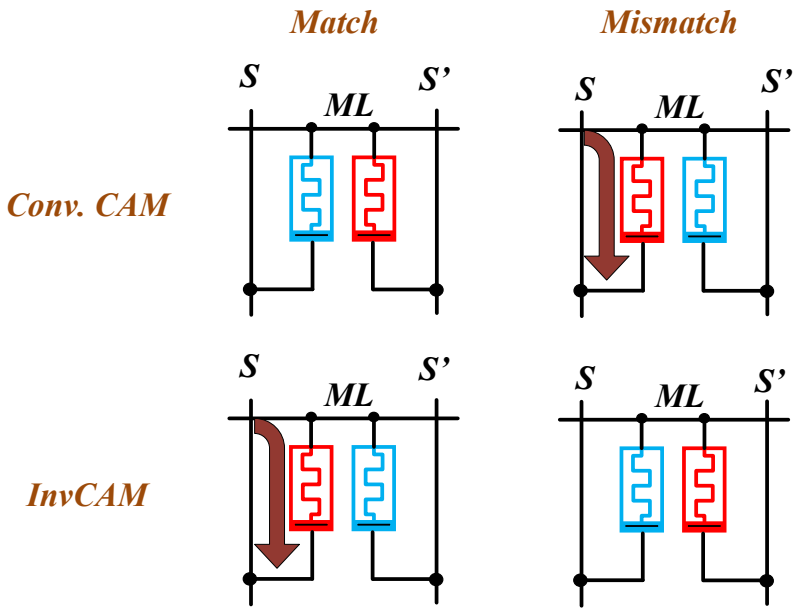
3.2 Background

Resistive CAM accelerators can perform faster and more energy efficient computations compared to general purpose floating point units. In order to have computation capability, lookup tables need to have the capability of searching for closest distance row. However, conventional CAMs do not have the ability to search for the nearest row. They can only determine a row which exactly matches with the input pattern (if there is any). Implementing a fully digital CMOS-based design, which can search for nearest row, is very inefficient in term of power and area because it needs (i) bit level comparison of the input pattern and store values, (ii) to count the number of matches in each row, and (iii) finally finding a row with the minimum distance. To enable the nearest distance search capability, we design an Invert CAM (InvCAM) and then exploit analog characteristic of the NVM-based CAM to efficiently search for nearest data.

3.2.1 InvCAM Cell

Figure 3.1 shows the structure of crossbar memristive CAM in normal and inverse (InvCAM) functionalities. In a conventional cell, the memristor devices and select lines are set such that the *ML* stays charged during the exact matching. In a match, there is no leakage current between the *ML* and ground, since the select line which stored 0 is connected to high resistance (H). The *ML* current also cannot discharge from the cell with the select line of 1 because of

same voltage across the memristor devices (even though the device is in low resistance mode). In the case of a mismatch in conventional cells, the select lines bias with inverse values, where the select line of a cell with low resistance (L) connects to the 0 and the high resistance to 1. Therefore, the *ML* discharges using the L resistance. We change functionality of CAM such that the *ML* counts the number of discharging cells in each row. InvCAM stores the opposite values on memristor devices. In case of a match, a cell with low resistance discharges the *ML*, while in case of mismatch the *ML* stays charged (as shown in Figure 3.1).



Search Data	Select lines
0	1 0
1	0 1

Store Data	Conv. CAM	InvCAM
0	L H	H L
1	H L	L H

Figure 3.1. Conventional and InvCAM cell in match and mismatch operations.

In InvCAM rows (consisting of several InvCAM cells), each matched cell adds a new discharging current component to the *ML*. So, during the search operation in InvCAM, all rows start discharging, except a row where all bits are mismatched with input operands. However, all InvCAM rows do not discharge at the same speed. In other word, in rows with more bit matches,

Table 3.1. *ML* hit/sampling time in 4-bit InvCAM having different number of mismatches/Hamming distances (HD) and different CAM sizes

InvCAM Mode	Number of Matches	128 rows	256 rows	512 rows	1024 rows	2048 rows
Exact	4	0.7ns	0.9ns	1.5ns	2.1ns	2.8ns
1-HD	3	0.9ns	1.1ns	1.8ns	2.4ns	3.4ns
2-HD	2	1.1ns	1.4ns	2.2ns	2.8ns	4.1ns
3-HD	1	1.4ns	1.7ns	2.6ns	3.3ns	4.6ns
4-HD	0	1.8ns	2.3ns	3.2ns	3.9ns	5.2ns

.

the cells will discharge *ML* faster. We exploit this analog characteristic of memristor devices to design a CAM which has the capability of searching for the nearest distance row. For robustly detecting the closet row, InvCAM needs to have a limited number of cells in the bitline, because a *ML* discharging current/time does not change linearly with the number of matches in a line. For example, in a 16-bit InCAM, rows with 15 or 16 matches have very similar discharging characteristics. In other words, to distinguish a row with the fastest discharging time, we require an ultra-fast sense circuitry (˜ps delay). To address this issue, we limit the bitline size in each CAM to <8-bits and use memristive devices with large ON resistances for search operations. Short bitlines of InvCAMs help us to identify the difference between the number of mismatches with reasonable detector circuitry delay. Table 3.2.1 shows the *ML* discharging current for a 4-bit InvCAM with different numbers of matching bits and different InvCAM sizes. When all four cells match the input operand, the *ML* has the maximum discharging current, which means it has fastest discharging speed. Having a fewer number of matches results in slower *ML* discharging current. In our design, rows with different number of matches have obvious discharging time distances. We exploit this characteristic to design a CAM with the capability of finding nearest hamming distance. This functionality is not easily implementable on conventional CAMs where *ML* stay charged in the case of a match. However, here we could provide this functionality by designing a circuity which can detect a row with the fastest discharging current.

3.2.2 InvCAM Architecture

To find an appropriate nearest neighbor row, we consider hamming distance between input data and stored values on a CAM. However, hamming distance metric is an aggressive metric for finding the nearest row, since hamming distance does not consider the impact of each bit indices in calculating the distance. In real computation, the most significant bits (MSBs) usually have higher impact on computation when compared to the least significant bits (LSBs). We exploit this feature to design a lookup table with lower power consumption and better accuracy. Figure 3.2 shows the overview of proposed design. In our design, we split the R-row*N-bit CAM block to m small size stages (i.e. B_1, B_2, B_m) where each stage contains N/m-bits. Partial blocks search for input data in serial stages. The search operation starts from the block with the most significant bit. Each block has the capability of configuring as a CAM or memory block. In memory configuration, it uses sense amplifiers at the tail of vertical bitline to read the memory rows. In CAM mode, each stage can find the nearest hamming distance row(s) using the sense circuitry in the horizontal MLs. The sense amplifier finds the row with the fastest ML discharging current. The nearest row corresponds to data with the maximum matching bits with the input operand. After the sense amplifier, we have analog detector circuitry which can sense the number of active rows in each CAM. As soon as the detector block senses an active row, it stops the search on the current CAM stage and selectively activates the rows of the next stage CAM. During the next cycle, a similar search on the second stage starts, but only in the selected rows. This search operation continues until our design reaches the last stage with a row with nearest distance to input operand.

For applications with multiple input operands, the first stage stores the first m-bits corresponding to all input operands and then uniformly searches for the stored data with the closest distance to the input key. All existing bits in the first stage have similar weight in our nearest distance search. Based on the rows activated by this first stage, the search on the next stages continues selectively.

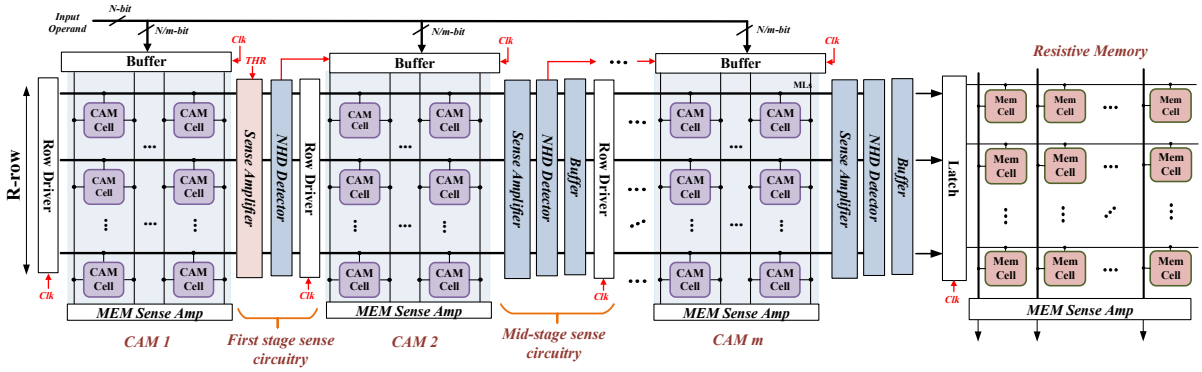


Figure 3.2. The overview of ALOOK structure using InvCAM blocks.

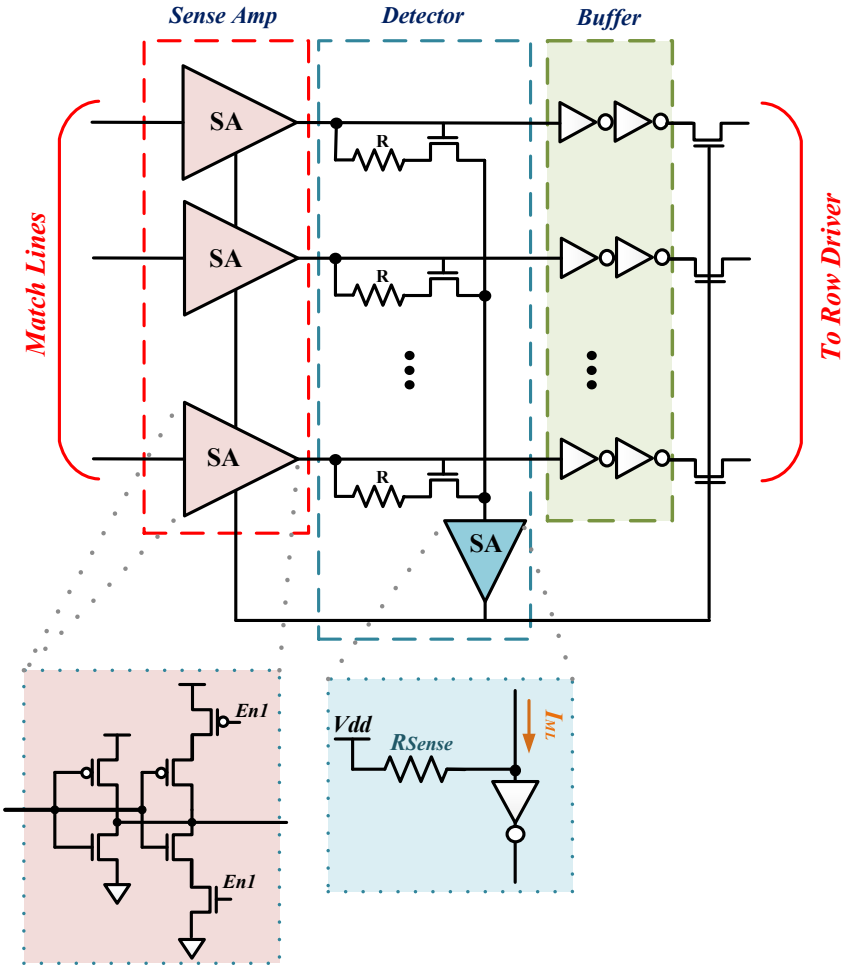


Figure 3.3. Details of the sense amplifier, detector and buffer circuitries of ALOOK block.

Figure 3.3 shows peripheral circuitry which supports the nearest distance search. For each CAM stage, we use three layers of peripheral circuitry to support nearest distance search.

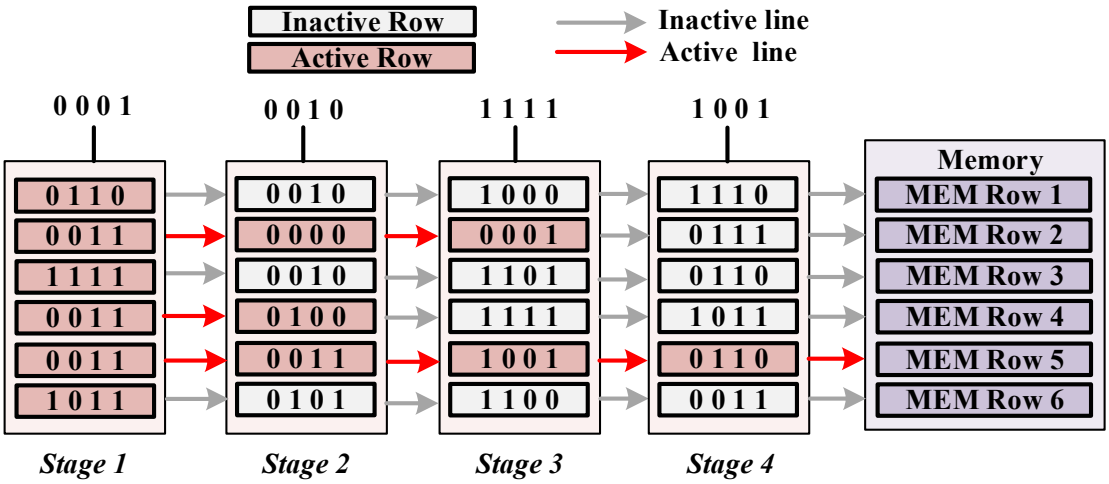


Figure 3.4. Example to show ALOOK search functionality in 4 stage search.

First, the sense amplifier reads the value of match lines (MLs) to find a row which matches with the input data. Based on our cell design, a row with more matched cells will start discharging the ML first. The matching results in a hit on the output of the sense amplifier. A detector circuitry, shown in Figure 3.3, checks for hits in CAM rows. In the case of any active rows, (i) it activates the access transistors to selectively activate the rows of the next CAM stage, and (ii) sends signals to the sense amplifier of the current CAM stage to stop the search operation from hitting additional rows. As our sense amplifier circuitry does not use a clock in its structure, so the delay of the buffer stage has an important rule on the correct functionality of our design. Because the detector circuitry is not an ideal circuit, and has a long delay, it may not be able to immediately sample the active rows. This delay can result in several matched rows and missing the rows which do not have minimum distance.

To better clarify the functionality of proposed multistage lookup, Figure 3.4 shows an example of search operation on a 4 stage, 8 row table. The search operation starts from the first stage, which is the most significant block, by searching for nearest Hamming distance row. The hit rows on the first stage selectively activate rows of the second stage. The second TCAM searches for the row nearest to 0010 on the three activated rows. The rows of the next TCAM stage activate serially, such that the design has a single active row in the final stage.

To guarantee the functionality of the proposed design, we add a buffer stage which has two main rules: **(i)** The block needs to delay the sense amplifier hits while the detector senses any active rows. In order to have stable detection, the delay of the buffer stage should be higher than the delay of the detector circuitry. To set the size of the buffer and detector circuitry, we consider 10% process variation on the transistors size and threshold voltage [68]. We designed our circuity for the worst-case scenario where there is only a single hit row on the CAM rows (maximum detector latency) and when CAM has a row with all matched cells and a row with a single mismatch. As we explained before, the ML discharging current starts saturating with an increase of the matched cells in each row. A row with all matched cells and a row with a single mismatch have fastest ML discharging speed. **(ii)** The second rule of buffer is to sample the activated rows (output of buffer stage) in case of a hit in detector circuitry.

The number of TCAM rows in the InvCAM structure depends on the precision of the detector circuitry. The detector circuitry should be able to identify a single activated row in CAM architecture. TCAMs with many rows suffer from large amounts of leakage currents through the sense circuitry output, resulting in a wrongly identified matching row by the detector. To have enough ratio between the leakage and matching currents, we need to have $I_{Matching} >> N * I_{Leakage}$, where the matching and leakage currents are the ON and OFF currents of sense amplifier output respectively. We use a diode connected transistor beside the detector resistance to control the OFF current ($I_{Leakage}$) of different rows.

3.2.3 Tunable Approximation

ALOOK, as a stand-alone processing unit, can provide enough accuracy for several applications. As we explained, there are large amounts of redundant input operands in computation. For example, in images, a large portion of background pixels are usually similar. Processing these high frequency patterns provides enough computation accuracy even using small sized and fast tables. However, uncommon/infrequent part of the workloads (with low data locality)

cannot run accurately. To provide high computation accuracy for general workloads, we need to store a large set of data in the tables, which results in high energy and performance overhead. To address this issue, we use both the existing exact FPU and ALOOK partially for each workload's computation. For the majority of data, which has close distance to stored values, our design uses approximate results to perform computation fast and efficiently, while the other part of input data with large distance to the stored values, can run on a precise hardware. Our goal is to have a hybrid computation which can set the level of accuracy by partially running the data on an exact processor and ALOOK. When the input data is far from the stored values in ALOOK, the data is sent to GPU pipeline to process.

ALOOK should have the ability of detecting the distance of input data to store values. As shown in Figure 3.5, the first CAM stage uses different sense circuitry, compared to the other stages, which can detect the ML discharging voltage corresponding to h-bit matching. Based on the h value, we set the sampling pulse of the sense circuitry (*THR*) to find the rows which have less than h-bit distance with pre-stored value. A detector circuitry checks the sense amplifier output of all rows and activates the row driver of next CAM stage accordingly. However, in case of missing data in detector circuitry, the EnL signal sends a signal to start running this data on exact processor instead. Based on desired accuracy, changing the clock time of the sense circuitry gives us different *THR* values. Using a late *THR* period, corresponds to deep approximation, while fast sampling means precise computation on the existing processor.

3.2.4 Early search termination

Our evaluation shows that in several cases we do not need to go through all CAM stages to find a nearest row. Instead, we can stop the search operation when the number of active rows in a block becomes one. Going further to InvCAM stages is unnecessary when we already found the row with the minimum distance, thus resulting in improved energy efficiency for the proposed design. We also observed that this condition occurs frequently. Even in a CAM with many

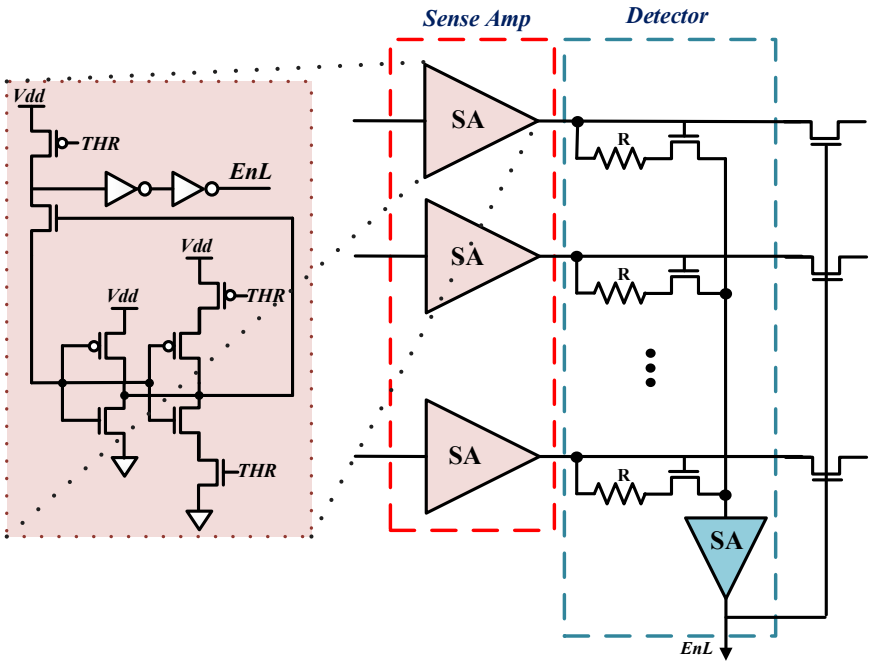


Figure 3.5. The sense circuitry of the first stage InvCAM in ALOOK structure

rows, searching a few stages of CAMs is enough to find the closest row. Therefore, our design exploits Early Search Termination (EST) detector circuitry (shown in Figure 3.6) which can identify a case that single CAM row is activated. EST is designed using an analog comparator circuitry [69], which samples the same current that the detector is sampling and can identify the case that a single row of a CAM is matched. Obviously, using ETS in more stages can accelerate the search operation, however, it increases the energy and area of the CAM. Therefore, for all tested applications in this paper we use EST circuitry in one of the middle stages to check the number of row activations once and then stop further search operations. The best stage can be found based on the configuration using profiling results from training mode.

Our design considers the impact of each bit indices on the computation accuracy by searching for a closest input data starting from most significant blocks. In addition, serial search operation reduces the number of active rows in TCAM block, since the rows of each can be activated by the hit of the previous stage. This reduces the number of active rows stage by stage, until we achieve a single row at the last stage. The energy savings achieved by selective row

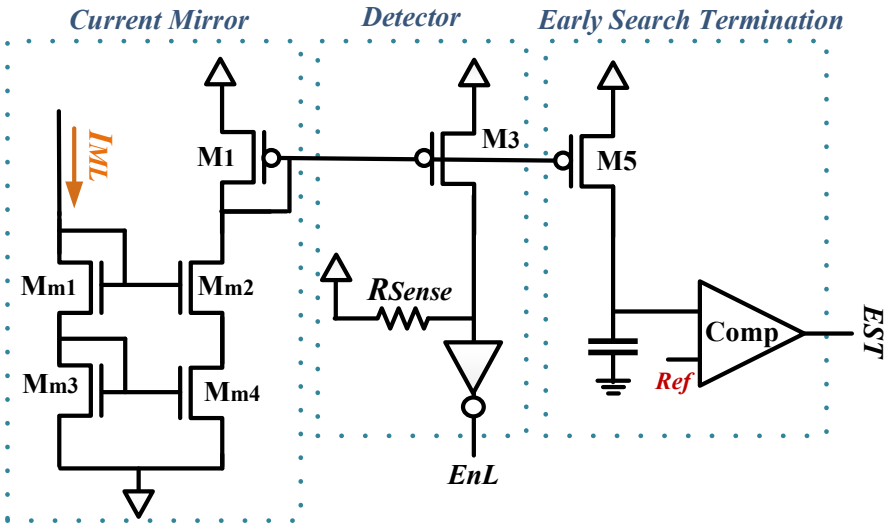


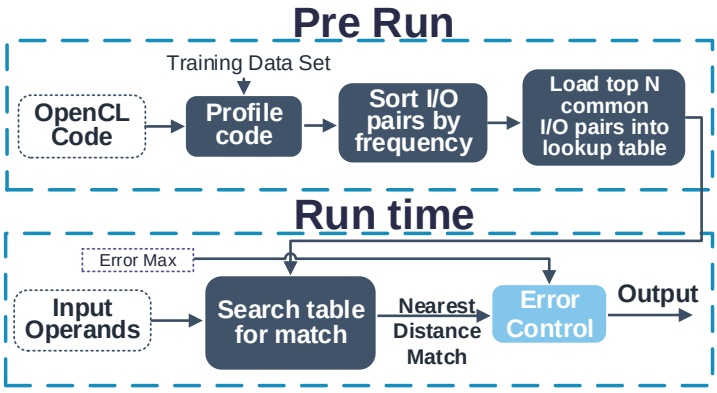
Figure 3.6. Early search termination technique using analog comparator block

activation depends on the InvCAM block size. InvCAM with smaller blocks reduces the overall number of active rows through CAM stages. However, this requires a larger number of sense amplifier and detector circuitries. The tradeoff between the size and energy consumption also impacts the computation performance.

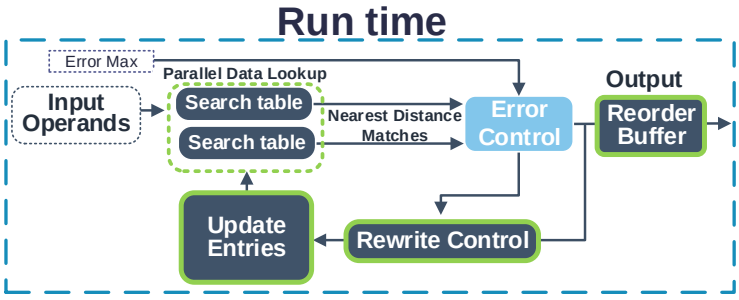
3.3 Adaptive Lookup Design

Many GPGPU workloads show large amounts of computational data locality with identical or highly similar operations recomputed repeatedly. In these applications, inputs and output pairs for frequently used operations can be stored in associative memory. Inputs for operations are used to search the memory to find the nearest distance match and return the associated output.

We propose ALOOK, a dynamic lookup table capable of adapting to short-term data locality, thus improving GPGPU performance and reducing power use by eliminating redundant computations. Our design provides three major improvements over previous work. First, we eliminate the need to profile the application before runtime for common operations to store in the table. Second, ALOOK updates data entries dynamically to adapt to changes within the



(a) Static Work Flow [2]



(b) ALOOK Work Flow

Figure 3.7. The process flow for a) the static lookup table [2] and b) the dynamic ALOOK.

application over time. Third, ALOOK uses multiple small lookup tables perform multiple parallel checks and speed up applications, unlike prior designs which only offer energy improvements.

3.3.1 Lookup Overview

Placing a small lookup table next to each floating point unit enables approximate computational reuse [2, 54]. Figure 3.7a) shows the pre-run and profiling stage of a static table [2]. During the pre-run stage, the user must flag the section of OpenCL code they wish to approximate. Then, the application is profiled with a training data set and the input and outputs of each arithmetic operation is recorded. Each unique I/O pair is counted and the pairs are sorted by frequency. Prior to a non-profiling run, the top N most common pairs are loaded into the lookup table, where N is the number of entries the table can store. At run-time, the lookup table searches for each operation's input values and returns the corresponding output. The table returns the

nearest distance match rather than recomputing the result. The nearest distance match is not guaranteed to be an exact match, so the final application accuracy is managed through error control. Error for a match grows as the difference between the input values and the closest match increases. Error control identifies matches with a large distance from the inputs and assigns them to hardware to compute exact results, while close matches are used directly as output, saving power.

Figure 3.7b) shows the improvements, in green, ALOOK offers over previous work. ALOOK entirely eliminates the prerun profiling steps required for static designs. The user must still flag code safe for approximation, but no longer needs to sample the application for I/O pairs and identify the most common ones. At run-time ALOOK searches for the nearest distance match to each input within the table. To speed up applications while still allowing accuracy control, two incoming operations are run in parallel. *Parallel data lookup* searches two duplicate tables simultaneously, then a reorder buffer ensures results are output in the order they arrived in the pipeline. The output values associated with the nearest match is used as the result for the operation rather than running on the FPU. Accuracy checks are made on each operation using error control, shown in Figure 3.7 in light blue, to ensure close matches for operations. ALOOK supports the closest distance search within a user set maximum error. If the input data has distance larger than the specified distance, it will not match any entries. Instead of using poor matches, the results are computed on exact hardware and the newly computed values replace stored entries in the table using an LRU policy. Rewriting on every miss is unnecessary, so *rewrite control* is added improve the energy efficiency of ALOOK. Updating the entries with recently computed values increases the search hit rate, reducing energy consumption and accelerating a wide range of applications.

3.3.2 Multi-Table Parallel Lookup

A critical drawback of prior lookup based designs is the lack of application acceleration. Placing a lookup table within the FPU saves power, but is tied to the pipeline [2]. Approximated

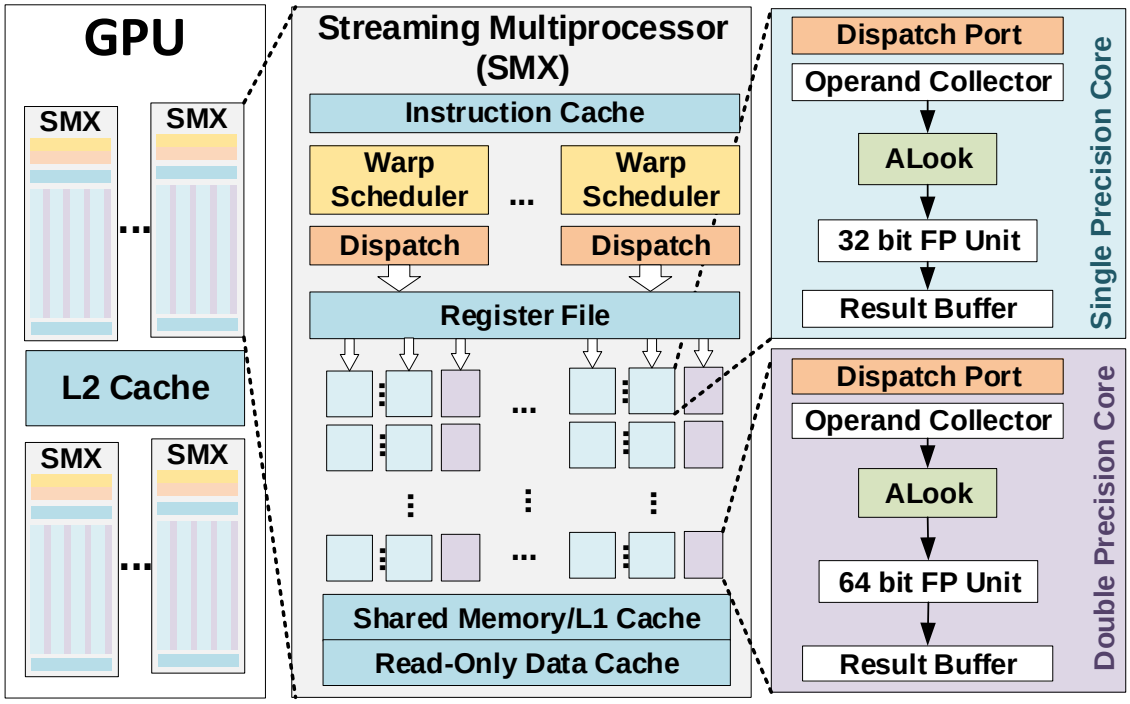


Figure 3.8. Implementation of ALOOK alongside FPU within AMD Southern Island Architecture

the pipeline stages for operations are clock gated to save power, but these stages are not utilized. Only one operation is searched for at a time, rather than checking several operations simultaneously. We place multiple tables outside the FPU to check several instructions concurrently. For each search that returns a usable result, the operation is removed from the input buffer avoiding the FPU altogether and accelerating the application.

Figure 3.8 shows ALOOK integrated into a GPU. We place ALOOK within each GPU core immediately after the operand collector, ensuring the values are available for lookup. To build the lookup tables for ALOOK, we use CAM, proposed in [2]. ALOOK provides nearest distance match using the inputs from each arithmetic operation run on the GPU core. The number of bit mismatches between the match and the input values predicts the output error. A better match has a lower error, while further distances lead to worse approximation. To control application accuracy, the operations are split between using results from ALOOK and running on exact FPU based on the match distance. Users set the maximum allowed error for individual

matches to trade off between accuracy and performance/energy. Higher error tolerance results in better acceleration and energy savings.

Figure 3.9(a) shows associative memory used to reduce energy consumption of an FPU [2]. This configuration places the lookup table adjacent to the first stage of the FPU as the search operation takes one cycle. When a match is detected, the FPU stages for the operation are clock gated to save power and the output is used rather than recomputing the value. This approach only checks one operation at a time, preventing application acceleration. ALOOK uses multiple lookup tables before the FPU pipeline to search incoming instructions in parallel. Figures 3.9(b) shows the implementation of ALOOK. To accelerate applications, we use *multi-table parallel lookup*, an architecture which compares multiple inputs in lookup tables simultaneously. The lookup table is placed outside the FPU. Incoming operations are stored in a queue before the FPU. Up to five of the inputs are searched for in the lookup table at the same time. The state of matches can be one of the following. (i) If one or more searches results in a hit, the stored value associated with the match is output by ALOOK. All matched operations are removed from the queue and the remaining ones are moved to the checked queue. (ii) If no matches are detected, values that received the passthrough signal are removed from the queue and the rest are moved to the checked queue. Values in the queue are processed sequentially. As the queue fills, we disable lookup tables to avoid overflows.

The lookup table hit rate has a substantial impact on the performance of ALOOK. Hit rate is impacted by the level of approximation allowed and the size of the lookup table (LUT). If three values are processed in parallel and two can be approximated, the performance increases by $3\times$. However, if the hit rate is low and all three miss, the performance remains the same, but the search operation increases overall energy consumption. The number of lookup tables enabled is adjustable. Each core has multiple tables, but the exact number represents a design trade-off. Additional tables improve acceleration but increase power draw. One method we use to improve our energy savings is to utilize multiple tables, but disable some during periods of low hit rate. As the number of good matches decreases, the checked queue will become full and additional

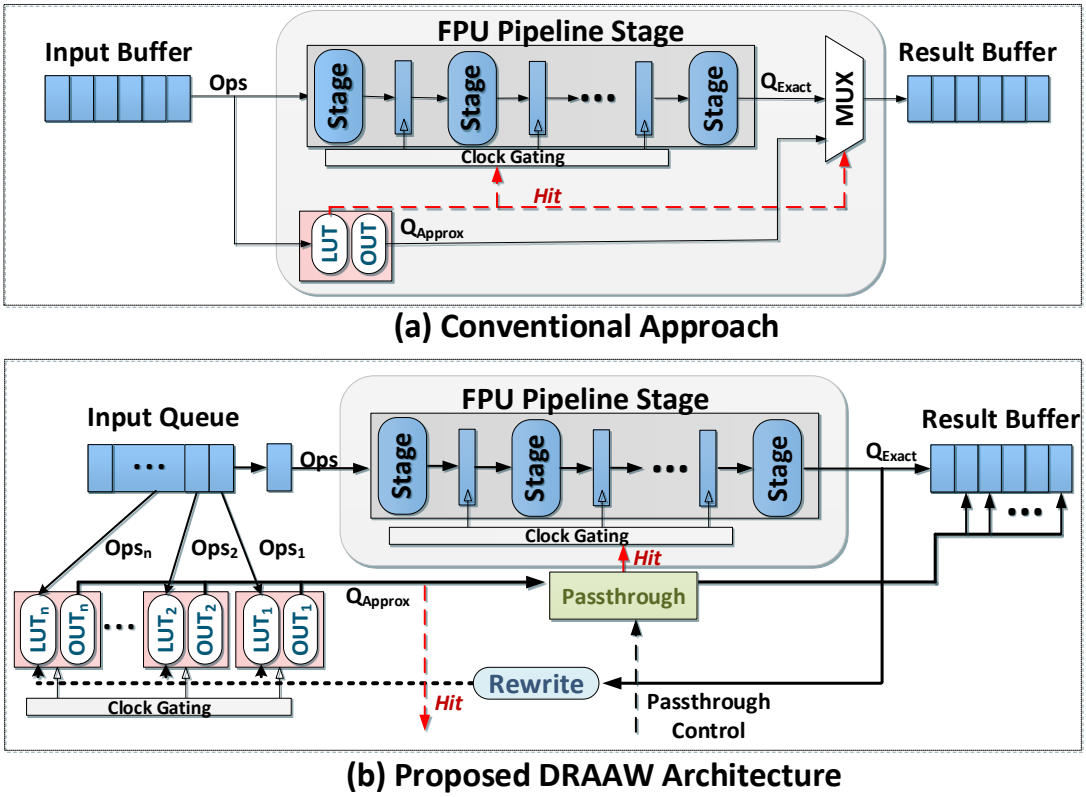


Figure 3.9. (a) Computational reuse in conventional FPU within GPU [2] (b) ALOOK architecture integrated lookup tables outside of FPU.

tables are disabled. For example, for an application with 80% hit rate, we need to enable all five lookup tables, where four inputs (out of five) on average will be bypassed with a match from the table. Input which misses, runs on the exact GPU pipeline. If on average more than one of the inputs misses, a queue of operations pending to be computed forms. A table cannot perform another search after a miss until the inputs enter the FPU pipeline. For example, if two tables miss, only the three which hit can be checked in the next step. Therefore, the number of active lookup tables is roughly based on the recent average hit rate. The lookup tables are mirrors of each other, so increasing them does not impact hit rate. However, as each incoming operation is only searched for in a table once, adding additional tables primarily impacts area overhead.

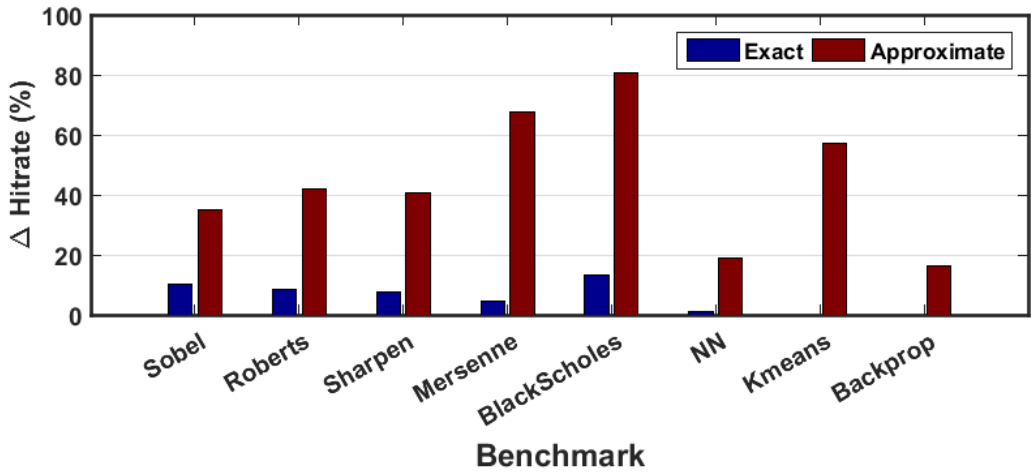
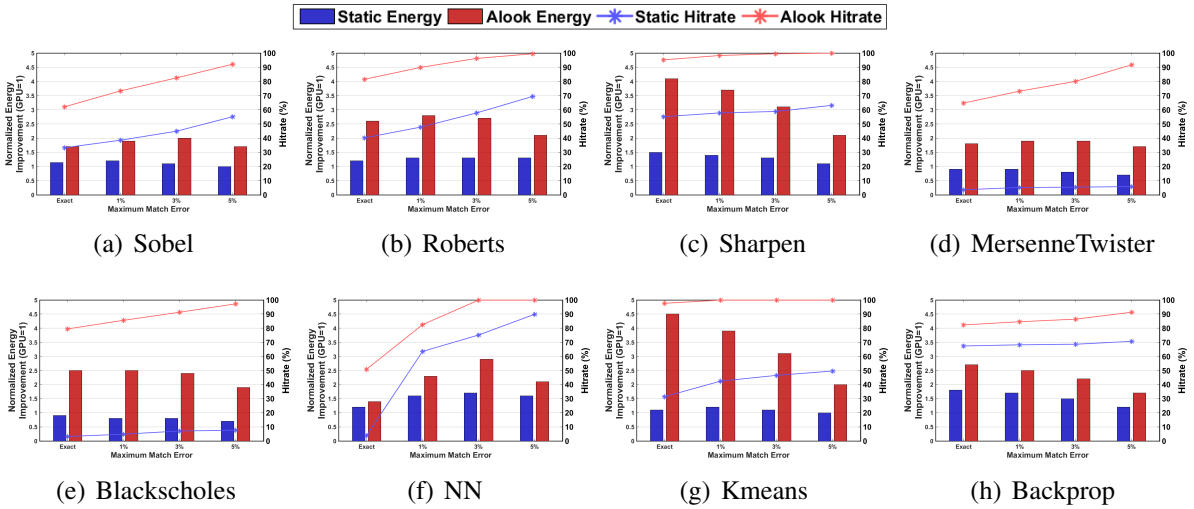


Figure 3.10. Increase in hitrate for a dynamic table over a static table [2] for exact matches and approximate matches with less than 5% error

3.3.3 Rewrite Control

Developers can adjust the level of accuracy for individual operations for each section of code, with the default setting only providing exact matches. In this way, even applications with high precision requirements benefit from ALOOK. If the error of the closest match is greater than a user-specified error max, the output is run on exact hardware instead. Error control selects which of the two possible results to output. When a search match exceeds the maximum error, the tables must be updated, adding additional overhead due to write costs. Rewrite control determines when and how often to rewrite data in the table. We show reducing rewrite rate decreases overhead without significantly impacting hitrate, resulting in higher energy savings.

We reduce the frequency of ALOOK rewrites in order to improve energy efficiency. Decreasing the rewrite rate saves energy, but also results in fewer hits. During periods of high hit rates, rewriting on misses can remove common operations prematurely. ALOOK replaces infrequently used entries with a least recently used (LRU) replacement policy. When a close match is found for given inputs, the counter is set to zero and the other counter values are incremented. When a miss occurs in ALOOK, the value with the highest count is evicted and replaced with the result computed on exact hardware. To enable LRU policy we use m bits, based



(a) Sobel (b) Roberts (c) Sharpen (d) MersenneTwister

(e) Blackscholes (f) NN (g) Kmeans (h) Backprop

Figure 3.11. Operation hitrate and energy improvement over unmodified GPU for a static lookup table [2] compared to ALOOK.

on the number of entries in the table, to keep track of the values in the dynamic lookup table. When two misses occur during parallel data lookup, only the furthest distance miss is used to update the tables to ensure consistent values.

For a hardware implementation, we used a Content Addressable Memory (CAM) using Non-Volatile Memory (NVM) [70, 71]. In particular, we used memristor CAM with 2 transistors and 2 resistors (2T-2R) for our implementation. Write speeds in CAM memory are slower and consume more energy than reads. When processing code sections with low data locality, the cost of constantly rewriting data becomes high and penalizes energy savings. We decrease the number of writes by only replacing values in ALOOK once for every 8 misses as determined by our experimental results. In Section 3.4.4 we examine the trade-off between energy efficiency and update frequency.

A dynamically updating table can handle larger and more varied data sets than static designs. Static profiled data may not accurately represent the runtime data. A training set needs to be varied enough to cover a wide variety of inputs, but this generalization can provide sub-optimal hit rates and accuracy. The profiling data set may not accurately reflect all use cases and contain gaps for some real-world cases. In many workloads, the data locality changes

significantly over time. The more distinct regions a data set has, the harder it is for a small static lookup table to cover the changes over time.

3.4 Experimental Results

3.4.1 Experimental Setup

The proposed ALOOK is implemented beside each floating point unit on an AMD Southern Island architecture GPU, Radeon HD 7970 device. We use Multi2sim, a cycle accurate CPU-GPU simulator and modify the kernel code for profiling and runtime simulation [64]. We test the design using 8 OpenCL applications: 5 from AMD APP SDK v2.5 [66] and 3 from the Rodinia 3.1 machine learning benchmark suite [48]. From AMD APP SDK we test *Sobel*, *Roberts*, *Sharpen*, *MersenneTwister*, and *BlackScholes*. From Rodinia we test *Knn*, K-means, and *BackProp*. We use the Caltech 101 computer vision dataset [72] for image processing applications. For other applications, we use application specific data sets. For image processing, we define PSNR as the accuracy metric. For general applications working with numbers, the average relative error is defined as the accuracy metric. For each machine learning algorithm, we define a unique accuracy metric to test the impact of approximation. We extract frequent patterns of four GPU floating point units; adder (ADD), multiplier (MUL), multiply accumulator (MAC) and square root (SQRT). To estimate core performance we use a 6-stage balanced FPU generated by FloPoCo [73] optimized using *Synopsys Design Compiler* [65]. The circuit level simulation of TCAM design performs using HSPICE simulator on 45nm technology. We use the Vdd=0.85V for blocks without accepting any computation error. To guarantee the impact of variation on circuit level design, we consider 10% process variation on the size and threshold voltage of transistors by running 10,000 Monte Carlo simulations.

3.4.2 Static vs Dynamic

ALOOK adapts to previously unseen applications by updating the values over time. Figure 3.10 shows the improvement a dynamic table provides over a same sized static table [2]. The dynamic table updates when it cannot provide a match, replacing old entries with values computed on exact hardware based on an LRU policy. On average the dynamic table improves the hit rate of exact matches for the tested applications by 6% compared to an identical static design. When matches are guaranteed to less than 5% error, the hit rate improvement increases to 44.8%. The adaptability of ALOOK allows fewer entries to produce the same or better results than static memory, resulting in significantly less area overhead and search power. In our testing, static tables require up to 8× more entries to provide comparable hit rates to ALOOK. The MersenneTwister application has temporal locality but exhibits drastic changes in computational values over its run. As shown in Figure 3.10, a static design using approximate matching provides 5% hit rate, while ALOOK can approximate 70% of operations. Despite 65% difference in hit rate, our adaptive approach produces less than 10% overall output error.

Figure 3.11 shows the hit rate and efficiency improvement of a GPU enhanced with a static table [2, 59] compared to ALOOK as the number of entries increases from 16 to 128 rows. We allow up to 5% error on individual matches. Before the application is run, the static table loads N pre-profiled pairs and these values remain fixed for the run duration. ALOOK does not require pre-profiled data, instead of updating entries on search misses allowing it to provide higher hit rates. Based on our results the best average energy improvement occurs when ALOOK has 32 rows, with an average energy improvement of 2.7× for the eight tested applications. The higher activation rate of larger tables is negated by the increased energy needed to search the additional rows. Our evaluation shows that for same sized tables with similar computational accuracy, ALOOK provides an average of 2.1× better energy savings compared to a static table [2].

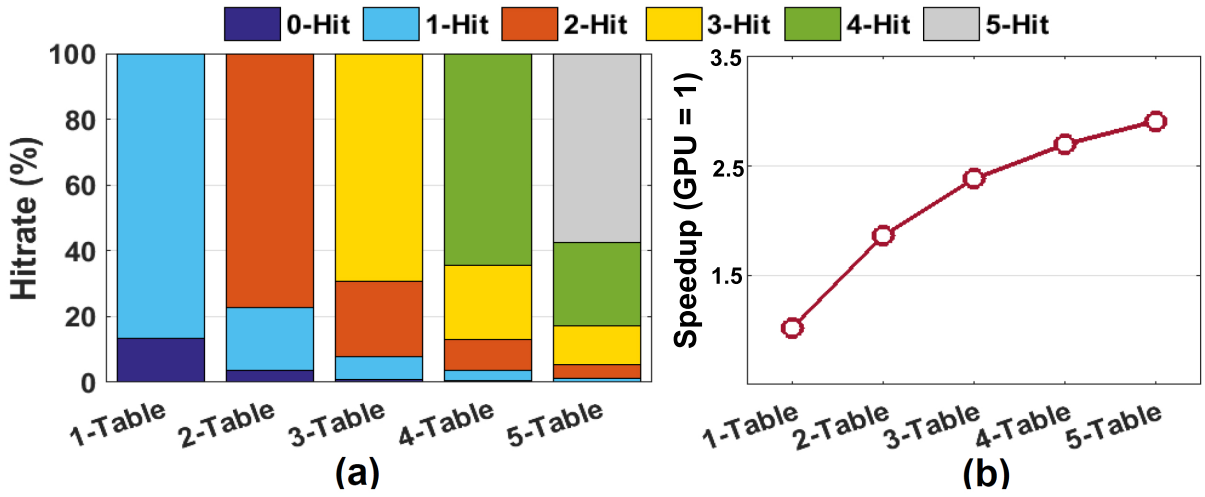


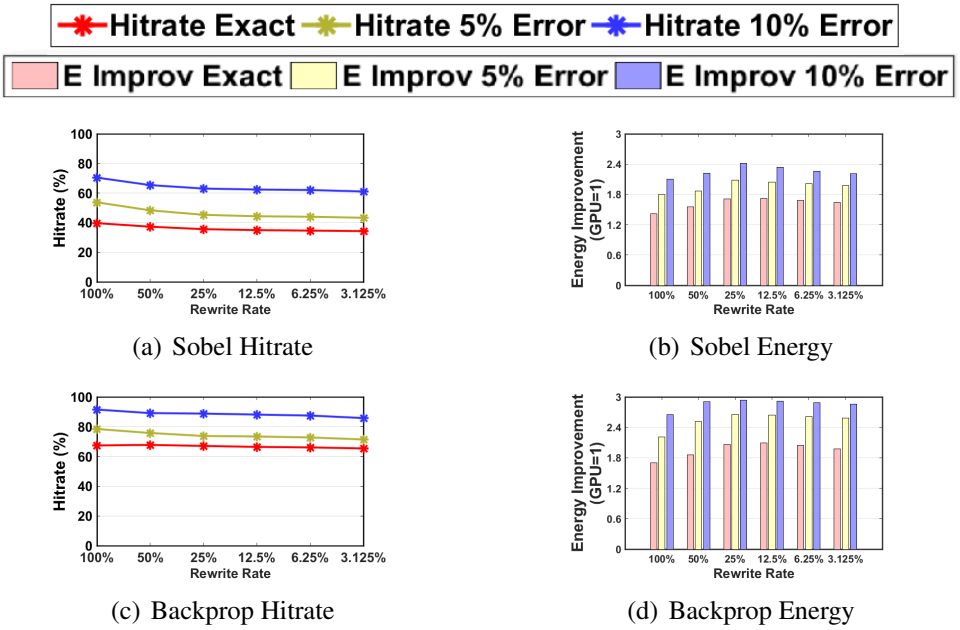
Figure 3.12. Comparison (a) ratio of hits for increasing the number of tables and (b) the speedup improvement provided for *MatrixMul*

3.4.3 Multi-Table Parallel Lookup

Using more than one table allows CFPU to accelerate applications. We examine the trade-offs associated with increasing the number of lookup tables. More tables increase the area overhead and power consumption of CFPU. Figure 3.12 shows acceleration as the number of tables increases for the *MatrixMultiply* application with less than 5% error. The application using one table has an overall hit rate of 86%. If five tables are used, 49% of checks result in all five tables identifying a close match, while the remaining have 4 matches or fewer. Each miss must be computed on exact hardware and prevents speedup. Additional tables provided decreased benefits to performance as the percentage of operations making use of all tables decreases.

3.4.4 Variable Rewrite

When a good match is not detected in ALOOK, the value must be computed on exact hardware and written to the CAM. Our design needs to have enough data entries so the effective hit rate overcomes the energy overhead for write operations. Increasing the number of entries shows diminishing returns for hit rate improvement and also raise the energy and time required to search the table. Figure 3.13 shows ALOOK hit rate and energy saving for decreased rewrite rates.



(a) Sobel Hitrate

(b) Sobel Energy

(c) Backprop Hitrate

(d) Backprop Energy

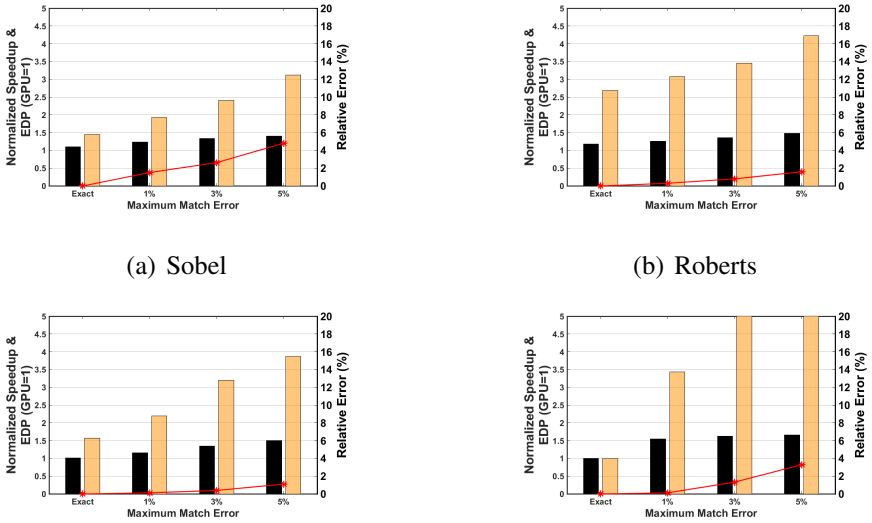
Figure 3.13. Hitrate and energy improvement as rewrite rate is decreased for Sobel and Backprop applications

Initially, a write occurs for every search resulting in a poor match. As rewrite rate decreases, the energy savings levels out and then begins to rise slightly. Decreasing rewrites past a point results in the energy savings from decreased matches impacting energy more than saving energy from fewer writes. Based on our results, then it is most efficient to rewrite once for every 8 misses, roughly 12% of misses.

3.4.5 Accuracy-Energy Trade-off

In this section, we discuss the impact of hardware approximation when trading accuracy for energy savings. In section 3.3.3 we discuss that ALOOK has the capability to control maximum match distance to control the level of approximation. Accuracy is controlled by setting the maximum acceptable distance of input data to the stored value in ALOOK. If the closest distance match error is more than the specified maximum, the input data is sent to precise FPUs to process. We adjust the maximum error distance to trade energy and accuracy in different applications. Figure 3.14 shows the energy-delay product (EDP) and performance speedup

which each application can achieve when running different applications using an ALOOK with 32 rows. As the error distance increases, more values are computed using ALOOK resulting in better speed up and more energy savings. The overall error is represented by the red line. In the eight tested applications, ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, with less than 5% output error.



(a) Sobel (b) Roberts

(c) NN (d) Kmeans

Figure 3.14. Normalized EDP and performance speedup of ALOOK enhanced GPU for increasing maximum error distances.

Figure 3.15 shows the accuracy of the K-means application when the ALOOK is in exact and approximate modes. For clustering algorithms, the accuracy is determined by identifying the number of incorrectly grouped points. The points wrongly classified by the approximate hardware occur at the boundaries of two clusters.

3.4.6 Comparison

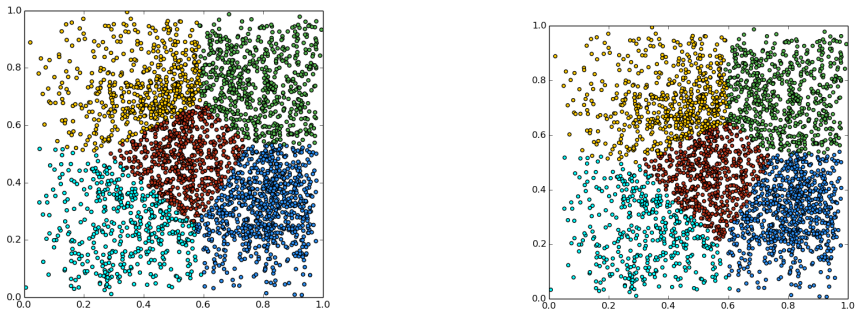
We compare the efficiency of the ALOOK with two state-of-the-art approaches enabling computational reuse in GPU architecture. First, ReCAM [9], which utilizes a static but configurable table to enable approximation, and second ACAM [58] which uses online learning to fill the lookup table values during runtime. We have implemented both ReCAM and ACAM

approaches in their best configurations which results in maximum energy saving. All evaluations have been performed using 32-row table and with 5% maximum quality loss. Table 3.2 compares the energy-delay product improvement of different designs as compared to conventional GPU architecture. Our evaluation shows that ALOOK can achieve 2.9× and 2.0× higher EDP improvement as compared to ReCAM and ACAM respectively. The higher ALOOK efficiency comes from (i) the low hit rate of the static table in ReCAM and the significant cost of the online learning algorithm to update the lookup table values in ACAM. (ii) ALOOK is capable of speeding up the GPU computation, while ReCAM and ACAM work with the same performance as conventional GPU.

3.4.7 Overhead

We modify four main floating point units in GPU architecture by duplicating their first stage and adding an associative memory next to them. The tested FPUs utilize deep pipelines with 23 stages, so the duplication of the first stage adds less than 4.5% area overhead to a conventional FPU. In this chapter, we exploit a crossbar lookup table that can be integrated at the top of FPUs with minor area overhead. Our evaluation shows that the peripheral circuits which enable the nearest search operation adds an extra 0.3% area overhead to the FPU (for a table with 32 rows).

The ALOOK search operations happen in a single cycle and in parallel with the FPU's computation, thus ALOOK does not add any performance overhead to the GPU. However, a miss in ALOOK adds 5.7% energy overhead to FPU operations, since we still need to pay the cost of FPU to process such data. To ensure no energy overhead of a particular GPGPU application, ALOOK needs to produce a hit rate of 17%. This hit rate is much lower than the numbers we saw in the tested applications.



(a) Exact (b) Approx

Figure 3.15. Output quality comparison for *K-means* application running on (a) exact hardware, (b) ALOOK in approximate mode resulting in 2.9% error.

Table 3.2. EDP improvement of ALOOK and other approximate approaches on GPGPU with 5% maximum quality loss.

	ReRAM [9]	***ACAM*** [58]	**Alook**		***ReRAM*** [9]	***ACAM*** [58]	**Alook**
Sobel	1.1×	1.3×	2.6×	***BlackScholes***	0.9×	1.6×	3.8×
Robert	1.2×	1.6×	4.0×	***NN***	1.6×	1.9×	3.5×
Sharpen	1.4×	1.9×	6.3×	***Kmeans***	1.2×	2.6×	5.9×
Merrnse	0.9×	1.3×	2.7×	***Bakcpropag***	1.7×	2.4×	3.5×

3.5 Conclusion

Associative memory in form of lookup table can decrease the energy consumption of the parallel processor by exploiting data locality and reducing the number redundant computation. We propose an adaptive associative memory, called ALOOK, which accelerates GPGPU computation by searching for the nearest distance value for incoming FPU operations. ALOOK consists of a small dynamic lookup table which adapts over time to an application. Our evaluation shows that ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. When both ALOOK and CFPU are used in conjunction, average EDP improves to 5.6×.

Both ALOOK and CFPU can be used to reduce energy of many applications. While ALOOK provides speedup, in many applications over 50% of the warps have threads running in both approximate and exact mode. The cores within the warp must remain in lockstep, so these

mixed warps are not accelerated. The next chapter addresses methods for avoiding bottlenecks and accelerating warps within GPUs.

Chapter 3 contains material from ”ALook: Adaptive Lookup for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapter 3 contain material from ”Resistive CAM Acceleration for Tunable Approximate Computing”, by Mohsen Imani, Daniel Peroni, and Tajana Rosing, which appears in IEEE Transactions on Emerging Topics in Computing (TETC), 2016. The dissertation author was a primary instigator and the second author of this paper.

Chapter 3 contains material from ”ARGA: Approximate Reuse for GPGPU Acceleration”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which appears in IEEE Design Automation Conference (DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapter 4

Warp level approximation

Our previous chapters described methods for approximating operations within GPU cores, but do not consider how these cores work in conjunction with each other. Instructions on GPUs are issued in groups called warps or wavefronts. In Nvidia and AMD architectures, these groups must wait until all operations are completed before processing the next task. This means individual operations can be sped up, but if a single operation must be computed using the exact hardware, then there is no performance benefit. Prior work [**?**, 2, 74] offers reductions to energy but their naive implementations lack the necessary load balancing required to improve warp performance.

In this chapter, we propose AWARP, a novel approximate computing architecture for accelerating warps run on GPUs. AWARP predicts the approximation error based on incoming operations and assigns the most inaccurate to compute in exact mode. *Warp passthrough* allows groups of threads with a large majority of hits to be accelerated with a minor penalty to output accuracy. We apply *warp value trading* to ensure approximate operations are grouped together in warps which can then be sped up. To estimate power we use a cycle-accurate simulator, Multi2sim, to implement our design on an AMD Southern Island 7970 GPU. We test 6 GPGPU applications and show our design speeds up the applications by an average of 1.8× and improves EDP by up to 5.7× for less than 5% application error compared to the unmodified GPU. We also test AWARP by modifying Tensorflow, then running LeNet and ResNet neural networks using

approximation on an Nvidia 1080 GPU. AWARP speeds up the inference of the networks by up to 2.4× with less than 1% loss in classification accuracy.

4.1 Related Work

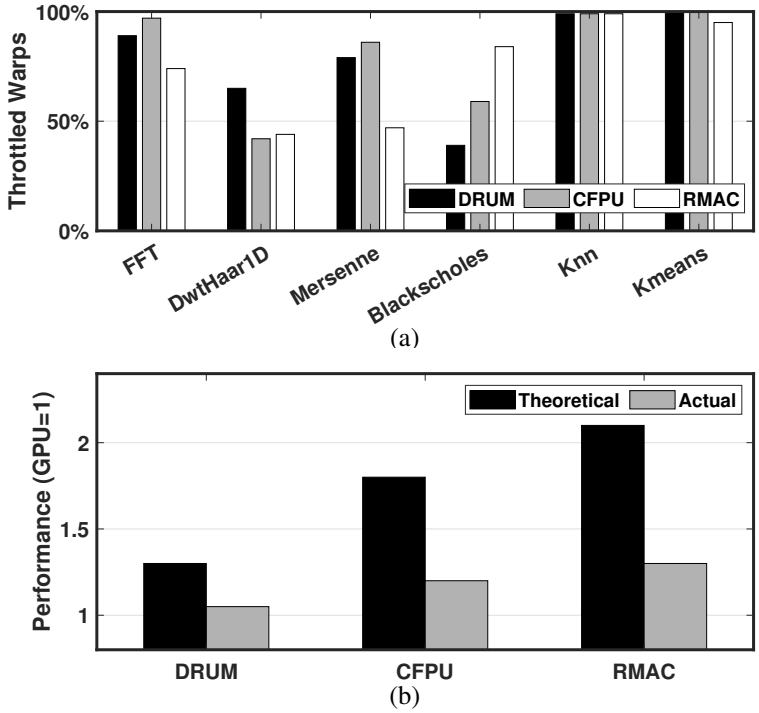


Figure 4.1. (a) The percentage of warps which are bottlenecked by exact operations during approximation and (b) the theoretical and realized performance of a NN sped up by approximate hardware

Many GPGPU applications, such as neural networks, involve many FPU multiplies. Floating point involves several sub-computations, the most expensive being mantissa multiplication. Our previous design, CFPU, proposes a floating point multiplier which avoids computing the resultant mantissa by using one of the input mantissa directly and dropping the other. RMAC [4] uses addition in order to achieve better accuracy. These designs add a small decision circuit to the FPU to enable it to compute in an approximate or exact mode. The error for operations can be predicted by examining the input values. This allows a tuning parameter to be set to control computation accuracy by running a portion of results in the exact mode. Although the addition in

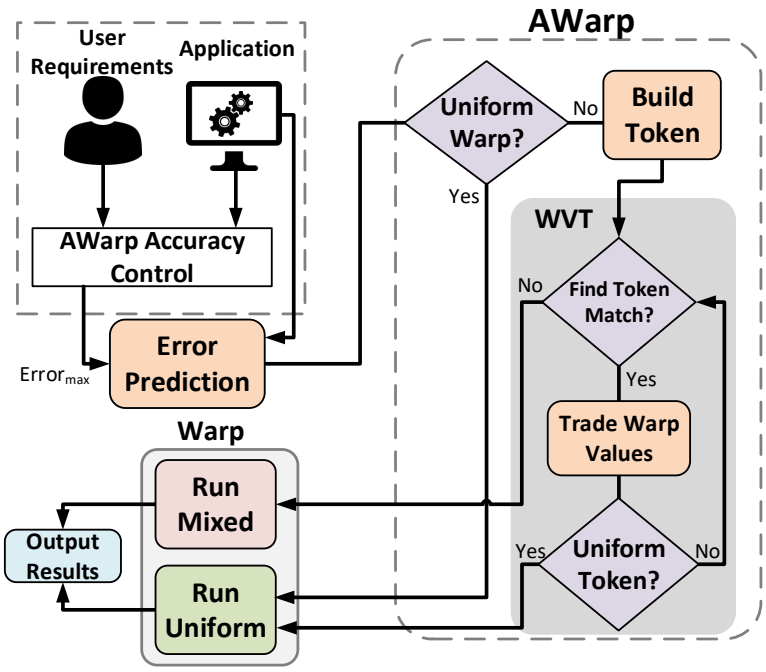


Figure 4.2. Overview of how AWARP takes user settings and approximates warps

RMAC is more computationally expensive than copying a mantissa in CFPU, fewer operations need to be run in exact mode. This results into better overall energy savings for the same level of accuracy. While these approaches provide good energy improvements for GPGPU applications, they do not accelerate applications well. The designs are implemented naively leading to warps running both approximate and exact computations simultaneously. Instructions are issued in warps of 32 threads, so a single operation run in exact mode bottlenecks the entire group. Prior work has proposed approximate warps which disable some threads in warps and use results from adjacent cores as output [75, 76]. This design does not speed up individual warps as it still requires exact computation on the remaining cores.

In this work, we remedy the shortcomings of prior work for approximating GPU computation. Our design, AWARP, is a framework which speeds up warps through approximate computation. AWARP groups safe to approximate operations together to be issued together in approximate warps allowing flexible accuracy control without sacrificing performance.

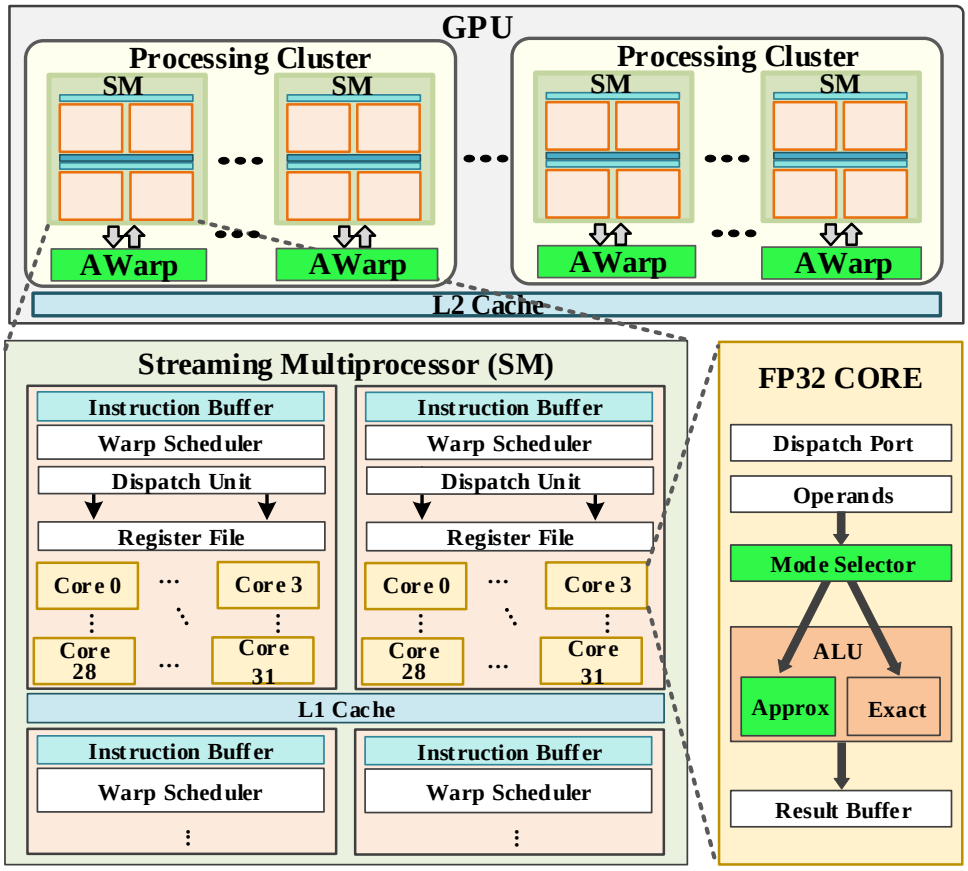


Figure 4.3. Implementation of AWARP within an Nvidia Pascal GPU

4.2 AWARP Architecture

Our design, AWARP, is a framework for integrating approximate arithmetic units into GPUs while enabling significant performance improvements at warp level. We propose a novel method of balancing data between warps to allow improved performance while ensuring a high level of user control over application accuracy.

4.2.1 AWARP Framework

Applications can have a wide range of accuracy requirements, so AWARP allows users to adjust the approximation rates of applications with high granularity. AWARP identifies operations with the highest error and runs them in exact mode.

When using an approximate multiplier, a GPU has several options for controlling accuracy.

The first option is to approximate all operations equally. This is not optimal as approximation error is not uniform for all inputs and can lead to high error in the final output. It also requires hardware level accuracy adjustment which may be difficult or impossible at run-time. The second method is to predict the most erroneous operations based on the inputs and run them in an exact mode. The ratio of approximate to exact operations can be easily adjusted over the course of an application for different sections of code based on user requirements. In theory, computing the highest error operations exactly maximizes the energy savings and performance improvements compared to output error. However, in GPUs operations are grouped into warps and in a naive implementation, a thread selects approximation mode without considering its peers in the warp. As shown in Figure 4.1a), many warps are throttled if a naive implementation is used for state-of-the-art approximate multipliers. These applications have a high potential for performance improvements based on the number of multiplies approximated, but these gains are unrealized as the warps cannot be sped up as long as at least one runs in exact mode. Figure 4.1b) shows the theoretical and actual performance improvements for the approximation of a LeNet [77] neural network during inference.

Figure 4.2 provides an overview for how AWARP functions. A user controls accuracy by adjusting $Error_{max}$ value. The upper bits of incoming operands are checked to predict output error. The more bits that are checked, the lower the error guaranteed by AWARP. Any operations which are identified to produce outputs with error larger than $Error_{max}$ are instead run in an exact mode using the original FPU multiply. The accuracy check metric is selected based on the approximate multiplier being used [3, 4]. After examining each set of operands the warp determines if it can be run uniformly in exact or approximate mode. If not, it constructs a token which is used to find other compatible warps with which operands can be rearranged. The process continues until the warp can either be run in a uniform mode or cannot find another compatible token.

Figure 4.3 shows the integration of our design within an Nvidia Pascal architecture. The Nvidia 1080 contains 4 Graphics Processing Clusters (GPC) each with 20 Streaming

Multiprocessors (SM). Each SM contains 128 computational cores grouped into sets of 32. The SM contains the thread scheduler and dispatcher to assign warps to cores. We utilize several modules in our design. First, AWARP is placed inside each GPC with communication to the SMs. AWARP collects information about a warp through the warp scheduler and compares it to other warps running the same instruction. The inputs for the multiplies which can be approximated are traded through AWARP from the warp with the lowest number to the one with the most. After each trade one of the two warps will be entirely approximate or entirely exact operations and can be safely issued. At the lowest level, we utilize approximate floating point multipliers to enable approximation. AWARP is not specific to a single approximate multiplier, and we test implementations of three different designs [5] [3] [4].

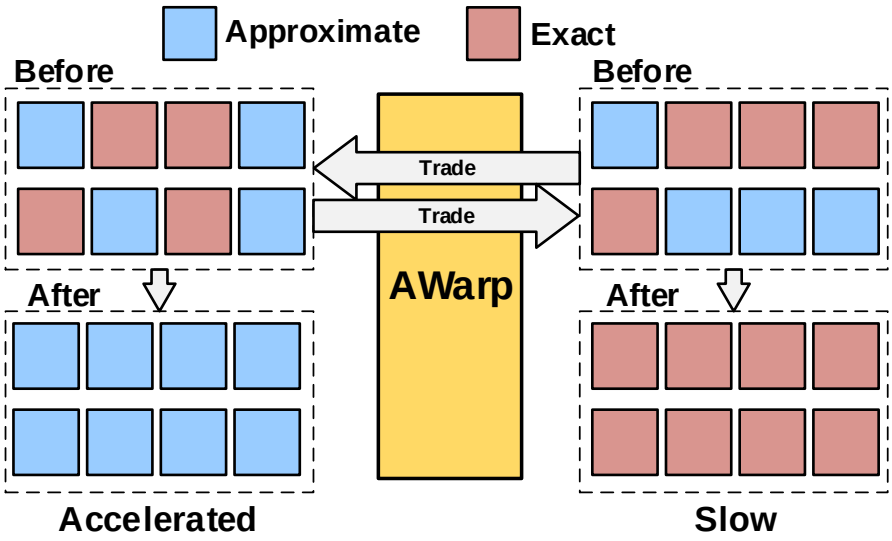


Figure 4.4. Example of *warp value trading*. AWarp organizes operations to maximize speed up in warps

4.2.2 Warp Passthrough

In GPU, instruction threads run in groups called warps. Each core is assigned the same instruction and all operations run in lockstep on a warp. This creates a problem where a single poor match in one of the cores may prevent the entire warp from accelerating that instruction. This is highlighted in Figure 4.6a, where 3 of the 16 cores bottleneck the operation. Here,

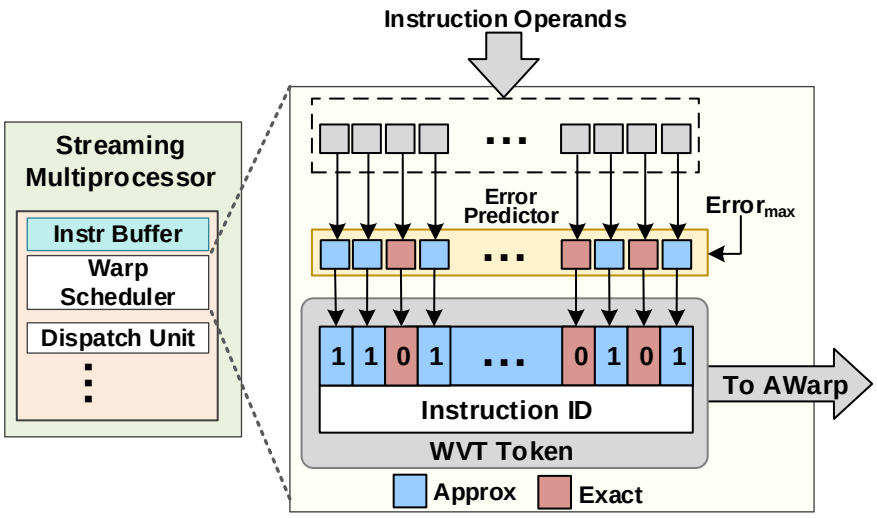


Figure 4.5. Tokens generation process in AWARP

individual operations have a higher error than the rest in the warp, but the overall warp error remains low. These operations are computed on exact hardware, increasing energy costs and preventing application acceleration. In order to accelerate the entire warp, all operations must be accelerated. In a tested application, *ScalarProd*, we find that although 86% of the operations are approximated, 70% of the warps are bottlenecked and not accelerated.

We propose warp passthrough, a scheme which identifies warps with high overall match rates and accelerates them in spite of a few poor matches. Figure 4.6b shows the same process in which passthrough is enabled. Instead of waiting on a small number of threads to compute on the FPU, all threads in the warp use the result produced by the approximate lookup. This, combined with multi-table parallel lookup, enables AWARP to accelerate applications. As shown in Figure 4.7 each core in a warp outputs a signal identifying the result produced by the lookup table as either a good or bad match. If a majority of cores identified good matches, it outputs a passthrough enable signal back to the cores. The cores with poor matches send the operation to be computed on exact FPU. The results of the exact operations then update the lookup table using an LRU policy to ensure the most recently computed values are populating the table. Finally, the computed results are reordered and placed in the result buffer.

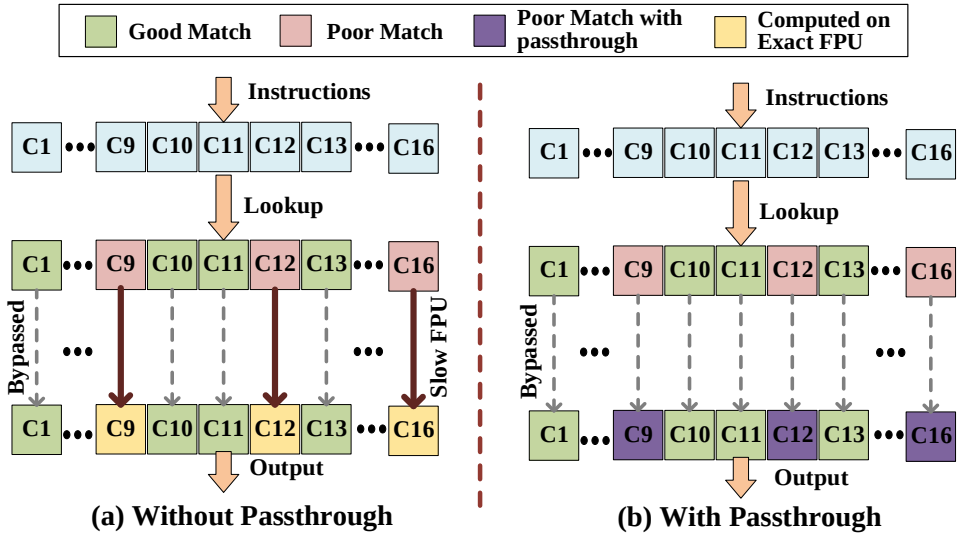


Figure 4.6. Instructions approximated using a lookup table (a) without passthrough [2] and (b) with the passthrough support.

4.2.3 Warp Value Trading

As previously explained, when attempting to control approximation accuracy on GPUs, we ensure all values in a warp are approximated in order to gain performance improvements. We address this through a process we call *warp value trading* (WVT) in which threads are grouped and assigned to warps based on whether they can be run approximately or exactly. As shown in Figure 4.4 AWARP preemptively reassigns operations across warps before they are issued. In the example, each warp contains four operations to be run in approximate mode and four in exact mode. AWarp rearranges the operands creating two uniform warps. Now, one of the warps can be sped up, rather than both being bottlenecked by the exact computations.

AWARP identifies the warp with the most approximate operations and assigns it to send swap exact inputs for the other approximate ones. In this way, the number of approximate threads in the warp is maximized. After the trade, at least one of the two warps will have all 32 threads set uniformly for approximate or exact mode. This warp is then issued to be run on hardware.

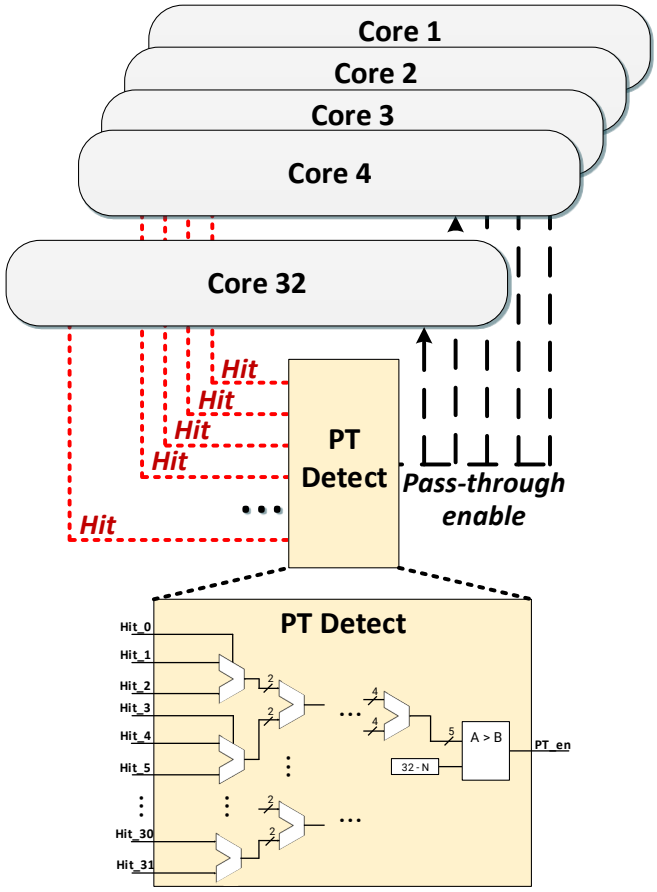


Figure 4.7. Check of match quality across threads in warp to enable passthrough.

4.2.4 Token Generation and Pooling

Prediction error of the operations is essential to ensuring AWARP minimizes application error. We show three different approximate multipliers which can be used in AWARP: CFPU [3], DRUM [5], and RMAC [4]. Each approximate multiplicatier can check a number of tuning bits to predict error.

Floating point is represented in the IEEE 754 standard. In this notation, each FP value has three components. The MSB is reserved for the sign bit, then the remaining bits are split to cover the exponent and mantissa bits. In FP32, 8 bits are reserved for the exponent and 23 are reserved for the mantissa. During a multiply operation, each of the three components of the

result is computed in a different way. The sign bits of the two inputs are XORed together and the exponent values are added. The mantissa bits are multiplied together, which requires the majority of the power and is the longest of the steps. In CFPU [3], the multiply avoided by copying a mantissa directly to the output and discarding the other. This results in an error of up to 50%. In RMAC [4], the multiply is replaced with an addition in order to approximate the result. RMAC is slower than CFPU for individual operations, but achieves much better accuracy. Using this method the error of the output is at most 11.1%. DRUM truncates the bottom bits of the multiply to accelerate performance, but the truncated multiplication takes more time than the previous two methods as shown in Table 4.2. Tuning bits are used to predict the output error of the different multipliers. Up to N-bits are checked per input based on user accuracy requirements. The more bits that are compared, the more strict the accuracy.

In order to enable *warp value trading*, two sets of warp instructions must be compared in a lightweight manner. To do this, we generate a token for each group of inputs. The inputs for each incoming operation is checked to determine whether it can be approximated based on $Error_{max}$. Shown in Figure 4.5, AWARP builds a 32 bit token with 1 bit per thread in the tentative warp. In the token, a '1' represents exact mode and '0' represents approximate mode. The tokens also contain an instruction ID to ensure the same instructions are compared. When a token is completed, it is sent to AWARP for processing.

4.2.5 AWARP

Before warps for a given instruction are issued, AWARP communicates with the warp scheduler to collect the tokens into a pool to examine. Figure 4.8 shows the architecture of AWARP. When a token arrives in AWARP, it is stored in a token pool to wait for comparison. The pool is large enough for multiple tokens per streaming multiprocessor to be stored. AWARP selects tokens from the pool and attempts to find another sharing the same instruction ID. If no such match occurs, the token is returned and the warp is issued. When a compatible match is found, the two are compared to identify the token with the highest number of approximate ops.

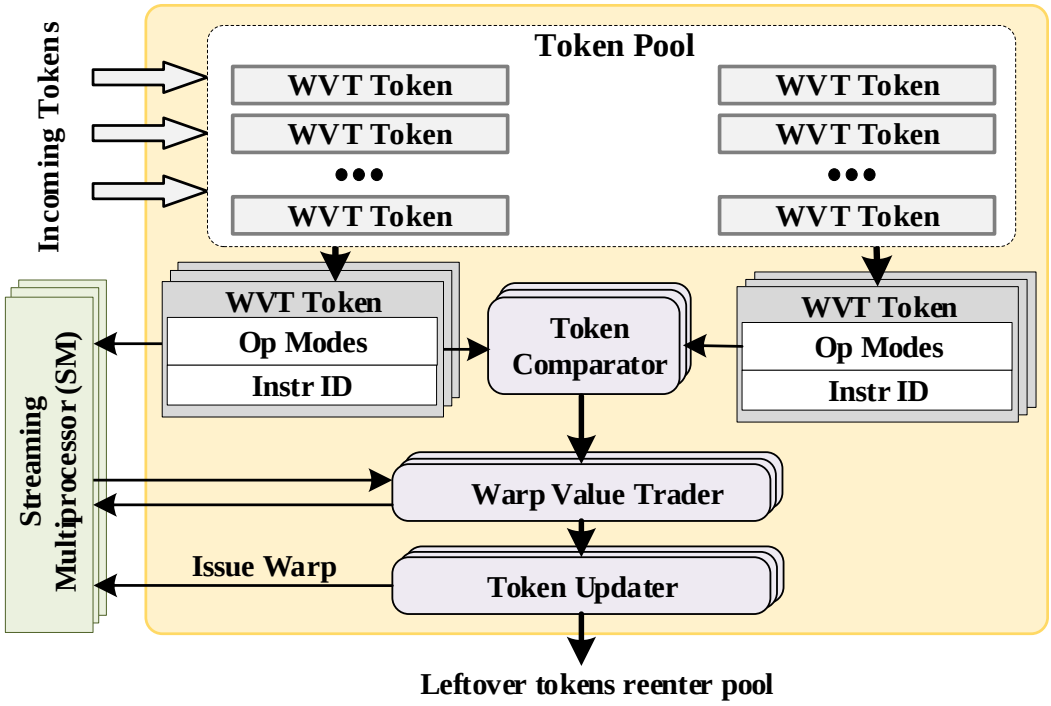


Figure 4.8. Overview of AWARP trading

Then, in AWARP the warp value trader unit enables swaps of values with approximate values migrating to the warp with the previously identified token. Swaps must occur within SM units to utilize the shared memory, keeping overhead low and allowing AWARP to reassign values quickly.

After *warp value trading*, the two tokens are updated. At least one will now contain either 32 exact or 32 approximate threads that can be issued. The remaining token is returned to the pool to be compared again. This process repeats until a single token with the instruction ID remains and is issued as a warp with both exact and approximate computations. This warp runs at the pace of the exact instructions, but will still save energy on any approximate operations it performs. By comparing against all warps running the same instruction, AWARP ensures at most 1 warp will remain with a mix of both approximate and exact operations.

4.3 Experimental Results

4.3.1 Experimental Setup

We simulate AWARP using Multi2sim [78], a cycle accurate CPU-GPU simulator. Multi2Sim is configured to model an AMD Southern Island Architecture 7970 GPU. For approximate multiplication, we use three approximate multipliers, CFPU [3], RMAC [4], and DRUM [5] to test our approximate warp technique. While DRUM is an approximate integer multiplier, we adapt it to FP by truncating the mantissa multiplier. Within the simulated GPU, each FPU multiplier is modified to allow it to perform mantissa approximation or the exact multiplication. To estimate power and performance we simulate AWARP using HSPICE in 45-nm process with a 1V supply voltage.

We test AWARP using 6 OpenCL GPGPU applications. Four of the applications are from AMD APP SDK v2.5: *FFT, Mersenne, BlackScholes* and *DwHaar1D*. On average, over 80% of these operations in these applications involve a floating point multiply. For the image processing applications, we use Caltech 101 for our input data set, while others use randomized input. We also test two applications from the Rodinia 3.1 machine learning benchmark suite [48]: *K-nearest neighbor* and *Kmeans*. For all applications, we use average relative error as the accuracy metric.

We also test the impact of AWARP on CNNs, *LeNet-5* [77] and a ResNet-20 [37], run on an Nvidia 1080 Pascal GPU. We modify Tensorflow to enable approximate multiplication and collect operation statistics to estimate AWARP performance and generate accuracy characteristics. LeNet classifies 28x28 pixel images of hand-written digit characters from the `MNIST` dataset [79]. The *LeNet-5* network is trained with $60K$ training images, and it provides an accurate classification for about 97% of $10K$ tested image samples. We test a ResNet-20 network using the CIFAR-10 image dataset. The pretrained network exhibits a baseline 91.6% classification accuracy when using a testing dataset of $10K$ images.

4.3.2 Warp passthrough

We show the acceleration improvement provided by warp pass through. We start without passthrough enabled and show the accuracy and performance improvements. Figure 4.9 shows the bottleneck of threads for the *ScalarProduct* application. Considered individually, 84% of the operations in the application can be approximated with less than 5% error. If no thread is allowed to violate the error threshold only 30% can be accelerated. By allowing passthrough for a single thread to return a poor match, we can accelerate 65% of threads. Although passthrough violates the amount of accuracy which AWARP ensures, it can be used to significantly accelerate the computation.

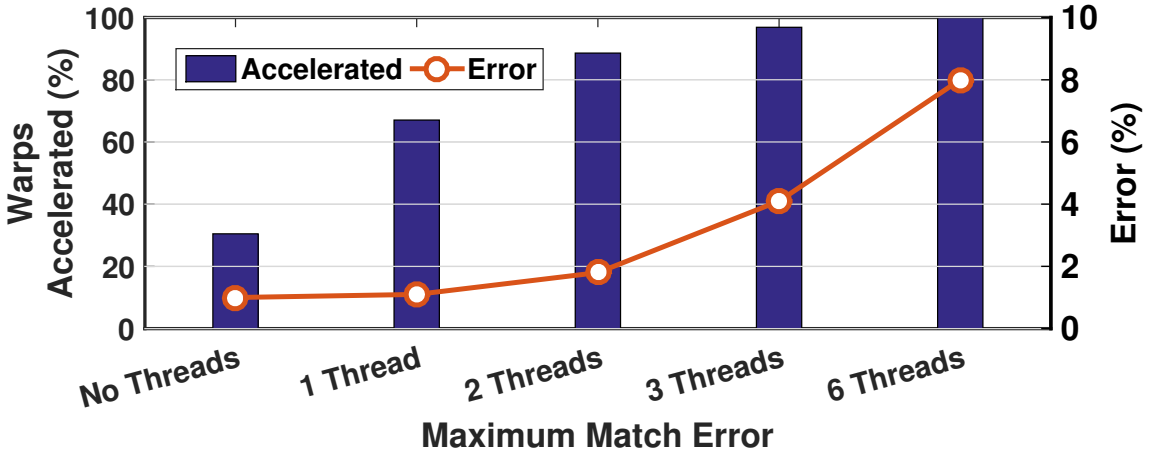


Figure 4.9. Warps accelerated by AWARP using warp passthrough and the change in output accuracy for *ScalarProd*

4.3.3 AWARP Performance

We test 6 different OpenCL applications on a simulated AMD GPU. Figure 4.10 shows the performance for the 6 tests OpenCL applications. The application speedup is normalized against an unmodified GPU and compared to a naive implementation which does not approximate warps. The application error is less than 5%. We show the performance of the three tested multipliers. RMAC can be accelerated more than the CFPU, but DRUM outperforms it in the

FFT and DwTHaar1D benchmarks. The average speedup over the unmodified GPU is 1.45× for CFPU, 1.8× for RMAC, and 1.6× for DRUM. AWarp improves speedup over the naive implementation by 34% for CFPU, 54% for RMAC, and 50% for DRUM. On average RMAC and DRUM approximate computations more accurately, allowing them to be used for a higher percentage of operations than CFPU which provides better overall application speedup.

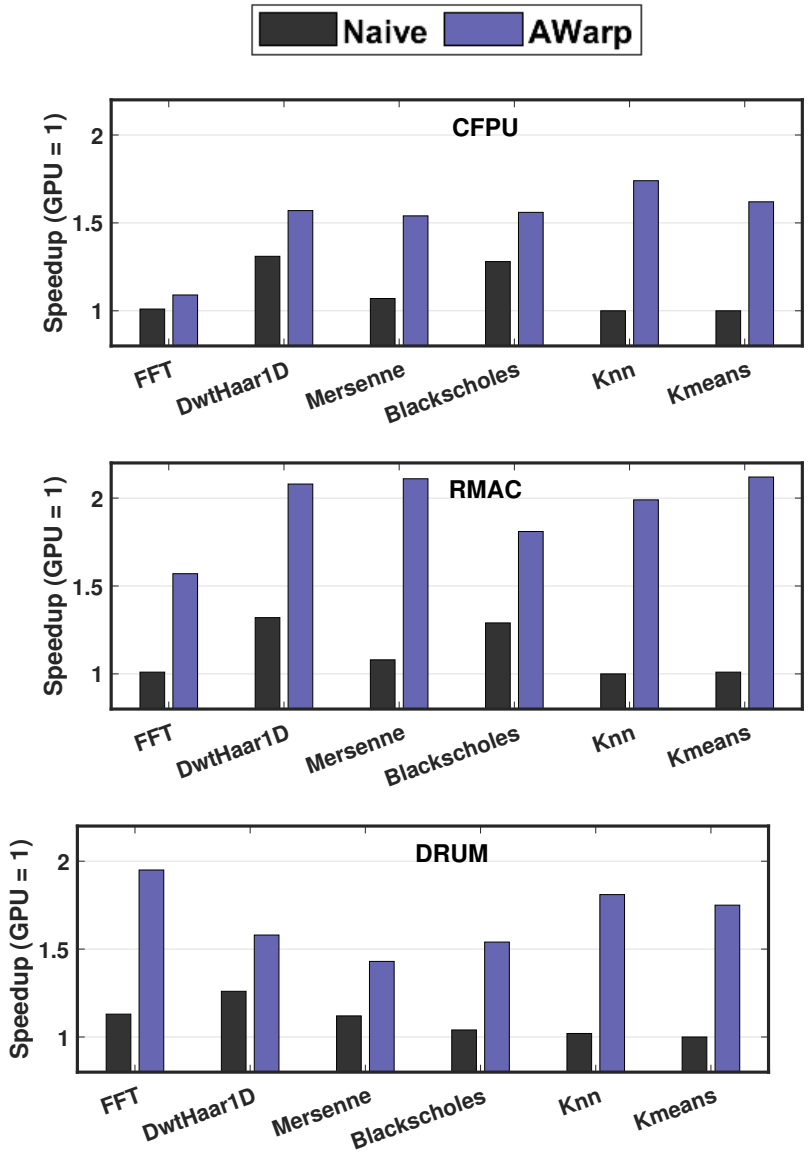


Figure 4.10. The performance improvement from AWARP over naive implementation when using a) CFPU [3] b) RMAC [4] and c) DRUM [5] for less than 5% application error

4.3.4 AWARP Efficiency

AWARP also offers power reduction for the 6 tested applications. Compared to prior, AWARP adds a small energy overhead of 1.4% on average to each operation over the naive implementations. However, the increased speed up offsets this small penalty. Figure 4.11 shows the EDP improvement for the three approximate multipliers across the tested applications. We ensure the maximum relative output error for each application remains less than 5%. We compare the optimization to the unmodified GPU. On average, CFPU has 5.7× EDP improvement, RMAC has 5.1×, and DRUM has 3.1× when application error is less than 5%. While CFPU shows less speedup improvement than RMAC or DRUM, its mantissa drop approximation requires significantly less power.

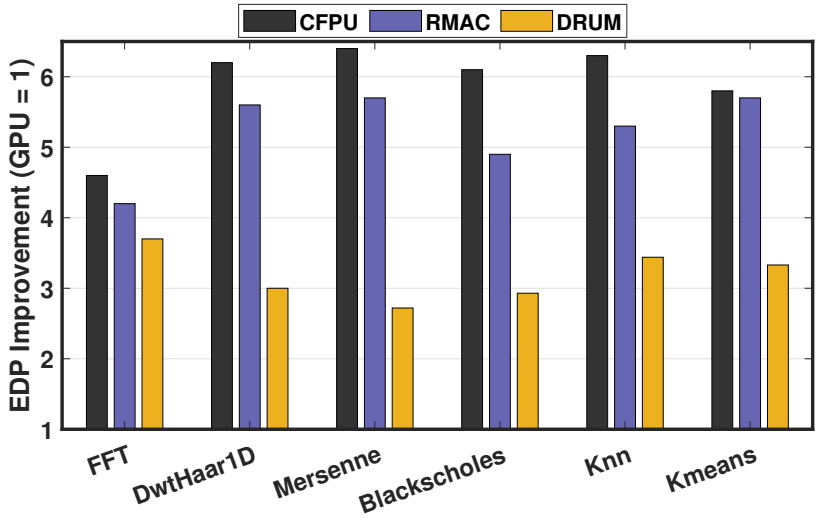


Figure 4.11. EDP improvement when using AWarp with approximate multipliers

4.3.5 AWARP & Neural Networks

Machine learning algorithms like neural networks have a high tolerance for noisy data. Approximate computation can be used to greatly improve the performance of these networks. We test our design on two networks. First, we examine a simple LeNet-5 network classifying the MNIST dataset. We also test a larger Resnet-20 network classifying images from the CIFAR-10 dataset. For this approximation, we show results for the RMAC [4] multiplier. Table 4.1 shows

the percentage of operations that are approximated and the impact on classification accuracy. We adjust the $Error_{max}$ for individual operations which determines the ratio of approximate to exact multiplies. Increasing $Error_{max}$ allows more approximate multiplies, but reduces overall application accuracy. Figure 4.12 shows the speedup improvement as more multiplies are approximated. The LeNet is smaller and tolerates noise better for classification. More operations can be approximated with only a minor impact on accuracy. The ResNet, on the other hand, appears to be more sensitive to noise and quickly degrades as $Error_{max}$ increases. For the LeNet, AWARP is able to achieve 2.4× speedup improvement with less than 1% classification accuracy. AWARP can achieve 1.5× speedup for the ResNet with only 0.2% degradation in classification accuracy.

Table 4.1. Percentage of multiplies approximated and classification accuracy in neural networks using AWARP

Error Rate		**0%**	**0.78%**	**1.61%**	**3.12%**	**6.25%**	**12.5%**
LeNet	*Hitrate*	65.9%	74.1%	77.8%	82.8%	90.0%	100%
	Accuracy	97.4%	97.4%	97.4%	97.3%	96.9%	95.2%
ResNet	*Hitrate*	45.8%	52.4%	61.7%	76.1%	82.4%	100%
	Accuracy	91.6%	91.6%	91.4%	90.8%	84.2%	76.4%

Figure 4.12. Inference speedup for a LeNet-5 running MNIST and a ResNet-20 running CIFAR on Nvidia 1080 GPU

4.3.6 Comparison

In this section we compare our results to state-of-the-art-work. There are several designs which used computational reuse for improving energy efficiency of GPU, but these do not speedup the computation because they check one operation per cycle. In the event of a hit, these designs clock gate the pipeline saving power, but do not accelerate computation. In contrast, our design not only accelerates the applications , but provides the improved power efficiency. For example, comparing AWARP with A^2M^2 [80] and NNCAM [81] shows that AWARP achieves significant energy savings. Figure 4.13 compares our results to these designs.

Table 4.2. Specification of different approximate multipliers at 6.3% maximum error rate.

Multipliers	Area(m^2)	Energy (pJ)	Execution (ns)
DRUM [5]	7915.3	0.28	0.72
CFPU [3]	7951.4	0.09	0.34
RMAC [4]	7819.2	0.15	0.52

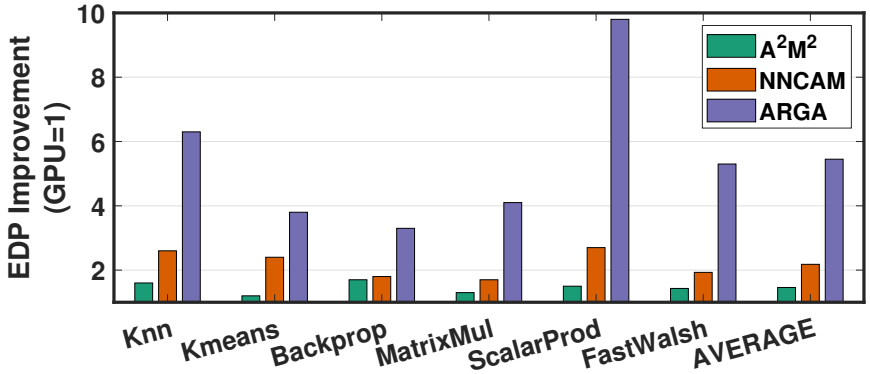


Figure 4.13. EDP improvement provided by AWARP compared to prior work

For the tested benchmarks, work in [80] [81] achieve on average 1.5× and 2.2× energy-delay product (EDP) improvements as compared to the unmodified GPU, while AWARP achieves an 5.6× improvement over baseline GPU. Overall, AWARP achieves an average of 2.5× higher EDP improvement while allowing 5% overall error for the tested applications compared to the prior state-of-the-art-work.

4.3.7 Overhead

Implementing AWARP requires two primary modifications to the GPU. First, the approximate multipliers increase the area of the FPU unit. Table 4.2 shows the overhead for each multiplier. CFPU adds 3.4% area overhead to the conventional FPU, RMAC adds 1.7% area overhead to the FPU and DRUM adds 2.9%. AWARP itself also increases the overhead of the system. To implement, we require memory to store the tokens, along with comparator circuits to check the tokens and enable warp value trading. The circuitry requires an area overhead of 1.2% compared to the base GPU to be implemented. The AWARP unit increases energy overhead by

1.4%.

4.4 Conclusion

In this chapter, we present AWARP, a framework for approximating and accelerating GPU warps. Reducing the complexity of multiply operations offers energy savings, but naive implementations have many throttled warps. We use warp pass through to minimize single threads bottlenecking warp acceleration. By using warp value trading, we are able to trade operation inputs across warps before they execute to reduce computational bottlenecks. AWARP can utilize different approximate multipliers to offer greater flexibility. We use a cycle-accurate simulator, Multi2sim, to implement our design on an AMD Southern Island 7970 GPU. We test 6 GPGPU applications and show our design speeds up the applications by an average of 1.8× and improves EDP by up to 5.7× for less than 5% application error compared to the unmodified GPU. AWARP speeds up the inference of LeNet by 2.4× and ResNet by 1.5× with less than 1% loss in classification accuracy.

AWARP, ALOOK, and CFPU combined provide both acceleration and energy savings for many applications. Our approximate hardware provides flexible error control to allow users to meet accuracy requirements. Further optimizations can be made to target specific classes of applications. In the next section we apply our work to maximize approximation of neural networks during both training and testing.

Chapter 4 contains material from "ARGA: Approximate Reuse for GPGPU Acceleration", by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which appears in IEEE Design Automation Conference (DAC), 2019. The dissertation author was the primary instigator and author of this paper.

Chapter 4 contains material from "Data Reuse for Accelerated Approximate Warps", by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which

currently is being prepared for submission for publication. The dissertation author was the primary instigator and author of this paper.

Chapter 4 contains material from ”Warp Level Approximation for GPU Acceleration”, by Daniel Peroni, Mohsen Imani, and Tajana Rosing, which was submitted for publication. The dissertation author was the primary instigator and author of this paper.

Chapter 5

Approximating Neural Networks

5.1 Introduction

Certain applications, such as neural networks, can leverage additional optimizations to maximize their utilization of approximate hardware. In this chapter, we utilize techniques from chapters 2-4 to approximate neural networks (NNs) along with several further enhancements specially for enhancing training and inference of NNs. Neural networks are very effective for image processing, video segmentation, detection and retrieval, speech recognition, computer vision, and gaming [13, 35, 36]. NNs exploit learned knowledge to deal with data which they have not previously encountered. Although NNs can outperform many other machine learning models, they require enormous resources to be executed. Many NN applications need to update their model at run-time in order to adapt to the environment or enable a personalization. For instance, in speech recognition, NNs personalize as a function of the user's context or accent [38]. Due to limited processing resources and power budgets, training and testing NNs has not been done on constrained embedded devices.

Most current computing systems deliver only exact solutions at high energy cost, while many algorithms, such as neural networks, do not require exact answers, due to their statistical nature [14, 39, 40]. Slight inaccuracy due to enabled HW approximation in neural networks often results in little to no quality loss. Prior work attempted to accelerate neural network by enabling approximation [11, 36, 82–87]. These prior designs are application specific, as the hardware could

not adapt the level of approximation at run-time. Moreover, these designs enable approximation on all input data regardless of their sensitivity to approximation, potentially yielding less accurate overall results that might be possible otherwise.

In this chapter, we propose DRAAW, a configurable approximate computing platform which significantly accelerates neural networks in both training and inference phases by exploiting their stochastic behavior. For training, we propose a Gradual Training Approximation (GTA) which significantly accelerates neural network computation, while providing a desirable quality of service. GTA starts training from deep approximation, and gradually reduces the level of approximation as a function of NN internal error, until the accuracy is sufficient. We use a hardware configurable floating point unit (FPU) which can tune the level of approximation at runtime. We also discuss *magnitude sensitive accuracy control*. In applications such as neural networks, larger computations impact accuracy more. We propose a novel error control scheme which accounts for both magnitude and error distance to maximize the number of approximated operations while minimizing the overall output error. Our accuracy control identifies match quality by using the most significant bits of floating point mantissa values. We examine the exponent bits of the inputs and increase accuracy strictness for larger values.

We provide a methodology to automatically select approximation controls for neural network inference using DRAAW called *neuron aware approximation*. Neural networks tolerate significant noise to their computation before prediction accuracy degrades significantly. However, once accuracy begins to decrease, it falls sharply. Inputs to neurons are summed together so approximation error in larger values impacts the output more drastically. We profile the neuron activations of each layer to identify output distributions which can then be used to maximize the operations approximated with minimal impact on prediction accuracy. We use this to predict safe approximation values for neural networks which can automatically be set at run time based on user selected maximum prediction accuracy loss.

We evaluate the accuracy and the efficiency of our design by integrating CFPU, AWARP, and ALOOK together with our NN specific design proposed in this chapter into AMD's Southern

Island GPU architecture. Our experimental evaluation shows that GTA achieves up to 4.84× (7.13×) energy savings and 3.22× (4.64×) speedup when running four different neural network applications with 0% (2%) quality loss as compared to baseline GPU. DRAAW accelerates inference of the tested neural networks by 2.9× and improves EDP by 6.2×.

5.2 Related Work

Neural networks can be adapted to run on a wide variety of hardware, including: CPU, GPGPU, FPGA, and ASIC chips [14, 83, 88–90]. Because they benefit from parallelization, a significant effort has been dedicated to utilizing multiple cores. On GPGPUs, neural networks get up to two orders of magnitude performance improvement as compared to CPU implementations [31].

Prior works attempted to leverage the stochastic properties of neural networks in order to relax the computation accuracy and improve the implementation efficiency [26, 39, 83, 91]. As shown in [91], implementing neural networks in fixed-point quantized numbers improves performance. Similarly, Lin et al. [82] also examined the use of trained binary parameters in order to avoid multiplication altogether. However, not all applications can handle this approach. Modifications of neural networks parameters during training require higher precision and have difficulties with additive quantization noise [92]. Unlike these works, our design allows the use of full floating point precision, giving it more flexibility when needed.

Han *et al.* [93, 94] investigated the use of model compression in NNs. They trained sparse models with shared weights to compress the model*et al.* [93]. The compressed parameters of [93] are used to design ASIC/FPGA accelerators [94]. Compression fails to improve the implementation in general purpose processors, which require the compressed parameters to be decompressed into the original parameters. Our method is orthogonal to all this previous work, as our design can further reduce power consumption and execution time by enabling gradual and adaptive approximation. In addition, our proposed design uses a general hardware-software

platform which accelerates neural network on CPU, GPU, FPGA, and even ASIC, by enabling configurable FPU approximation.

5.3 Neural Network Acceleration

5.3.1 Neural Networks

Neural networks are computationally intensive. Figure 5.1 details the breakdown their performance. The between 55 to 90% of time is spent within convolutional layers which are primarily multiply adds. DRAAW is a good option for neural networks because they are tolerant to noise [95, 96] and over 90% of the computational time spent involves multiply or multiply add operations. While NNs show resilience to noise applied to the system, once they reach a critical point, additional error compounds into exponential decreases in prediction accuracy. It is necessary to offer fine grain approximation control to maximize benefits from DRAAW. However, the design space to find the optimal network configuration is immense. For example, if there were five error settings per layer, a ResNet152 would have 81 trillion possible accuracy configurations. DRAAW offers far more error control settings, so identifying acceptable settings is essential.

5.3.2 Neural Network in Training & Inference

Figure 5.2a shows the overall structure of a neural network consisting of input, output and hidden layers. The input data dimension and the number of output classes determines the number of neurons in the input and output layers respectively. The number and size of hidden layers depends on the network topology. As Figure 5.2b shows, in neural networks, each neuron is a small processing unit with one or more inputs and a single output. Each input has an associated weight determining the strength of the input data. The neuron simply multiplies inputs with their weights and adds them to calculate an output. Finally, the output value passes through an activation function, which is historically a *Sigmoid* function. Neural networks have training

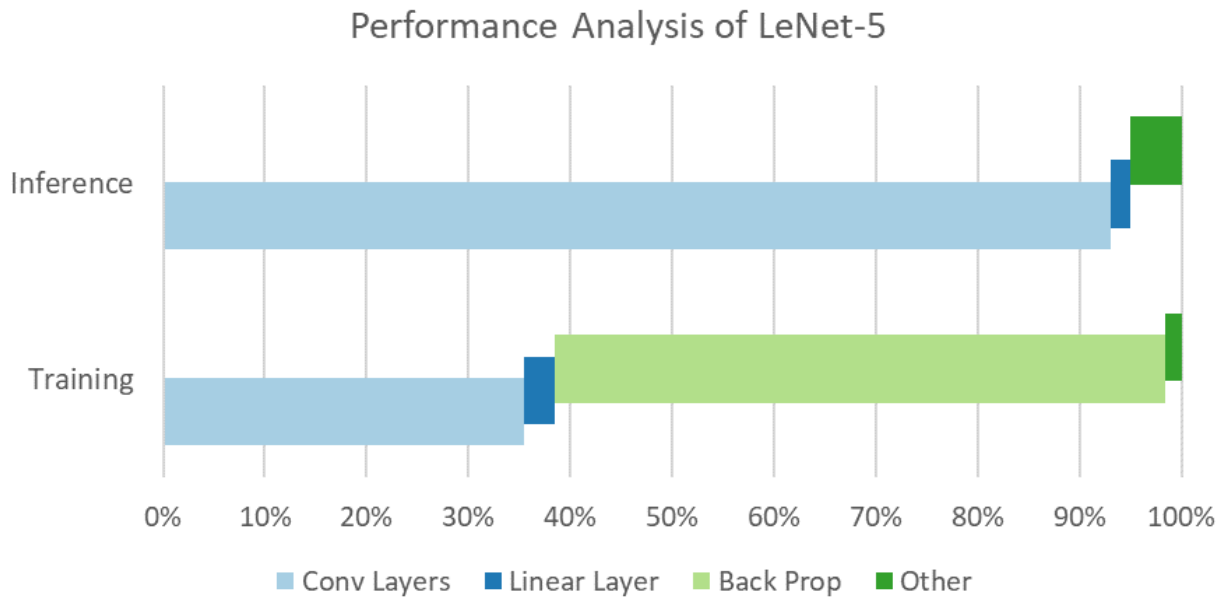


Figure 5.1. Neural network performance breakdown

and testing phases. During the first training iterations, weights and biases are assigned random values. The training phase finds the best weight values which result in maximum classification accuracy. To find such weights, input data (from the training dataset) passes to the network in a feed forward fashion. Based on the errors measured at the output stage, the network adapts the weights and biases values in back propagation mode. When the network is trained, the trained weights and biases can be used to classify the inputs in he dataset. Two fully connected neural network layers have a huge number of multiplications between them. Figure 5.2c shows that these operations can be modeled as matrix multiplication, where each row of the matrix represents the weights corresponding to each neuron. The output of each neuron can be computed as:

$$x^i = f(\sum_k W_k^i * x_k^{i-1} + b^i) \tag{5.1}$$

where multiplications exist between the output of neurons in $i-1^{th}$ layer and the weight matrix in i^{th} layer, W^i, and f is an activation function. Each layer has its own bias vector, b^i. This vector adds to the output signal of each neuron. In back propagation, the i^{th} neural network layer has N inputs and M outputs. The error δ propagates backwards from the output to update the weights

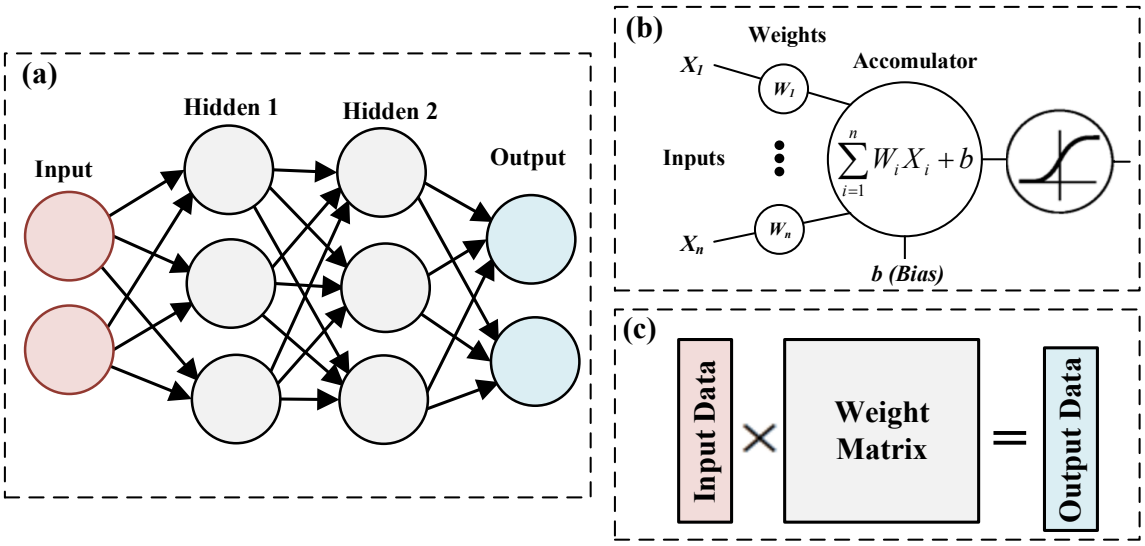


Figure 5.2. (a) Neural network structure with two hidden layers, (b) computing model of each neuron and (c) matrix multiplication representation between two NN layers.

and the bias value. Here is the gradient descent equation for updating the weight and bias values:

$$\Delta W = \eta[\delta \odot h^{'}(W * x + b)]x^{T} \tag{5.2}$$

$$\Delta b = \eta[\delta \odot h^{'}(W * x + b)] \tag{5.3}$$

where x is input to the layer, η is learning rate, and $\odot$ shows the element-wise multiplication. After updating the weights and bias value, *delta* also needs to be updated using:

$$\delta = (W^{T}\delta) \odot h^{'}(W * x + b) \tag{5.4}$$

In Equation 5.2, the outer product of $h^{'}(W * x + b)$ and x is the main multiplication cost. The input x has higher potential for bounding rather than $h^{'}(W * x + b)$, since the second term is determined by the cost function and network parameter. The term $(W * x + b)$ is not computed again in back propagation, as this term was previously calculated in the forward pass and can be

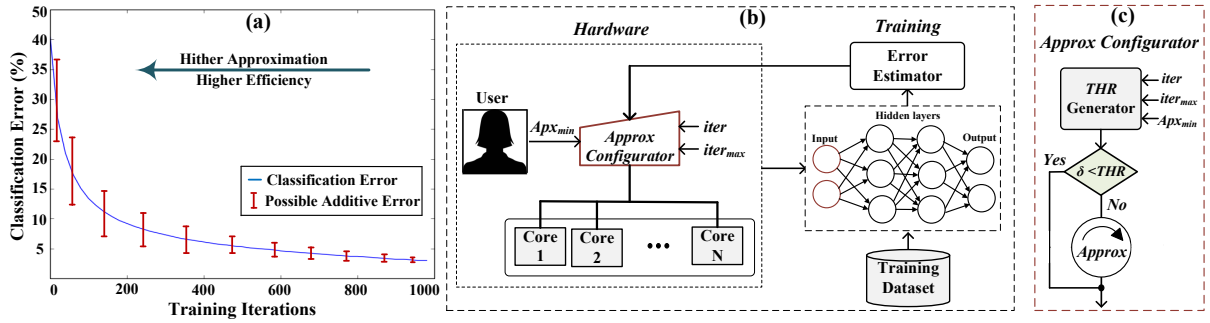


Figure 5.3. (a) MNIST classification accuracy in different training iterations (b,c) GTA framework to accelerate neural network training by enabling Adaptive approximation.

Algorithm 1. Gradual Training Approximation (GTA)

inputs: NN Parameters, Training Data, $Iter_{max}$, Apx_{min}
outputs: NN Trained Model

Initialize weights and biases to random values
Initialize Approx-level

for $iter = 1 \ldots iter_{max}$ **do** out_{iter}= *feed_forward (input)*
$\mathscr{P}$=*error_estimation (out)*
isConveraged($\mathscr{P}$) & isApprox(Apx_{min}) **Break** $approx\_configurator(\mathscr{P}, iter, iter_{max}, Apx_{min})$
$back\_propagate(\Delta W, \Delta b, \delta_l)$

reused. Similar to other machine learning algorithms, neural networks are stochastic in nature, meaning that they accept a part of inaccuracy in their computation. The goal of this chapter is to exploit this scholastic behavior to accelerate both training and inference of a neural network by enabling configurable approximation.

5.3.3 DRAAW Acceleration During Training

Uniform Training Approximation

It is essential to use the precision of floating point units (FPU) for neural network training, as FPUs cover a wide range of numbers appearing during the back propagation. Training neural networks on such approximate hardware accelerates the process while maintaining the desired level of accuracy. The level of approximation is user and application dependent. For example,

for an easy image classification task (e.g. MNIST Handwritten digits [77]), training on hardware with deep approximation might provide the same quality of service that hardware with light approximation can provide over more complex datasets (e.g. *ImageNet* [97]). Thus, there is not a fixed optimal approximation level which is acceptable for all applications. In order to generalize the existing core so it accelerates the neural network during the training phase, we use our approximate configurable FPU. Depending on the running application and its accuracy needs, our framework changes the level of hardware approximation to an optimal level.

Table 5.1 shows the impact of uniform approximation on the quality loss, energy consumption, and execution time of a neural network when evaluating the MNIST Handwritten digit dataset. This network consists of four fully connected layers with 784, 500, 500, and 10 neurons in each layer, respectively. The application classifies handwritten digitsinto ten different classes, 0–9 . Quality loss is an additive error, defined as the classification error of neural network training on full precision and approximate FPUs:

$$\Delta e_{train} = e_{Approx} - e_{FPU}$$

This result shows that increasing the level of hardware approximation does not automatically imply a degradation in classification accuracy. For example, for this network, our design works with the same accuracy as a full range 32-bit FPU, even when it trains with only four configuration tuning bits. At this level of approximation the FPUs are running 33% of the time in approximate mode (Hit-rate=33%), resulting in a training speedup of 2.2× and energy efficiency improvement of 3.2× as compared to the hardware with full FPU precision. Increasing the level of approximation to no tuning bits further accelerates the training by 5.1× and results in 7.9× energy efficiency improvement with 3.2% quality loss.

Table 5.1. Quality loss, normalized energy consumption and execution time of neural network running on GPGPU with different level of approximation (tuning bits).

Approximation	Exact	4-bit	3-bit	2-bit	1-bit	0-bit
Quality loss (Δe_{train})	0%	0%	0.9%	1.3%	1.7%	3.2%
Norm.Energy	1	0.31	0.25	0.21	0.18	0.12
Norm. Execution	1	0.45	0.38	0.34	0.31	0.19

Gradual Training Approximation

Neural network training accuracy changes during the training phase. During the first training iteration, the weights are assigned randomly, resulting in larger classification error. These weights adapt during the training phase using stochastic gradient decent. Figure 5.3a shows the error rate of the MNIST dataset over 1000 training iterations. The red line in this graph illustrates the visual range of approximation that a network can accept during the training. The error reduces significantly during early training iterations, but then saturates. Different error rates while training suggests that uniform approximation is not the best method for approximation because all training iterations do not have the same impact on the final network classification accuracy. During the first iterations, the weights are fairly random, so accepting large approximation should not impact the final classification accuracy. However, during the last iterations, the weights are close to optimal, thus even small hardware approximation may degrade the classification accuracy noticeably.

We propose a gradual training approximation framework, called GTA, which accelerates the neural network training. Our framework, shown in Figure 5.3b, starts the training from the hardware with maximum level of approximation (zero tuning bits). Then, it updates the level of hardware approximation at each iteration, as the network converges. The convergence controls when the slope of accuracy improvement is less than a threshold value, THR. This THR value adaptively changes in our design to ensure that training completes with acceptable accuracy by the final iteration ($iter_{max}$). As Figure 5.3c shows, our framework updates the THR value by using four inputs: (i) the current training iteration $iter$, (ii) the maximum number of iterations

Algorithm 2. Layer-based Inference Approximation

inputs: NN Parameters, Validation Data, Test Data, *QoS*

outputs: NN Layer Configuration

Initialize weights and biases based on trained model

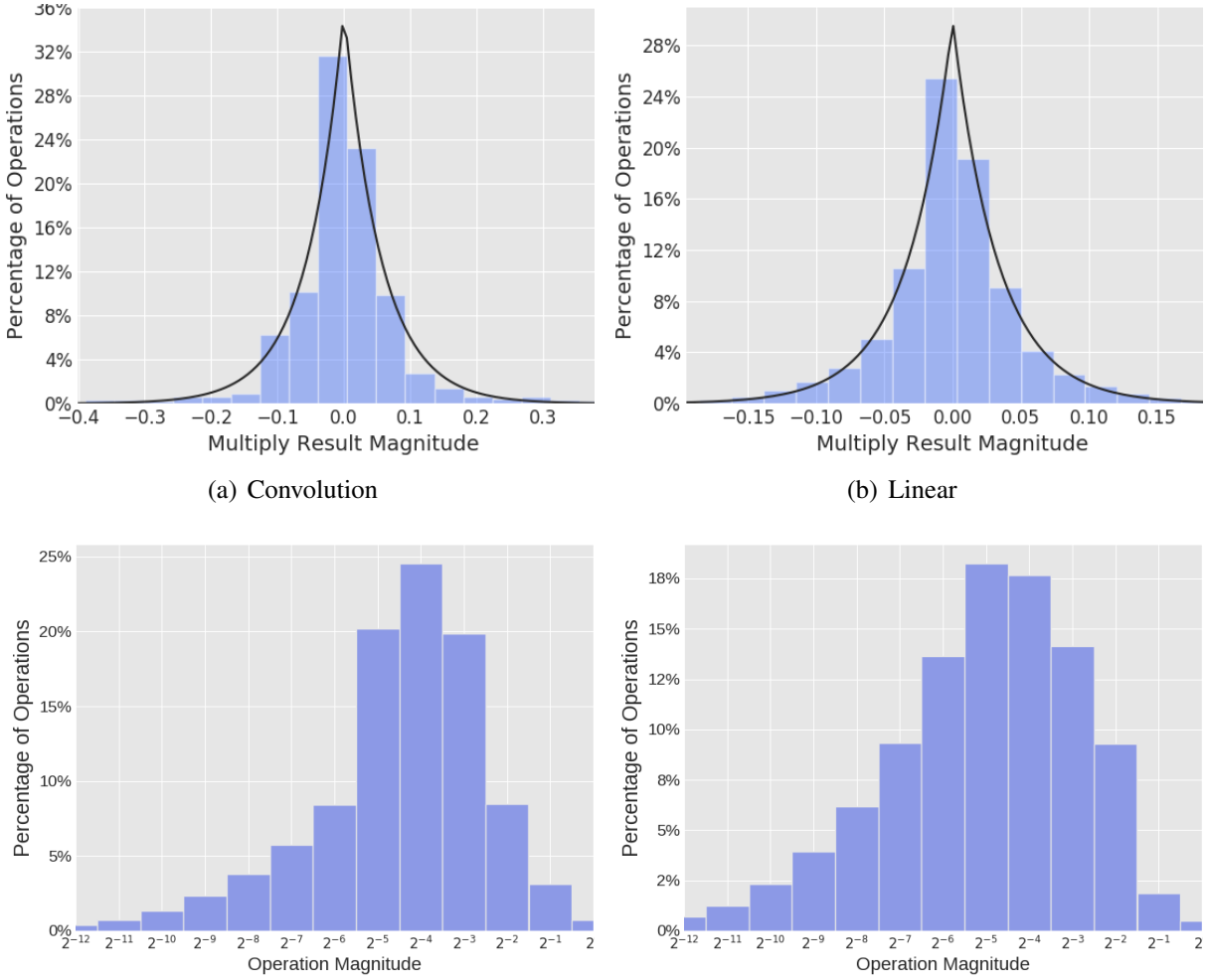
for $i = 1 \ldots N$ **do** *config(i)=approx*$[(\mathrm{l}_1 \ldots l_{i-1}, l_{i+1} \ldots l_n)=\mathrm{Apx}_0,\ l_i = Apx_{max}]$
out$_i$*= feed_forward (validation-data, config(i))*
$\mathscr{C}_i$*=error_estimation (out*$_i$*)*

for $Apx(j) = Apx_{max} \ldots Apx_{min}$ **do** $\mathscr{S}_j$*=selective_approx(*$\mathscr{C}_i$*,* $Apx(j)$*)*
out$_j$*= feed_forward (test-data,* $\mathscr{S}_j$*)*
$\mathscr{E}_j$*=error_estimation (out*$_j$*)*

if (*QoS* ¿ $\mathscr{E}_j$) **then** *save_config(*$\mathscr{S}_j$*)*
Break

($iter_{max}$), (iii) neural network error (δ), and (iv) the maximum hardware precision (Apx_{min}). Based on the updated THR, GTA checks the convergence in each training iteration by measuring the slope of classification accuracy over last 50 iterations. If the slope is smaller than a THR value, our framework reduces the level of approximation by a single step.

Algorithm 1 outlines the details of our gradual training approximation. The first steps assign the weights and biases random values and set the hardware approximation to the maximum level (Apx_{max}). The iterative training starts by feed forward (line 6), where input patterns pass through NN and generate the output class. Our algorithm estimates the network error (line 7) and checks for the convergence (line 8). If training converges and the hardware is at the most precise level (Apx_{min}, defined by user), the algorithm terminates. Otherwise, it updates the level of approximation (line 11) and performs back propagation for the next training iteration (line 13). This iterative procedure terminates when either the network converges during the final hardware approximation, or the iteration reaches the maximum ($iter_{max}$).



(a) Convolution

(b) Linear

(c) Convolution

(d) Linear

Figure 5.4. Distribution of multiply result magnitudes produced by convolutional and linear layers within a LeNet network trained for MNIST

5.3.4 DRAAW Accuracy Control

Previously we controlled accuracy using a uniform error control scheme. For each operation, the error is estimated by the number of bit mismatches. The computation is run on exact hardware if more bit differences occur than a specified cutoff. Applying a uniform approximation scheme to applications is not always optimal, especially in the case of neural networks. Each neuron in a layer receives a series of inputs which are then multiplied by a weight. Neurons sum the results of the multiplications, so larger values impact this summation more. Figure 5.4

shows distributions of multiplication results in a neural network layer. Approximation error in the largest values has a larger impact on the output accuracy. The tolerance for approximations must be adjusted based on the distribution of operations within the neural network. Using a *magnitude sensitive accuracy control* scheme, DRAAW is able to approximate more operations without decreasing prediction accuracy.

Approximate computing techniques must be capable of providing reliably controllable output accuracy. If the accuracy degrades too much, then the results may become unusable. DRAAW offers the ability to adjust accuracy by setting the maximum error individual operations may produce. In many GPU applications, floating point operations are the bulk of the computation time and energy. As a result, determining match quality for these operations is relatively straightforward. *IEEE 754* floating point representation includes three components: the sign bit, the exponent bits, and the mantissa bits. The exponent bits provide magnitude, while the mantissa bits represent the value. When determining match quality, each additional bit match starting from the most significant bits guarantees a maximum output error decrease of 50%. The maximum possible result error can be predicted with the following equation where A is the input value and N is the number of bits checked.

$$Error = \sum_{i=0}^{N-1} 2^{(((N-i)A_{N-i-1}))}. \tag{5.5a}$$

A user selects the maximum error for an operation, ErrorMax, which is then used to generate N, the number of tuning bits checked in the mantissa.

$$N = log_2(\frac{1}{Error_{Max}}). \tag{5.6}$$

While uniformly setting the maximum error per operation ensures high error matches are

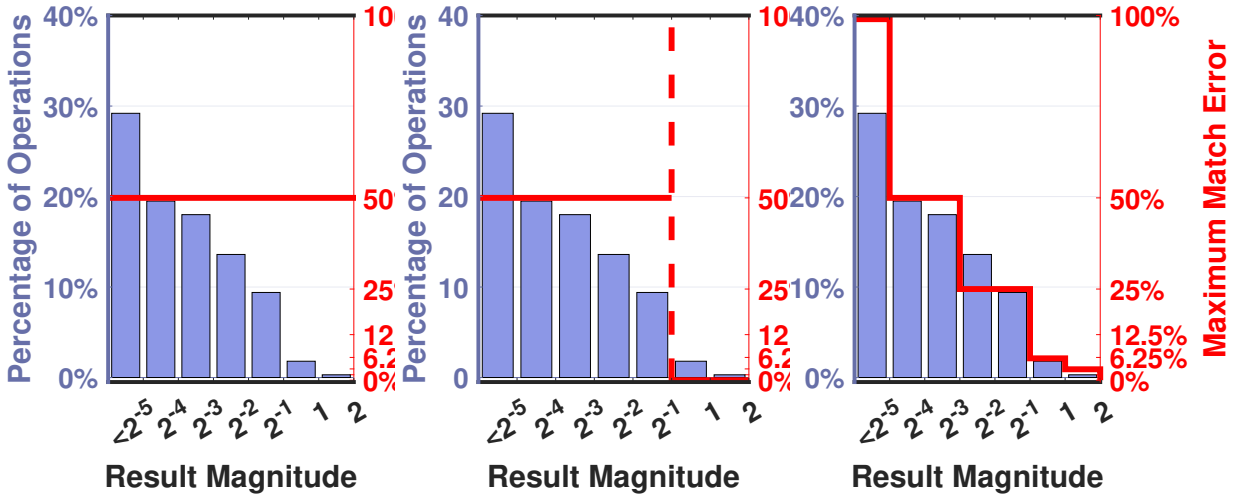


Figure 5.5. Maximum error per operation at different result magnitudes using a) uniform error control, b) magnitude cutoff, and c) magnitude scaling.

not used, it will not necessarily result in high utilization of DRAAW. As shown in Figure 5.4, applications such as neural networks do not have a uniform distribution of output magnitudes. There are exponentially more values closer to zero and the distribution of exponent values forms a Gaussian distribution. If the larger values have a more significant impact on the output accuracy, then an alternative accuracy control scheme may provide better results than the error distance alone. We test two alternatives as shown in Figure 5.5. First, we show *magnitude cutoff* in which a key exponent value is selected above which all values are computed exactly. Below this threshold, values are approximated if DRAAW predicts they will fall below ErrorMax. In the second case, *magnitude scaling*, we attempt to map the maximum error to the distribution of values and gradually increase ErrorMax as the magnitude decreases. Based on our testing, we are able to approximate more operations while maintaining application output accuracy than our previous design [98].

5.3.5 Neuron Aware Approximation

To quickly and automatically select values for *Magnitude sensitive accuracy control* in neural networks, we propose *Neuron Aware Approximation*. Testing many potential con-

figurations of neural networks to find the optimal accuracy settings is time consuming, so we attempt to estimate the accuracy control parameters based on neuron values within a network. When summing the inputs for each neuron, larger values will impact the result of the summation more than smaller ones, so DRAAW attempts to approximate inputs producing larger outputs more precisely. DRAAW profiles the network prior to inference and examines layer to identify the absolute values of of the multiply operations. We sample each layer to generate scaling approximation values to determine how much each operation can be approximated. We used the following equation to estimate the floating point exponent value where we no longer check the mantissa bits under scaled approximation. C represents magnitude constant to be identified through evaluation. M is the maximum magnitude allowed and σ is the layer's standard deviation.

$$M = floor(log_2(\sigma) + C)$$

Figure 5.6 shows our design. The user specifies the output accuracy requirement of the system for *neuron aware approximation* to meet. The maximum output accuracy is given in the form of Δe the value representing the maximum percent drop in prediction accuracy allowable. This value is keyed to a C value to find an appropriate setting. To find C, a pre-trained neural network is run using the training data set. The distribution of operation magnitudes and standard deviation for each layer in the network is found. The standard deviation determines the point in which mantissa bits are not checked under the scaled error control scheme and the operation distribution determines how the scaled error curve is mapped to the remaining magnitudes. *Neuron aware approximation* uses a series of values based on profiling smaller networks to extrapolate to larger networks. The accuracy settings are set on a layer by layer basis at run time. For ResNets, the profiling can be performed at a block level as well. DRAAW provides a user-friendly method of automatically approximating networks while meeting accuracy requirements.

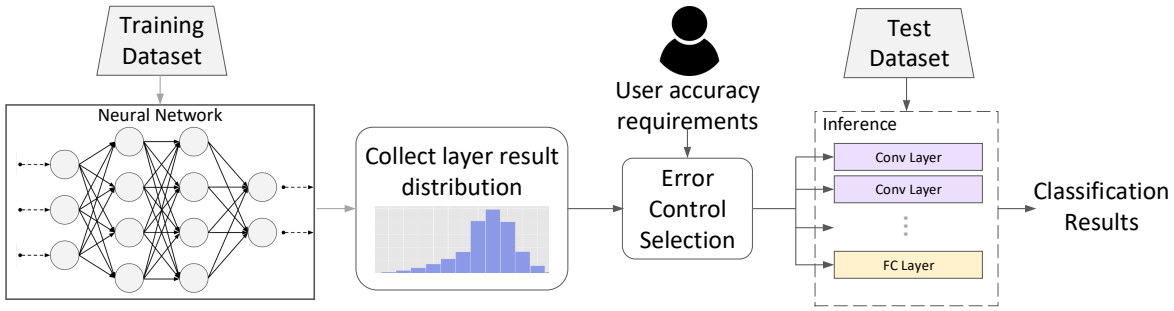


Figure 5.6. Neuron level approximation

5.4 Experimental Results

5.4.1 Experimental Setup

We integrate configurable FPUs on the AMD Southern Island GPU, Radeon HD 7970 device, a recent GPU architecture with 2048 streaming cores. We test our design by modifying PyTorch [99] to approximate operations at runtime. We use *Synopsys Design Compiler* to calculate the energy consumption of the balanced FPUs in GPU architecture in 45-nm ASIC flow. We perform circuit level simulations to design configurable FPU using HSPICE simulator in 45-nm TSMC technology. Neural networks are realized using OpenCL, an industry-standard programming model for heterogeneous computing. We tested the application of proposed design on four general neural network applications:Handwritten Image Recognition (MNIST) [77], Voice Recognition (ISOLET) [100], Hyperspectral Imaging (HYPER) [101], Human Activity Recognition (HAR) [102].

We test the following data sets as shown in Table 5.2. For ISOLET, UCIHAR, and FACE we test using a fully connected network with two hidden layers. For MNIST, we test using a CNN, *LeNet-5* [77]. We evaluate *neuron aware approximation* using two deep networks, AlexNet and ResNet20, running the CIFAR-10 dataset.

For each of the data sets, we compare the baseline accuracy of the train and inference phases with those when using the proposed DRAAW framework. We compare the designs in terms of run time and power consumption. Stochastic gradient descent with momentum [103] is

used for training. The momentum is set to 0.1, the learning rate is set to 0.001, and a batch size of 10 is used. Dropout [104] with drop rate of 0.5 is applied to hidden layers to avoid over-fitting. The activation functions are set to "Rectified Linear Unit" clamped at 6. A "Softmax" function is applied to the output layer.

Table 5.2. Datasets: (n = Features, K = Classes)

Application	n	K	Train Set Size	Test Set Size	Description
ISOLET	617	26	6238	1559	Voice Recognition [100]
UCIHAR	561	6	7352	2947	Human Activity Recognition [105]
FACE	608	2	22441	2494	Facial Recognition [106]
MNIST	784	10	60000	10000	Handwritten digit classifier [79]
CIFAR10	3072	10	60000	10000	Image prediction [107]

Table 5.3. Network Configuration

Application	Network Topology	Test Accuracy (%)	e_{test}(%)
ISOLET	617→ 500 → 500 → 26	95.7%	4.3%
UCIHAR	561→ 500 → 500 → 6	94.6%	6.4%
FACE	608→ 500 → 500 → 2	97.0%	3.0%
MNIST	LeNet-5 [77]	99.1%	0.9%
CIFAR10	AlexNet [97]	73.6%	26.4%
CIFAR10	ResNet20 [37]	93.8%	6.2%

Table 5.4. Configuration of different neural networks running on GPGPU with uniform and GTA approximation, providing different quality of service.

Applications		**MNIST**				**ISOLET**				**HYPER**				**HAR**			
Training Error (Δe_{train})		**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**	**0%**	**1%**	**2%**	**4%**
GTA	*Apx_{min}*	5-bits	4-bits	1-bits	0-bits	7-bits	4-bits	2-bits	1-bit	8-bits	6-bits	5-bits	3-bits	8-bits	6-bits	3-bits	2-bits
	Approx Hit Rate	56%	69%	88%	100%	52%	60%	79%	92%	23%	44%	56%	82%	38%	51%	72%	86%
Uniform	*Tuning bits*	4-bits	3-bits	1-bit	0-bits	5-bits	3-bits	2-bits	1-bit	8-bits	5-bits	4-bits	3-bits	6-bits	4-bits	2-bits	1-bit
	Approx Hit Rate	33%	42%	54%	100%	29%	38%	51%	83%	6%	14%	27%	37%	11%	26%	47%	66%

5.4.2 Training Accuracy-Efficiency

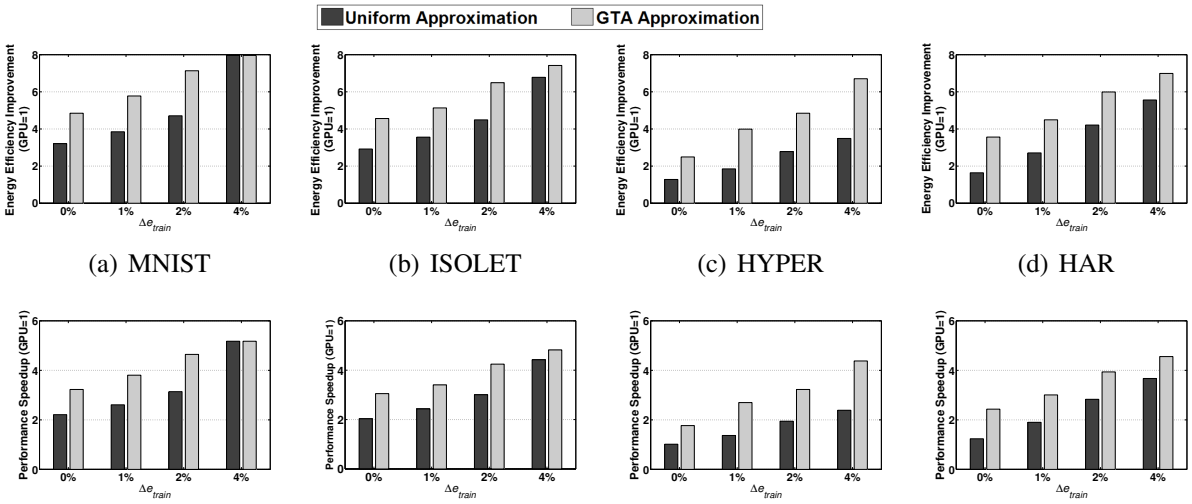
Here we compare the classification accuracy of different NN applications training on GPGPU with uniform and GTA approximation. In uniform mode, the hardware approximation

is fixed during the training mode, while GTA changes the approximation adaptively depending on the training error (explained in Section 5.3.3). Table 5.4 shows the configuration of different applications training on uniform and GTA approximation, providing 0% to 4% quality loss (Δe_{train}). For uniform approximation, the table shows the number of tuning bits in hardware which provides the desired accuracy. For GTA, the level approximation is set by defining the final level of hardware approximation. The table also shows the approximation hit rate, which is the ratio of running FPUs in approximate mode to the total accesses, for each configuration. For instance, for the MNIST application the uniform approximation provides 0% quality loss using 4 tuning bits which results in a roughly 36% approximation hit rate. For the same quality of service, GTA adaptively changes the tuning bits from 0 to 5 bits, resulting in average 70% approximation hit rate.

Figure 5.7 shows the energy efficiency improvement and the speedup of different applications running on GPGPU with uniform and GTA approximation. The results are normalized to GPGPU using exact 32-bit FPUs. Our experimental results shows that at the same level of accuracy, the GTA always outperforms the efficiency of the uniform approximation. This higher efficiency of comes from GTA ability to put the GPGPU in approximate mode for higher portion of time (as compared to uniform approximation, as shown in Table 5.4). The result shows that ensuring 0% additive error, GTA (uniform) design can achieve 3.86× (2.26×) energy efficiency improvement and 2.62× (1.62×) speedup as compared to exact mode. For GTA (uniform) design, this improvement increases to 4.84× and 6.11× (2.99% and 4.04%) in energy efficiency and 3.23× and 4.01× (2.07× and 2.37×) in performance accepting 1% and 2% additive errors.

5.4.3 Magnitude sensitive accuracy control

Figure 5.8 shows the change output accuracy as hit rate increases under each of the magnitude sensitive schemes. Cutoff and scaled error control approaches result in higher hit rates with lower degredation in error. The networks are highly sensitive to approximation error in the larger magnitude values, while smaller operations can safely be approximated. Table 5.5



(a) MNIST (b) ISOLET (c) HYPER (d) HAR

(e) MNIST (f) ISOLET (g) HYPER (h) HAR

Figure 5.7. Energy efficiency improvement and speedup of different NN applications training on GPGPU with uniform and GTA approximation.

compares the hit rate for the different applications based on the error control used. The scaled approximation approach can approximate an average of 94.2% of operations during inference with only 0.4% average decrease in prediction accuracy.

Neuron aware approximation

Table 5.6 shows the mean and standard deviation of the four smaller networks tested. Figure 5.9 shows network prediction accuracy as the magnitude in which only exponent bits are checked for scaled error control changes. The points where degradation begins are marked. The mean values are close to zero and do not appear to impact the point where accuracy decreases, so we focus on standard deviation. For the tested applications, face shows a rapid drop at 2^{-6}, ISOLET and UCIHAR at 2^{-5}. This appears to correlate to the standard deviations presented in Table 5.6 giving the predicted magnitude constant C as 1.38, 0.92, and 0.98 respectively. For MNIST, the standard deviation gives 1.45 for the convolution layers and 0.82 for the linear layers. We average the values to set our testing C to 1 as the cutoff where quality drops drastically and estimate -2 to achieve less than 1% error based on the MNIST network results.

With a value of M selected, we can predict a viable configuration for larger neural

Table 5.5. Hit Rate of DRAAW using different error control schemes for Δe_{test} of less than 1%.

Data set	Uniform		Cutoff		Scaled	
	Hit Rate	$\Delta\ e_{test}$	*Hit Rate*	$\Delta\ e_{test}$	*Hit Rate*	$\Delta\ e_{test}$
ISOLET	36.9%	0.3%	85.5%	0.9%	94.2%	0.3%
UCIHAR	72.0%	1.0%	87.5%	0.4%	96.0%	0.7%
FACE	68.5%	0.9%	93.1%	1.0%	95.6%	0.2%
MNIST	61.8%	0.3%	81.7%	0.2%	90.8%	0.3%

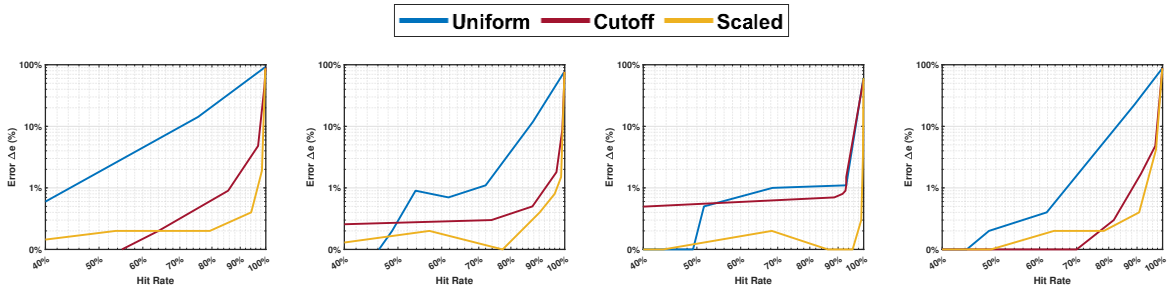
Table 5.6. Network Neuron Magnitude Analysis

Application	Linear μ	Linear σ	Conv μ	Conv σ
ISOLET	0.001	0.033	NA	NA
UCIHAR	-0.001	0.035	NA	NA
FACE	-0.0005	0.012	NA	NA
MNIST	0.0004	0.091	0.002	0.142

networks. Tables 5.7 and 5.8 show the predicted configuration values for the two larger networks running CIFAR10. We examine the blocks within the ResNet for our prediction. DRAAW sets the point where mantissa bits are not longer checked based on the predicted value, then uses the distribution adjust the scaled error curve. For each layer or block, DRAAW sets the predict configuration and we compare the performance neuron aware approximation to the networks using uniform selection and scaled. Based on our results shown in Table 5.9 we improve hit rate by 5% compared to the scaled performance. However, the primary benefit of neuron aware approximation is an automated selection process based on user accuracy requirements. It is not guaranteed to provide the optimal configuration for the network, but sets near maximum approximation parameters for DRAAW running the network.

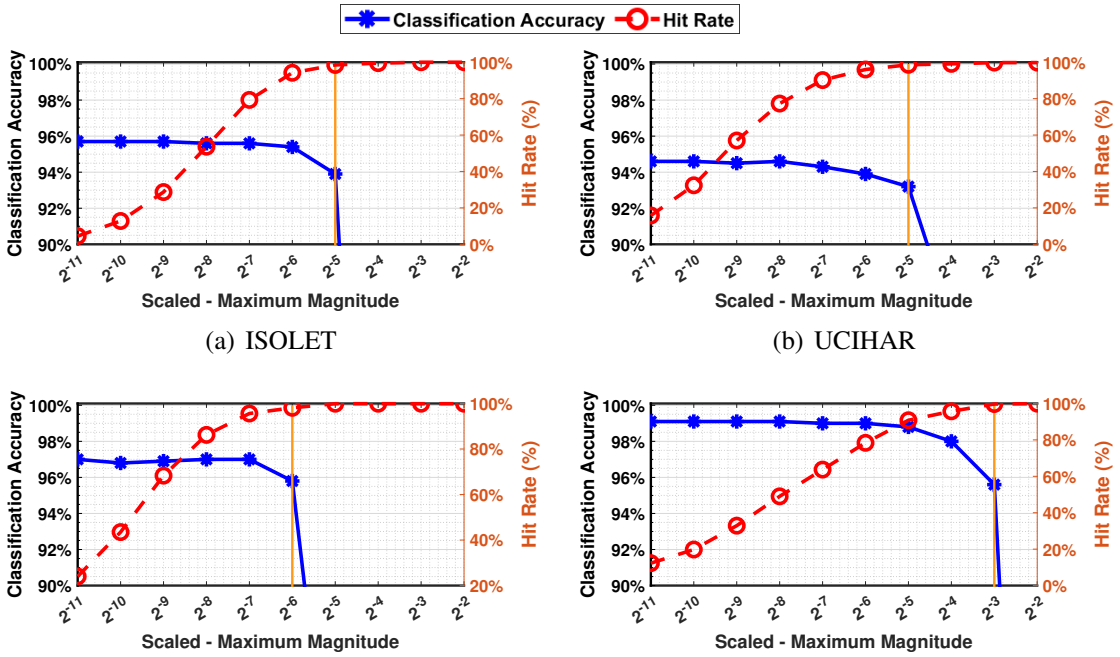
5.4.4 Energy Reduction and Acceleration

DRAAW provides significant improvements to both acceleration and energy reduction for the tested applications. Figure 5.11 shows the speedup and energy savings for the tested neural networks. Uniform error control provides an average of 2.9× EDP improvement and



(a) ISOLET (b) UCIHAR (c) Face (d) MNIST

Figure 5.8. Neural network applications error as hitrate increases for three different error control schemes



(a) ISOLET (b) UCIHAR

(c) Face (d) MNIST

Figure 5.9. Hit Rate as magnitude where mantissa bits are not checked for scaled error control increases. Magnitude when prediction accuracy sharply drops is marked.

1.27× speedup over the baseline GPU for these networks. Enabling scaled approximation set by neuron aware approximation increases the EDP improvement to 6.2× and speedup to 2.8×. We show a 2.2× increase in speedup and 2.1× improvement to energy over our design in [98] while maintaining less than 1% decrease in prediction accuracy.

Table 5.7. Predicted Cutoffs for AlexNet convolution and fully connected layers

Layer	C1	C2	C3	C4	C5	FC1	FC2	F3
Std Dev	0.11	0.018	0.0251	0.034	0.055	0.143	0.38	0.39
Predicted Cutoff	-6	-8	-8	-7	-7	-5	-4	-4

Table 5.8. Predicted Cutoffs for ResNet blocks and layers.

Layer	C1	B1	B2	B3	B4	FC1
Std Dev	0.129	0.011	0.005	0.003	0.001	0.066
Predicted Cutoff	-5	-9	-10	-11	-12	-6

Table 5.9. Hit Rate of DRAAW using different error control schemes for Δe_{test} of less than 1%.

Data set	Uniform		Scaled		Neuron Aware	
	Hit Rate	$\Delta\ e_{test}$	*Hit Rate*	$\Delta\ e_{test}$	*Hit Rate*	$\Delta\ e_{test}$
AlexNet	73.9%	0.9%	88.4%	0.5%	92.7%	0.2%
ResNet	71.0%	0.6%	89.2%	0.6%	95.8%	0.3%

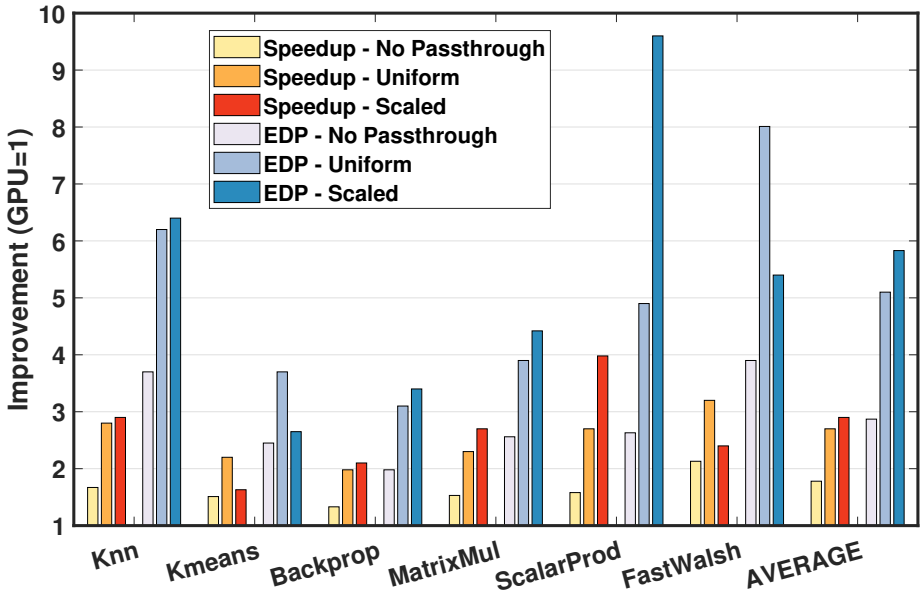


Figure 5.10. Speedup and EDP improvement provided by DRAAW for 6 GPGPU applications.

5.4.5 Scalability and Overhead

DRAAW is a general framework which can accelerate several supervised machine learning algorithms which have iterative training procedure or layer-based inference structure.

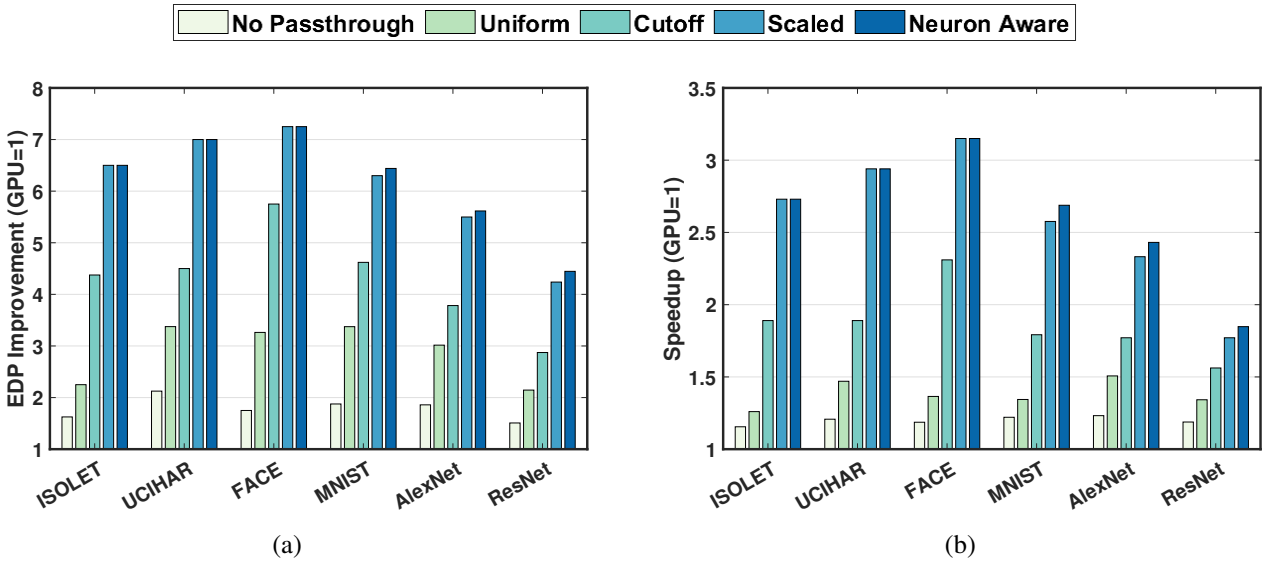


Figure 5.11. a) EDP improvement and b) speedup for neural networks with less than 1% Δe_{test}.

DRAAW is a scalable design in terms of the neural network size and supports the approximation on both convolution and fully connected layers. If the number of neurons surpasses the number of GPGPU cores, our design sequentially runs the network and configures the cores at run-time accordingly. Our design is able to reconfigure all cores in a single GPU cycle with negligible impact on the neural network training execution. In inference, DRAAW sensitivity analysis and adaptive approximation performs just once at offline over all applications, which results in a negligible overhead. In terms of area, our evaluation shows that the proposed configurable FPU can be designed by adding less than 2.6% area overhead to the existing FPU. This area is negligible considering 7.13× energy savings and 4.64× speedup that DRAAW can provide.

5.5 Conclusion

In this chapter we propose DRAAW, a novel framework to accelerate neural network computation in both training and inference modes by enabling configurable approximation. We utilize our previous techniques along with several new ones two maximize approximation of neural network operations. DRAAW supports gradual approximate training which enables the hardware approximation adaptively based on on the network accuracy. Our framework starts the

training from deep approximation then changes this level adaptively based on the neural network error rate. For inference, DRAAW proposes a layer-based approach which enables approximation on neural network layers based on their sensitivity to approximation. Our experimental evaluation shows that DRAAW 4.84× energy savings and 3.22× speedup when running four different neural network applications with 1% decrease in classification accuracy. DRAAW automatically selects parameters based on user prediction requirements for neural networks and improves speedup by 2.9× speedup and EDP by 6.2× of inference across six neural networks. Through the use of our approximate hardware we are able to provide significant performance improvements and energy savings for GPGPU applications.

Chapter 5 contains material from ”CANNA: Neural Network Acceleration using Configurable Approximation on GPGPU”, by Mohsen Imani, Max Masich, Daniel Peroni, Pushen Wang, and Tajana Rosing, which appears in Asia and South Pacific Design Automation Conference (ASP-DAC), 2018. The dissertation author was one of the primary instigators and a secondary author of this paper.

Chapter 5 contains material from ”Data Reuse for Accelerated Approximate Warps”, by Daniel Peroni, Mohsen Imani, Hamid Nejatollah, Nikil Dutt, and Tajana Rosing, which currently is being prepared for submission for publication. The dissertation author was the primary instigator and author of this paper.

Chapter 6

Summary and Future Work

Alternative computing strategies are needed process the increasing amounts of data being produced. GPUs accelerate applications which can be parallelized to a high degree, but they require a lot of energy. Approximate computing provides a method of lowering energy consumption and speeding up the applications which can tolerate small amounts of error. This dissertation improved GPU energy efficiency and performance of many applications by using approximate computing. We target arithmetic operations through the use of CFPU and use ALOOK to improve performance of applications with high redundancy. The previous designs provide energy savings, but to avoid bottlenecks and accelerate warps we use AWARP. Finally, we show with application specific techniques specifically aimed at neural networks that our design significantly improves training and inference energy consumption and performance.

6.1 Approximate Arithmetic

Floating point approximate multiplies make up the bulk of computation in many GPGPU applications including machine learning. We propose a configurable floating point unit, CFPU, to perform approximate multiplications. CFPU eliminates the most costly component of multiplication and replaces a much more energy efficient mantissa copy. Applications with a large number of power of two values can utilize our design to a high degree as CFPU computes these exactly. Our results in Chapter 2 show we are able to improved EDP (Energy Delay Product) by

4.1× over the unmodified GPU while ensuring less than 10% accuracy loss.

6.2 Computational Reuse

For applications with a high amount temporal similarity we propose Alook. We use small lookup tables to store frequently computed operations in order to avoid recomputing them. Because we can reduce the size of the lookup tables compared to previous static implementations, we can utilize more tables to perform lookup in parallel. Alook achieves a 3.6× EDP and 1.3× performance speedup for less than 10% error compared to the unmodified GPU. Using ALOOK along with CFPU results in a 5.6× average EDP improvement for the same amount of error, as shown in Chapter 3.

6.3 Warp-level Acceleration

Warps are an integral part of how GPU instructions are executed, so they must be accounted for when accelerating applications. While many instructions within the warps may be approximated, in many cases bottlenecks occur and prevent acceleration. In the event a small number of threads needs to compute in exact mode, we propose warp pass-through, which signals all cores to use approximate results. To target warps with a larger number of exact operations, we propose warp value trading in which we use the predicted error to map threads to uniform approximate or exact warps. Using the two approaches increases speedup to 1.8× on average for the tested applications, as shown in Chapter 4.

6.4 Approximating Neural Networks

Neural networks are popular for solving many machine learning tasks. However, as they grow deeper in layers, they take more time to train and run. We apply approximate computing to improve the performance of training and inference in Chapter 5. For training we show that gradually reducing the maximum allowed error per operation results in 3.2× speedup and 4.8×

EDP improvement with less than 1% drop in prediction accuracy. For inference we are able to automatically select parameters based on user prediction requirements for neural networks and improve speedup by 2.9× and EDP by 6.2× of inference with less than 1% decrease in prediction accuracy.

6.5 Future Work

Approximate computing offers a potential solution to improving computation efficiency in error tolerant applications. Our work has shown many energy and performance improvements over state-of-the-art approximate units, however further work is needed to utilize these designs more fully. Similar to neuron aware approximation, additional research is needed to explore automated error control for a wider range of general purpose applications. Additionally, further exploration of machine learning algorithms may yield improved information of redundant or approximate optimizations.

6.5.1 Approximate Activation Functions

Recently, reinforcement learning has been used to find alternative neural network optimizations to improve accuracy beyond increasing depth [108–110]. Each neuron in the network takes activations which are multiplied by weights and summed together. This value is then passed through an activation function to generate the next activation value. The activation function is a fundamental core aspect of neural networks. Recent research has examined optimizing these functions to improve classification accuracy on difficult tasks. Google showed replacing ReLU activation functions with the Swish function [110] improved top-1 ImageNet classification accuracy of two state of the art neural networks by almost 1%. The authors note that the improvement from the Inception V3 to Inception-ResNet-v2 required a year of architecture tuning and improvement resulting in only 1.3% top-1 accuracy improvement. Currently the most commonly used activation function is rectified linear unit (ReLU). ReLU clamps negative values to zero otherwise it outputs the input value. Alternative activation functions, such as swish,

require more computation overhead compared to ReLU, limiting their adoption. Activation functions such as sigmoid and tanh are often used in long short term memory (LSTM) units that make up recurrent neural networks (RNNs). A majority of this work focused on approximating multiply and multiply add operations. However, as these are accelerated, the computation such as activation functions take a larger portion of the total time. Approximate computing may be able to approximate these operations in order to provide fewer training iterations and classification accuracy without incurring the additional computation time penalties.

6.5.2 Hyperdimensional computing

Hyperdimensional (HD) computing provides an alternative to neural networks which offers faster training and inference [111–113]. HD computing encodes values into high dimensional spaces and creates class hyper-vectors consisting of as many as 10,000 bits. HD computing has high tolerance to noise which can be exploited by approximate computing. However, storing the vectors during training requires a large memory footprint. It is necessary to explore methods for optimizing encoding and training schemes to allow this computing method to scale to larger and more difficult tasks. Approximate computing offers potential solutions to increasing the efficiency of these algorithms.

Bibliography

[1] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low-cost binary128 floating-point fma unit design with simd support," *IEEE Transactions on Computers*, vol. 61, pp. 745–751, May 2012.

[2] M. Imani, D. Peroni, and T. Rosing, "Program acceleration using nearest distance associative search," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pp. 43–48, IEEE, 2018.

[3] M. Imani, D. Peroni, and T. Rosing, "CFPU: Configurable floating point multiplier for energy-efficient computing," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2017.

[4] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "RMAC: Runtime configurable floating point multiplier for approximate computing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 12, ACM, 2018.

[5] S. Hashemi, R. Bahar, and S. Reda, "DRUM: A dynamic range unbiased multiplier for approximate applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 418–425, IEEE Press, 2015.

[6] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[7] G. E. Moore, "Cramming more components onto integrated circuits," 1965.

[8] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Test Symposium (ETS), 2013 18th IEEE European*, pp. 1–6, IEEE, 2013.

[9] M. Imani, A. Rahimi, and T. S. Rosing, "Resistive configurable associative memory for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1327–1332, IEEE, 2016.

[10] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: an approximate computing framework for artificial neural network," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 701–706, EDA Consortium, 2015.

[11] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "RAPIDNN: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.

[12] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 120, ACM, 2015.

[13] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing, "Efficient neural network acceleration on GPGPU using content addressable memory," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1026–1031, IEEE, 2017.

[14] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient FPGA implementation," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 85–92, IEEE, 2017.

[15] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, Feb 2017.

[16] 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 *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 164–174, ACM, 2011.

[17] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, pp. 922–927, July 2005.

[18] M. Imani and U. Braga-Neto, "Optimal state estimation for boolean dynamical systems using a boolean kalman smoother," in *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 972–976, IEEE, 2015.

[19] X. Lin and K. G. Hubbard, "Sensor and electronic biases/errors in air temperature measurements in common weather station networks," *Journal of Atmospheric and Oceanic Technology*, vol. 21, no. 7, pp. 1025–1032, 2004.

[20] Analog Devices, *Small, Low Power, 3-Axis 3 gAccelerometer*, 2009.

[21] Analog Devices, *Low Voltage Temperature Sensors*, 2015.

[22] Texas Instruments, *AmbientLight Sensor(ALS)*, 2017.

[23] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, pp. 62:1–62:33, Mar. 2016.

[24] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, J. Park, G. Mururu, O. Mutlu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 493–494, Aug 2014.

[25] P. Yin, C. Wang, W. Liu, and F. Lombardi, "Design and performance evaluation of approximate floating-point multipliers," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 296–301, IEEE, 2016.

[26] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 27–32, ACM, 2014.

[27] A. Suhre, F. Keskin, T. Ersahin, R. Cetin-Atalay, R. Ansari, and A. E. Cetin, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 1123–1127, IEEE, 2013.

[28] M. Imani, S. Patil, and T. S. Rosing, "MASC: Ultra-low energy multiple-access single-charge TCAM for approximate computing," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 373–378, EDA Consortium, 2016.

[29] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 603–614, IEEE, 2015.

[30] W. José, A. R. Silva, H. Neto, and M. Véstias, "Efficient implementation of a single-precision floating-point arithmetic unit on FPGA," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4, IEEE, 2014.

[31] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[32] "White paper:NVIDIAs next generation CUDA compute architecture: Kepler TMGK110/210," tech. rep., Nvidia, 2014.

[33] "Reference guide: Southern islands series instruction set architecture," tech. rep., AMD Accelerated Parallel Processing Technology, 2012.

[34] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, Feb 2014.

[35] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1717–1724, 2014.

[36] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "LookNN: Neural network with no multiplication," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1775–1780, March 2017.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[38] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pp. 117–122, ACM, 2015.

[39] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 95, European Design and Automation Association, 2014.

[40] C.-H. Lin and C. Lin, "High accuracy approximate multiplier with error correction," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 33–38, IEEE, 2013.

[41] M. Imani, S. F. Ghoreishi, D. Allaire, and U. Braga-Neto, "MFBO-SSM: Multi-fidelity Bayesian optimization for fast inference in state-space models," AAAI, 2019.

[42] M. Courbariaux, Y. Bengio, and J.-P. David, "Low precision arithmetic for deep learning," *3rd International Conference on Learning Representations (ICLR2015)*, 12 2014.

[43] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 62, 2016.

[44] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for fpgas," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pp. 185–194, IEEE, 2003.

[45] S. P. Gnawali, S. N. Mozaffari, and S. Tragoudas, "Low power spintronic ternary content addressable memory," *IEEE Transactions on Nanotechnology*, 2018.

[46] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim, "Energy-efficient approximate multiplication for digital signal processing and classification applications," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 23, no. 6, pp. 1180–1184, 2014.

[47] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, pp. 346–351, IEEE, 2011.

[48] 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*, pp. 44–54, Ieee, 2009.

[49] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun, "Elastic-core: enabling dynamic heterogeneity with joint core and voltage/frequency scaling," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 151, ACM, 2015.

[50] P. K. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[51] M. Imani, A. Rahimi, P. Mercati, and T. Rosing, "Multi-stage tunable approximate search in resistive associative memory," *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[52] 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*, pp. 409–414, IEEE Press, 2011.

[53] M. Imani, S. Patil, and T. Rosing, "Approximate computing using multiple-access single-charge associative memory," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[54] M. Imani, D. Peroni, A. Rahimi, and T. Rosing, "Resistive cam acceleration for tunable approximate computing," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[55] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[56] X. Yin, M. T. Niemier, and X. S. Hu, "Design and benchmarking of ferroelectric fet based TCAM," *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1444–1449, 2017.

[57] X. Chen, X. Yin, M. Niemier, and X. S. Hu, "Design and optimization of FeFET-based crossbars for binary convolution neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 1205–1210, IEEE, 2018.

[58] M. Imani, Y. Kim, A. Rahimi, and T. Rosing, "ACAM: Approximate computing based on adaptive associative memory with online learning.," in *ISLPED*, pp. 162–167, 2016.

[59] M. Imani, P. Mercati, and T. Rosing, "ReMAM: low energy resistive multi-stage associative memory for energy efficient computing," in *Quality Electronic Design (ISQED), 2016 17th International Symposium on*, pp. 101–106, IEEE, 2016.

[60] M. Imani, D. Peroni, A. Rahimi, and T. Rosing, "Resistive cam acceleration for tunable approximate computing," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[61] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *European Solid-State Circuits Conference, ESSCIRC Conference 2016: 42nd*, pp. 465–468, Ieee, 2016.

[62] S. A and R. C, "Carry speculative adder with variable latency for low power VLSI," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 2400–2402, March 2016.

[63] M. Nazemi and M. Pedram, "Deploying Customized Data Representation and Approximate Computing in Machine Learning Applications," *ArXiv e-prints*, June 2018.

[64] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, pp. 335–344, IEEE, 2012.

[65] Synopsys, "Design compiler," 2000.

[66] " AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)." http://developer.amd.com/sdks/amdappsdk/.

[67] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE Computer Society, 2012.

[68] S. N. Mozaffari and A. Afzali-Kusha, "Statistical model for subthreshold current considering process variations," in *Quality Electronic Design (ASQED), 2010 2nd Asia Symposium on*, pp. 356–360, IEEE, 2010.

[69] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu, and H. Jiang, "A spiking neuromorphic design with resistive crossbar," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 14, ACM, 2015.

[70] Y. Kim, M. Imani, and T. Rosing, "ORCHARD: Visual object recognition accelerator based on approximate in-memory processing," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 25–32, Nov 2017.

[71] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2018.

[72] "Caltech Library." http://www.vision.caltech.edu/Image_Datasets/Caltech101/.

[73] "FloPoCo [online]." http://flopoco.gforge.inria.fr/.

[74] D. Peroni, M. Imani, and T. Rosing, "ALook: adaptive lookup for GPGPU acceleration," in *ASPDAC 2019, Tokyo, Japan*, pp. 739–746, 2019.

[75] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 176–187, March 2016.

[76] Z. Liu, D. Wong, and N. S. Kim, "Load-triggered warp approximation on gpu," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, (New York, NY, USA), pp. 26:1–26:6, ACM, 2018.

[77] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[78] X. Gong, R. Ubal, and D. Kaeli, "Multi2sim kepler: A detailed architectural GPU simulator," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 269–278, April 2017.

[79] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[80] A. Rahimi, A. Ghofrani, K. Cheng, L. Benini, and R. K. Gupta, "Approximate associative memristive memory for energy-efficient gpus," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1497–1502, March 2015.

[81] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing, "Efficient neural network acceleration on GPGPU using content addressable memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1026–1031, March 2017.

[82] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[83] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2016.

[84] M. Imani, S. F. Ghoreishi, and U. M. Braga-Neto, "Bayesian control of large mdps with unknown dynamics in data-poor environments," in *Advances in Neural Information Processing Systems 31* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 8146–8156, Curran Associates, Inc., 2018.

[85] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, IOT '18, (New York, NY, USA), pp. 38:1–38:6, ACM, 2018.

[86] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "RNSnet: In-memory neural network acceleration using residue number system," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–12, Nov 2018.

[87] F. Imani, C. Cheng, R. Chen, and H. Yang, "Nested gaussian process modeling for high-dimensional data imputation in healthcare systems," in *2018 Institute of Industrial and Systems Engineers Annual Conference and Expo, IISE 2018*, 2018.

[88] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," IEEE, 2018.

[89] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 110, ACM, 2016.

[90] B. D. Rouhani, M. Samragh, T. Javidi, and F. Koushanfar, "Curtail: Characterizing and thwarting adversarial deep learning," *arXiv preprint arXiv:1709.02538*, 2017.

[91] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," pp. 2849–2858, 2016.

[92] D. D. Lin and S. S. Talathi, "Overcoming challenges in fixed point training of deep convolutional networks," *arXiv preprint arXiv:1607.02241*, 2016.

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

[94] 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," pp. 243–254, 2016.

[95] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016.

[96] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017.

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

[98] D. Peroni, M. Imani, H. Nejatollahi, N. Dutt, and T. Rosing, "ARGA: Approximate reuse for GPGPU acceleration," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ACM, 2019.

[99] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[100] R. Cole and M. Fanty, "UCI machine learning repository," 2994.

[101] "Hyperspectral remote sensing scenes." http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes.

[102] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.

[103] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," pp. III–1139–III–1147, 2013.

[104] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[105] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *ESANN*, 2013.

[106] Y. Kim, M. Imani, and T. Rosing, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 25–32, Nov 2017.

[107] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, vol. 55, 2014.

[108] C. Liu, B. Zoph, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. L. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," *CoRR*, vol. abs/1712.00559, 2017.

[109] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, vol. abs/1611.01578, 2016.

[110] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017.

[111] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "VoiceHD: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2017.

[112] M. Imani, J. Hwang, T. Rosing, A. Rahimi, and J. M. Rabaey, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.

[113] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.