Approximate Computing Techniques: From Logic Synthesis to Deep Learning

By

Jingxiao Ma

B.S., University of Nottingham, UK, 2018

M.S., Brown University, 2020

Thesis

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Engineering at Brown University

PROVIDENCE, RHODE ISLAND

September 2024

This dissertation by Jingxiao Ma is accepted in its present form by the School of Engineering as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Sherief Reda, Advisor

Recommended to the Graduate Council

Date _____ _____

Sherief Reda, Reader

Date _____ _____

Jacob Rosenstein, Reader

Date _____ _____

James Tompkin, Reader

Approved by the Graduate Council

Date _____ _____

Thomas A. Lewis, Dean of the Graduate School

Curriculum Vitae

Jingxiao Ma was born and raised in Qingdao, China. He received his BSc degree of Computer Science from University of Nottingham in 2018, and MSc degree from Brown University in 2020. He then joined the SCALE lab at Brown University. Initially, his research focused on electronic design automation and approximate computing, leading to his development of a comprehensive methodology for generating approximate circuits using Boolean matrix factorization. Presently, his research has expanded to encompass Deep Learning, Adaptive/Dynamic Neural Networks, and Edge Computing. He developed a dynamic neural network framework for efficient inference and a low-precision training algorithm for efficient training. Additionally, he integrated his knowledge in electronic design automation with Large Language Models to investigate their potential in revolutionizing circuit design and analysis.

jingxiao_ma@brown.edu

Brown University, RI, USA

Education

2020–2024 Ph.D., Engineering
Brown University (Providence, RI, USA)

2018-2020
M.Sc., Computer Science
Brown University (Providence, RI, USA)

2014–2018
Honors B.Sc., Computer Science
University of Nottingham (Nottingham, UK)

Selected Publications

1. **Ma, J.**, Panda, P., and Reda, S. (2025). FF-INT8: INT8 Forward-Forward Algorithm for Efficient DNN Training on Resource-Constrained Devices. The 9^{th} Workshop on Approximate Computing.

2. Abdelatty, M., **Ma, J.**, and Reda, S. (2025). MetRex: A Benchmark for Verilog Code Metric Reasoning Using LLMs. IEEE Asia and South Pacific Design Automation Conference.

3. **Ma, J.**, and Reda, S. (2023). WeNet: Configurable Neural Network with Dynamic Weight-Enabling for Efficient Inference. ACM/IEEE International Symposium on

Low Power Electronics and Design.

4. **Ma, J.**, and Reda, S. (2023). RUCA: RUntime Configurable Approximate Circuits with Self-Correcting Capability. IEEE Asia and South Pacific Design Automation Conference.

5. **Ma, J.**, and Reda, S. (2021). Runtime Configurable Approximate Circuits with Self-Correcting Capability. International Workshop on Logic and Synthesis.

6. **Ma, J.**, Hashemi, S., and Reda, S. (2021). Approximate Logic Synthesis Using Boolean Matrix Factorization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

7. **Ma, J.**, Hashemi, S., and Reda, S. (2019). Approximate Logic Synthesis Using BLASYS. Workshop on Open-Source EDA Technology.

Acknowledgments

This thesis would not have been possible without the constant support, guidance and inspirations of many kind individuals. First and foremost, I would like to express my immense gratitude to my advisor and mentor, Prof. Sherief Reda, whose guidance, support, and valuable insights during the course of my research has made this thesis possible. I would also like to thank Prof. Jacob Rosenstein and Prof. James Tompkin for being on my defense committee and taking the time to review my thesis.

I am extremely thankful for the productive collaborations with all my collaborators. I would especially like to thank Soheil Hashemi and my advisor, Prof. Sherief Reda, for their guidance and support during the early stages of my PhD journey. I would also like to thank Prof. Priyadarshini Panda and Manar Abdelatty for their valuable insights and guidance. I am grateful to all my fellow graduate students and friends in Prof. Reda's group at Brown, including Soheil Hashimi, Abdelrahman Hosny, Farnaz Nouraei, Abdelrahman Hussein, Marina Neseem, Ahmed Agiza, Jon Nelson, Manar Abdelatty, Mahdi Boulila, Tasneem Shaffee, Lisa Korver, Maryam Nouh and many others who have made the past four years memorable.

I cannot express enough my immense gratitude to my family for their constant support and love. I am particularly thankful to my parents, Yin Zhang and Haiwen Ma, and my grandparents, Dianyi Chen and Chengde Zhang, whose support has been the cornerstone of my achievements. Special thanks go to my uncle, Ning Zhang, my aunt, Mei Li, and my cousins, Zibing Zhang and Helena Zhang, for their invaluable help during my time in

the United States.

Last but certainly not least, the insights in this thesis would not have been possible without the unwavering support of my friends. I am deeply grateful to Kai Liu, whose constant inspiration and challenges pushed me beyond my limits, and to Jac Hu, whose steadfast support during long study sessions was invaluable. My sincere thanks also go to the Brown Japanese group in the Department of East Asian Studies, especially Hiramatsu-sensei and Yamakawa-sensei, for nurturing my interest in Japanese language and culture.

Abstract of Approximate Computing Techniques: From Logic Synthesis to Deep Learning, by Jingxiao Ma, Ph.D., Brown University, September 2024

Computational efficiency is a critical factor in the design of both software and hardware systems. As technology progresses, modern computing systems are growing increasingly complex. For example, OpenAI reports that the computational resources required to train the largest deep neural networks (DNNs) have been doubling every six months since 2010. Furthermore, computing devices are continuously shrinking—from desktop computers to smartphones and even smaller wearable technologies. With rising demand for high-performance computing in resource-constrained environments, especially for deep learning applications, optimizing power consumption and computational efficiency has become essential. This is where approximate computing comes into play, allowing trade-offs in computational accuracy for gains in energy efficiency or performance. This dissertation presents novel methodologies in approximate logic synthesis, dynamic neural network inference, and low-precision training to meet these demands.

In the hardware domain, the research introduces an approximate logic synthesis framework based on Boolean matrix factorization (BMF), which effectively reduces circuit complexity by permitting controlled errors, thus balancing accuracy and efficiency. This approach is further extended to runtime configurable approximate circuits that can adapt at runtime.

On the software side, the dissertation explores dynamic neural network architectures that adjust their computational load in real-time, optimized for deployment across heterogeneous hardware platforms. Additionally, it delves into low-precision training techniques, focusing on the Forward-Forward (FF) algorithm. By integrating INT8 quantization into the FF algorithm, the research achieves significant reductions in memory usage and energy consumption during training, making it particularly suitable for edge devices where computational resources are limited.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Importance of Energy-Efficient Computing

Computational efficiency and energy consumption are now critical factors in the design of modern software and hardware systems. With the rapid advancement of technology, the demand for more powerful yet efficient computational systems has risen sharply, particularly in areas like natural language processing, augmented reality, and computer vision. These domains involve highly complex computations, often relying on deep neural networks (DNNs) and other sophisticated algorithms, all while operating on devices with constrained resources. As computational complexity grows to meet the need for processing more intricate data, the push toward smaller, more compact devices — ranging from personal computers to smartphones and now even wearable technology — has heightened the importance of efficiency. This convergence of challenges has sparked increased interest in energy-efficient computing strategies that minimize hardware size, reduce energy consumption, and shorten delay.

Achieving energy efficiency in computing is not merely a hardware concern; it is a holistic challenge that encompasses software algorithms, hardware architectures, and the intricate synergy between the two. One promising approach to tackling this challenge is approximate

computing, a technique that intentionally sacrifices some degree of computational precision in exchange for improvements in energy efficiency, area, and speed. This approach is particularly valuable in applications where minor inaccuracies in computation do not significantly degrade the overall output quality, making it highly suitable for a range of deep learning tasks, multimedia applications, and even sensor-based systems. Approximate computing can be leveraged to design not only more efficient hardware but also optimized software systems, such as deep learning models, where the trade-off between accuracy and energy consumption can be fine-tuned. In this context, the growing emphasis on approximate computing reflects a broader trend towards creating adaptable, efficient systems that meet the stringent demands of modern applications without compromising on functionality or user experience.

1.2 Problem Statement

As computational systems grow in complexity, the power and energy consumption required to maintain high levels of performance have also increased significantly. This has posed a challenge for the deployment of advanced computing methods, particularly in environments with limited power resources, such as mobile and embedded devices. Traditional methods of improving energy efficiency, such as optimizing individual components or utilizing lower power modes, are becoming insufficient as the computational demands continue to rise. There is a pressing need for innovative methodologies that can address these challenges holistically, considering both software and hardware aspects of the system.

Approximate computing has emerged as a promising solution, offering the potential to reduce power consumption and improve energy efficiency by allowing for controlled errors in computation. However, implementing approximate computing techniques requires a careful balance between accuracy and efficiency, as well as the development of new design methodologies that can automate this process across different computational domains.

This thesis explores energy-efficient approximate computing methods, focusing on both hardware designs and software algorithms. The primary objectives of this thesis are:

- To investigate algorithms and methodologies for the synthesis of approximate circuits, which optimize chip area, power consumption and circuit delay.
- To explore approximate computing in deep learning context to improve efficiency for both model inference and training.
- To explore dynamic and flexible methodologies that allow for real-time adjustments to computational accuracy, thereby optimizing power consumption and performance based on the specific requirements of the task.

1.3 Contributions of the Thesis

The contributions of this thesis are multifaceted, addressing both theoretical and practical aspects of energy-efficient computing. The key contributions include:

- In Chapter 3, we introduce an approximate logic synthesis methodology based on truth table factorization. In addition, we provide an automatic circuit partitioning approach and a design space exploration heuristic to navigate the search space. We implement our methodology using a full stack of open-source tools, and thoroughly evaluate our methodology on a number of representative circuits showcasing the benefits of our proposed methodology for approximate logic synthesis. Especially, we compared 23 approximate designs of unsigned multipliers against the state-of-the-art method, where BLASYS has better power utilization in 17 cases, and better circuit delay in 16 cases.
- In Chapter 4, we present a novel framework which aims to synthesize runtime configurable approximate circuits based on arbitrary input circuits. By decomposing the truth table, our approach aims to approximate and separate the input circuit

into multiple configuration blocks which support different accuracy levels, including a corrector circuit to restore full accuracy. Power gating is used to activate different blocks, such that the approximate circuit is able to operate at different accuracy-power configurations. To improve the scalability of our algorithm, we also provide a design space exploration scheme with circuit partitioning. We evaluate our methodology on a comprehensive set of benchmarks. For 3-level designs, RUCA saves power consumption by 43.71% within 2% error and by 30.15% within 1% error on average.

- In Chapter 5, we present a configurable neural network architecture, where the weights of neural network can be dynamically enabled or disabled to switch between different sub-networks, so that we are able to balance the trade-off between inference time, energy consumption and model accuracy. We extend the methodology to convolutional layers using group convolution and channel shuffling. We also propose a design space exploration approach to search for the optimal sub-network for different scenarios. We thoroughly evaluate our methodology using a number of DNN architectures on different hardware platforms, showing that WeNet provides a large number of energy-efficient operation modes, 73.2% of which provide better accuracy-efficiency trade-off compared to other methodologies.

- Chapter 6 introduces an INT8 quantized training method based on the Forward-Forward (FF) algorithm, which was proposed recently. Additionally, we propose a novel loss function and updated training procedure to enhance model accuracy. Experiments show that our FF-INT8 method accelerates training by 0.7%, decreases energy consumption by 6.0%, and significantly reduces memory footprint by 22.2%, while maintaining high accuracy compared to state-of-the-art INT8 training algorithms.

CHAPTER 2

Background

In this chapter, we describe the background and a concise overview of the relevant prior work related to the techniques proposed in this dissertation. As outlined in Chapter 1, this dissertation aims to enhance the efficiency of deep learning computing from both hardware and software perspectives. Accordingly, we will explore key concepts and prior work from both domains. We begin in Section 2.1 by introducing the methodologies of approximate computing in circuit and transistor levels, *i.e.* approximate circuits and approximate logic synthesis. We then transition to the software domain in Section 2.2, where we review previous work aimed at enhancing the efficiency of inference in deep learning models. Finally, in Section 2.3, we conclude with an overview of efficient training methodologies for deep learning models. Additional details regarding the related work for the specific techniques discussed in this dissertation are included in their respective chapters.

2.1 Approximate Circuits and Approximate Logic Synthesis

Recent advancements in Deep Learning, particularly through Deep Neural Networks (DNNs), have achieved state-of-the-art results in various application domains, including computer vision and natural language processing. Convolutional Neural Networks (CNNs), the most widely used DNN architecture, heavily depend on multiplier-accumulate (MAC) operations [6]. For instance, AlexNet, a CNN architecture proposed in 2012, requires 724 million MAC operations to classify an image with a resolution of 227×227 pixels [7]. This number dramatically increases to 15.5 billion for more complex models like VGG-16 [8]. To meet such computational demands, DNN accelerators integrate thousands of MAC units on a single chip. For example, Google's TPU contains 64K MAC units, while Samsung's neural processing unit (NPU) incorporates 6K MAC units [9]. However, the sheer number of MAC units, coupled with high parallelization, leads to significant energy consumption, which is particularly challenging for edge devices like smartphones and smartwatches. Therefore, designing more efficient circuits is essential.

Approximate computing, an emerging paradigm, challenges the traditional requirement for absolute precision by employing less accurate functions to enhance power efficiency and performance. While the idea of sacrificing accuracy might seem counterintuitive, this approach is highly effective in application domains where small inaccuracies in output are acceptable. Such tolerance can arise from various sources, including noise in input data, inherently approximate calculations, or the human ability to tolerate variations in outputs [10]. This tolerance is particularly relevant in deep learning applications. For example, in image classification, input images often contain noisy pixels that are indistinguishable to the human eye. Additionally, training a neural network is fundamentally an approximation of an unknown function. These characteristics make approximate computing well-suited for deep learning tasks. Within the approximate computing paradigm, approximations can

be introduced in many different levels of the computing stack [11; 12], ranging from the software and algorithm [13; 14; 15; 16; 17], to system architectures [18; 19; 20], and circuit and transistor levels [21; 22; 23; 24; 25]. In this section, we mainly focus on hardware level.

One class of approximate computing techniques is voltage over-scaling (VOS), which reduces power consumption by operating circuits at lower-than-nominal supply voltages. By scaling down the voltage, the dynamic power dissipation, which is proportional to the square of the supply voltage, is significantly reduced. However, this reduction in voltage comes at the cost of increased susceptibility to timing errors, as lower voltages slow down the circuit's switching speed. These timing errors can lead to incorrect computations, making VOS particularly challenging in applications requiring high reliability [21]. To mitigate these errors, designers often implement error detection and correction mechanisms. For instance, AED-C [26] enhances the efficiency of Adaptive Voltage Over-Scaling (AVOS) for error-resilient applications, while implementing a tunable detection window that dynamically adjusts error detection accuracy to make the result of computation more stable. Huang *et al.* also proposed to use an inexact full adder (AMA1), which can operate at significantly lower supply voltages with reduced energy consumption while maintaining higher computation accuracy compared to exact adders [27].

Compared to VOS, logic approximation of underlying hardware has been explored as a more stable alternative. These approaches can be broadly categorized into two classes, which will be introduced separately below:

- **Approximate Circuits**: Architectural approximations of specific hardware components, such as adders and multipliers.
- **Approximate Logic Synthesis**: Approximation of logic synthesis procedure, which automatically generates approximations of arbitrary circuits.

Architectural approximations of specific hardware components involve intentionally

simplifying or altering the design of these components to achieve improved power efficiency, reduced area, or faster computation at the expense of precision. Such methodologies usually apply to arithmetic units, such as adders or multipliers, because of their structural architecture. For instance, ACA allows for runtime adjustments of accuracy to balance performance and power consumption, which involves cutting the carry chain to reduce critical-path delay, implementing error detection and correction circuits, and using a pipelined architecture to achieve both high performance and low power consumption with configurable accuracy based on application needs [22]. Mannepalli *et al.* proposes two novel approximate multipliers using optimized lower part constant OR adder (OLOCA) and hardware-optimized approximate adder with normal error distribution (HOAANED) to enhance performance, reduce power consumption, and minimize area for edge detection applications, maintaining relatively high accuracy when compared to exact multipliers [28]. Ahmad *et al.* also proposed a methodology for designing low-error, efficient approximate adders specifically for FPGAs [29].

On the other hand, the goal of Approximate Logic Synthesis (ALS) is to automate the design of approximate logic from arbitrary accurate circuits [30; 31; 32; 33; 34; 35; 36; 37]. To achieve this goal, a number of methods in the literature map an approximate synthesis problem into an instance where established logic synthesis techniques can be applied. For example, in SALSA, a difference circuit is created to compute the error between the original circuit and the approximated circuit [31]. The don't cares of the outputs of the approximate circuit with respect to outputs of the difference circuit can be used to simplify the approximate circuit using regular logic synthesis techniques. In SASIMI [33], a technique is proposed to identify similar signals, such that their values agree over a large number of input test cases, and then substitute one for the other, simplifying the logic. Froehlich *et al.* describe the use of formal methods such as binary decision diagrams (BDDs) and symbolic computer algebra (SCA) to generate single-output and multi-output approximate circuits respectively [36; 37]. For higher-level synthesis, ABACUS

generates variants of an input high-level Verilog description file by applying a set of possible transformations on the circuit to generate a set of mutant approximate circuit variants [30]. A multi-objective design space exploration technique is then used to identify the best set of approximate variants. Vasicek *et al.* propose evolutionary approaches, EvoApprox, on datapath circuits that are composed of basic arithmetic blocks (e.g., adders and multipliers) and logic blocks [35], where the exact circuit is encoded in a string-based representation as a "chromosome" and then a genetic algorithm mutates the circuit to create approximate versions as long as the error is kept below target.

2.2 Efficient Inference Methods for Deep Learning

As deep learning continues to make strides across various application domains, the deployment of deep DNNs on resource-constrained devices such as smartphones, IoT devices, and edge computing platforms has become increasingly common. However, these environments are often characterized by limited computational power, memory, and energy resources, making the efficient inference of DNNs a critical challenge. The demand for real-time performance and extended battery life in these devices necessitates the development of inference methods that can achieve a balance between computational efficiency and model accuracy.

2.2.1 Model Compression Techniques

One of the most widely explored strategies for efficient inference is model compression, which aims to reduce the size of the neural network while maintaining its performance. Model compression techniques include pruning, quantization, and knowledge distillation.

Pruning is a technique that involves removing redundant or less important parameters (weights) from a neural network. This reduction in parameters leads to a sparser network, which requires fewer computations during inference. There are several pruning methods,

including:

- Magnitude-based Pruning: This method removes weights with the smallest absolute values, assuming that these weights contribute less to the overall performance of the network. After pruning, the model is usually fine-tuned to recover any accuracy lost during the pruning process. For instance, LAMP [38] proposes to select layer-wise sparsity based on an adaptive importance score and use this sparsity to prune the weights.

- Structured Pruning: Unlike unstructured pruning, which removes individual weights, structured pruning removes entire structures such as neurons, filters, or layers [39]. This method is particularly effective in reducing the computational complexity and memory footprint, as it directly impacts the architecture of the network. For instance, DBLP [40] considers filters with smaller norms to have a weak activation and contribute less to the final classification decision.

- Dynamic Pruning: In this approach, the network is pruned dynamically during inference, allowing the model to adjust its complexity based on the input data. This technique is useful in scenarios where different inputs require varying levels of computational resources. FDNP [41] proposes a dynamic pruning scheme in the frequency domain to compress convolutional neural networks. Spatial-domain convolutional is converted into frequency-domain multiplication, in which weight pruning is performed to compress the model.

Pruning techniques can significantly reduce the size of the network and the number of operations required for inference, making them particularly valuable for deploying DNNs on edge devices with stringent resource constraints.

Quantization is another powerful compression technique that reduces the precision of the weights and activations in a neural network. Typically, neural networks are trained using 32-bit floating-point (FP32) arithmetic, which is computationally expensive and

memory-intensive. Quantization reduces the precision to 8-bit integers (INT8) or even lower, significantly reducing the memory footprint and accelerating the computation, especially on hardware that supports low-precision operations. Traditionally, quantization is performed after training, which is called post-training quantization (PTQ). AdaRound [42] proposes a better weight-rounding PTQ scheme that adapts to the data and loss function. Quantization-aware training (QAT) further enhances this process by incorporating quantization effects into the training process, resulting in models that are more robust to the reduced precision [43]. As an extension of QAT, PikeLPN proposed QuantNorm, allowing for quantizing the batch normalization parameters without compromising the model performance [44].

Knowledge distillation is a technique where a smaller, more efficient "student" model is trained to mimic the behavior of a larger, more complex "teacher" model. The student model learns from the teacher by minimizing the difference between its predictions and those of the teacher, often using a softened output distribution from the teacher. This approach allows the student model to achieve similar accuracy to the teacher while being significantly smaller and faster, making it more suitable for real-time inference on resource-limited devices [45].

2.2.2 Neural Architecture Search and Hardware-aware Optimization

Another avenue for achieving efficient inference is Neural Architecture Search (NAS), a technique that automates the design of neural network architectures optimized for specific hardware constraints. NAS explores a vast space of possible network designs to identify architectures that offer the best trade-offs between accuracy, latency, and energy efficiency. To avoid slow training process of NAS, Mellor *et al.* propose non-training NAS approach, which evaluates the overlap of activations between datapoints in untrained networks, using a measure based on the Hamming distance between binary activation

patterns [46]. Hardware-aware NAS further refines this process by incorporating hardware characteristics directly into the search process, ensuring that the resulting models are not only accurate but also optimized for the target deployment environment [47].

2.2.3 Dynamic Inference and Early Exiting

Dynamic inference methods have gained traction as a means of further optimizing the efficiency of DNNs. These methods allow the network to adapt its computational requirements based on the complexity of the input data or the available computational resources. One such technique is early exiting, where the model is designed with multiple exit points. During inference, the model can output a prediction after processing only a subset of the network's layers, effectively reducing the computation required for "easier" inputs while maintaining full network capacity for more challenging ones [48].

The dynamic nature of these methods makes them particularly well-suited for deployment in heterogeneous environments, where the available computational resources may vary over time or across different devices.

2.3 Efficient Training Methods for Deep Learning

Training deep learning models is a resource-intensive process, often requiring vast computational power, memory, and energy, especially for large-scale networks like Convolutional Neural Networks (CNNs) and Transformers. These challenges are magnified when training needs to be performed on resource-constrained devices or in energy-sensitive environments. Efficient training methods have thus become a crucial area of research, with various strategies being developed to optimize the training process without compromising model performance.

2.3.1 Transfer Learning

Transfer learning is a widely used technique that leverages knowledge gained from pre-trained models on a large dataset and adapts it to a related but different task with a smaller dataset. This approach significantly reduces the computational cost and time required for training because it allows the reuse of previously learned features, which can be fine-tuned for the new task [49]. For instance, a model pre-trained on a large dataset like ImageNet can be fine-tuned on a smaller dataset for a specific task, such as medical image classification. By freezing the initial layers of the pre-trained model and only training the final layers, the training process becomes more efficient, as fewer parameters need to be updated. This not only accelerates the training process but also mitigates the risk of overfitting, particularly when working with limited data.

Transfer learning is particularly advantageous in scenarios where computational resources are limited, as it allows the deployment of sophisticated models without the need for extensive training from scratch. Moreover, it enables the utilization of state-of-the-art models even in environments where training a large model from the ground up would be infeasible.

2.3.2 Parameter-Efficient Training

Parameter-efficient training techniques focus on optimizing the number of parameters that need to be learned during training, thereby reducing the computational load and memory usage. These methods are particularly useful in scenarios where deploying large models is constrained by hardware limitations.

One popular approach in parameter-efficient training is model pruning during training [50]. Unlike post-training pruning, where the model is pruned after training, this method involves gradually pruning unimportant parameters during the training process. Lottery ticket hypothesis suggests that a smaller, well-initialized subnetwork can be found early in training that can achieve similar or even better performance than the original full

network when trained in isolation [51].

Another parameter-efficient approach is low-rank factorization, where the weight matrices of the neural network are factorized into products of lower-rank matrices [52]. This reduces the number of parameters that need to be learned and can lead to faster training and inference times.

2.3.3 Low-Precision Training

Low-precision training is a powerful technique that reduces the precision of the arithmetic operations involved in training neural networks. Traditional deep learning models are typically trained using 32-bit floating-point (FP32) arithmetic, which is both memory-intensive and computationally expensive. Low-precision training reduces this precision, commonly to 16-bit floating-point (FP16) or 8-bit integer (INT8), leading to significant savings in both memory usage and computational power.

One of the major challenges in low-precision training is maintaining the stability and accuracy of the model. Lower precision can introduce quantization noise, which may lead to instability in the training process or loss of model accuracy [5]. Mixed-precision training have been developed to address these issues, where critical parts of the network are trained using higher precision, while less critical parts use lower precision [53]. This approach strikes a balance between the efficiency gains of low-precision training and the stability of high-precision computations.

CHAPTER 3

Approximate Logic Synthesis using Boolean Matrix Factorization

3.1 Introduction

Since the emergence of power as the main factor limiting the scale of the computational power, novel techniques have been proposed aiming at reducing the power and energy footprint of conventional computing systems. As introduced in Section 2.1, one of such emerging low-power techniques is approximate computing, where computational accuracy is traded for improvements in hardware cost and complexity, e.g. design area, power consumption or energy cost. Approximate computing is effective for application domains that inherently tolerate small inaccuracies in their output, such as signal processing, deep learning, and computer graphics.

A primary challenge of approximate computing is to devise techniques for automated approximate circuit synthesis that can generate approximate circuits from arbitrary exact input circuits, while offering a wide range of trade-off between accuracy and hardware metrics. Such techniques, while less optimized for specific designs, enable a more versatile approach where any input design amenable to approximations, can be readily optimized

without requiring added guidance from the designer.

The proposed approach utilizes recent advance in multivariate analysis, namely Boolean matrix factorization [54], that can reduce the dimensionality of the problem, by identifying the common bases which can be later combined to yield the original Boolean matrix. Our methodology operates on truth tables and introduces approximations in the circuit by simplifying the input truth table based on statistical analysis [54; 55]. Compared to our previous publications, this article provides the following contributions.

- We provide a unified approach to approximate logic synthesis utilizing matrix factorization. Our approach utilizes three factorization techniques, relying on different algebra. Such methodology introduces an exponentially large search space which requires careful navigation.
- In order to improve the scalability of our methodology, we partition an input circuit into manageable subcircuits [56], and perform a detailed design space exploration over factorization degrees of subcircuits to optimize the resulting approximate top-level design. Meanwhile, our approach is able to handle various error metrics, such as Normalized Hamming Distance (HD), Mean Absolute Error (MAE), *etc.*
- We provide a more comprehensive set of experiments, including the well established EPFL [57] and ISCAS '85 [58] benchmark suites commonly used in the literature. Evaluation on 15 total benchmarks, we clearly demonstrate the versatility of our proposed technique. Furthermore, we compare our approximation designs against EvoApproxLib, a library of approximated adder and multiplier circuits, in order to show that our approach reaches state-of-the-art performance.
- We implement our approach using a full stack of open-source tools, while adopting a more runtime aware approach and introducing techniques, e.g. parallelization and computation reuse, to reduce the runtime overhead of our methodology.

The organization of this chapter as follows. In Section 3.2 we overview relevant previous

work on approximate logic synthesis and the broader approximate computing paradigm. Next, in Section 3.3 we discuss the necessary background on Boolean matrix factorization, as it related to our methodology. In Section 3.4, we describe our new approaches, mainly the XOR field-based circuit approximation method. We also describe the integration of our approach in a circuit decomposition and design space exploration technique. We provide a comprehensive set of experimental results in Section 3.5. The conclusions of this work are summarized in Section 3.6.

3.2 Previous Work

Within the approximate computing paradigm, approximations can be introduced in many different levels of the computing stack [11; 12], ranging from the software and algorithm [13; 14; 15; 16; 17], to system architectures [18; 19; 20], and circuit and transistor levels [21; 22; 23; 24; 25]. In this section, we briefly discuss some of the existing researches to explore different aspects of approximate computing and their findings.

First, in software and algorithmic domains, one popular methodology is loop perforation, where the iterative computation can be stopped prematurely, to reduce the computation cost while introducing errors in precision [13; 14]. In this domain, approximations based on approximate GPU kernels [15], approximate compression [16], and approximate parallelization [17] have also been proposed.

In addition, on the computer architecture front, approximate instruction set architectures (ISA) have also been explored. Esmaeilzdeh *et al.* proposed an approximate processing pipeline within which approximate versions of all main arithmetic and logical operations are implemented as an ISA extension [18]. Similarly, utilization of approximate computing techniques for many specific computing components, such as dynamic random access memories (DRAM) [19], and cache and register file subsystems [18; 20] have also been proposed.

On circuit level, voltage over-scaling (VOS) has received significant attention [21]. Here, the operating voltage is reduced beyond safe operation thresholds reducing the energy consumption. However, the indeterministic nature of such approximations has resulted in limited applicability of such methodologies. Logic approximation of the underlying hardware have also been explored. Here, two main approaches have been evaluated; (i) architectural approximations of specific designs (such as adders and multipliers), and (ii) automated approximations of arbitrary circuits. Arithmetic blocks, due to their utilization in many other applications, have received significant attention. Here, approximate adders [22], multipliers [23; 24], and dividers [25] are few examples where architectural approximations for specific hardware blocks are proposed.

Approximate synthesis methodologies operating on arbitrary circuits have also been proposed [30; 31; 32; 33; 34; 35; 36; 37]. For example, in SALSA, a miter is created to compute the error between the original circuit and the approximated circuit [31] using existing methodologies in logic synthesis. The don't cares of the outputs of the approximate circuit with respect to outputs of the difference circuit can be used to simplify the approximate circuit using regular logic synthesis techniques. This approach was extended in ASLAN [32] to model error arising over multiple cycles. In SASIMI [33], a technique is proposed to identify similar signals, such that their values agree over a large number of input test cases, and then substitute one for the other, simplifying the logic.

For higher-level synthesis, ABACUS generates variants of an input high-level Verilog description file by applying a set of possible transformations on the circuit to generate a set of mutant approximate circuit variants [30]. A multi-objective design space exploration technique is then used to identify the best set of approximate variants. Vasicek *et al.* propose evolutionary approaches, EvoApprox, on datapath circuits that are composed of basic arithmetic blocks (e.g., adders and multipliers) and logic blocks [35], where the exact circuit is encoded in a string-based representation as a "chromosome" and then a genetic algorithm mutates the circuit to create approximate versions as long as the error is kept

below target. Raising the approximate synthesis to C-based design, Lee *et al.* propose a new technique to synthesize approximate circuit directly from C descriptions [59].

Finally, approximate computing techniques have also been deployed in specific applications such as deep learning [60], and computer vision. More recently, impact of approximate computing on end-to-end systems such as biometric security [61], and smart camera system [62] has also been studied.

3.3 Background

In this section, we describe problem of Boolean matrix factorization (BMF), as it forms the basis of our methodology. Then, we briefly discuss some existing algorithms of BMF. Matrix factorization (or decomposition) is a class of algorithms that propose to factor an input matrix $n \times m$ $\mathbf{A}$ into two matrices: a $n \times f$ matrix, $\mathbf{B}$, and a $f \times m$ matrix, $\mathbf{C}$, such that $\mathbf{A} \approx \mathbf{BC}$. In many applications the *factorization degree*, f, is required to be smaller than m in approximations in the multiplication results. Note that one can interpret the columns of $\mathbf{B}$ as *factors* or *bases* that are linearly combined using $\mathbf{C}$.

While generic matrix factorization algorithms allow for both negative and positive matrix entries, non-negative matrix factorization (NNMF) restricts the elements to non-negative values [63]. Non-negative values occur in many physical domains, such as computer vision and document clustering [64]. More recently, NNMF has been extended to Boolean matrix factorization, where all elements of all matrices are limited to '0's and '1's. Different algebra can be used for the arithmetic [65; 66]. Boolean matrix factorization algorithms have many applications, including data mining, noise detection, and document clustering.

Boolean matrix factorization has been proved to be NP-hard [67], which can also be

formulated as an optimization problem solving,

$$\text{argmin}_{\mathbf{B},\mathbf{C}}|\mathbf{A} - \mathbf{BC}|, \tag{3.1}$$

where the elements of **A**, **B** and **C** are '0 or '1'. Therefore, many algorithms take a heuristic approach. For example, in ASSO, an association matrix is computed as candidates of *bases* vectors using association rule mining [67]. Intuitively, the association matrix evaluates the likelihood among all pairs of columns in the input matrix. Then, for each candidate *base* in the association matrix, ASSO calculates a paired column by enumerating all possibilities, and picks the optimal pair in order to greedily cover '1's in the input matrix. As it is fast and straightforward, there exists one drawback, such that errors of covering '0's by '1's are irreversible. Therefore, some improvements have also been proposed, such as clustering input matrix before factorization or transposing input matrix.

Besides heuristic approach, some other methods have also been studied, which first solve non-negative matrix factorization problem and then extend to binary case [68]. Penalty function algorithm attempts to build a loss function and optimize by computing derivatives. Thresholding algorithm aims to solve **B** and **C** in real numbers, and then find thresholds to binarize two matrices. Recently, more methodologies are proposed, such as using Minimum Description Length principle [66] or Message Passing [69].

3.4 Proposed Methodology

In this section, we first describe our proposed BLASYS methodology for utilizing Boolean matrix factorization (BMF) in automated approximate logic synthesis. Here, we also discuss our techniques for improving accuracy and versatility of our methodology, by introducing XOR algebra and weighting schemes in subsection 3.4.1. Later on, we discuss the consideration required, when applying the proposed methodology on larger circuits.

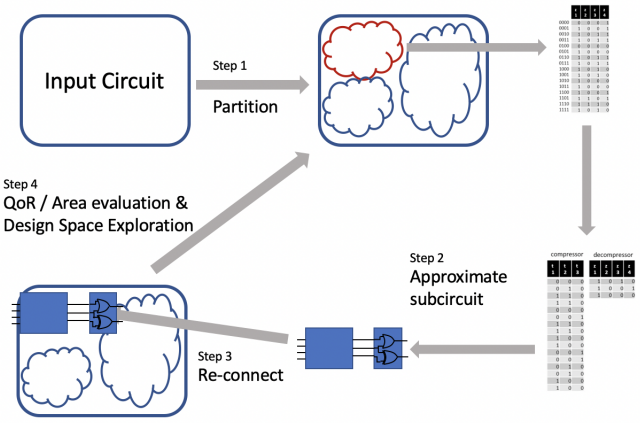


Figure 3.1: General flow of BLASYS for approximate logic synthesis using Boolean matrix factorization (BMF).

Since, the proposed BMF based methodology operates on truth tables, in order to keep the truth table within manageable size, we propose to use circuit partitioning. We then introduce methodologies for design space exploration (DSE) of the resulting search space in subsection 3.4.2.

Figure 3.1 illustrates the general flow of BLASYS algorithm. As demonstrated in the figure, an input circuit can optionally be decomposed into smaller subcircuits, if required by its input size. Next, each subcircuit is approximated to a specific degree, and the approximate components are connected together to generate the approximate design. For each approximate design, the Quality of Results (QoR) and design area are evaluated, which is then used for design space exploration and guide the factorization degree during next iteration. Next subsections describe, in more details, the exact inner workings of the proposed technique.

3.4.1 Approximate Synthesis Using Boolean Matrix Factorization

As discussed in Section 6.3, Boolean matrix factorization is a special extension of matrix factorization, where all elements of all matrices are limited to '0's and '1's. There exists an inherent connection between logic circuits and Boolean matrix, where truth

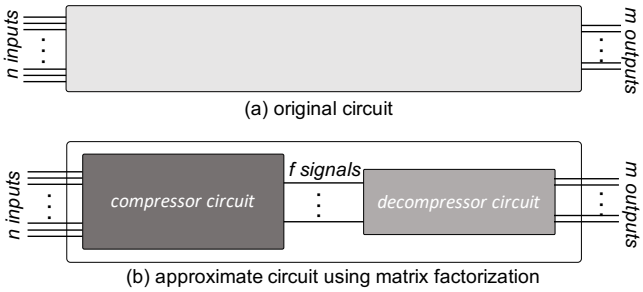


Figure 3.2: Utilization of binary matrix factorization for approximate logic synthesis. (a) an arbitrary input circuit, and (b) the compressor and decompressor circuits used in binary matrix factorization methodology.

tables of circuits can be represented by Boolean matrix.

To use Boolean matrix factorization methods for approximate logic synthesis, the truth table of the input circuit is first generated and given as the input matrix for a binary matrix factorization algorithm, where the factorization degree, f, is chosen to be smaller than the number of outputs of the original circuit. The two factorized matrices from the algorithm are then treated as truth tables synthesized into two subcircuits and connected together to generate the approximate circuit as illustrated in Figure 3.2. In Figure 3.2, the first subcircuit receives the n outputs as the original circuit, but instead produces $f < m$ outputs, and thus referred to as the *compressor circuit*. The second subcircuit receives $f < m$ inputs and produces m outputs and thus referred to as the *decompressor* circuits. In prior work where only semi-ring Boolean algebra is considered, the implementation of the decompressor is very simple as it uses a network of only OR gates [54].

Factorization Algebra

Boolean matrix factorization aims at minimizing the number of mismatches between an input matrix and the approximate multiplication result of the factorized matrices. In Boolean matrix factorization, the multiplications are carried out using the logical AND operation, and the addition operation can be either based on semi-ring Boolean algebra, or field modulo-2 algebra. In the case of Boolean matrix factorization (BMF), the algebra implements a semi-ring algebra, where the addition is carried out using logical

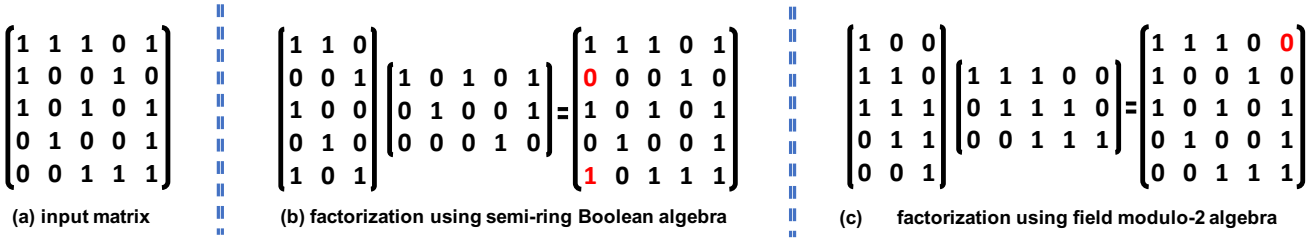


Figure 3.3: Example of binary matrix factorization using different algebra. (a) input matrix, (b) matrix factorization using Boolean algebra where addition is carried out using logical ORs, and (c) matrix factorization using modulo-2 algebra, where the addition is carried out using logical XORs. The errors are highlighted in red.

OR, i.e., $1 + 1 = 1$. In the case of field modulo-2 algebra, the addition is carried out using logical XOR, i.e., $1 + 1 = 0$ Figure 3.3 shows an example of an input matrix as well as the factorized matrices and their multiplication result for both Boolean and Modulo-2 arithmetic.

Using different arithmetic can result in significantly different characteristics in the factorized matrices as well as the best approximation degree. In the specific case of Figure 3.3, modulo-2 algebra generates better quality of results. Next we describe the utilization of binary matrix factorization methodologies in the approximate logic synthesis problem.

One possible drawback of using OR-based Boolean arithmetic is that the number of bases from $\mathbf{B}$, i.e., outputs of the compressor circuits, that can be combined to produce one column in $\mathbf{C}$, i.e., output of the decompressor circuit, is limited. ORing two bases from $\mathbf{B}$ with a '1' in the same location will lead to a '1' in the corresponding location in the resulting output column, and this result will not change regardless of any additional bases that can be further ORed with the two. In contrast, in modulo-2 algebra, $1 + 1 = 0$, thus a '1' can be reduced back to '0' and therefore combining additional bases in modulo-2 implementation can offer more diversity in the results. Interestingly, modulo-2 based approximate logic synthesis closely resembles that of the Boolean based approach, where the only differences are (1) a modulo-2 approach is utilized for the matrix factorization, and (2) the decompressor circuit needs to be mapped to network of XOR gates instead of a OR gates.

Currently there are no modulo-2 matrix factorization algorithms and the complexity of the problem is unknown [66]. Note that the Boolean counterpart is proven to be NP-Hard, and therefore all existing algorithms are based on heuristics. To enable our methodology using modulo-2 arithmetic, we devise a simple heuristic based on the methodologies used for the Boolean matrix factorization. More specifically, we use ASSO [65; 66] for initial matrix factorization, where we further do an exhaustive search for the decompressor matrix to minimize the error assuming modulo-2 arithmetic. Note that this operation incurs a timing complexity of $O(m2^n)$ as different columns of the decompressor circuit can be identified independently.

Finally, as different columns of the decompressor matrix represent different combinations of the compressor circuits, one can mix the OR-based and XOR-based methodologies, where some outputs are implemented using OR and other outputs are implemented using XORs, i.e., the decompressor circuit uses both OR and XOR gates. We refer to this approach as XOR/OR, as it chooses the better outcome of OR versus XOR results to implement. We will evaluate OR, XOR and OR/XOR methodologies in the experimental results highlighting the benefit of each in different circumstances.

Output Weights

In BMF algorithms, the objective is to minimize $||\mathbf{M} - \mathbf{BC}||_2$, which translates to Hamming distance in Boolean systems. In approximate circuit design, however, such metric does not provide a good representation of QoR in many cases. As an example, if the columns of a m-column matrix represent an m bit signal, minimizing the Hamming distance as the cost function can lead to significant errors in numerical value. For instance, a bit flip in the least significant bit will lead to a numerical error of 1, whereas a bit flip in the n^{th} bit leads to an error of 2^{n-1}.

To account for the bit significance, we augment existing BMF algorithms with custom QoRs enabling weighted cost functions. Specifically, we propose to define the cost function

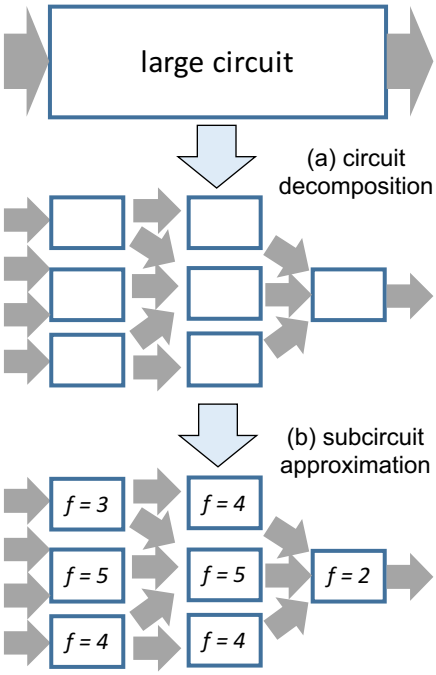


Figure 3.4: Illustrated methodology for partitioning circuits.

as $||(\mathbf{M} - \mathbf{BC})\mathbf{w}||_2$, where $\mathbf{w}$ is a constant weight vector, instead of $||\mathbf{M} - \mathbf{BC}||_2$ as the standard hamming distance cost function. Here, if the numerical difference is the objective QoR, then $\mathbf{w}$ will be defined to introduce bit significances based on powers-of-two (e.g., 8, 4, 2, 1); therefore, giving different numerical weights for different bit positions. In our experiments, we modify the ASSO [66] algorithm as to penalize mismatches on higher bit indices more than lower significant bits. We will provide experimental results showcasing the benefits of such weighting schemes in contrast to uniform weights (Hamming distance) in Section 6.5.

3.4.2 Partitioning and Design Space Exploration

Since the truth table size of a circuit grows exponentially with the number of its inputs, we break down any large circuit into sub-circuits, where each sub-circuit has a limited number of inputs (e.g., $n \leq 10$) and then approximate each sub-circuit individually using the proposed binary matrix decomposition method with mixed OR/XOR decompressor implementation.

As our methodology operates on the truth table of the input circuit, the size of the input matrix, i.e. the number of rows, grows exponentially as the number of primary inputs increases. Furthermore, BMF is a NP-hard problem, and the existing methodologies are based on heuristics [63; 65; 66]. Therefore, the applicability of our method can be limited as the complexity of the circuit increases. Therefore, we propose a circuit decomposition technique to scale the BMF algorithm for larger circuits. The overall idea of our method is to first partition a large circuit into a number of subcircuits, such that each subcircuit has a maximum of k inputs as illustrated in Figure 6.4.a and then each of the subcircuits is approximated as shown in Figure 6.4.b. The values for k and m, the number of outputs, are determined based on the afforded runtime of the factorization algorithm.

To limit the number of inputs and outputs in subcircuits, we propose to use hypergraph partitioning algorithm [56] recursively until all subcircuits have a maximum of k inputs and maximum of m outputs. Also, we will discuss the relation between size of subcircuits and performance of approximation in Section 6.5.

Dividing a large circuit into smaller subcircuits of size $k \times m$ requires a change to the way we compute the QoR. More specifically, we can no longer evaluate the accuracy of a subcircuit in isolation, as errors in one component can propagate through the circuit leading to larger errors in the final outputs. Therefore, in our work instead of evaluating the QoR of a subcircuit individually, we evaluate the QoR of the entire approximate circuit, denoted by $Cir(s_i \rightarrow T_{s_i,f_i})$, where an accurate subcircuit, s_i, is substituted by its approximate version, T_{s_i,f_i}, with a factorization degree of f_i.

Greedy Heuristic DSE

Our design space exploration algorithm starts by identifying the sub-circuits; we calculate the possible approximate realizations for each sub-circuit using various factorization degrees, OR/XOR implementations. We then greedily explore the space of generated approximate sub-circuits to identify a good approximation order. We assess the QoR

as measured by a user-defined error metric for each of its approximate realization by substituting the original subcircuit by its approximate realization and evaluating the outcome using the primary outputs of the circuit. The sub-circuit that leads to the smallest value of loss function is then chosen, and its approximated realization is then substituted in the main circuit. This sub-circuit approximation process is repeated until the maximum target error is reached.

Since a large input circuit will have multiple subcircuits, the order and the degree to which the approximations are introduced to the circuit has to be carefully analyzed. We devise Algorithm 1 to gradually approximate the circuit. In our algorithm, first, the circuit is partitioned into smaller subcircuits (line 1). In the next stage (lines 3-9) and for each subcircuit, the set of *potential* approximate versions under various approximation degrees are profiled. Next, starting from the accurate design, approximations are gradually added to the input design by exploring the neighbors of the current design (lines 14-24). Here, neighbors of a given design are defined as top-level circuits for which the degrees of approximation only reduce by *one* in *one* subcircuit. Here in lines 16-20, each neighbor is synthesized, where its QoR metric and chip area are assessed. The subcircuit with the least loss value, defined in line 18, is then chosen to replace the current circuit for next iteration in lines 21-23. The process is repeated iteratively until the QoR gets higher than a predefined threshold. The output approximation Cir is the one with smallest chip area in explored design space.

Loss function

In Algorithm 1, our goal is to reduce design area and power consumption as much as possible with a fixed error threshold. We choose design area as an estimation of approximation degree, and propose the following loss metric to greedily explore the design space. Assuming we denote design area of accurate circuit by $area(ACir)$, the approximate circuit by $area(Cir_i)$, and degradation in QoR by $QoR(Cir_i)$, the loss is

Algorithm 1: BLASYS: Boolean Level Approximate Circuit Synthesis

Input : Accurate Circuit $ACir$, Error Threshold
Output : Approximate Circuit Cir

subcircuits=Decompose input circuit $ACir$ by using k-way hypergraph partitioning recursively
// Factorization profiling Phase
for *each subcircuit* s_i *with* $m_i \leq m$ *outputs* **do**
    M=Construct truth table of s_i
    // profile for every possible factorization degree
    for f=*1 to* m_i*-1* **do**
        $[\mathbf{B}, \mathbf{C}]$ = BMF(**M**, f)
        $T_{s_i,f}$=Construct truth table of **BC**
    end
end
// Circuit Space Exploration Phase
Cir=ACir;
ExploredSpace=Empty List;
Let $f_i = m_i$ for all subcircuits s_i
while $QoR(Cir) \leq$ *threshold* $+\epsilon$ **do**
    for *each subcircuit* s_i *with* $f_i > 1$ **do**
        Cir'=$Cir(s_i \rightarrow T_{s_i,f_i-1})$
        $loss_i = (area(Cir') - area(ACir)) \,/\, QoR(Cir')$
        Add Cir' into ExploredSpace
    end
    $b = \arg\min_i(loss_i)$
    $Cir = Cir(s_b \rightarrow T_{s_b,f_b-1})$
    $f_b = f_b - 1$
end
Cir=Best design in ExploredSpace
return Cir

defined as

$$L_i = \frac{area(Cir_i) - area(ACir)}{QoR(Cir_i)} \tag{3.2}$$

For each iteration, we choose the neighbor with smallest loss to replace the current circuit. Recall that neighbors of a given design are defined as top-level circuits for which the degrees of approximation only reduce by *one* in *one* subcircuit. To minimize this loss metric, on one hand, a larger degradation in design area is preferable. On the other hand, since the loss value is negative, a smaller degradation in QoR is also preferable in order to minimize the loss. The intuition of the loss function is that, the design space of approximate circuit is expected to reduce sharply, while the design accuracy should remain relatively high. Thus, we balance the trade-off between reduction in design area and QoR. Although design area and power consumption are not strictly proportional to

each other, design area is a better representative of circuit complexity, and able to reflect the changes in other metrics in general.

This loss metric performs even better with output-weighting scheme. Since different outputs could have different weights in QoR estimation, our loss metric will first explore design space which approximates less significant output bits, and then gradually move to more significant ones.

The loss function may be further modified in a stepwise manner. In each iteration, we first choose from designs with very small degradation of QoR (*e.g.* 0.01%). If there is no better design in this range, we then gradually increase the range of QoR degradation. The reason for this stepwise loss metric is to prevent design accuracy from dropping rapidly.

Error Metric

In previous discussion, we analyze that QoR in Equation 3.2 can be either Hamming distance error or any output-weighting scheme. In practice, however, we may further generalize the error metric to any function that takes both simulation results of exact circuit and approximate circuit as input.

In Section 3.4.1, since we directly factorize the truth table of the exact input circuit, the output-weighting scheme is proposed to approximate arithmetic circuits with natural numbers. However, in Algorithm 1, we factorize the truth table of each subcircuit (line 7). It is difficult to analyze the significance of inputs of a subcircuit, or in other word, how inputs of a subcircuit impact primary outputs. Therefore, when factorizing the truth tables of subcircuits, we just use Hamming distance error (line 7), and later use objective error metric to choose the optimal design in design space exploration (line 15-20). In practice, to compute QoR, we simulate both exact circuit and approximate circuit with the same testbench, and compare the simulated results. If we denote simulated results of exact circuit as T and ones of approximate circuit as T', we may use any function f of T

and T' to guide the design space exploration, such as

$$QoR = f(T, T') \tag{3.3}$$

For example, one common error metric is Mean Absolute Error (MAE), where we convert simulation results to real numbers and compute mean of absolute of the difference. Assume that our testbench has N test vectors. For i^{th} test vector, the simulation result of exact circuit is t_i and the one of approximate circuit is t'_i, QoR can be formulated as

$$\text{QoR} = \frac{1}{N}\Sigma_{i=1}^{N}|\mathcal{R}(t_i) - \mathcal{R}(t'_i)|, \tag{3.4}$$

where $\mathcal{R}$ is the function that converts simulated results from binary to real numbers.

3.4.3 Hyperparameters

Besides Algorithm 1, we also introduce a few hyperparameters, in order to control the range of explored design space and balance trade-off between runtime complexity and approximate performance.

Step size

In Algorithm 1, $Cir' = Cir(s_i \rightarrow T_{s_i,f_i-1})$ (line 17) means that the factorization degree for one subcircuit is decreased only by one. In practice, in order to factorize truth table efficiently, large input circuits might be partitioned into hundreds of subcircuits. To speed up Algorithm 1, we are able to set a larger integer as step size. With a larger value, each approximation realization will take a larger step, meaning that there will be a more significant reduction in design area and QoR. In this case, the algorithm will converge more quickly with a set error threshold. On the other hand, larger step size will ignore many approximate design in-between, lead to smaller exploration space.

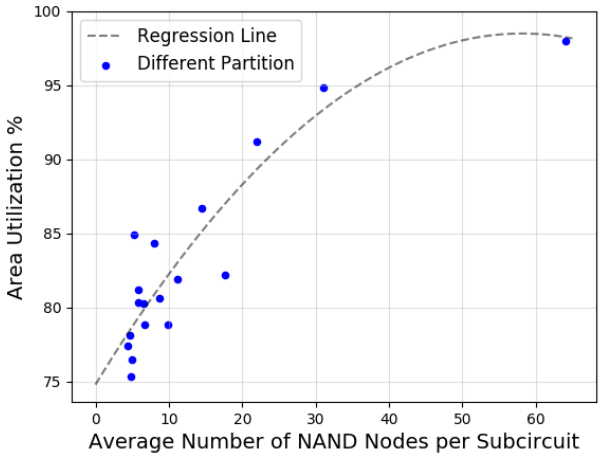


Figure 3.5: Relationship between average size of subcircuits and design area, for 0.1% Mean Absolute Error on 7-bit unsigned multiplier.

Size of subcircuits

The first step of Algorithm 1 is to break down input circuit in subcircuits, whose number of inputs and outputs is limited. Algorithm 1 calls k-way hypergraph partitioning recursively, and may further break down subcircuits to smaller ones, which introduces more subcircuits. Figure 3.5 demonstrates relationship between average size of subcircuits, which is assessed by average number of NAND gates, and the area of output circuit. We test on 7-bit unsigned multiplier with 0.1% error threshold. Generally, when average size of subcircuits is smaller, which means input circuit is partitioned into more pieces of subcircuits, the approximate circuit has smaller design area. On one hand, our algorithm relies on synthesis capacity. Smaller subcircuit corresponds to smaller truth table, which then leads to smaller truth tables of *compressor* and *decompressor*. In practice, it is easier to optimize synthesis result with a smaller truth table. On the other hand, with smaller subcircuits, each of them represents less information in terms of top-level design, and each step of approximation leads to a slower degradation in QoR. With a fixed error threshold, we are able to explore more designs with smaller subcircuits, which is more likely to end up with a better approximate design.

However, with smaller subcircuits, the algorithm take longer to converge to the error

threshold. As Algorithm 1 (line 16) suggests, for each iteration, it will evaluate n designs, where n is the number of subcircuits. In practice, having more subcircuits is more likely to improve approximation results, but will dramatically increase runtime.

Multi-path exploration

In Algorithm 1 (line 22), the current design is substituted by the best approximation realization in each iteration based on the loss metric. In order to expand explored design space for global optimum, we also propose a multi-path version of greedy DSE. Instead of only choosing the best approximation realization, the first b best design are chosen as current designs and explored in each iteration. Specifically, all neighbors of b designs are assessed. Then again, among all neighbors, best b designs are chosen to substitute original b designs as starting point of next iteration. Multi-path exploration has a larger explored design space, which is roughly b times than before, and thus often leads to a better design with the same error threshold.

3.5 Experimental Results

In this section, we discuss our experimental results and highlight the benefits offered by the proposed methodology. For hardware metrics, all designs are implemented in Verilog and synthesized using ABC logic synthesis tool [70] using an industrial 65 nm technology node at the typical processing corner. We evaluate combinational benchmarks available in ISCAS [58] and part of EPFL arithmetic benchmark suite [57]. For smaller benchmarks, we generate the truth table and directly pass the truth table to the factorization algorithm. For the larger ones, however, we first decompose the circuit as described in Subsection 3.4.2. Furthermore, we compare approximate designs from our algorithm against EvoApproxLib, a library of approximate arithmetic circuits, to demonstrate that our algorithm is able to reach state-of-the-art performance.

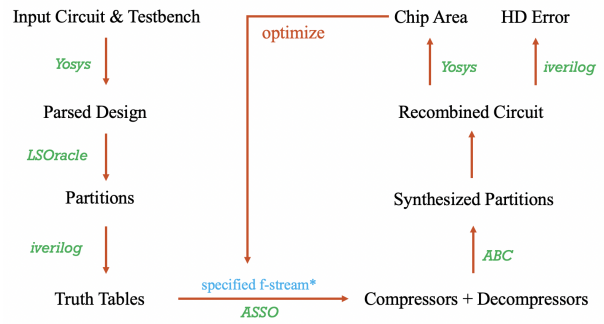


Figure 3.6: Structure of BLASYS Tool-chain.

For design accuracy, we report the normalized Hamming distance (HD), which is defined as

$$\text{Normalized HD} = \frac{|\mathbf{A} - \mathbf{BC}|}{Nm}, \tag{3.5}$$

and mean absolute error (MAE) defined as

$$\text{MAE} = \frac{1}{N}\Sigma_{i=1}^{N}\frac{|R_i - R_i'|}{2^m}, \tag{3.6}$$

for logical and binary numerical outputs, respectively. Here, N represents the size of the test vectors while R_i and R_i', represent the accurate and approximate numerical results. m is the number of primary outputs. Furthermore, for smaller circuits, we define the accuracy over all possible inputs, while for larger networks, we estimate standard deviation of QoR with different number of test vectors, and choose a proper size as discussed in the first subsection.

3.5.1 Work Flow

In this subsection, we briefly describe the work flow of our methodology. Figure 3.6 demonstrates various tools in BLASYS tool-chain, which is used for all following experiments [71].

To begin with, Yosys [72] parses the input exact circuit and assesses its chip area with a given liberty file, which in our case, is an industrial 65 nm technology node. Using the provided set of test vectors, Icarus Verilog [73] simulates the input exact circuit, which is then used for QoR estimation.

Next, LSOracle [74] is used to partition the input circuit to multiple subcircuits, each of which has a similar size. Considering runtime efficiency of Boolean matrix factorization, our methodology partitions an input circuit until all subcircuits have less than 10 inputs and 10 outputs. Then a set of test vectors is generated for each subcircuit. We use the ASSO algorithm [66] to factor each truth table based on a vector called *f-stream*, which consists of factorization degree for each subcircuit. This vector is determined by the design space exploration method as discussed in Section 3.4.2. As a result, each truth table is factorized into a compressor and decompressor. We use ABC [70] to synthesize the compressor matrix to a circuit and uses a network of logic OR or XOR to represent decompressor, depending on heuristic search of XOR/OR-based approach. Thus, an approximated version of the input circuit can be obtained by recombining all approximated subcircuits. Afterwards, we use Yosys to estimate the chip area of the approximate circuit and executes a simulation using the input set of test vectors. From the original and approximated simulation results, QoR can be defined arbitrarily based on the functionality of input circuit. In our experiments, we consider the Normalized Hamming Distance error (HD) and Mean Absolute Error (MAE). The area reduction ratio and QoR are used to optimize f-stream iteratively as mentioned in Algorithm 1.

The implementation of work flow is available at `http://github.com/scale-lab/blasys`.

3.5.2 Number of test vectors

Before experimenting our methodology with various benchmarks, we need to create testbench for each benchmark. Since most benchmarks have large number of inputs, it

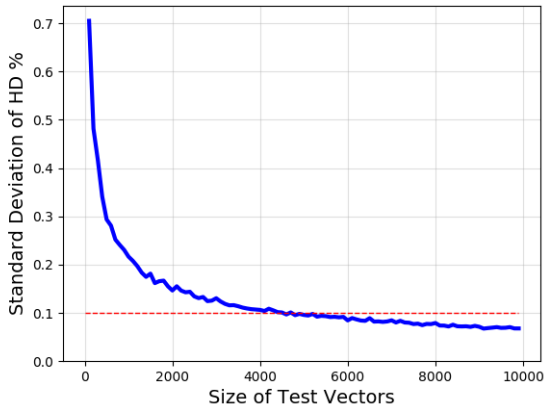


Figure 3.7: Relationship between number of test vectors and standard deviation in Normalized Hamming Distance in benchmark Max.

Table 3.1: Size of test vectors required to achieve 0.1% and 0.2% standard deviation of Hamming distance.

Name	I/O	Area (um^2)	0.2% σ Size	0.1% σ Size
Adder	256/129	1743.48	700	2100
Barrel shifter	135/128	4878.00	600	2100
Max	512/130	4320.00	1500	4600
Multiplier	128/128	37799.28	500	2000
Sine	24/25	8308.44	2300	9400
Square	64/128	25733.16	5400	-

is impossible to enumerate all possible combination of test vectors. Therefore, for each benchmark, we generate a set of distinct random test vectors of size s. To find out proper size for each benchmark, we evaluate standard deviation of Normalized Hamming Distance with different sizes of test vectors. Specifically, for one benchmark, we generate 200 random sets of test vectors respectively, from size 100 to 10,000 for every 100, and assess standard deviation of Normalized Hamming Distance for each size. Figure 3.7 illustrates relationship between number of test vectors and standard deviation of Normalized Hamming Distance in Max circuit of EPFL benchmarks. After reaching 0.1%, reduction of standard deviation becomes slower and standard deviation begins to converge. Considering runtime efficiency of our algorithm, the number of test vectors cannot be arbitrary large. Therefore, sizes with 0.1% standard deviation is reasonable in terms of both accuracy and efficiency. Table 3.1 demonstrates the number of test vectors required to achieve below 0.1% and 0.2% standard

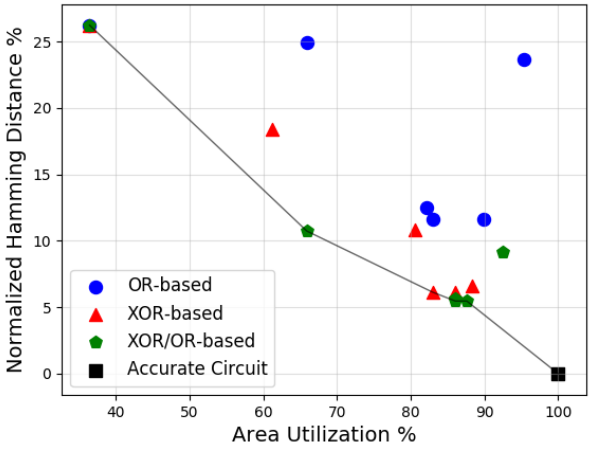


Figure 3.8: Difference between OR-based, XOR-based and XOR/OR-based method on x2 benchmark.

deviation of Normalized Hamming Distance in EPFL arithmetic benchmarks.

3.5.3 BMF-based Approximate Logic Synthesis

Semi-Ring vs Field Algebra

In this section, we compare approximate results among different boolean matrix factorization algebra. As Section 3.4.1 mentions, semi-ring boolean algebra is implemented by ASSO algorithm, which is also referred to as OR-based. In order to implement field modulo-2 algebra (XOR-based), we perform an exhaustive search over the results of semi-ring algebra. Specifically, for $\mathbf{A} \approx \mathbf{BC}$, we fix $\mathbf{B}$ and greedily replace columns in $\mathbf{C}$ with field modulo-2 algebra. Moreover, we mix OR-based and XOR-based method and derive XOR/OR-based method. After computing OR-based $\mathbf{A} \approx \mathbf{BC}$ by ASSO algorithm, we fix $\mathbf{B}$, and for each column in $\mathbf{C}$, we do an exhaustive search with both semi-ring algebra and field modulo-2 algebra. Then the one which leads to smallest QoR degradation is chosen. In this case, the decompressor circuit uses both OR and XOR gates. We evaluate OR-based, XOR-based and XOR/OR-based method on x2 benchmark in LGSynth 91. Since x2 is a small benchmark, we generate the truth table and directly pass the truth table to the factorization algorithm without partitioning. Figure 3.8 demonstrates the

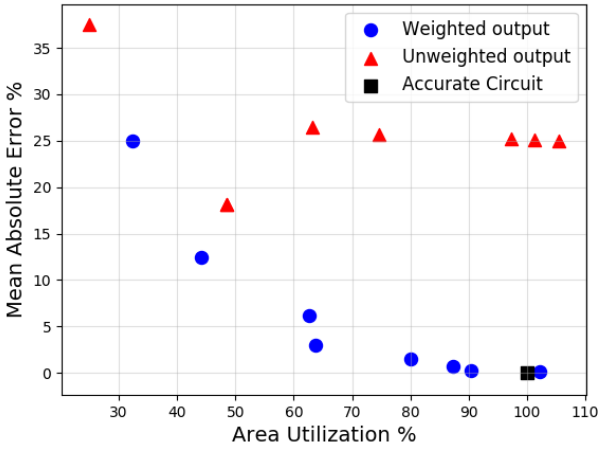


Figure 3.9: Benefit offered by output weight scheme on 8-bit unsigned adder.

approximate results from three methods. x2 benchmark has 7 output bits. Therefore, each method derives 6 approximate designs, ranging from approximation degree 1 to 6. According to Figure 3.8, with XOR-based and XOR/OR-based method, we make huge improvement in terms of area saving with similar Hamming distance error. And in most case, XOR/OR-based method has best performance. With 5.47% Hamming distance error, XOR/OR-based method can save 14.00% design area. For designs with higher error, XOR/OR-based method can save 34.11% design area with 10.74% Hamming distance error, which significantly outperforms other two methods.

Output Weight Schemes

As Section 3.4.1 mentions, considering that significance of output bits may be different, output weights in BMF algorithm sometimes improve approximate results. For example, with arithmetic circuit which outputs binary numbers, bit flips in least significant bit and a more significant bit have different impact on QoR. Therefore, for unsigned arithmetic circuits, we introduce output weight into ASSO algorithm, where n^{th} output bit has weight 2^{n-1}. We approximate 8-bit unsigned adder with both unweighted and weighted BMF algorithm. To eliminate the interference of exhaustive search in XOR-based method and highlight the benefit of using output weights, we only use OR-based method in this section.

Table 3.2: ISCAS '85 benchmarks evaluated using the proposed methodology with normalized Hamming distance.

Name	Original			5% Error Metric			10% Error Metric			15% Error Metric		
	Area (um^2)	Power (uW)	Delay (ns)	Area %	Power %	Delay %	Area %	Power %	Delay %	Area %	Power %	Delay
c1355	457.92	64.20	0.81	6.8	6.8	1.7	6.1	6.1	1.7	5.4	5.4	1.7
c17	-	-	-	-	-	-	-	-	-	-	-	-
c1908	339.84	52.90	1.25	39.3	37.8	28.8	23.9	22.9	23.2	20.0	19.9	23.7
c2670	625.68	219.00	1.16	36.0	28.5	65.4	24.3	19.0	50.6	15.2	13.2	34.5
c3450	959.76	222.00	1.75	60.7	71.6	93.1	56.2	67.1	88.8	50.3	64.4	90.7
c432	152.64	38.60	1.62	85.6	77.7	85.6	83.0	75.1	86.3	71.5	53.6	67.4
c499	460.80	91.50	0.88	47.0	39.8	99.3	21.3	21.9	57.9	19.1	18.7	57.8
c5315	1543.68	487.00	1.31	59.3	58.3	77.2	36.3	32.9	72.1	21.3	18.9	60.7
c6288	3066.84	264.00	4.39	96.3	83.3	91.4	93.7	92.8	92.0	90.3	110.2	97.3
c880	362.16	75.90	1.34	56.6	50.0	53.0	34.5	29.0	32.8	14.3	11.3	25.0
Average				54.2	50.4	66.2	42.1	40.8	56.2	34.2	35.1	51.0

Figure 3.9 demonstrates the necessity of using output weights. Since outputs of adder are numerical results, we use mean absolute error (MAE) as QoR metric. As Figure 3.9 shows, output weight scheme provides decent approximate results with good QoR performance, while approximate designs from unweighted scheme have much higher mean absolute error, which are all above 18%. If no output weight is provided, BMF algorithm will factorize truth table while minimizing number of total flipped bits. However, the algorithm does not consider bit significance. Therefore, although more bits in truth table are accurate, more significant bits might be flipped, which leads to much higher mean absolute error.

3.5.4 Design Space Exploration

In previous subsection, we approximate small benchmarks x2 and 8-bit unsigned adder by directly passing the truth table to the factorization algorithm without partitioning. As Section 3.4.2 mentioned, the size of truth table grows exponentially with the number of primary inputs. In order to approximate larger circuit, we first partition input circuit into subcircuits with maximum 10 inputs and 10 outputs, generate truth table for each subcircuits, and perform BMF on each truth table of subcircuit. Since output bit significance within each subcircuit is hard to analyze, when approximating each subcircuit, we use XOR/OR-based method and the target QoR metric to evaluate the simulation results that guide the design space exploration. In this section, we demonstrate the

Table 3.3: EPFL arithmetic benchmarks evaluated using the proposed methodology with normalized Hamming distance.

Name	Original			5% Error Metric			10% Error Metric		
	A. (um^2)	P. (uW)	D. (ns)	A. %	P. %	D. %	A. %	P. %	D. %
Adder	1325.16	59.40	11.56	89.4	94.8	90.8	79.4	84.0	80.9
Barrel shifter	2828.88	1270.00	2.69	95.8	79.5	105.6	90.0	64.7	88.5
Max	3131.28	851.00	13.45	91.0	65.5	114.3	77.6	58.0	94.3
Multiplier	30417.48	1230.00	12.24	87.7	78.4	99.4	80.5	57.6	93.8
Sine	6608.16	754.00	10.08	84.3	81.2	93.1	71.7	65.2	79.9
Square	24736.32	876.00	9.48	95.8	93.6	85.8	88.5	80.7	75.5
Average				90.7	82.2	98.2	81.3	68.4	85.5

approximate result with design space exploration on ISCAS and EPFL benchmarks. We also compare our approximate results against EvoApproxLib, which is a well-established library of adders and multipliers.

Table 3.2 demonstrates approximate designs of ISCAS '85 benchmarks. Since these benchmarks are not arithmetic circuits, we use Normalized Hamming Distance as QoR metric. For each benchmark, we set 3 error thresholds, which are 5%, 10% and 15%, and evaluate best approximate designs for them. Since c17 benchmark only has 2 primary outputs, our algorithm has only 1 factorization degree, where Hamming distance is above 15%. Within 5% hammming distance error, on average the area utilization is 54.18% and power consumption is 50.44% of original. Within 10% Hamming distance error, the area utilization drops to 42.14% and power utilization is 40.75%. Therefore, our algorithm shows remarkable saving of area and power on ISCAS '85 benchmark.

Furthermore, Table 3.3 summarizes approximate designs of EPFL arithmetic benchmarks. This benchmark suite has 10 circuits, which have larger chip areas than ISCAS '85. Due to computational capacity, we test our algorithm on 6 benchmarks. Since EPFL benchmark suite does not provide bit numbering of outputs, we use normalized Hamming distance as QoR metric. For each benchmark, we set two error thresholds for approximate design, which are 5% and 10%. Within 5% Hamming distance error, the area utilization drops to 90.7% and power utilization drops to 82.2%.

Table 3.4: Comparison between EvoApproxLib and BLASYS on 7-bit unsigned multiplier

EvoApproxLib			BLASYS			Naive		
QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)
0.0299%	448.20	82.40	0.0290%	445.68	74.40	-	-	-
0.0515%	417.60	79.00	0.0488%	421.92	75.00	-	-	-
0.1400%	351.72	63.10	0.1337%	356.40	61.70	-	-	-
0.2428%	272.16	44.80	0.2369%	317.16	62.40	-	-	-
0.4583%	225.36	41.80	0.4532%	252.72	39.00	0.3860%	365.11	57.62
1.1530%	133.20	22.30	1.1203%	125.28	19.70	1.1491%	258.53	38.74
2.2738%	80.64	14.00	2.2298%	69.12	12.30	-	-	-
5.0938%	30.96	4.26	4.4771%	30.96	4.36	2.6384%	135.14	23.78

3.5.5 Comparison of Arithmetic Blocks

Finally, we test our method on four commonly used arithmetic circuits and compare results against EvoApproxLib, which provides approximate designs for adders and multipliers. As baseline, we also compare against a naive approximate computing approach, where we only compute with n-most significant bits and ignore the less significant ones. For example, the verilog code below shows a naive approximate design of 8-bit unsigned multiplier, which only takes 6-most significant bits into consideration and assigns the rest bits to 0. Notice that such approximate design is effectively a 6-bit unsigned multiplier.

```
1 module mult8u_approx(A, B, O);
2   input [7:0] A, B;
3   output [15:0] O;
4   wire [11:0] N;
5
6   assign N = A[7:2] * B[7:2];
7   assign O = {N, 4'b0000};
8 endmodule
```

Since circuits in EvoApproxLib are synthesized from a different standard cell library, we first synthesize their approximate designs with the same industrial 65 nm technology

Table 3.5: Comparison between EvoApproxLib and BLASYS on 8-bit unsigned multiplier

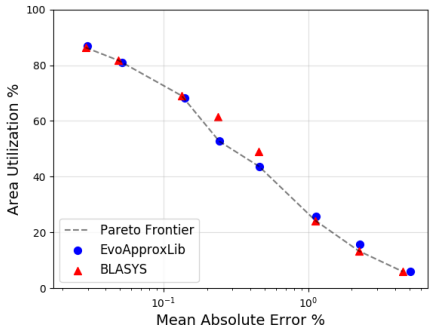
EvoApproxLib			BLASYS			Naive		
QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)
0.0002%	682.92	120.00	-	-	-	-	-	-
0.0014%	666.72	113.00	0.0011%	640.08	92.10	-	-	-
0.0076%	612.00	106.00	0.0069%	622.44	92.80	-	-	-
0.0370%	522.00	88.20	0.0346%	534.96	72.60	-	-	-
0.1952%	358.56	47.40	0.1757%	413.64	54.60	0.1941%	507.01	87.60
0.8859%	170.64	24.10	0.7973%	239.76	31.60	0.5802%	365.11	57.62
4.8338%	26.28	3.42	4.4782%	52.56	5.13	2.8324%	135.14	23.78

Table 3.6: Comparison between EvoApproxLib and BLASYS on 16-bit unsigned multiplier

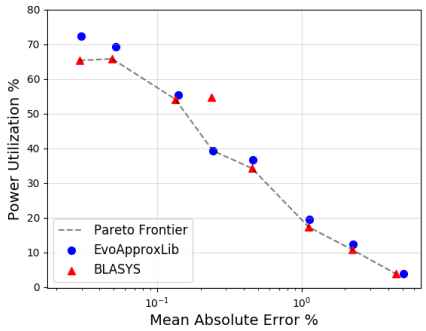
EvoApproxLib			BLASYS			Naive		
QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)
3e-10	3056.40	287.00	0.00%	3038.76	265.00	-	-	-
5.7e-09	2900.88	275.00	5.1e-09	2925.72	251.00	-	-	-
4.5e-08	2665.80	246.00	3.4e-08	2702.16	242.00	-	-	-
7.5e-07	2291.76	218.00	7.5e-07	2322.36	215.00	-	-	-
7.7e-06	1735.92	170.00	7.3e-06	1864.80	155.00	7.5e-06	2532.07	237.29
0.0150%	1182.24	121.00	0.0110%	1100.16	95.70	0.0115%	1635.71	152.67
0.1000%	732.24	65.60	0.0958%	654.48	49.20	0.0963%	810.26	75.28
1.5500%	225.72	18.30	1.4824%	166.68	10.60	1.5448%	258.53	38.74
18.750%	2.16	0.09	-	-	-	10.9306%	31.48	4.94

Table 3.7: Comparison between EvoApproxLib and BLASYS on 16-bit unsigned adder

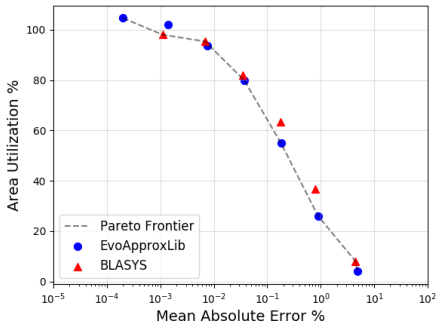
EvoApproxLib			BLASYS			Naive		
QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)	QoR	Area (um^2)	Power (uW)
0.0002%	167.40	56.50	-	-	-	-	-	-
0.0018%	134.28	46.40	0.0015%	164.16	51.60	0.0010%	172.18	61.26
0.0063%	119.52	41.10	0.0034%	145.44	49.10	0.0053%	155.31	52.85
0.0210%	101.16	34.60	0.0147%	138.60	44.10	0.0114%	141.72	48.90
0.0570%	85.68	27.60	0.0289%	119.88	35.80	0.0481%	125.80	44.97
0.2000%	63.72	20.10	0.1996%	92.88	24.90	0.1944%	102.59	32.07
0.9100%	42.12	13.10	0.7864%	77.40	17.50	0.7802%	81.20	21.72
3.5200%	24.84	6.42	3.2475%	60.84	11.20	3.1273%	54.01	19.21



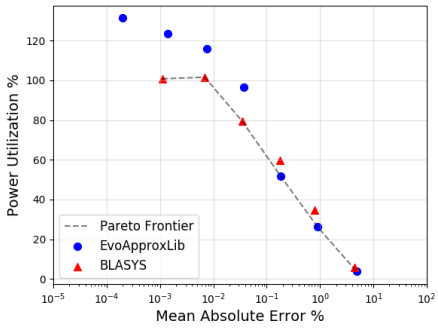
(a) Area utilization of 7-bit unsigned multiplier



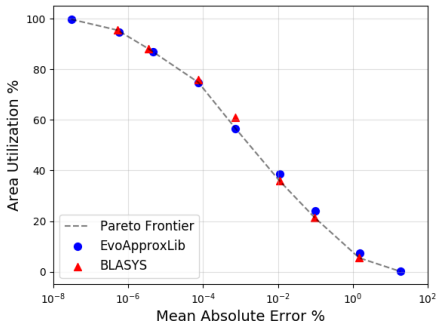
(b) Power utilization of 7-bit unsigned multiplier



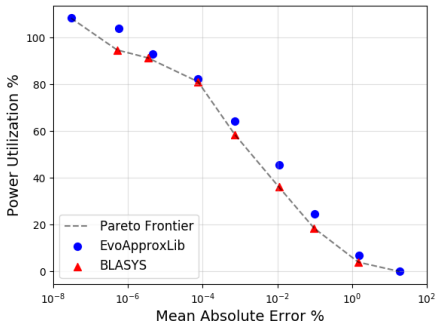
(c) Area utilization of 8-bit unsigned multiplier



(d) Power utilization of 8-bit unsigned multiplier



(e) Area utilization of 16-bit unsigned multiplier



(f) Power utilization of 16-bit unsigned multiplier

Figure 3.10: Comparison between EvoApproxLib and BLASYS. Red points represent designs explored by BLASYS. Blue points represent designs provided by EvoApproxLib.

node. Then we use our algorithm and naive approach to generate designs using their QoR metrics as thresholds, and compare area and power utilizations. Since outputs represent numerical value, we use mean absolute error (MAE) as QoR metric. Table 3.4 to 3.7 compare approximate designs between EvoApproxLib, BLASYS and the naive approach

on 7-bit unsigned multiplier, 8-bit unsigned multiplier, 16-bit unsigned multiplier and 16-bit unsigned adder respectively.

As Table 3.4 to 3.7 demonstrate, the naive approach has an obvious limitation, *i.e.* approximate designs with small error are missing. Since the naive approach generates approximate circuits by ignoring a certain number of less significant bits, the number of possible approximate designs is fixed. Take 7-bit unsigned multiplier in Table 3.4 as an example. Even ignoring the least significant bit results in 0.3860% MAE, while approximate designs with less error cannot be generated. Meanwhile, if comparing chip area and power consumption using the same error threshold, our BLASYS algorithm is a lot better than the naive approach.

As a unified approach that generates approximate designs for general circuit, our algorithm outperforms EvoApproxLib in terms of power consumption. Among 24 designs of unsigned multipliers, our algorithm has better power utilization in 17 designs. Although we only beat 6 designs in terms of area utilization, the numbers are close in other designs while ours have better QoR. Figure 3.10 illustrates the explored design space of our algorithm compared to designs of EvoApproxLib, where blue points are designs from our algorithm, and red points are designs from EvoApproxLib. It shows that our algorithm is competitive in terms of area utilization, and outperforms EvoApproxLib in terms of power utilization. Therefore, our algorithm is able to reach state-of-the-art performance in many commonly used circuits. Table 3.7 shows that our algorithm has worse results in 16-bit unsigned adder. Since it is a relatively small design, it has less number of subcircuits, which leads to a small explored design space. In this case, lack of design space exploration might sometimes affect performance.

3.5.6 Runtime Characterization

In this subsection, we briefly discuss the improvement of runtime. As mentioned in Section 3.4.1, the time complexity of exhaustive search for XOR/OR-based method is

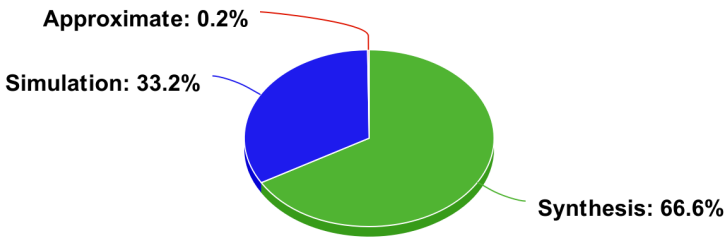


Figure 3.11: Runtime distribution. Approximate corresponds to the time of approximating subcircuits. Synthesis corresponds to the time of synthesizing top-level design from subcircuits. Simulation corresponds to QoR estimation.

$O(m2^n)$, where m is the number of output bits, and n is the number of input bits. In order to speed up this process, we break down input circuits into subcircuits with maximum 10 inputs and 10 outputs. Also, in practice, to exhaustively search columns in truth table of decompressor, we compute all possible combinations of columns at first, and then choose the best for each column.

Then, instead of approximating all subcircuits at once as Algorithm 1 suggests, in practice we approximate subcircuits on-demand. The approximation realizations of subcircuits are stored and can be reused later for other designs. With a multi-core system, we are able to parallel evaluation of designs in each iteration, since the degrees of approximation is reduced by step size in different subcircuits.

After implementing improvements mentioned above, our method is speed up by 35%. Figure 3.11 illustrates the distribution of runtime. Due to on-demand approximation of subcircuits and reusing, subcircuit approximating only takes 0.2% of runtime. Simluation, which is QoR estimation of approximate designs, takes 33.2% of runtime. And most of runtime is spent on synthesizing top-level designs from approximate subcircuits using Yosys.

3.6 Conclusions

In this paper we proposed a new approach for approximate circuit synthesis by generalizing matrix factorization techniques to incorporate field (XOR) and semi-ring (OR) algebra implementations. This generalization leads to a wider range of possible approximate circuit realizations that can be explored to identify the best trade-off. We integrated our approach into a design space exploration method with the capability to partition large circuits into subcircuits for approximation. We implemented and evaluated our approach on a large range of circuits using a number of error metrics such as numerical differences and Hamming distances, and we have demonstrated that our method is able to result in state-of-the-art performance while being flexible for all kinds of input design. Furthermore, we elucidated the large space of possible approximate designs generated from our approach, and the trade-off between accuracy and design metrics such as power and area.

In our experiments, we compared 23 approximate designs of unsigned multipliers against the state-of-the-art method, where BLASYS has better power utilization in 17 out of the 23, and better circuit delay in 16 of the designs. Although BLASYS only achieves smaller area in 8 designs, the numbers are very close. We also noticed that the input distribution of each subcircuit could potentially be used to improve truth table factorization and design space exploration. In future work, we plan to analyze the influence of input distributions of subcircuits on approximation results, and further improve BLASYS by deriving such distributions.

CHAPTER 4

Runtime Configurable Approximate Circuits with Self-Correcting Capability

4.1 Introduction

As circuit customization is developed to meet the requirements of various applications, power consumption becomes a main factor limiting the scale of computational capacity. As discussed in previous sections, one of such low-power techniques is approximate computing, which can be widely used in application domains that have inherent resilience to small inaccuracies in the outputs [75]. Such resilience can originate from various sources including, noise in input data, inherent approximate calculations, or human tolerance to variations in the outputs, while different applications may have different resilience. Thus, one trend is to design approximate circuits which are able to dynamically switch among various accuracy levels (including full accuracy) at runtime, each of which is associated with different power consumption. By properly configuring accuracy levels at runtime, power consumption could be substantially saved.

The last few years have seen various techniques for approximate logic synthesis [54; 76; 77; 78]. Most of them focus on approximating circuits with "fixed" accuracy, while some other works start to explore the flexibility of runtime configuration [2; 3; 79; 80; 81; 82]. In this chapter, we propose a novel *RUntime Configurable Approximate* (RUCA) methodology based on *truth tables factorization*, which generates approximate circuits with multiple accuracy levels, including full accuracy. The contributions of this chapter are as follow.

- Using Boolean Matrix Factorization algorithm, RUCA approximates an *arbitrary* input circuit and separates it into multiple *configuration blocks* using truth tables decomposition. By activating different blocks at runtime using power gating, we can dynamically choose the expected accuracy-power configuration to optimize power and delay. A corrector circuit is introduced to restore 100% accuracy.
- To improve the scalability of our approach, a large input circuit is first *partitioned* into subcircuits, and a *design space exploration* scheme is used to choose the proper subcircuits to approximate in the runtime configurable manner.
- We evaluate RUCA framework on a number of commonly used arithmetic circuits from Benchmarks for Approximate Circuit Synthesis (BACS) [75]. We also compare our methodology against other accuracy-configurable frameworks, Approximate through Logic Isolation [2] and 8-bit QCM [3], showcasing that RUCA efficiently improves power and delay with the flexibility of operating under multiple modes.

The organization of this chapter is as follow. In Section 4.2, we overview relevant previous work on Approximate Logic Synthesis (ALS). In Section 4.3, we introduce our novel RUCA methodology. We provide our experimental results in Section 4.4. Finally, we summarize our conclusion in Section 4.5.

4.2 Previous Work

Various approaches have been proposed for Approximate Logic Synthesis (ALS) [54; 76; 77; 78]. Compared to ALS that generates "fixed" approximate circuits, quality-configurable approximate design is less explored. One category of runtime configuration is Voltage Over-Scaling (VOS), where the power and accuracy of operation can be dynamically adjusted by tuning the voltage. However, the application of VOS is limited since it may cause uncontrollable errors that affect the most significant bits. Also, VOS increases delays on all timing paths, which may affect the performance of the whole system and even lead to the failure of operation [83].

To design stable and predictable circuits, methodologies based on logic synthesis have been proposed. SASIMI [79] proposed the first methodology to generate quality-configurable design from an arbitrary input circuit by identifying similar signals and substituting one for the other to simplify the logic. However, when full accuracy is required, the approximate circuit may need an additional clock cycle to detect errors and re-compute the substituted signals. Thus, SASIMI turns a combinational circuit into a variable latency circuit, which may not be applicable to large systems.

To mitigate the possibly doubled delay, an approximation approach through logic isolation is proposed [2], which aims to isolate parts of circuit that significantly contribute to power consumption while having less effect on overall accuracy, where power gating is then used to control the activation of these parts.

Another method based on clock gating has been proposed [80], where multiple approximate designs of input circuit are first instantiated. Then area-saving gating mechanisms are used to exploit synthesis relaxation, which leads to total energy saving. This approach was further extended using cross-layer approach [81]. While such methodologies reduce dynamic power consumption significantly, a large amount of area overhead is introduced. Also, clock gating does not reduce leakage power, which is significant in today's tech-

nology node. Besides methodologies that take arbitrary circuits as input, some runtime configurable designs with arithmetic circuits have been proposed [3; 82]. For example, QCM [3] designs configurable multipliers using genetic algorithms.

4.3 Proposed Methodology

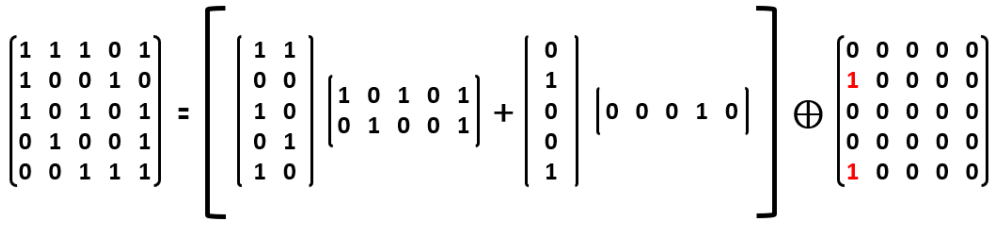
In this section, we describe our proposed methodology of designing RUntime Configurable Approximate (RUCA) circuit by factorizing and separating truth table, together with the method of self-correcting by corrector circuit. Due to the complexity of BMF algorithm, we partition a large input circuit into subcircuits and use design space exploration to improve the scalability of our methodology. We also discuss the approaches of reducing design overhead.

4.3.1 RUCA with Corrector Circuit

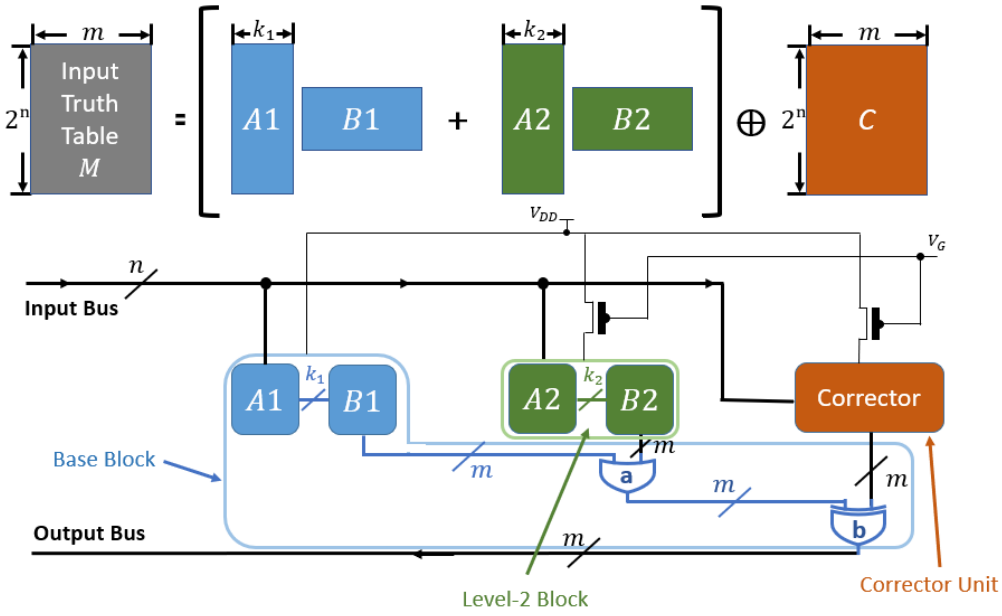
According to the rule of matrix multiplication, after factorizing a matrix $\mathbf{M}$ into $\mathbf{A}$ and $\mathbf{B}$, we may separate them into individual columns and rows, as Equation 4.1,

$$\mathbf{M} \approx \mathbf{AB} = (\mathbf{a_1} \cdots \mathbf{a_f}) \begin{pmatrix} \mathbf{b_1} \\ \vdots \\ \mathbf{b_f} \end{pmatrix} = \mathbf{a_1 b_1} + \mathbf{a_2 b_2} + \cdots + \mathbf{a_f b_f} \tag{4.1}$$

where $\mathbf{a_i}$ is the i^{th} column in matrix $\mathbf{A}$ and $\mathbf{b_j}$ is j^{th} row of matrix $\mathbf{B}$. As discussed in Chapter 3, due to the *heuristic* property of BMF algorithm, as we add terms from $\mathbf{a_1 b_1}$ to $\mathbf{a_f b_f}$, the hamming distance $||\mathbf{M} - \mathbf{AB}||_0$ keeps decreasing in a *greedy* manner. To enable runtime configuration, our goal is to factorize the input matrix $\mathbf{M}$ and separate into multiple terms which may satisfy different error thresholds. For example, suppose that we want to factorize $\mathbf{M}$ such that there exists two configurable accuracy (e.g., error 2% and 1%). Starting from factorization degree $f = 1$ with only the first term $\mathbf{a_1 b_1}$, we gradually increment f and sum $\mathbf{a_i b_i}$ terms, until the approximation error becomes no larger than 2%. Assume the current factorization degree is $f = k_1$. In this case, we can



(a)



(b)

Figure 4.1: Example of a 3-level approximate circuits using RUCA. (a) BMF with multiple accuracy levels. (b) Runtime configurable circuit design, where power gating is used to activate different blocks.

stack vectors from $\mathbf{a_1}$ to $\mathbf{a_{k_1}}$ as $\mathbf{A_1}$, and stack vectors from $\mathbf{b_1}$ to $\mathbf{b_{k_1}}$ as $\mathbf{B_1}$, such that the error between $\mathbf{M}$ and $\mathbf{A_1B_1}$ is no larger than 2%. We then keep incrementing f until 1% error threshold is met. Assuming now $f = k_1 + k_2$, vectors from $\mathbf{a_{k_1+1}}$ to $\mathbf{a_{k_2}}$ are stacked as $\mathbf{A_2}$, and vectors from $\mathbf{b_{k_1+1}}$ to $\mathbf{b_{k_2}}$ are stacked as $\mathbf{B_2}$, such that the error between $\mathbf{M}$ and $\mathbf{A_1B_1} + \mathbf{A_2B_2}$ is no greater than 1%. In other words, factorized matrices $\mathbf{A}$ and $\mathbf{B}$ are separated as

$$\mathbf{M} \approx \mathbf{AB} = \mathbf{A_1B_1} + \mathbf{A_2B_2} = (\mathbf{a_1} \cdots \mathbf{a_{k_1}}) \begin{pmatrix} \mathbf{b_1} \\ \vdots \\ \mathbf{b_{k_1}} \end{pmatrix} + (\mathbf{a_{k_1+1}} \cdots \mathbf{a_f}) \begin{pmatrix} \mathbf{b_{k_1+1}} \\ \vdots \\ \mathbf{b_f} \end{pmatrix} \tag{4.2}$$

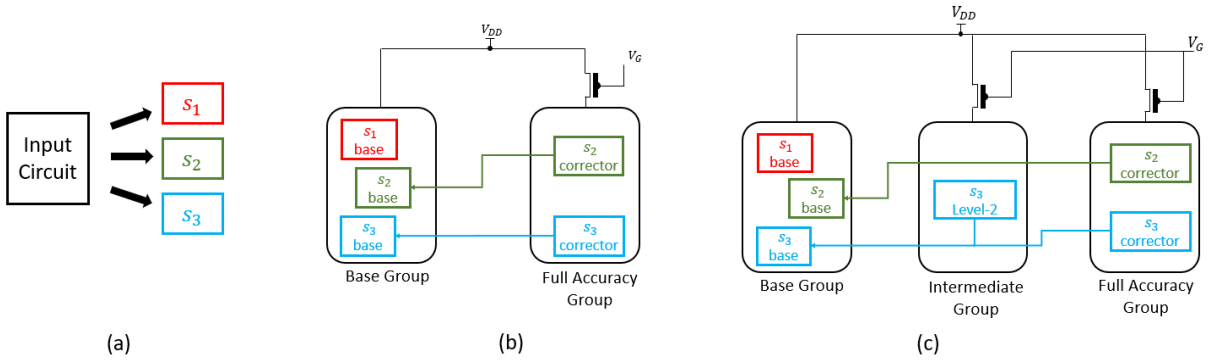


Figure 4.2: An example of runtime configurable designs for a large input circuit. (a) Input circuit is partitioned into three subcircuits. (b) *Subcircuits* are approximated into 2-level runtime configurable designs. Base blocks of 3 *subcircuits* are synthesized together as the base group of the *top-level circuit*. Corrector circuits are grouped together as the full-accuracy group of the *top-level circuit*. (c) Additional accuracy levels can be introduced by re-arranging RUCA blocks of *subcircuits* into intermediate group(s).

If more accuracy levels are needed, this procedure is repeated until we obtain matrices $\mathbf{A_i}$ and $\mathbf{B_i}$ for each accuracy level. We propose to synthesize each $\mathbf{A_iB_i}$ term into its own circuit blocks. To implement binary addition, bitwise OR gates (gate a in Figure 4.1b) are used to connect each block of $\mathbf{A_iB_i}$ term. Therefore, starting from block of $\mathbf{A_1B_1}$, as we activate more $\mathbf{A_iB_i}$ blocks, the error between original truth table $\mathbf{M}$ and summation of $\mathbf{A_iB_i}$ terms keeps decreasing, where different error thresholds can be achieved.

In order to support critical applications which require full accuracy, we propose to use a corrector circuit to restore the original functionality when needed. Here, field modulo-2 algebra (logic XOR) is used to correct flipped bits, where '1's can be used to flip bits such that $1 \oplus 1 = 0$ and $1 \oplus 0 = 1$. After input truth table $\mathbf{M}$ is factorized and separated into summation of terms $\mathbf{A_iB_i}$, bitwise XOR is computed between $\mathbf{M}$ and $\Sigma_i \mathbf{A_iB_i}$ to obtain the corrector matrix $\mathbf{C}$. This matrix can be used to restore input truth table $\mathbf{M}$ by $\mathbf{M} = (\Sigma_i \mathbf{A_iB_i}) \oplus \mathbf{C}$. Figure 4.1a demonstrates a factorization algebra with three accuracy levels. Corrector matrix $\mathbf{C}$ is computed to restore the input matrix by XOR operation. Figure 4.1b demonstrates structure of a 3-level runtime configurable circuit. Firstly, an input circuit with n inputs and m outputs is simulated to obtain the $2^n \times m$ truth table $\mathbf{M}$. Then, $\mathbf{M}$ is factorized into two matrices $\mathbf{A}$ and $\mathbf{B}$, which are then separated into $\mathbf{A_1B_1}$

and $\mathbf{A_2B_2}$. All matrices are synthesized into corresponding parts of the circuit. Corrector matrix $\mathbf{C}$, which is used to synthesize the corrector circuit, is computed for restoring input truth table $\mathbf{M}$. As Boolean algebra indicates, $\mathbf{A_1B_1}$ and $\mathbf{A_2B_2}$ are connected using bitwise OR (gates a), which is then connected to the corrector circuit using bitwise XOR (gates b). Thus, if all parts are activated, it will produce equivalent functionality as original circuit, where circuit operates in full-accuracy mode:

$$\mathbf{M} = (\mathbf{A_1B_1} + \mathbf{A_2B_2}) \oplus \mathbf{C} \tag{4.3}$$

In order to enable runtime configuration, we allocate these parts into different *configuration blocks*, and use power gating to control their activation. In this example, $\mathbf{A_1}$, $\mathbf{B_1}$ and all connecting gates compose the *base* block, which is always activated by default. When only activating the *base* block, the circuit operates in approximate mode with lowest accuracy, where the output matrix $\mathbf{M}'$ is

$$\mathbf{M}' = \mathbf{A_1B_1} \tag{4.4}$$

$\mathbf{A_2}$ and $\mathbf{B_2}$ compose the *level-2* block, which can be additionally activated for higher-accuracy mode. And the output matrix $\mathbf{M}''$ is

$$\mathbf{M}'' = \mathbf{A_1B_1} + \mathbf{A_2B_2} \tag{4.5}$$

Following this framework, we are able to design runtime configurable circuits with arbitrary number of accuracy levels.

4.3.2 Partitioning and Design Space Exploration

The number of rows in a truth table grows exponentially with the number of primary inputs in the circuit, which makes BMF algorithm computationally expensive for circuits with large number of inputs. To scale our approach, we propose a *divide-and-conquer*

method using circuit partitioning and design space exploration technique. As illustrated in Figure 6.4a, to begin with, a given circuit is partitioned into a number of subcircuits with limited number of inputs and outputs, each of which is approximated using RUCA approach as Figure 4.1. A design space exploration technique is used to find proper subcircuits and factorization degrees, where the priority is to approximate the subcircuits that consume more power while having less impact on final accuracy. Figure 6.4b demonstrates a 2-level configurable circuit, where base blocks of the approximate subcircuits are grouped in an individual power domain and synthesized together as the *base group* of the top-level design. Corrector circuits of approximate subcircuits are also grouped together into *full-accuracy group*, which enables the top-level design to restore full accuracy. Moreover, if additional accuracy levels are expected, more *intermediate groups* can be created by re-allocating different blocks of approximate subcircuits, as illustrated in Figure 6.4c.

Algorithm 4 describes the overall procedure of approximating subcircuits and re-allocating blocks into groups. To begin with, the input circuit is partitioned into subcircuits (line 1) using hypergraph partitioning algorithm [56]. Then the truth table of each subcircuit is factorized as Equation 4.1 to prepare for RUCA design (line 3).

In design space exploration scheme (line 7-19), we approximate and replace subcircuits with RUCA design, whose configuration blocks are re-allocated into groups of top-level design. In other words, for each subcircuit s_i, we search proper factorization degrees f's to separate factorized truth tables as Equation 4.2. Factorization degree f_i for each subcircuit is initialized as the number of primary outputs (line 4) and decrement by 1 at each iteration (line 9,13), where RUCA($s_i, f_i - 1$) denotes a runtime configurable design based on subcircuit s_i with *factorization degree* $f_i - 1$. For each candidate design, a loss is computed based on accuracy Err_i, power in full-accuracy mode P_{acc} and in approximate mode P_{app} (line 10,12). Whenever an error threshold is reached, all new added blocks are placed into corresponding group (line 14-19).

Algorithm 2: Runtime Configurable Approximate Circuit with Design Space Exploration

Input : $ICir$: input circuit
ϵ: list of error thresholds, sorted in ascending order
Output : $RCir$: list of groups in output RUCA circuit

```
S_Cir = Partition ICir into list of subcircuits
for each subcircuit s_i in S_Cir do
    Factorize truth table for s_i
    Set current factorization degree f_i = m_i where m_i is the number of primary outputs
      of subcircuit s_i
end
n = 0 // Indexing RCir list
while ϵ is not empty do
    for each subcircuit s_i in S_Cir do
        TCir_i = RUCA(s_i, f_i − 1)
        loss_i = Err_i · [P_acc + P_app]
    end
    k = arg min_i loss_i
    f_k = f_k - 1
    if QoR ≥ ϵ[0] then
        Replace s_i with TCir_i for each subcircuit s_i
        Add all new blocks (except base blocks) into RCir[n]
        n = n + 1
        ϵ.pop(0)
    end
end
Add all base blocks of subcircuits into RCir[n]
return RCir
```

4.3.3 Reducing Design Overhead

Design overhead is considered as an important criterion in accuracy-configurable designs, which is defined as additional chip area, power and delay running in full-accuracy mode compared to original input circuit. In RUCA, the design overhead mainly comes from two sources: (1) As Figure 4.1b shows, additional bitwise OR and XOR gates are used to connect blocks, which is inevitable in RUCA design. To mitigate such overhead in design space exploration, we should limit number of levels for each approximate subcircuit. In our practice, each subcircuit is approximated with no more than *three levels*, such that each of these contains at most *one* bitwise OR gate and *one* bitwise XOR gate.

(2) Since each block is synthesized and optimized individually, we may lose the

opportunity of logic optimization across different blocks, which leads to logic redundancy. Such logic redundancy becomes severe with a dense corrector matrix. As discussed in subsection 4.3.1, the corrector circuit is synthesized from corrector matrix to flip wrong bits in approximate truth table. Normally, the difference between approximate and original truth table is not too large, and the corrector matrix is sparse as shown in Figure 4.1a. In this case, the overhead caused by corrector circuit is small. However, if input circuit is partitioned and design space exploration is performed, for some subcircuits, the difference between approximate and original truth table may be large. In this situation, the corrector matrix is dense, which will be synthesized into large corrector circuit, sometimes even larger than the original subcircuit. In this case, rather than using a corrector circuit to achieve full accuracy, we use the original subcircuit instead. In our design space exploration algorithm, once the corrector circuit is synthesized, we compare the power consumption between corrector circuit and original one. If a corrector circuit consumes less power, we follow the algorithm described in Section 4.3.2. However, if the corrector circuit consumes more power than the original subcircuit, we directly include original subcircuit for full-accuracy mode. In this case, instead of XOR gates, a multiplexer is used to choose between original subcircuit and the approximate versions.

4.4 Experimental Results

In this section, we evaluate our proposed methodology on a number of arithmetic circuits which are commonly deployed in approximate computing from Benchmarks for Approximate Circuit Synthesis (BACS) [75]. Table 4.1 summarizes the characteristics of evaluated benchmarks. To begin with, we directly generate runtime configurable designs of 8-bit adder, where the trade-off between design overhead and choices of error thresholds is discussed. The remaining benchmarks are first partitioned into subcircuits, and then design space exploration is performed as Algorithm 4.

For hardware metrics, all designs are implemented in Verilog and synthesized with

a $7nm$ predictive process design kit. Cadence Genus is used to synthesize each design and estimate chip area, circuit delay and power consumption under the maximum clock frequency of original circuit. For QoR metric, we report normalized mean absolute error (MAE) defined as

$$\text{MAE} = \frac{1}{N}\Sigma_{i=1}^{N}\frac{|R_i - R_i'|}{2^m}, \tag{4.6}$$

where N denotes the size of the test vectors while R_i and R_i' denote the accurate and approximate numerical results.

In the first set of experiment, we analyze the trade-off between design accuracy, power consumption and design overhead. RUCAs are generated for 8-bit adder with different error thresholds. Besides full accuracy, only *one* approximate level is considered for each design in this experiment. Since the original circuit has 9 primary outputs, after factorizing its truth table, first f pairs of columns and rows are synthesized into *base* block as approximate mode, where f ranges from 1 to 8. For each RUCA design, an associated corrector circuit is created to restore errors in full-accuracy mode. In Figure 4.3, we report the power consumption of the corrector circuit, and the RUCA design in both approximate mode and full-accuracy mode. In approximate mode, power consumption keeps reducing while error increases. However, in full-accuracy mode, the corrector circuit becomes more substantial and power-consuming, especially when factorization degree f is small. As MAE exceeds 5%, where factorization degree $f < 4$, power consumption of full-accuracy mode increases substantially due to the corrector circuit. Therefore, to limit the overhead in full-accuracy mode, error thresholds in approximate mode need to be limited, *e.g.*, below 5% MAE.

In the second set of experiments, for the remaining six benchmarks in BACS in Table 4.1, we generate three RUCA designs with 2 levels, 3 levels and 4 levels respectively. We use 0.1%, 1% and 2% as error thresholds. In order to highlight the benefits of our methodology, we report *relative power* as the ratio between power of RUCA design (under certain accuracy level) and power of the original circuit. Figure 4.4 illustrates relative

Table 4.1: Characteristics of evaluated benchmarks.

Bench-mark	Name	Function	I/O	Area (um^2)	Power (uW)	Delay (ns)
BACS	adder8	8-bit adder	16/9	47.58	24.70	0.81
	abs_diff	absolute difference	16/9	67.41	22.68	0.90
	adder32	32-bit adder	64/33	167.03	32.20	2.83
	buttfly	butterfly structure	32/34	174.26	42.30	3.05
	mac	multiply-add	12/8	94.48	33.76	1.14
	mult8	8-bit multiplier	16/16	364.61	82.21	1.97
	mult16	16-bit multiplier	32/32	1084.52	245.06	3.74

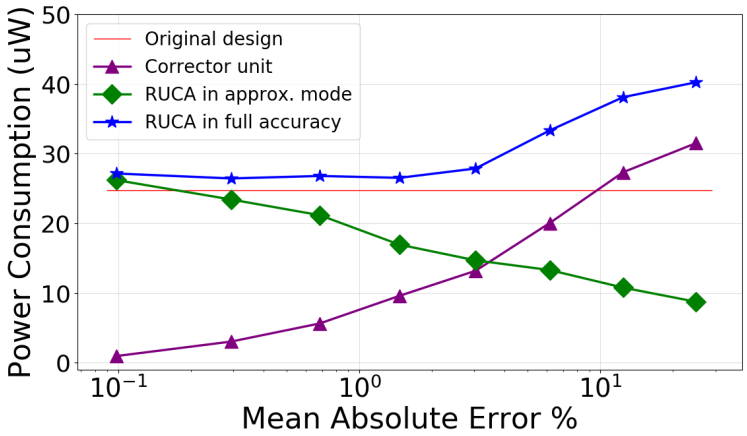


Figure 4.3: 2-level approximate design of 8-bit adder: Power consumption with different error thresholds.

power of RUCAs for each benchmark. Compared to original circuit, RUCA substantially saves power in approximate mode, and use slightly extra power to enable corrector circuit for full-accuracy mode. However, as the number of accuracy levels increases, RUCA approximates an input circuit into more configurable blocks, which potentially reduces opportunities to optimize logic synthesis and increases power consumption in full-accuracy mode.

In Table 4.2 and Table 4.3, we thoroughly evaluate all benchmarks and compare the performance against another runtime configurable framework named Approximation through Logic Isolation [2]. Under each accuracy level, we report *total area*, *power* and *delay* as ratio against original input circuit. We use 3-level runtime configurable designs and set error thresholds as 1% MAE and 2% MAE. Red numbers represent better performance between two methodologies. On average, we are able to reduce 30.15% power and 22.96%

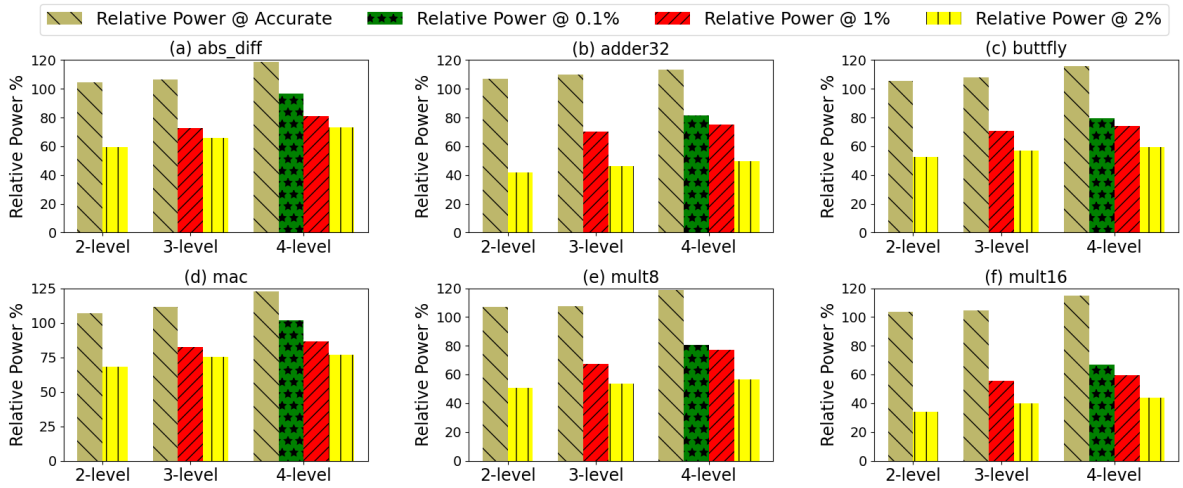


Figure 4.4: Relative power of RUCAs for each benchmark, with 2-4 levels.

Table 4.2: Comparison of total area and power between RUCA and Approximation through Logic Isolation (ISO) [2] (using 3-level runtime configurable design)

Name	Total Area		Power					
	RUCA	ISO	Under 2% MAE		Under 1% MAE		Full Accuracy	
			RUCA	ISO	RUCA	ISO	RUCA	ISO
abs_diff	138.38%	152.09%	65.79%	55.82%	72.73%	82.54%	105.68%	109.85%
adder32	142.74%	134.47%	46.17%	51.89%	70.13%	60.93%	109.86%	113.10%
buttfly	147.73%	136.10%	56.84%	51.82%	70.95%	72.34%	107.93%	107.38%
mac	133.43%	138.22%	75.31%	69.50%	82.39%	83.38%	111.45%	117.42%
mult8	129.23%	133.09%	53.70%	71.29%	67.41%	87.23%	107.41%	112.49%
mult16	117.78%	128.66%	39.90%	54.16%	55.47%	72.45%	104.52%	109.73%
Average	134.89%	137.11%	56.29%	59.08%	69.85%	74.81%	107.81%	111.66%

Table 4.3: Comparison of delay between RUCA and Approximation through Logic Isolation (ISO) [2] (using 3-level runtime configurable design)

Name	Delay					
	Under 2% MAE		Under 1% MAE		Full Accuracy	
	RUCA	ISO	RUCA	ISO	RUCA	ISO
abs_diff	56.19%	64.90%	85.40%	88.94%	116.34%	124.92%
adder32	62.37%	68.50%	84.92%	82.56%	129.73%	124.99%
buttfly	42.27%	74.24%	76.08%	92.03%	121.05%	137.70%
mac	52.73%	65.10%	74.20%	87.76%	134.79%	129.53%
mult8	40.91%	71.46%	72.00%	91.26%	138.84%	134.26%
mult16	33.05%	62.07%	71.62%	79.41%	126.50%	136.83%
Average	47.92%	67.71%	77.04%	86.99%	127.87%	131.37%

delay with 1% error threshold; and reduce 43.71% power and 52.08% delay with 2% error threshold. To run in full accuracy mode, RUCA consumes 7.81% additional power than the original circuit. However, it is expected that with approximate computing, the circuits

Table 4.4: Comparison between RUCA and the naive approach

Benchmark	RUCA				Naive			
	Area	MAE	Power	Delay	Area	MAE	Power	Delay
8-bit unsigned multiplier	129.33%	2.00%	53.70%	40.91%	149.20%	2.83%	63.05%	52.18%
		1.00%	67.41%	72.00%		1.34%	77.15%	73.90%
		0.00%	107.41%	138.84%		0.00%	129.14%	133.07%
16-bit unsigned multiplier	117.78%	2.00%	39.90%	33.05%	157.37%	2.23%	54.94%	45.87%
		1.00%	55.47%	71.62%		0.97%	78.01%	73.00%
		0.00%	104.52%	126.50%		0.00%	136.41%	143.47%

will run approximately most of the time, and only in a few occasions, full accuracy will be needed and enabled. Compared to Logic Isolation, our RUCA framework has smaller total area in 4 designs out of 6 benchmarks, which on average saves 2.22% area compared to Logic Isolation. In terms of power consumption, our approach has 3 better results under 2% error level, and 5 better results under 1% error level and full-accuracy level respectively. In general, compared to Logic Isolation, RUCA is able to use smaller chip area and consumes less power to implement the same functionality of runtime configurable design, especially in higher-accuracy mode.

Similar to Section 3.5.5, using 8-bit multiplier and 16-bit multiplier, we compare 3-level RUCA against 3-level approximate circuits generated by the naive approach, which dynamically "ignores" a certain number of less significant bits. To enable the functionality of dynamic configuration, the blocks of "ignored" bits need to be synthesized individually, so that power gating can control their activations. Similar to our observation in Section 3.5.5, the accuracy level in naive approach is limited. Thus, when generating configurable approximate circuits with the naive approach, we choose the closest accuracy level from our RUCA experiments. Table 4.4 demonstrates the results. For both multipliers, RUCA significantly saves more power consumption and reduces more circuit delay in approximate mode. In full accuracy mode, RUCA also has less overhead compared to the naive approach. The main reason is that, in naive approach needs to synthesize each blocks separately as mentioned before. However, RUCA is able to optimize logic synthesis across all subcircuits using design space exploration.

Table 4.5: Comparison between RUCA and QCM [3]

RUCA				QCM			
Area	MAE	Power	Delay	Area	MAE	Power	Delay
129.33%	2.00%	53.70%	40.91%	135.01%	1.40%	53.93%	47.61%
	1.00%	67.41%	72.00%				
	0.00%	107.41%	138.84%		0.00%	112.40%	128.46%

Finally, in Table 4.5, using 8-bit multiplier, we compare a 2-level QCM [3] against 3-level RUCA, where RUCA has more accuracy levels with even smaller total area, demonstrating that the design overhead of RUCA is relatively lower.

4.5 Conclusion

In this chapter, we proposed a novel methodology RUCA to design runtime configurable approximate circuit with Boolean matrix factorization. Factorized matrices are separated to synthesize each configuration block, while a corrector circuit is created to restore full accuracy. Moreover, we integrated our methodology with design space exploration scheme to improve scalability. We evaluated RUCA on a set of benchmarks, and demonstrated that RUCA significantly reduce power and delay, while providing flexibility to balance the accuracy-power trade-off. Comparing against other approaches, on average RUCA additionally saves power consumption by 3.87% and circuit delay by 11.08% while reducing chip area by 2.22%, which highlights the state-of-the-art performance.

CHAPTER 5

Configurable Deep Neural Network with Dynamic Weight-Enabling for Efficient Inference

5.1 Introduction

Deep Neural Networks (DNN) are widely used in many applications, *e.g.* object detection, gesture recognition and augmented reality, which are often deployed on edge devices with limited computational resources. While it is possible to train DNNs on the cloud, inference on edge devices needs to meet timing and energy constraints. Different devices have different computing power, while other running tasks also limit the amount of available computational resources. To deploy the same DNN model in different scenarios, one solution is to train a *dynamic DNN* with *multiple configurations of operation modes*, where we are able to choose the most applicable configuration at runtime to meet the timing and energy constraints while optimizing accuracy.

The last few years have seen various methodologies for dynamic DNNs, which can be categorized into three classes: flexible width [1; 84; 85], flexible depth [86; 87] and flexible

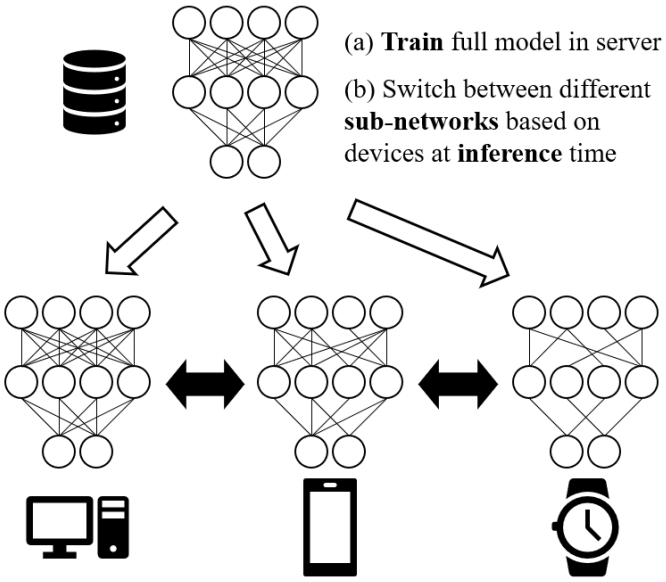


Figure 5.1: Flexible weight-enabling methodology, where weights can be dynamically enabled to form different sub-networks for different hardware platforms.

precision [88; 89]. In this chapter, we propose a novel orientation, *flexible weight-enabling*, where connective weights between layers can be dynamically enabled or disabled. As Figure 5.1 illustrates, the entire model is first trained on server. For inference, the model can be dynamically configured to different sub-networks by enabling different set of weights to meet the timing and energy constraints of different devices. We name our methodology *dynamic Weight-enabling Network* (WeNet). The contributions of this chapter are as follow.

- We introduce a novel dynamic DNN architecture, WeNet, that is able to dynamically enable different subsets of weights at runtime. We propose a weight-enabling pattern, that for each layer, the subset of weights forms a sub-network with several *independent groups*.
- We extend WeNet to convolutional layers by using *flexible group convolution* and *channel shuffling operation*.
- During training, random sub-networks are sampled at each iteration, where *switchable batch normalization* [85] is used. At inference time, we propose a *design space exploration* method to search optimal sub-networks and balance the trade-off between

efficiency and accuracy.

- We evaluate WeNet using multiple DNN architectures, and measure inference time, energy consumption and accuracy with different configurations of sub-networks. By comparing against other dynamic DNNs, we demonstrate that WeNet provides better accuracy-efficiency trade-off.

The organization of this chapter is as follow. In section 5.2, we overview relevant previous works. In section 5.3, we introduce WeNet and its training algorithms. We provide our experimental results in section 5.4. Finally, we summarize our conclusion and directions for future works in Section 5.5.

5.2 Previous Work

A number of methodologies for efficient DNNs have been proposed [90; 91; 92]. While these methodologies aim to shrink model size and reduce computational cost, none of them have the flexibility of adjusting model efficiency at runtime. The first design with such flexibility is "Big/Little" implementation [93], where two networks are trained and the "big" network is triggered only if the result from "small" network is not deemed confident enough. However, such methodology needs to store multiple models, and the latency for switching between different models is not negligible. Thus, a more efficient approach is to train *one single* network, with the flexibility to switch between different modes at runtime. Recent works on dynamic network can be categorized into three classes:

- **Flexible Width:** At runtime, the width of each layer can be shrunk to reduce computational cost, or expanded to increase accuracy. Tann *et al.* [84] propose one of the first runtime configurable DNN with flexible width, where DNN is trained incrementally at each width ratio. Slimmable neural network (SNN) [85] further improves the idea with switchable batch normalization. US-Nets [1] extends SNN to execute at arbitrary width ratio.

Table 5.1: Comparison between Different Dynamic Network Methods

	Memory Footprint	Num. of Neurons	Special Device
Flex. Width	Reduced	Reduced	Not Needed
Flex. Depth	May Increased	Reduced	Not Needed
Flex. Precision	Reduced	Same	Needed
Flex. W.E.	Reduced	Same	Not Needed

- **Flexible Depth:** Flexible depth means that the depth of DNN is adjustable at runtime. One common approach is to introduce early-exiting points, where the rest layers are ignored [87]. BranchyNet [86] proposes to add side branches to exiting points, where a forward pass can exit earlier from the main branch with higher confident inputs.

- **Flexible Precision:** Flexible precision means that the precision of operands, including both weights and inputs, can be adjusted dynamically at runtime. Pagliari *et al.* [88] find that many inputs do not need full precision to make accurate classification, and propose to reduce precision of operations when the confidence of input is high enough. SP-Nets [89] propose to train a multi-precision network using switchable batch normalization.

Table 5.1 compares three categories of dynamic networks, where all of them aim to reduce inference time and energy consumption. For *flexible width*, the number of neurons activated at each layer can be reduced at runtime, which leads to less memory footprint. For *flexible depth*, however, due to the additional side branches, memory footprint may even increase. Both *flexible width* and *flexible depth* reduce number of neurons, where crucial information may be lost and accuracy may drop significantly. For *flexible precision*, each weight uses less memory space by decreasing its floating-point precision, which reduces memory footprint. Since the number of neurons remains the same as original network, it is more likely to retain extracted features and remain high accuracy. But this class of dynamic networks cannot be used on any arbitrary devices, since the device has to support operations with multiple precision to exploit the efficiency of low-precision

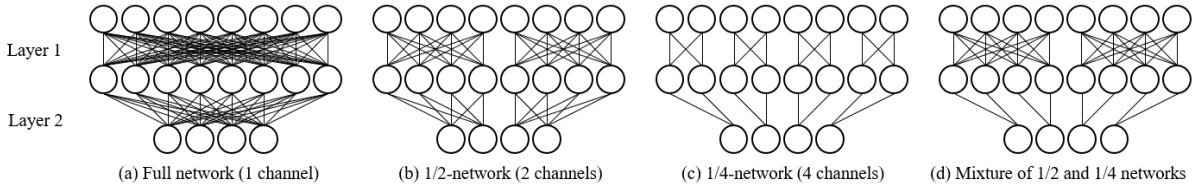


Figure 5.2: Example of a WeNet on dense layers. (a) Full network: Enable all weights to restore original network with highest accuracy. (b) 1/2-network: Enable 1/2 weights and form 2 separated channels. (c) 1/4-network: Enable 1/4 weights and form 4 separated channels. (d) Combination of 1/2- and 1/4-network.

operands. Comparing to all three categories, we expect our methodology to reduce memory footprint for energy-efficient inference, keep the number of neurons unchanged for higher accuracy, while generalizing to all types of devices. Thus, as Table 5.1 shows, we propose *flexible weight-enabling* (Flex. W.E.), where different subsets of weights can be enabled dynamically at runtime.

5.3 Proposed Methodology

In this section, we propose the methodology of Weight-enabling Network (WeNet). As Figure 5.2 shows, WeNet enables different subsets of weights to dynamically switch between different sub-networks, where computational cost can be adjusted without changing the number of activated neurons. Such network architecture can be executed on any types of hardware platforms, though as discussed later, it will benefit even more with parallelism. After discussing WeNet and its extension to convolutional layers, we will introduce training algorithm and design space exploration method for optimizing the trade-off between efficiency and accuracy at inference time.

5.3.1 Dynamic Weight-enabling Network (WeNet)

WeNet dynamically balances between accuracy and efficiency by switching between different sub-networks, each of which enables a subset of weights, following a specific pattern as shown in Figure 5.2. Assume for a dense layer, input vector is $\mathbf{x}$ and output vector is $\mathbf{y}$. In a standard dense layer, every neuron in $\mathbf{y}$ is connected to every neuron in

$\mathbf{x}$, which leads to $\ell(\mathbf{x}) \cdot \ell(\mathbf{y})$ weights, where $\ell(\mathbf{x})$ denotes the number of neurons in $\mathbf{x}$. In WeNet, we propose to divide both input $\mathbf{x}$ and output $\mathbf{y}$ into n groups, respectively, such that $\mathbf{x} = \mathbf{x}_1 \cup \mathbf{x}_2 \cup \ldots \cup \mathbf{x}_n$ and $\mathbf{y} = \mathbf{y}_1 \cup \mathbf{y}_2 \cup \ldots \cup \mathbf{y}_n$. To reduce the computational cost, instead of connecting all input and output neurons, only neurons in $\mathbf{y}_i$ are connected to neurons in $\mathbf{x}_i$, which forms a partially-connected sub-network. In other word, for each layer, ($\mathbf{x}_i$,$\mathbf{y}_i$) pairs form *independent groups*. By adjusting the number of groups n at runtime, WeNet dynamically controls the number of enabled weights, and thus adjusts computational cost.

Notice that all *independent groups* can be executed in parallel. To optimize inference time and energy consumption, each layer is evenly divided, in other word, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ have same number of neurons. Evenly-divided layers have two benefits: (1) For layers with n groups, evenly division leads to the least number of weights as well as computational cost

$$O\left(\frac{L(\mathbf{x}) \cdot L(\mathbf{y})}{n}\right) \tag{5.1}$$

(2) Each group has the same amount of FLOPs, which further improves efficiency with parallel execution.

Figure 5.2(a) shows the full network. To improve efficiency, we may disable half of the weights and execute the 1/2-network as Figure 5.2(b) shows. If faster inference time or lower energy consumption is expected, we may then switch to 1/4-network shown in figure 5.2(c). As less weights are enabled during this process, WeNet loses more information, which leads to lower accuracy as trade-off. To further improve the flexibility, we may create more operating modes, where each layer may have individual weight-enabling ratio. Figure 5.2(d) shows a combination of 1/2- and 1/4-network. By enabling different subset of weights, WeNet dynamically switches between sub-networks to balance between efficiency and accuracy.

5.3.2 WeNet on Convolutional Layers

WeNet transforms standard dense layers into a set of switchable partially-connected sub-networks. Nowadays, the most widely used layers in DNNs are convolutional layers, which itself is a partially-connected layer. In this subsection, we extend the idea of WeNet to convolutional layers.

Similar to enabling subset of weights in dense layers as shown in Figure 5.2, in convolutional layer, WeNet dynamically enable *subset of kernels* between feature maps, which forms *independent groups of channels*, or in other word, using group convolution with the flexibility of switching number of groups. We consider a standard convolutional layer as the full network of convolutional WeNet. Assume that it takes an input tensor L_i of size $h_i \times w_i \times c_i$, and applies convolutional kernel $K \in \mathbb{R}^{k \times k \times c_i \times c_o}$ to produce an output tensor L_o of size $h_o \times w_o \times c_o$. The computational cost of standard convolution is

$$T_{full} = h_i \times w_i \times c_i \times c_o \times k \times k \tag{5.2}$$

Similar to WeNet on dense layer, input tensor L_i with c_i feature maps is divided *evenly* into n groups, each of which has c_i/n feature maps, such that $L_i = L_i^1 \cup L_i^2 \cup \ldots \cup L_i^n$. Output tensor L_o is also divided in same pattern $L_o = L_o^1 \cup L_o^2 \cup \ldots \cup L_o^n$. In standard convolution, there exist kernels that convolve any input feature map into any output feature map. In sub-network of WeNet, however, only kernels between feature maps of L_i^j and L_o^j is enabled. Figure 5.3 demonstrates the example of dividing input and output tensors into 4 groups, where input feature maps in L_i^j are only convolved to output feature maps in L_o^j, using only the kernels shown in same color. Since WeNet effectively forms 4 *separate groups of channels*, only 1/4 of kernels in each filter are enabled. Thus, computation cost is reduced to 1/4 of original cost T_{full}.

$$T_{1/4} = h_i \times w_i \times \frac{c_i \times c_o \times k \times k}{4} = \frac{T_{full}}{4} \tag{5.3}$$

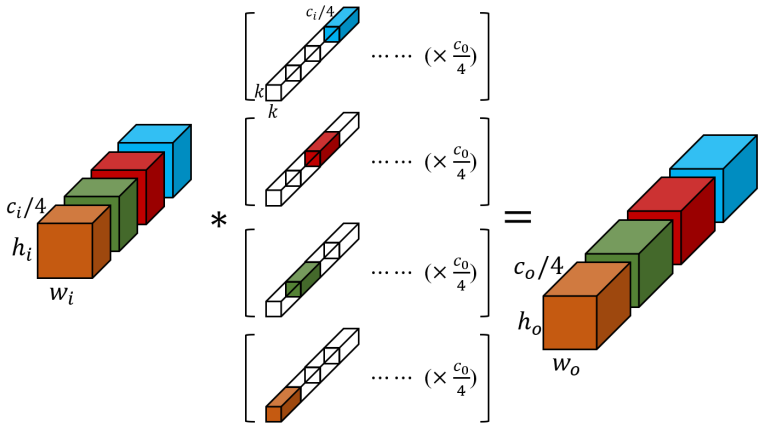


Figure 5.3: 1/4-network of convolutional layer

Similar to dense layer, each group of channels is independent, which can be executed in parallel to further improve efficiency at inference time. However, on the other hand, each feature map can only receive inputs from its own group. Compared to standard convolution, feature maps in each group receive limited input information for features extractions, where accuracy may drop significantly. To mitigate the accuracy loss caused by *isolated* groups, we adopt a channel-shuffling operation proposed by ShuffleNet [91]. A channel-shuffling operation is performed between two convolutional layers. As Figure 5.4 shows, after the first convolutional layer, each group of features is first split, and then shuffled and merged to form new groups of features as input to the next convolutional layer. Although a standard convolution is replaced by several independent groups of channels, channel-shuffling allows each group to learn from the others, which significantly improves accuracy. Meanwhile, parallelism can still be used on each group to improve efficiency.

Implementation of channel-shuffling is straightforward. Suppose a convolutional layer of WeNet has *maximum* g independent groups (in experiment, we set $g = 16$), where each group has n feature maps. All $g \times n$ feature maps are first reshaped into (g, n), which are then transposed to (n, g) and flattened as shuffled groups. Channel-shuffling operation only involves matrix transposing, where the computational cost is negligible compared to model inference.

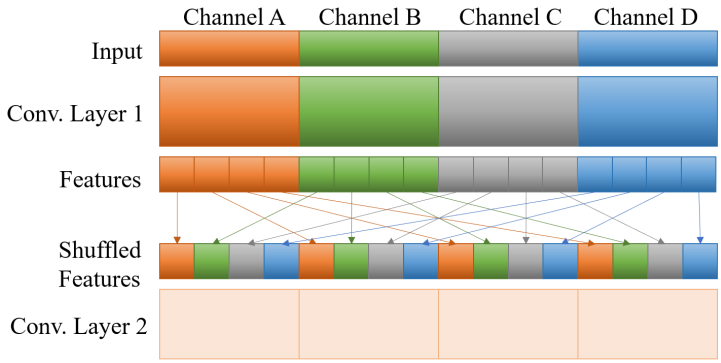


Figure 5.4: Channel shuffling after 1/4-network

Algorithm 3: Train with random sampling and S-BN

Input : Training set $(\mathbf{X}, \mathbf{Y})$ with features $\mathbf{X}$ and labels $\mathbf{Y}$, Number of iterations n_{iter}, Number of sampled sub-network per iteration s, Loss function $loss\_fn$, Optimizer opt

Output : WeNet M

Initialize WeNet M
Initialize S-BN layers for each weight-enabling pattern
for $num\_iter = 1, 2, \ldots, n_{iter}$ **do**
    Get next batch of data and labels $(\mathbf{x}, \mathbf{y})$ from $(\mathbf{X}, \mathbf{Y})$
    Clear gradients $opt.zero\_grad()$
    Execute full network $\hat{y} = M(x)$
    Compute loss $loss = loss\_fn(\hat{y}, y)$
    Accumulate gradient $loss.backward()$
    Adjust S-BN to smallest sub-network
    Execute smallest sub-network $\hat{y} = M'(x)$
    Compute loss $loss = loss\_fn(\hat{y}, y)$
    Accumulate gradient $loss.backward()$
    Randomly sample $s - 2$ sub-networks
    for *each sampled WeNET* **do**
        Adjust S-BN according to weight-enabling patterns
        Forward training data $\hat{y} = M'(x)$
        Compute loss $loss = loss\_fn(\hat{y}, y)$
        Accumulate gradient $loss.backward()$
    end
    Update gradients to weight $opt.step()$
end
return M

5.3.3 Training WeNet with Switchable Batch Normalization

WeNet dynamically enables subset of weights or kernels to form sub-networks, where each layer may have different number of *independent groups*. Since it is impractical to enumerate and train all possible sub-networks, we propose to *randomly sample* sub-

networks at each iteration.

We also need to reconsider the implementation of Batch Normalization (BN) layers, which normalize feature maps across each batch. Nowadays, BN is widely used to reduce internal covariate and stabilize training process. Using y and y' to denote inputs and outputs, computation of BN layers is defined as

$$y' = \gamma \cdot \frac{y - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{5.4}$$

where μ, σ^2 are means and variance of feature maps of current batch, γ, β are learnable variables. Since each layer has a number of possible weight-enabling pattern, where each neuron receives different inputs as Figure 5.2, μ and σ^2 are not consistent, which leads to inaccurate learning of γ and β. Yu *et al.* propose Switchable Batch Normalization (S-BN) in SNN [85], which privatizes μ, σ^2, γ, β settings for different pattern of the same layer. In other words, each weight-enabling pattern has its own customized BN layer, which is switchable according to the selected pattern. The number of learnable variables in BN layers is negligible compared to other weights and kernels. Notice that in WeNet, although there is a large number of possible sub-networks, for each layer the possible number of weight-enabling patterns is limited (in our experiment, maximum 16 independent groups). Thus, it is still efficient to use customized S-BN layers for all weight-enabling patterns in each layer. Algorithm 3 describes the process of training WeNet using S-BN. We initialize DNN with independent BN layers for each weight-enabling pattern (line 1-2). For each iteration, we accumulate gradients from full network (line 6-8), the smallest sub-network (line 9-12) and other $s - 2$ random sub-networks (line 14-19).

5.3.4 Design Space Exploration

After training with Algorithm 3, theoretically WeNet can be configured into operation modes corresponding to all possible sub-networks. But in practice, some sub-networks performs better in terms of efficiency-accuracy trade-off. Given an accuracy threshold, to

Algorithm 4: Design space exploration of WeNet

Input : Full network of WeNet M, Accuracy threshold δ
Output : Optimal sub-network M'

1 $M' = M$ // Keep track of most efficient sub-network so far
2 acc = Accuracy of M // Accuracy of current sub-network
3 **while** $acc > \delta$ **do**
4 **for** *each layer* i *in* M' *that has* < 16 *groups* **do**
5 Increase the number of groups in layer i as sub-network m_i
6 acc_i = Accuracy of m_i
7 t_i = Inference time of m_i
8 $loss_i = t_i / acc_i$
9 **end**
10 $k = \arg\min_i loss_i$
11 $M' = m_k$ and $acc = acc_k$
12 **end**
13 **return** M'

determine the optimal operation mode on a specific hardware platform, we propose to explore the design space of possible sub-networks, using *design space exploration* (DSE) approach as Algorithm 4.

We start from the full network (line 1-2). In each iteration of DSE, we consider a number of candidate sub-networks by increasing the number of groups in one of the WeNet layers, respectively (line 5). We set an upper bound on number of groups (16 in experiment) to prevent accuracy dropping rapidly. As our objective is to minimizing computational cost (represented by inference time t) while keeping accuracy acc high, for each candidate sub-network i, we evaluate t_i and acc_i using targeted hardware device, and compute a loss using t_i/acc_i (line 8). By minimizing the loss in each iteration, we keep reducing computational cost while optimizing the trade-off against accuracy. When accuracy of current sub-network drops below threshold, DSE is finished.

Using DSE, we effectively explore a number of possible sub-networks. At each iteration, sub-network with smallest loss forms Pareto Frontier, which demonstrate the optimal trade-off between efficiency and accuracy.

5.4 Experimental Results

In this section, we first describe our experiment setup. To highlight the benefits of channel-shuffling, we evaluate WeNet with and without channel-shuffling operations. We also explore the design space of WeNet and compare optimal execution modes against other methodologies, showing that WeNet provides better trade-off. Finally, we measure inference time and energy consumption, showing the benefits on different types of real devices.

5.4.1 Experiment Setup

1. *Datasets and models:* We implement and evaluate WeNet using *ResNet-50* [94], *MobileNet-V2* [90] and *EfficientNet-B0* [92] on ImageNet classification problem.

2. *Software setup:* We implement weight-enabling operations with *group convolution* in *PyTorch*. We use default training settings for each benchmark, with one additional hyper-parameter $s = 20$, meaning that in each iteration we randomly sample 20 sub-networks.

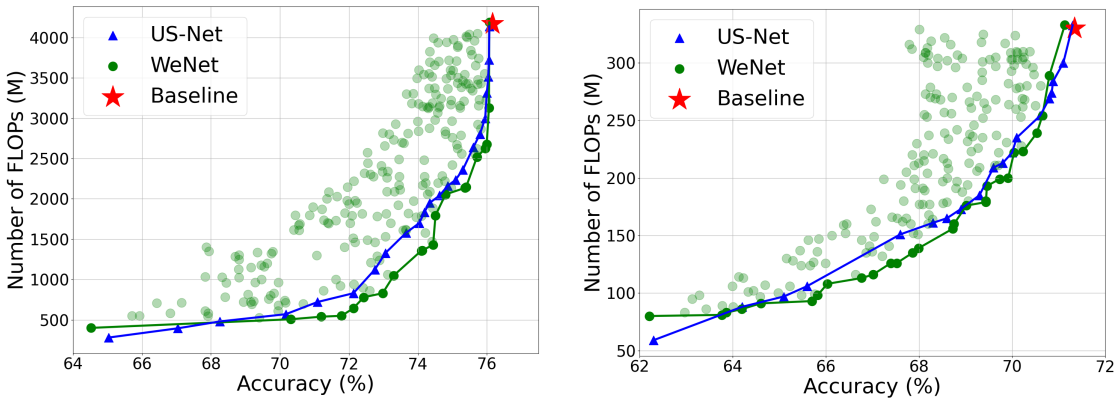
3. *Hardware setup:* We measure inference time and energy consumption on the NVIDIA Jetson Nano board.

5.4.2 Channel-Shuffling

In the first experiment, we demonstrate the benefit of channel-shuffling operation by evaluating five operating modes of WeNet with and without channel-shuffling. Table 5.2 shows the comparison result using ResNet-50, where each residual block has one channel-shuffling operation after the first 1×1 convolution layer. Channel-shuffling operation significantly improves accuracy for all five sub-networks. On average, channel-shuffling increases accuracy by 2.98%. Meanwhile, channel-shuffling introduces additional computational cost. On average, these operations use additional $3.36ms$ out of $118.38ms$ inference

Table 5.2: Comparison of top-1 accuracy and inference time with and without channel-shuffling operations using ResNet-50.

Sub-Network	Without Shuffling		With Shuffling	
	Acc.(%)	Time(ms)	Acc.(%)	Time(ms)
1/16	60.88	41.68	64.24(+3.36)	43.41(+1.73)
1/8	65.81	82.79	70.43(+4.62)	85.62(+2.38)
1/4	69.38	124.14	74.07(+4.69)	127.35(+3.21)
1/2	73.49	127.88	75.63(+2.41)	132.17(+4.29)
Full	75.99	198.62	76.07(+0.08)	203.38(+4.76)
Average	69.11	115.02	72.09(+2.98)	118.38(+3.36)



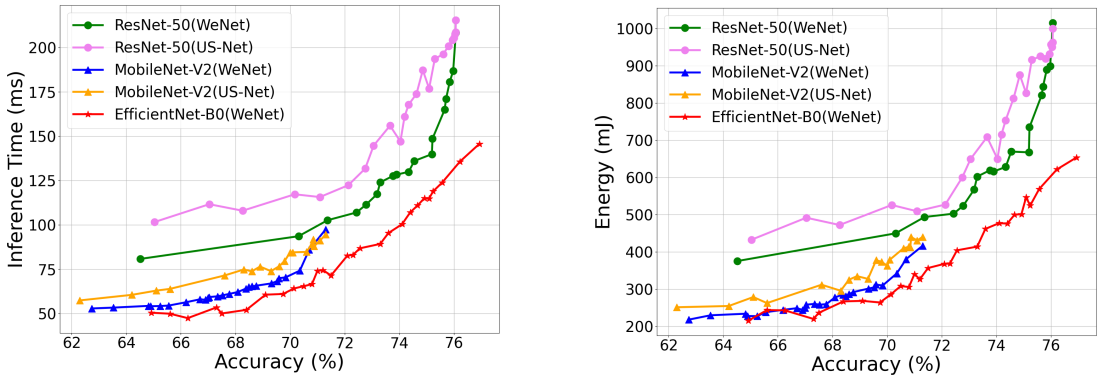
(a) ResNet-50 (b) MobileNet-V2

Figure 5.5: Comparison between WeNet and US-Net [1]

time. Compared to significant improvements on accuracy, increasing of inference time is negligible.

5.4.3 WeNet v.s. US-Net

In the second set of experiment, we compare WeNet against US-Net [1], which provides multiple operation modes by adjusting width ratio for each layer, using two benchmarks, i.e. *ResNet-50* [94] and *MobileNet-V2* [90]. After training WeNet using Algorithm 3, we explore the design space of WeNet operation modes using Algorithm 4, where computational cost is measured by number of Floating-Point Operations (FLOPs). Figure 5.5 shows the comparison results, where red star represents the original model. Compared to US-Nets, WeNet has less number of FLOPs in most accuracy range, which demonstrates the state-of-the-art performance. For ResNet-50, WeNet provides more efficient operating



(a) Inference time (b) Energy consumption

Figure 5.6: Inference Time and Energy Consumption on Jetson Nano Board

modes when accuracy is higher than 69%. For MobileNet-V2, WeNet performs better when accuracy is between 64% and 71%.

5.4.4 Inference Time and Energy Consumption

In the third set of experiment, we implement WeNet using all three benchmarks and plot operation modes on Pareto Frontier. Using Jetson Nano board, we evaluate inference time and energy consumption of these optimal points. Figure 5.6 shows the evaluation results, including US-Net in Section 5.4.3 as comparison. As Figure 5.6a shows, for all three benchmarks, WeNet substantially saves inference time by trading-off small amount of accuracy. Compare to number of FLOPs in Figure 5.5, the improvement of inference time is more significant, which is always lower then US-Net. Since each layer of WeNet consists of *independent* groups, which can be executed in parallel, inference time can be further saved if there are idle threads (or warps). Thus, the benefit of inference time is more significant. As Figure 5.6b shows, WeNet also substantially improves energy consumption compared against US-Net.

5.4.5 Evaluation on Different Devices

Finally, in Figure 5.7, we measure inference time of models trained in Section 5.4.3 using three devices, *i.e. Tesla P40* (GPU), *Tegra X1* (GPU) and *Xeon E5-2680* (CPU).

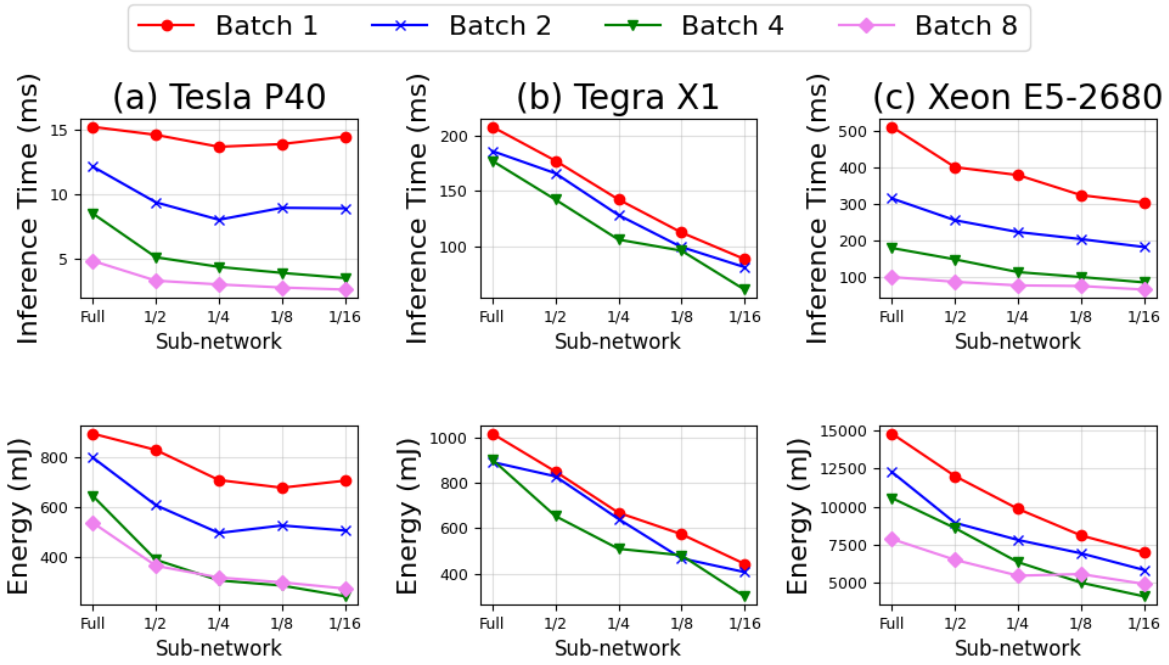


Figure 5.7: Evaluation of *ResNet50* on three devices, with different batch size

High-performance GPUs (*Tesla P40*) are already well-optimized for tensor operations. Thus, on these devices, inference time of *full network* is already fast enough. Disabling part of weights does not accelerate inference time as expected, but does improve energy efficiency due to fewer FLOPs. If we increase batch size, inference time and energy consumption keep reducing, since there is still enough computational resources to do parallel computing for each batch. On the other hand, for low-performance devices (*Tegra X1*), all threads (or warps) are busy even if batch size is 1. In this case, increasing batch size does not make much difference. But inference time and energy consumption vary a lot according to sub-network, where disabling more weights improves efficiency a lot. Since CPU (*Xeon E5-2680*) is not designed specifically for tensor operations, the inference time and energy consumption is much higher compared to GPU. But disabling more weights still improves energy efficiency significantly.

5.5 Conclusion

In this chapter, we proposed a novel dynamic network methodology, WeNet, which enables different subsets of weights on the fly to trade-off between accuracy and inference time. By enabling smaller subsets of weights, WeNet effectively forms a "sparser" sub-network with multiple separate groups of channels, where parallelism can be used to further improve efficiency. We also extended WeNet to convolutional layers using group convolution and channel shuffling, trained with switchable batch normalization and explored design space of possible sub-networks. We thoroughly evaluate our methodology using a number of DNN architectures on different hardware platforms, showing that WeNet provides a large number of energy-efficient operation modes, 73.2% of which provide better accuracy-efficiency trade-off compared to other methodologies.

CHAPTER 6

Low-precision Training using Forward-Forward Training Algorithm

6.1 Introduction

Deep neural networks (DNNs) have recently achieved state-of-the-art results across a variety of application domains, including computer vision, natural language processing, and more. To enhance their accuracy, the architectures of these networks are becoming increasingly complex and deeper. It has been reported that the computational resources needed to train the largest DNNs are doubling every 6 months from 2010 to 2022 [95]. Training and deploying these more intricate networks require significantly more energy and a larger memory footprint to accommodate the increase in parameters. For example, ResNet-50, a computer vision model proposed in 2015, comprises 26 million parameters, requires 7.5 GB of local DRAM, and demands 14 days of training on an NVIDIA M40 GPU [96]. On the other hand, GPT-3, a large language model released in 2020, which contains 175 billion parameters, requires 3,640 PF-days of computation for a training run, equivalent to 355 years of single-processor computing time, consuming 284,000 kWh of energy and costing over 4.6 million US dollars [97]. Concurrently, the evolution of DNNs

has driven a significant demand for training capabilities on edge devices. Training on the edge, as opposed to centralized servers, not only enhances model personalization but also ensures data privacy and security. However, these devices are typically constrained by limited memory and computing power, making real-time training increasingly challenging as DNN models grow in size. Therefore, minimizing the time, memory requirements, and energy consumption of DNN training has become crucial.

Numerous methodologies have emerged recently to enhance the training efficiency of DNNs. As an innovative alternative to the traditional backpropagation, the Forward-Forward (FF) algorithm [98] has been introduced to address the inefficiencies associated with the backward pass of gradient computation by replacing it with an additional forward pass. The FF algorithm brings about efficiency improvements in several ways. Firstly, it operates in a layer-by-layer greedy manner during training, thereby storing only the current layer in memory instead of the entire computational graph. This reduction in memory footprint is substantial. Additionally, the introduced forward pass in FF takes less time compared to the backward pass. Moreover, without propagating derivatives backward, the FF algorithm shows potential for implementation on analog devices.

This chapter delves into yet another benefit of the FF algorithm, specifically its applicability in implementing INT8 quantized training algorithms. Traditional backpropagation algorithms face instability when directly quantizing gradients, leading to computational challenges. Our observation highlights the effectiveness of FF's layer-wise greedy approach in mitigating the problem of accuracy loss accumulation often encountered in traditional backpropagation methods. The contributions of this chapter are outlined below.

- To the best of our knowledge, FF-INT8 is the first work to devise a low-precision (INT8) training method leveraging the Forward-Forward algorithm, which is advantageous as it learns effectively from a stream of input data while maintaining a smaller memory footprint, making it especially well-suited for resource-constrained edge devices. To date, no other studies have explored network quantization specifically

using the Forward-Forward algorithm.

- We propose an INT8 training method based on the Forward-Forward algorithm, named FF-INT8. This innovative approach involves greedily training each layer of a deep neural network independently, using INT8 precision. By confining gradient quantization to the current layer, our method effectively prevents the cumulative degradation of accuracy typically associated with precision loss in traditional training methodologies.

- Since the Forward-Forward algorithm is a greedy procedure, what is learned in earlier layers cannot be affected by what is learned in later layers, which typically results in low convergence accuracy and slow convergence rate. To address these limitations, we propose a novel loss function that accounts for interactions across layers, with a particular focus on the final output layer. Additionally, we introduce an updated training procedure termed the Look-afterward Forward-Forward algorithm, which is designed to better integrate this new loss computation. This enhancement aims to improve both the accuracy and speed of convergence in low-precision training environments.

- Utilizing FF-INT8, we train multiple DNNs on the Jetson Orin Nano board, which is outfitted with an INT8 engine. Our comprehensive evaluation covers model accuracy, training time, memory footprint, and energy consumption. The results demonstrate that our low-precision training algorithm not only significantly enhances efficiency but also maintains a high level of accuracy. With FF-INT8, we achieve speedups of 0.7% in DNN training. Furthermore, FF-INT8 also reduces the memory footprint by 22.2% and energy consumption by 6.0% compared to a state-of-the-art INT8 training algorithm.

The organization of this chapter is as follows. In section 6.2, we overview relevant previous works in DNN quantization. We then briefly introduce Forward-Forward algo-

rithm as preliminary in section 6.3. In section 6.4, we introduce FF-INT8 and its training algorithms. We provide our experimental results in section 6.5. Finally, we summarize our conclusion in Section 6.6.

6.2 Previous Work

Recent years have witnessed a proliferation of methodologies aimed at enhancing the efficiency of deep neural network (DNN) computations. These approaches include pruning, quantization, and neural architecture search, and others [99]. Particularly, quantization significantly reduces the memory footprint and computational cost by lowering the precision of weights and activations without altering the number of learned features. Neural network quantization can be categorized into post-training quantization (PTQ) and quantization-aware training (QAT). PTQ reduces the precision of a neural network model after it has been fully trained with floating-point precision. AdaRound [42] proposes a better weight-rounding PTQ scheme that adapts to the data and loss function. Smoothquant [100] proposes a PTQ solution for large language models by smoothing the activation outliers. PTQ4DM extends PTQ to diffusion models by adjusting calibration dataset [101]. To make the model more robust to the reduced precision, QAT methodologies are proposed to incorporates quantization directly into the training process. This means the model is aware of the quantization effects during training and can adapt to them, which typically leads to better performance and accuracy with quantized models compared to PTQ [43]. Additionally, QAT reduces the sensitivity of the model to weight initialization, as it learns quantization-friendly weights during training. This can result in more stable and consistent performance across different training runs. As an extension of QAT, PikeLPN proposed QuantNorm, allowing for quantizing the batch normalization parameters without compromising the model performance [44].

Compared to huge amount of studies on accelerating inference by model quantization, few works explore low-precision training including backward pass comprehensively.

MPTraining [102] uses 16-bit floating-point (FP16) to train DNNs with accuracy comparable to full-precision model. Minifloat [103] also proposes to use 8-bit floating-point numbers to train DNNs. Floating-point quantization requires specific hardware platform to achieve acceleration, which is not conducive to the practical deployment on resource-constrained devices. On the other hand, compared to floating-point, quantizing into 8-bit integer (INT8) has great potential in hardware acceleration, since INT8 operations are widely supported by recent GPUs. Besides, the 8-bit integer arithmetic is theoretically and practically 2× faster than FP16 and 4× faster than FP32. UI8 [4] utilizes direction sensitive gradient clipping and deviation counteractive learning rate scaling to implement a fully unified INT8 training for convolutional neural networks. DAI8 [104] achieves better gradient quantization by performing channel-by-channel gradient distribution perception, but it sets too many channel dimension quantization parameters, adding additional computational complexity. GDAI8 [5] also uses a data-aware dynamic quantization scheme to quantize various special gradient distributions. In summary, most INT8 training methods require extra computation to analyze and adapt to gradient distribution.

6.3 Background

Before introducing the Forward-Forward (FF) algorithm [98], we first briefly discuss backpropagation (BP) algorithm. Nowadays, most training algorithms for DNNs are built upon BP, which consists of a forward pass and a backward pass. In forward pass, input data is passed through DNN layer-by-layer to compute the final outputs and loss function. In backward pass, the loss is propagated backward through the network and gradients are computed using the computational graph to update weights and biases. Despite high accuracy, BP algorithm is inefficient in many aspects. First, it requires large storage to store the full computational graph in memory, which leads to a large memory footprint. Also backward pass has a larger delay and consumes higher energy. Moreover, many have argued that BP algorithm is not biologically plausible.

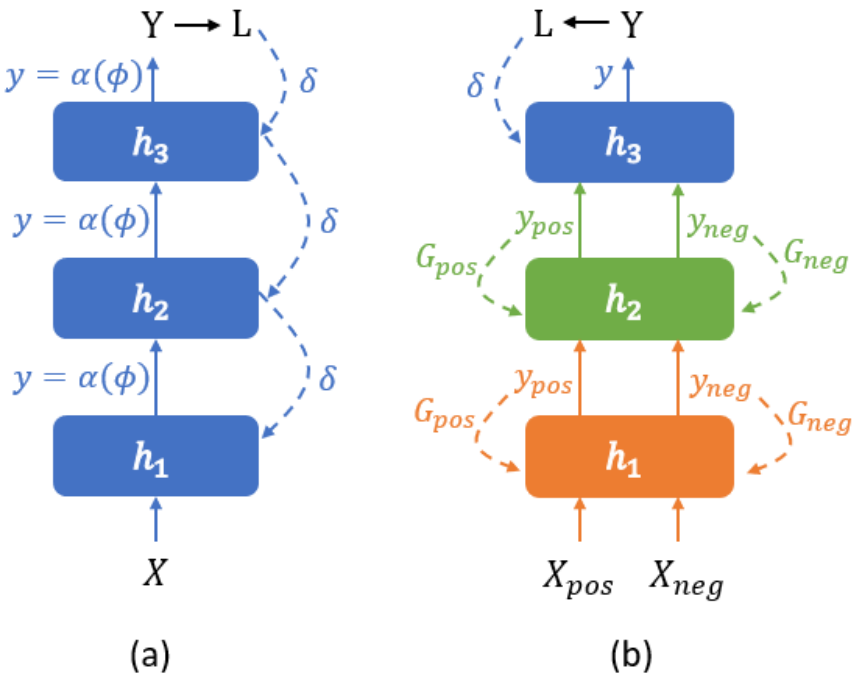


Figure 6.1: (a) Backpropagation consists of a forward pass and a backward pass. (b) The Forward-Forward algorithm uses "positive" and "negative" datasets, and trains each layer individually using "goodness" function G.

Alternatively, the Forward-Forward training algorithm [98] provides a brand new direction for DNN training, where two forward passes are used to replace the forward and backward passes in BP to improve efficiency. Different from BP, two forward passes of FF algorithm use different data with opposite objective, and trains DNN layer-by-layer in a greedy manner. Figure 6.1 compares training procedure between BP and FF. The first step is to generate "positive" and "negative" datasets from input data. One way to generate datasets is to annotate the input data using one-hot encoding vector of the label. More specifically, the one-hot vectors in positive samples are pointing to the true labels, while the ones in negative samples are pointing to the wrong labels. Besides loss function, FF algorithm uses *goodness* function to qualify how well each layer is trained. The positive pass operates on positive samples and adjusts the weights to increase the goodness in every hidden layer, while the negative pass operates on negative samples and adjusts the weights to decrease the goodness. One common measurement of goodness

function is the sum of squared neural activities, such that

$$G(\mathbf{x}) = ||\mathbf{y}||^2 = \sum_j y_j^2 \tag{6.1}$$

where $\mathbf{x}$ represents the vector of input neurons to the layer, $\mathbf{y}$ represents the vector of output neurons of the layer. FF algorithm aims to make the goodness function well above a certain threshold θ for positive data and well below θ for negative data, where θ can be considered as a hyperparameter that controls scale of the weights. In other words, it is a classification problem such that whether this set is positive or negative. We can express the probability of being positive or negative sample by applying logistic function σ to goodness function G,

$$p(positive) = \sigma(G - \theta) = \frac{1}{1 + e^{-(||\mathbf{y}||^2 - \theta)}} \tag{6.2}$$

$$p(negative) = 1 - \sigma(G - \theta) = \sigma\left(-(G - \theta)\right) = \frac{1}{1 + e^{(||\mathbf{y}||^2 - \theta)}} \tag{6.3}$$

FF algorithm trains each layer once until convergence and then optimize its successor. The main idea behind such greedy scheme is to make each layer "excited" about positive samples and, at the same time, less excited about negative samples, so that positive samples can be trained to match the correct label on the last layer.

FF algorithm improves efficiency in many aspects. First, instead of the entire computational graph, FF algorithm only stores current layer in memory, which significantly reduces memory footprint. Also, forward pass takes less time than backward pass. Moreover, the absence of backward pass gives FF algorithm the potential to be executed on analog devices. However, the heuristic property prevents earlier layers to learn from the later ones and final outputs. where training of earlier layers can be misleading. Thus, as a trade-off, FF algorithm usually suffers from accuracy loss.

6.4 Proposed Methodology

In this section, we describe our proposed methodology of INT8 Forward-Forward algorithm for efficient DNN training (FF-INT8). In the beginning, we analyze the effect of network depth in gradient quantization, as a motivation of using Forward-Forward algorithm in low-precision training. Then we describe our FF-INT8 training methodology in detail. We also modify the procedure of FF algorithm with a novel goodness function to improve convergence accuracy and convergence rate.

6.4.1 Network Depth and Gradient Quantization

Our FF-INT8 training method builds upon symmetric uniform quantization (SUQ), which is one of the most efficient quantization methods due to its hardware-friendly computation [105]. More specifically, given data x (*i.e.* weights, activations, gradients) in range (l, u) and a clipping value $c \in (0, \max(|l|, |u|))$, SUQ can be fomulated as

$$q(x) = \text{round}(\frac{\text{clamp}(x, c)}{s}) \tag{6.4}$$

where scale $s = c/127$ and clamp function is defined as

$$\text{clamp}(x, c) = \begin{cases} x & \text{if } |x| \leq c \\ c \times \text{sign}(x) & \text{if } |x| > c \end{cases} \tag{6.5}$$

The clipping value c directly affects quantization error of gradients, where quantization error is defined as the accuracy loss post quantization. The simplest and most efficient way is to set $c = \max(|x|)$ to cover the entire expressible data range of x.

As a preliminary experiment, we train ResNet-18 using CIFAR-10 dataset, with backpropagation and INT8 quantization including gradients. Figure 6.2 compares the changes of loss and accuracy per epoch between 32-bit floating-point (FP32) and INT8

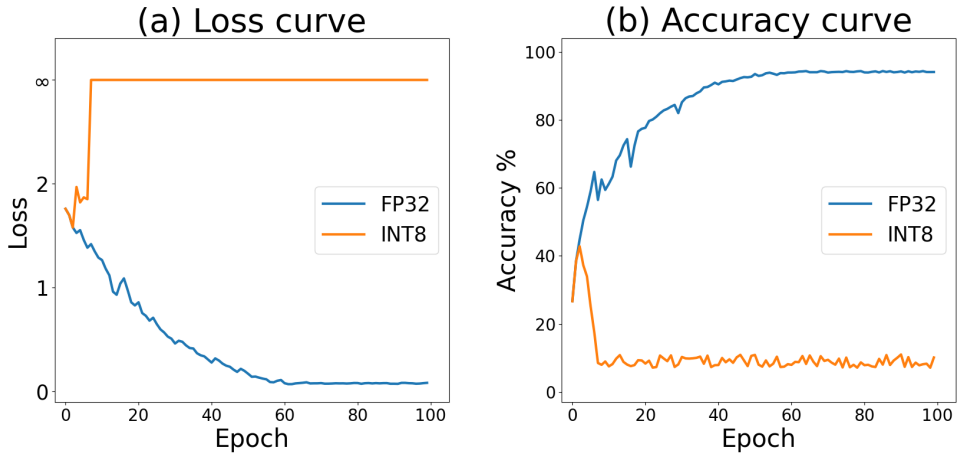


Figure 6.2: Loss and accuracy of ResNet-18 on CIFAR-10 when directly quantizing gradients to INT8.

training. The network is trained properly with FP32 precision. However, the loss of INT8 training increases dramatically as soon as we start training, while the accuracy drops to random level. We assumes that INT8 training fails due to the accumulation of quantization error through backward propagation of derivatives. To further prove this hypothesis, we train several fully-connected layers on MNIST dataset with different number of hidden layers, each of which is trained with FP32 and INT8 precision respectively. As Table 6.1 shows, as the number of hidden layers increases, the accuracy of FP32 training increases until the model overfits. On the contrary, the accuracy of INT8 training decreases dramatically as the network becomes deeper. The accuracy difference between FP32 and INT8 training is considerably small with a single-layer network, but increases significantly as we include the first hidden layer. We may conclude that quantization error accumulates as network becomes deeper. Instead of deep networks, INT8 training can be directly executed on a single-layer network. Finally, we plot gradient distribution of the *first* layers of different networks in Table 6.1, which are trained using FP32. As Figure 6.3 shows, for deeper networks, gradient distribution of earlier layers are sharper with larger extreme values, while distribution of single-layer network is more even. Thus, direct quantization in deeper network leads to large quantization error, since most gradients gather in a small range and we cannot tell the differences in such small range after quantization. On the other hand, direct quantization in single-layer network is less error-prone.

Table 6.1: Accuracy of fully-connected layers on MNIST dataset with different number of hidden layers and training precision. Each hidden layer consists of 500 neurons. Networks are trained using 32-bit floating-point or 8-bit integer.

Number of Hidden Layers	FP32 Acc. (%)	INT8 Acc. (%)	Accuracy Difference (%)
0	89.5	87.9	-1.6
1	93.4	73.8	-19.6
2	94.5	62.4	-32.1
3	94.3	65.2	-29.1

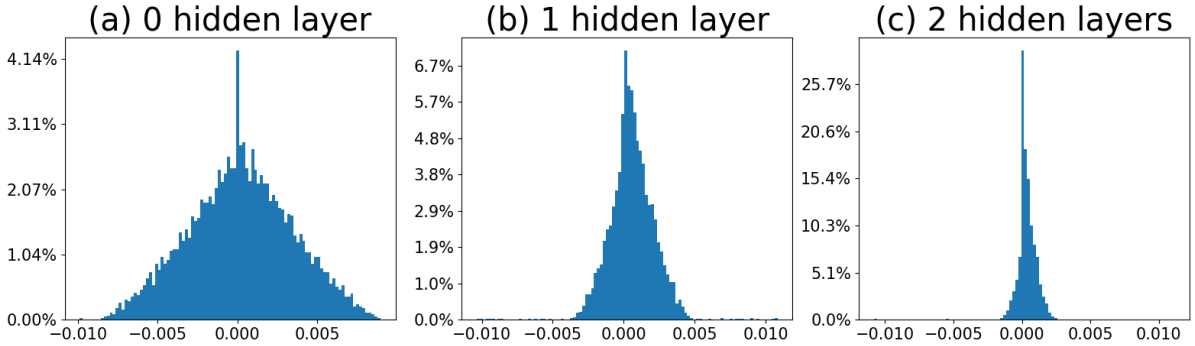


Figure 6.3: Gradient distribution of first layer with different number of hidden layers.

In summary, directly training multi-layer networks with INT8 is risky due to the accumulation of quantization error. Gradient distribution of earlier layers also poses a challenge to quantization of gradients in small range. On the other hand, single-layer network does not have these problems and is more suitable for INT8 training. Although almost all DNNs have multiple layers, FF algorithm proposes to train each layer individually in a greedy manner, making FF algorithm and INT8 training a perfect combination.

6.4.2 INT8 Forward-Forward Algorithm

To devise INT8 Forward-Forward algorithm, we first need to decide the clipping value c and round function in Equation 6.4. Since gradient distribution is smooth in single-layer network as shown in Figure 6.3(a), it is safe to set $c = \max(|x|)$ to cover the entire range. To ensure the quantized gradients maintain an unbiased expectation, we adopt stochastic

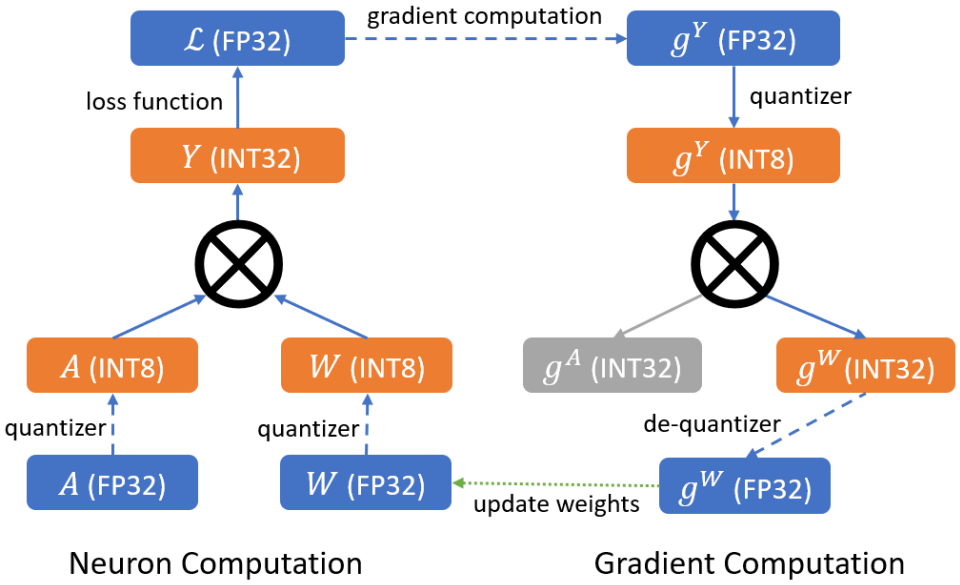


Figure 6.4: Dataflow of INT8 Forward-Forward algorithm on a single layer (dense or convolution layer).

Algorithm 5: INT8 Forward-Forward Algorithm

Input : Training set $(\mathbf{X}, \mathbf{Y})$, Number of epochs n,
Output : Trained network M with k layers

Initialize neural network M
Generate positive and negative sets from $(\mathbf{X}, \mathbf{Y})$
for $layer = 1, 2, \ldots, k$ **do**
  for $num\_epoch = 1, 2, \ldots, n$ **do**
    Quantize input data A and weights W into INT8 A_{int} and W_{int}
    Compute layer outputs $Y_{int} = W_{int} \bigotimes A_{int}$
    Compute loss function $\mathcal{L}$
    Compute gradient g^Y of Y to the loss function
    Quantize gradient g^Y into INT8 g^Y_{int}
    Propagate gradient to g^W_{int}
    De-quantize and update weights W
  end
end

rounding [106] as follows

$$\text{round}(x) = \begin{cases} \lfloor x \rfloor & \text{w.p. } 1 - (x - \lfloor x \rfloor) \\ \lfloor x \rfloor + 1 & \text{w.p. } x - \lfloor x \rfloor \end{cases} \tag{6.6}$$

FF-INT8 algorithm trains each layer individually with both positive and negative datasets in a greedy manner, where only one layer is trained at a time. For either positive

or negative samples, our INT8 training workflow of a single layer is demonstrated as Figure 6.4. We first use SUQ with stochastic rounding to quantize both input data A and weight W, as Equation 6.4 and 6.6, where INT8 MAC operation is used to compute dot product in either dense layer or convolution layer. Negative log-likelihood is used as loss function. For positive dataset, loss function $\mathcal{L}_{pos}$ is computed as

$$\mathcal{L}_{pos} = -\log p(positive) = \log\left(1 + e^{-(||\mathbf{y}||^2 - \theta)}\right) \tag{6.7}$$

For negative dataset, loss function $\mathcal{L}_{neg}$ is computed as

$$\mathcal{L}_{neg} = -\log p(negative) = \log\left(1 + e^{(||\mathbf{y}||^2 - \theta)}\right) \tag{6.8}$$

where $\mathbf{y}$ represents the vector of output neurons of the current layer, and θ is the hyperparameter that represents threshold in Equation 6.2 and 6.3. Two loss functions $\mathcal{L}_{pos}$ and $\mathcal{L}_{neg}$ can be rewritten into one function as

$$\mathcal{L}(\mathbf{x}) = \log\left(1 + e^{-\beta(||\mathbf{y}||^2 - \theta)}\right) \tag{6.9}$$

where $\beta = 1$ means that input $\mathbf{x}$ is a positive sample, and $\beta = -1$ means that input $\mathbf{x}$ is a negative one. By optimizing this loss function, we encourage outputs of positive samples to be large and outputs of negative samples to be small. Then gradient g^Y is computed and quantized into INT8. Since gradients are not back-propagated to previous layers, there is no need to compute the gradients of input data g^A. Only gradients of weights g^W are computed and updated to current weights. Thus, for computation of neuron activity Y and gradient g_W, we replace floating-point computation using INT8, which saves a large amount of computation. We summarize the entire procedure of FF-INT8 in Algorithm 5.

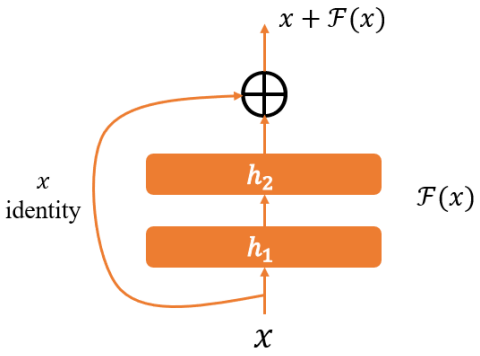


Figure 6.5: Residue block is commonly used in many modern DNN architectures.

6.4.3 FF-INT8 Algorithm with Look Afterward

In Section 6.4.2, we propose FF-INT8 training algorithm to speed up MAC operations in computation of neuron activity and gradient. As discussed in Section 6.3, FF algorithm is efficient, but only allows one layer to learn from its input and prevents it from communicating with later layers, especially the final outputs. Thus, earlier layers are only trained to classify between positive and negative samples, where the task of classifying specific labels is only left to the very last layer. Once a layer is trained, FF algorithm continues to optimize the next layer and never looks back. Without receiving feedback from the final outputs, training of earlier layers can be misleading. As a result, FF algorithm usually has lower accuracy and requires much more epochs to converge compared to backpropagation, as demonstrated later in Section 6.5.3. Also, FF algorithm does not fit certain structure, *e.g.* residue blocks, which are used in many modern DNN architectures. As Figure 6.5 shows, residue block allows inputs to be skipped over certain layers and accumulated to block's output [94]. Thus, all layers within same block should be optimized interactively. However, due to the limitation of FF algorithm, each layer within the same block is trained individually without knowing later layers, which may cause significant accuracy loss.

To solve these issues, our solution is to enhance the connection between different layers, especially with final outputs, while keeping the advantage of efficiency in FF algorithm. We name this modifed algorithm "FF-INT8 Algorithm with *Look Afterward*". Previously,

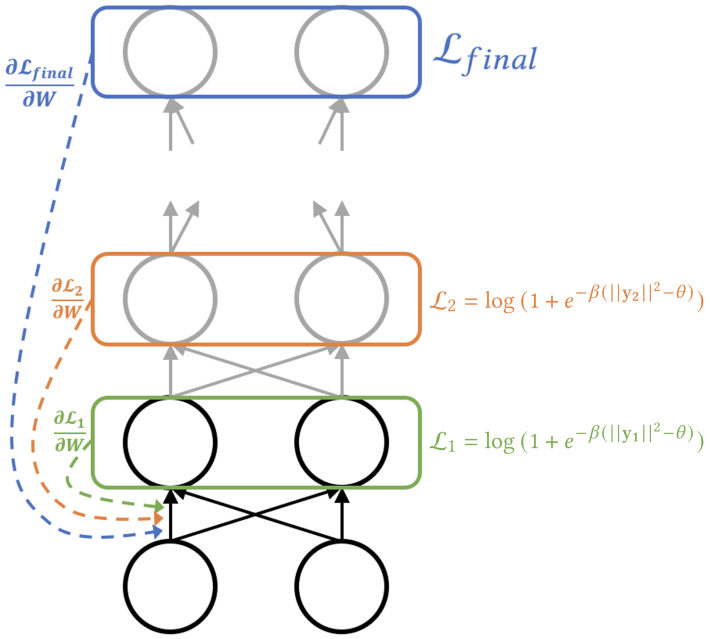


Figure 6.6: Gradient computation of FF algorithm after modification, where loss functions of later layers are considered.

Algorithm 6: FF-INT8 Algorithm with Look Afterward

Input : Training set $(\mathbf{X}, \mathbf{Y})$, Number of epochs n
Output : Trained network M with k layers

Initialize neural network M
Initialize hyperparameter $\lambda = 0$ in loss function
Generate positive and negative sets from $(\mathbf{X}, \mathbf{Y})$
for $num\_epoch = 1, 2, \ldots, n$ **do**
    Execute forward pass using INT8 precision
    Compute goodness function $G = ||\mathbf{y}||^2$ for each layer
    for $current\ layer = 1, 2, \ldots, k$ **do**
        Compute loss for current layer
        Update weights using gradients in INT8 precision
    end
    Increase hyperparameters λ in loss function
end

when training one layer, only goodness function of current layer is considered in the loss function as described in Equation 6.9. To receive feedback from different layers, we rewrite the loss function to take goodness functions of later layers into consideration. Assume that $\mathcal{L}_1$ denotes the loss of current layer, $\mathcal{L}_2, \mathcal{L}_3, \ldots$ denote the loss of following layers, and $\mathcal{L}_{final}$ denotes the loss of final outputs at the last layer. The new loss function is as

follow:

$$\mathcal{L}_{new} = \mathcal{L}_1 + \lambda \times (\mathcal{L}_2 + \mathcal{L}_3 + \cdots + \mathcal{L}_{final}) \tag{6.10}$$

where λ is a coefficient to balance between current layer and later layers. Figure 6.6 demonstrates gradient computation with the new loss function, which is

$$\frac{\partial \mathcal{L}_{new}}{\partial W} = \frac{\partial \mathcal{L}_1}{\partial W} + \lambda \times \frac{\partial (\mathcal{L}_2 + \mathcal{L}_3 + \cdots + \mathcal{L}_{final})}{\partial W} \tag{6.11}$$

Previously, Algorithm 5 trains each layer with n epochs individually without knowing information in later layers. For neural network with k layers, Algorithm 5 requires $k \times n$ epochs in total, which is a large number, especially considering the growing depth of DNNs nowadays. Although each forward pass only needs to reach the current layer to be trained, there are still n epochs requiring complete forward passes in order to train the final layer.

For new loss function in Equation 6.10, since it consists of losses of later layers including the final one, we have to execute complete forward passes for each layer. For the sake of efficiency, we propose Algorithm 6 to utilize one forward pass to update all layers. More specifically, for each epoch, we execute complete forward pass (line 5), and compute goodness function as Equation 6.1 (line 6). We can then construct loss functions of all layers as Equation 6.10 using *only* goodness function of each layer (line 8). Gradients can be computed directly as Equation 6.11 *without any backpropagation process* (line 9). Therefore, "FF-INT8 Algorithm with Look Afterward" managed to optimize all layers with n' epochs, which is much faster than $k \times n$ epochs in previous one. Also, it enables interaction between different layers and allows different layers to optimize together, which improves accuracy. As for memory footprint, similar to previous algorithm, since "FF-INT8 Algorithm with Look Afterward" does not have backpropogation process, it does not require memory usage for that. Besides basic memory usage for forward pass, it only needs addtional memory usage for goodness function (Equation 6.1) and loss function

Table 6.2: DNN Architectures and Datasets

DNN	Dataset	Num. of Params (M)
Multi-layer perceptron (MLP) (2 hidden layers)	MNIST	1.79
MobileNet-V2 [90]	CIFAR10	2.24
EfficientNet-B0 [92]		3.39
ResNet-18 [94]		11.19

(Equation 6.10) for each layer, which is negligible compared to other memory usage.

One more hyperparameter in "FF-INT8 Algorithm with Look Afterward" is λ, which balances between current layer and other layers. For first few epochs, since each layer is less optimized, our priority is to train each layer so that it can basically tell between positive samples and negative samples. In other word, we want to discourage the interaction between layers at the beginning of training, since later layers are not optimized and information of later layers does not help the optimization of current layer. After a few epochs, each layer is better optimized. We then gradually increase λ to encourage interaction between layers to improve convergence accuracy.

6.5 Experimental Results

In this section, we first introduce our experimental setup. The first experiment is to compare FF-INT8 algorithm with and without "Look Afterward", to show the benefit of it. We then theoretically analyze and compare the number of each operation, demonstrating that our methodology reduces computational cost. Finally, we comprehensively use our methodology to train different DNNs, and compare against other methodologies to demonstrate state-of-the-art performance.

6.5.1 Experimental Setup

1. Datasets and models: We train 4 different models using FF-INT8, as listed in Table 6.2. Note that the sizes of feature maps and kernels are adjusted based on

Table 6.3: Technical Specifications of NVIDIA Jetson Orin Nano

GPU	1024-core NVIDIA Ampere architecture GPU
CPU	6-core Arm®Cortex®-A78AE v8.2 64-bit CPU
Memory	8GB 128-bit LPDDR5 68 GB/s
Power	7-15W
AI Performance	40 TOPs

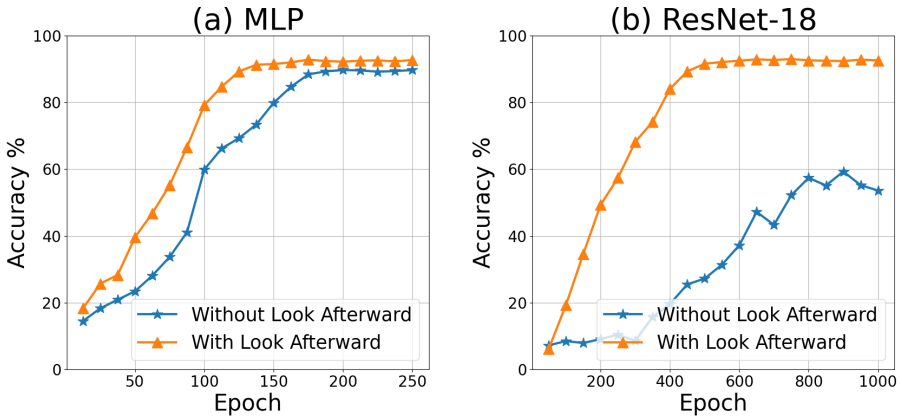


Figure 6.7: Test accuracy for different number of epochs where MLP and ResNet-18 are trained using FF-INT8, with and without "Look Afterward" respectively.

dataset used. Also, since we experiment our methodology over edge devices, batch size is limited to 32.

2. Hardware setup: NVIDIA Jetson Orin Nano board with an INT8 engine is used to measure the training time, energy consumption and memory footprint. Specifications of the device are listed in Table 6.3, showing that this board is representative of edge devices, which is resource-constrained for computationally intensive tasks such as DNN training.

3. Quantization method: We adopt stochastic rounding method as described in Section 6.4.2. An 8-bit optimizer [107] is used for gradient quantization and update.

4. Other hyperparameter: In Equation 6.2 and 6.3, threshold θ is used to separate positive and negative samples, while controlling the scale of weights. To avoid gradient explosion or vanishing, we set $\theta = 2.0$. In Equation 6.10, λ is a coefficient to balance between current layer and later layers, which is initialized to 0, and increased by 0.001 for each epoch.

6.5.2 Training with "Look Afterward"

In the first set of experiment, we show that "Look-Afterward" scheme improves accuracy for FF-INT8 as described in Section 6.4.3. The vanilla FF-INT8 algorithm without "Look Afterward" is described in Algorithm 5. The updated FF-INT8 algorithm with "Look Afterward" is described in Algorithm 6. We first train MLP with 2 hidden layers using both FF-INT8 algorithms and demonstrate results in Figure 6.7(a). Without "Look Afterward" scheme, the accuracy converges to nearly 90% with 180 epochs. However, with "Look Afterward" scheme, MLP reaches slightly higher convergence accuracy with only 130 epochs. Thus, we argue that by interaction with later layers, earlier layers compute gradients and update weights in a "more optimized way", which leads to higher convergence accuracy and faster convergence speed.

Figure 6.7(b) demonstrates the results in ResNet-18, which consists of residue blocks as Figure 6.5. Different from MLP, if ResNet-18 is trained without "Look Afterward" scheme, the accuracy converges to only 60%, while the accuracy curve is very unstable. There are two reasons for the bad performance:

1. The scale of ResNet-18 is much larger compared to MLP, which means it is much difficult for a greedy approach like FF algorithm to find the optimized solution.

2. Residue block allows input to be skipped over few layers and accumulated to block's output. However, without "Look Afterward" scheme, when previous layers in a block are trained, the vanilla FF algorithm does not consider the accumulation operation in block's output layer. And when block's output layer is trained, the previous layers have already trained and cannot be modified. Thus, the optimization process is limited, which leads to low convergence accuracy.

As shown in Figure 6.7(b), by layer interaction, "Look Afterward" scheme solves both problems and significantly improves convergence accuracy. Given that many modern DNN architectures incorporate residual blocks, the "Look Afterward" scheme significantly

Table 6.4: Comparison of computational cost between INT8 Forward-Forward algorithm and INT8/FP32 backpropagation

Computation		Operation	Counts (OPs)
FF-INT8	Quantization Phase	32-bit CMP	32.4K
		32-bit FADD	165.9K
	MAC Phase	8-bit MUL	23.8M
		8-bit ADD	23.8M
BP-FP32	MAC Phase	32-bit FADD	898.2M
		32-bit FMUL	898.2M
GDAI8 [5] (BP-INT8)	Quantization Phase	32-bit CMP	7.2K
		32-bit FADD	18.4K
	MAC Phase	8-bit MUL	898.2M
		8-bit ADD	898.2M

Table 6.5: Summary of model accuracy, training time, energy consumption and memory footprint between different approaches. Training algorithms are based on either BP (backpropagation) or FF (the Forward-Forward algorithm). The suffix denotes the precision, where FP32 means 32-bit floating-point, and INT8 means quantizing to 8-bit integer. UI8 [4] refers to unified INT8 training algorithm, and GDAI8 [5] refers to gradient distribution-aware INT8 training algorithm.

Model	Training Algorithm	Accuracy (%)	Time (s)	Energy (J)	Memory (MB)
MLP	BP-FP32	94.5	482.3	2315.0	247.6
	BP-INT8	52.4	326.1	1206.6	213.9
	BP-UI8 [4]	92.3	335.2	1277.1	197.0
	BP-GDAI8 [5]	93.8	344.9	1345.4	182.6
	FF-INT8	94.0	321.7	1097.0	140.7
MobileNet-v2	BP-FP32	91.7	2370.8	11593.2	649.8
	BP-INT8	5.9	1851.6	7836.0	571.6
	BP-UI8 [4]	87.2	1960.0	7618.5	592.6
	BP-GDAI8 [5]	90.9	1790.7	6528.1	578.9
	FF-INT8	90.5	1743.9	6344.3	491.0
EfficientNet-B0	BP-FP32	89.4	2692.8	13356.2	861.0
	BP-INT8	11.8	2095.0	8563.9	703.9
	BP-UI8 [4]	85.3	2230.8	8656.2	735.5
	BP-GDAI8 [5]	88.9	2177.1	8589.9	692.0
	FF-INT8	88.5	2219.9	8203.8	505.2
ResNet-18	BP-FP32	93.6	3853.0	18764.1	1096.4
	BP-INT8	7.2	2676.1	10436.8	885.8
	BP-UI8 [4]	89.7	2873.8	11466.5	920.7
	BP-GDAI8 [5]	92.9	2751.6	10291.0	894.1
	FF-INT8	92.1	2876.9	10526.5	682.3
Avg. difference between FF-INT8 and BP-GDAI8		Reduce 0.3%	Save 0.7%	Save 6.0%	Save 22.2%

enhances the FF-INT8 algorithm.

6.5.3 Analysis of Computational Cost

In Section 6.5.2, our experiment suggests that FF-INT8 training requires a large number of epochs to converge. In this section, we analyze the theoretical computational cost to demonstrate that although FF-INT8 training has much more epochs compared to backpropagation, it is still efficient. We count the number of required operations to train 4-layer MLP using MNIST dataset with three settings, *i.e.* the proposed FF-INT8 training method with "Look Afterward", and backpropagation with 32-bit floating-point (BP-FP32) as baseline. We also include Gradient Distribution-Aware INT8 training algorithm (GDAI8) [5] for comparison, which is an INT8 training algorithm based on backpropagation. Table 6.4 summarize the amount of computation required for training a mini-batch of 10 samples for three approaches. In FF-INT8, we have quantization phase and multiply–accumulation (MAC) phase, where computation of quantization phase is negligible compared to MAC. Comparing MAC phase, FF-INT8 training requires 23.8M 8-bit MAC operations, whereas backpropagation approach (BP-FP32 or GDAI8) requires 898.2M MAC operations (FP32 or INT8). This is because FF algorithm does not have large matrix multiplication to back-propagate derivatives from the last layer to the first. Thus, per mini-batch, FF-INT8 only requires 2.6% of MAC operations in backpropagation approach. INT8 arithmetic is also 4x faster than FP32 in hardware, which makes FF-INT8 hundres of times more efficient compared to BP-FP32 per epoch. Additionally, FF-INT8 only computes forward pass and does not compute backward pass of gradients. In many devices, forward pass is more efficient due to hardware optimization for model inference. In conclusion, although FF-INT8 training needs a large number of epochs, theoretically it is still more efficient compared to both backpropagation approaches.

6.5.4 Accuracy, Time, Energy, Memory Footprint

For the last set of experiment, we compare model accuracy, training time, energy consumption between multiple training algorithms, and demonstrate results in Table 6.5.

As baseline, we train DNNs using backpropagation (BP) with 32-bit floating-point operations, denoted as *BP-FP32*. If directly quantizing gradients to INT8 using BP (BP-INT8), although we observe significant time, energy and memory savings, accuracy decreases dramatically due to the accumulation of quantization error, as analyzed in Section 6.4.1. With deeper architecture, direct quantization makes training much worse. Thus, for simple architectures such as MLP, training accuracy of BP-INT8 reaches 50%. On the other hand, deeper models such as MobileNet or ResNet have training accuracy below 10%.

As a comparison, we train each DNN with two existing INT8 training algorithms, *i.e.* Unified INT8 Training Algorithm (BP-UI8) [4] and Gradient Distribution-Aware INT8 Training Algorithm (BP-GDAI8) [5]. Both algorithms are based on BP, and quantize gradients based on analysis of gradient distribution. Since they use BP as basic with additional operation of gradient monitoring and analysis, the computational cost of both is slightly higher than direct quantization (BP-INT8). The training accuracy is much higher compared to BP-INT8, and is close to traditional BP with FP32 operands. BP-GDAI8 has the state-of-the-art performance in terms of trade-off between model accuracy and efficiency, which is chosen as the comparison model to our methodology.

Finally, we implement Forward-Forward Algorithm with "Look Afterward" scheme (FF-INT8) as described in Section 6.4.3. As analysis in Section 6.4.2, gradient quantization works better on FF algorithm due to its layer-by-layer greedy manner. Thus, the training accuracy is much higher compared to direct quantization in BP (BP-INT8), and is also higher compared to BP-UI8. By comparing to BP-GDAI8, we show that training accuracy of FF-INT8 is very close to the state-of-the-art INT8 training algorithm, with a 0.3% reduction. For training time and energy consumption, since Jetson Orin Nano board has an INT8 engine, we observe a significant improvement compared to training with FP32 operations. As analysis in Section 6.5.3, although FF-INT8 requires more epochs compared to BP-base algorithms, each epoch is much faster. Thus, our FF-INT8 algorithm is more efficient compared to BP-GDAI8, with 0.7% saving in training time and 6.0%

saving in energy consumption. We also notice that FF-INT8 has even better performance in smaller models (such as MLP) compared larger ones (such as ResNet).

As mentioned in Section 6.3, one of the major benefits of FF algorithm is smaller memory footprint. Normally, BP algorithms rely on automatic differentiation, which requires to store the large computational graph for gradient backpropagation in memory. But since FF algorithm does not have backward pass, it does not need to store this computational graph. Such improvement in memory footprint is further enhanced by INT8 operations. Thus, compared to BP-GDAI8, FF-INT8 saves memory footprint by 22.2%.

6.6 Conclusion

In this chapter, we proposed a novel low-precision training methodology, *i.e.* INT8 Forward-Forward Algorithm (FF-INT8). We found that error from gradient quantization increases as DNN becomes deeper, while INT8 quantization can be implemented on a single-layer network. Since Forward-Forward algorithm trains DNNs in a greedy manner, which only trains one layer at one time, we implemented INT8 training on Forward-Forward algorithm. To improve accuracy, we proposed a novel loss function which also considers the following layers, and modified training procedure to make it more effective. Compared against the state-of-the-art approach, FF-INT8 accelerates training by 0.7%, decreases energy consumption by 6.0%, and significantly reduces memory footprint by 22.2%, while maintaining high accuracy.

CHAPTER 7

Summary and Possible Extensions

This dissertation has explored various techniques to enhance the efficiency of deep learning computing from both hardware and software perspectives. The primary contributions of this work include the development of advanced approximate computing methodologies, the introduction of dynamic deep neural network architectures, and the design and implementation of novel low-precision training algorithms.

1. Approximate Logic Synthesis: We developed a comprehensive methodology for approximate logic synthesis using Boolean matrix factorization. This approach enables the generation of approximate circuits that achieve significant reductions in design area and power consumption while maintaining acceptable levels of computational accuracy. The proposed methodology has been shown to be versatile, scalable, and effective across a wide range of benchmark circuits.

2. Runtime Configurable Approximate Circuits: The dissertation introduced the concept of runtime configurable approximate circuits (RUCA) with self-correcting capabilities. This innovation allows circuits to dynamically adjust their accuracy levels in response to varying computational demands and power availability, thereby optimizing both performance and energy efficiency. The experimental results demon-

strated the potential of RUCA designs to significantly reduce power consumption in various application scenarios.

3. Dynamic Neural Networks: We proposed and evaluated a configurable deep neural network architecture, WeNet, which leverages dynamic weight-enabling techniques for efficient inference. This approach allows the network to adapt its computational complexity at runtime, making it suitable for deployment on resource-constrained devices without sacrificing performance.

4. Low-Precision Training Algorithms: The Forward-Forward (FF) algorithm was introduced as a novel approach to low-precision training of deep neural networks. The FF-INT8 algorithm, in particular, was shown to reduce the memory footprint and computational cost associated with traditional backpropagation methods while maintaining high levels of accuracy. This advancement is especially relevant for training on edge devices with limited computational resources.

Overall, this dissertation has demonstrated the feasibility and effectiveness of integrating approximate computing techniques into logic synthesis and deep learning. The proposed methodologies not only address the growing need for energy-efficient computing but also provide practical solutions for deploying advanced machine learning models in resource-constrained environments.

While this dissertation has made significant contributions to the field of efficient deep learning computing, there are several avenues for future research that could further enhance the impact of this work:

1. Extension to Other Deep Learning Models: While the methodologies developed in this dissertation were primarily applied to convolutional neural networks (CNNs), they could be extended to other types of deep learning models, *e.g.* transformers, which forms the basis of large language models nowadays. Investigating the applicability of approximate computing techniques to these models could lead to new

insights and further optimization opportunities.

2. Integration with Emerging Hardware Technologies: As new hardware technologies, such as neuromorphic computing and quantum computing, continue to evolve, there is potential to integrate the techniques developed in this dissertation with these emerging platforms. Exploring how approximate computing methodologies can be adapted to take advantage of the unique characteristics of these technologies could lead to significant breakthroughs in energy-efficient computing.

3. Advanced Design Space Exploration: The design space exploration techniques employed in this dissertation focused on specific metrics such as area, power, and accuracy, and mainly uses greedy search methods. Future work could investigate more advanced multi-objective optimization techniques that simultaneously consider a broader range of metrics, such as fault tolerance and reliability. And methods such as reinforcement learning could also be used to improve design space exploration. This could lead to more robust and versatile approximate computing methodologies.

4. Exploration of SW/HW Co-design: Combining approximate computing methodologies proposed in this dissertation could lead to new architectures that are both energy-efficient and scalable. Investigating how these paradigms can be harmonized to create holistic solutions for distributed AI systems would be a promising direction for future research.

In conclusion, this dissertation has laid a strong foundation for the development of energy-efficient hardware circuits and deep learning models. By exploring the possible extensions outlined above, future research can build on this work to create even more advanced and impactful solutions for the challenges facing modern computing systems.

References

[1] J. Yu and T. S. Huang, "Universally slimmable networks and improved training techniques," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1803–1811, 2019.

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

[3] V. Mrazek, Z. Vasicek, and L. Sekanina, "Design of quality-configurable approximate multipliers suitable for dynamic environment," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 264–271, IEEE, 2018.

[4] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, "Towards unified int8 training for convolutional neural network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1969–1979, 2020.

[5] S. Wang and Y. Kang, "Gradient distribution-aware int8 training for neural networks," *Neurocomputing*, vol. 541, p. 126269, 2023.

[6] P. Raj, K. Saini, and C. Surianarayanan, *Edge/Fog Computing Paradigm: The Concept, Platforms and Applications.* Academic Press, 2022.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[8] K. Simonyan, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[9] G. Armeniakos, G. Zervakis, D. Soudris, and J. Henkel, "Hardware approximate techniques for deep neural network accelerators: A survey," *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–36, 2022.

[10] G. Zervakis, H. Saadat, H. Amrouch, A. Gerstlauer, S. Parameswaran, and J. Henkel, "Approximate computing for ml: State-of-the-art, challenges and visions," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 189–196, 2021.

[11] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

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

[13] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 124–134, ACM, 2011.

[14] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible approximation with selective dynamic loop perforation," in *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, (New York, NY, USA), pp. 341–351, ACM, 2018.

[15] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal, "Sfu-driven transparent approximation acceleration on gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 15:1–15:14, ACM, 2016.

[16] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 13–24, Dec 2013.

[17] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix-up: Relaxing program semantics to unleash parallelization," in *Proceedings of the 13th Annual*

IEEE/ACM International Symposium on Code Generation and Optimization, CGO '15, (Washington, DC, USA), pp. 235–245, IEEE Computer Society, 2015.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *SIGPLAN Not.*, vol. 47, pp. 301–312, Mar. 2012.

[19] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, pp. 213–224, Mar. 2011.

[20] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Exploiting staleness for approximating loads on cmps," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 343–354, Oct 2015.

[21] G. Karakonstantis and K. Roy, "Voltage over-scaling: A cross-layer design perspective for energy efficient systems," in *20th European Conference on Circuit Theory and Design (ECCTD)*, pp. 548–551, IEEE, 2011.

[22] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 820–825, 2012.

[23] S. Hashemi, R. I. 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*, ICCAD '15, (Piscataway, NJ, USA), pp. 418–425, IEEE Press, 2015.

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

[25] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate

applications," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

[26] R. G. Rizzo, A. Calimera, and J. Zhou, "Approximate error detection-correction for efficient adaptive voltage over-scaling," *Integration*, vol. 63, pp. 220–231, 2018.

[27] J. Huang, T. N. Kumar, and H. Abbas, "Simulation-based evaluation of approximate adders for image processing using voltage overscaling method," in *2020 IEEE 5th International Conference on Signal and Image Processing (ICSIP)*, pp. 499–505, IEEE, 2020.

[28] Y. Mannepalli, V. B. Korede, and M. Rao, "Novel approximate multiplier designs for edge detection application," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pp. 371–377, 2021.

[29] W. Ahmad, B. Ayrancioglu, and I. Hamzaoglu, "Low error efficient approximate adders for fpgas," *IEEE Access*, vol. 9, pp. 117232–117243, 2021.

[30] K. Nepal, S. Hashemi, H. Tann, R. I. Bahar, and S. Reda, "Automated high-level generation of low-power approximate computing circuits," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–13, 2016.

[31] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: Systematic logic synthesis of approximate circuits," in *Design Automation Conference*, pp. 796–801, 2012.

[32] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: Synthesis of approximate sequential circuits," in *Design, Automation & Test in Europe Conference*, pp. 1–6, 2014.

[33] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Automation and Test in Europe*, pp. 1367–1372, 2013.

[34] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 779–786, 2013.

[35] Z. Vasicek and L. Sekanina, "Evolutionary design of complex approximate combinational circuits," *Genetic Programming and Evolvable Machines*, vol. 17, no. 2, pp. 169–192, 2016.

[36] S. Frohlich, D. Grobe, and R. Drechsler, "Error Bounded Exact BDD Minimization in Approximate Computing," in *International Symposium on Multiple-Valued Logic*, pp. 254–259, 2017.

[37] S. Frohlich, D. Grobe, and R. Drechsler, "Approximate hardware generation using symbolic computer algebra employing grobner basis," in *Design, Automation and Test in Europe*, pp. 889–892, 2018.

[38] J. Lee, S. Park, S. Mo, S. Ahn, and J. Shin, "Layer-adaptive sparsity for the magnitude-based pruning," *arXiv preprint arXiv:2010.07611*, 2020.

[39] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2023.

[40] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016.

[41] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[42] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in *International Conference on Machine Learning*, pp. 7197–7206, PMLR, 2020.

[43] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and

D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.

[44] M. Neseem, C. McCullough, R. Hsin, C. Leichner, S. Li, I. S. Chong, A. G. Howard, L. Lew, S. Reda, V.-M. Rautio, *et al.*, "Pikelpn: Mitigating overlooked inefficiencies of low-precision neural networks," *arXiv preprint arXiv:2404.00103*, 2024.

[45] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[46] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *International conference on machine learning*, pp. 7588–7598, PMLR, 2021.

[47] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," *arXiv preprint arXiv:2101.09336*, 2021.

[48] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–30, 2022.

[49] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[50] M. Shen, P. Molchanov, H. Yin, and J. M. Alvarez, "When to prune? a policy towards early structural pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12247–12256, 2022.

[51] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[52] S. Swaminathan, D. Garg, R. Kannan, and F. Andres, "Sparse low rank factorization for deep neural network compression," *Neurocomputing*, vol. 398, pp. 185–196, 2020.

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

[54] J. Ma, S. Hashemi, and S. Reda, "Approximate Logic Synthesis Using Boolean Matrix Factorization," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[55] S. Hashemi and S. Reda, "Generalized matrix factorization techniques for approximate logic synthesis," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1289–1292, IEEE, 2019.

[56] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via n-level recursive bisection," in *18th Workshop on Algorithm Engineering and Experiments*, pp. 53–67, 2016.

[57] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *24th International Workshop on Logic & Synthesis*, 2015.

[58] D. Bryan, "The iscas'85 benchmark circuits and netlist format," *North Carolina State University*, vol. 25, p. 39, 1985.

[59] S. Lee, L. K. John, and A. Gerstaluer, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *Design, Automation and Test in Europe*, 2017.

[60] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 27–32, IEEE, 2014.

[61] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda, "Approximate computing for biometric security systems: A case study on iris scanning," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 319–324, March 2018.

[62] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system*," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2017.

[63] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, pp. 788–791, 1999.

[64] W. Xu, X. Liu, and Y. Gong, "Document clustering based on non-negative matrix factorization," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pp. 267–273, 2003.

[65] P. Miettinen and J. Vreeken, "Model order selection for boolean matrix factorization," in *17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 51–59, 2011.

[66] P. Miettinen and J. Vreeken, "Mdl4bmf: Minimum description length for boolean matrix factorization," *ACM Transactions on Knowledge Discovery from Data*, vol. 8, no. 4, pp. 18:1–31, 2014.

[67] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila, "The discrete basis problem," *IEEE transactions on knowledge and data engineering*, vol. 20, no. 10, pp. 1348–1362, 2008.

[68] Z. Zhang, T. Li, C. Ding, and X. Zhang, "Binary matrix factorization with applications," in *7th IEEE International Conference on Data Mining*, pp. 391–400, IEEE, 2007.

[69] S. Ravanbakhsh, B. Póczos, and R. Greiner, "Boolean matrix factorization and

noisy completion via message passing.," in *International Conference on Machine Learning*, pp. 945–954, 2016.

[70] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, pp. 24–40, Springer, 2010.

[71] J. Ma, S. Hashemi, and S. Reda, "Approximate logic synthesis using blasys," 2019.

[72] C. Wolf, "Yosys open synthesis suite." `http://www.clifford.at/yosys/`, 2016.

[73] S. Williams, "Icarus verilog." `http://iverilog.icarus.com/`, 2006.

[74] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "Lsoracle: a logic synthesis framework driven by artificial intelligence," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2019.

[75] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[76] J. Castro-Godinez, H. Barrantes-Garcia, M. Shafique, and J. Henkel, "Axls: A framework for approximate logic synthesis based on netlist transformations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 8, pp. 2845–2849, 2021.

[77] S. Boroumand, C. S. Bouganis, and G. A. Constantinides, "Learning boolean circuits from examples for approximate logic synthesis," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 524–529, 2021.

[78] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2014.

[79] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified

design paradigm for approximate and quality configurable circuits," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1367–1372, IEEE, 2013.

[80] T. Alan, A. Gerstlauer, and J. Henkel, "Runtime accuracy-configurable approximate hardware synthesis using logic gating and relaxation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1578–1581, IEEE, 2020.

[81] T. Alan, A. Gerstlauer, and J. Henkel, "Cross-layer approximate hardware synthesis for runtime configurable accuracy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 6, pp. 1231–1243, 2021.

[82] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate applications," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[83] S. Reda and M. Shafique, *Approximate Circuits*. Springer, 2019.

[84] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 1–10, IEEE, 2016.

[85] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," *arXiv preprint arXiv:1812.08928*, 2018.

[86] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, IEEE, 2016.

[87] S. Laskaridis, A. Kouris, and N. D. Lane, "Adaptive inference through early-exit networks: Design, challenges and directions," in *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pp. 1–6, 2021.

[88] D. J. Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[89] L. Guerra, B. Zhuang, I. Reid, and T. Drummond, "Switchable precision neural networks," *arXiv preprint arXiv:2002.02815*, 2020.

[90] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[91] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

[92] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[93] E. Park, D. Kim, S. Kim, Y.-D. Kim, G. Kim, S. Yoon, and S. Yoo, "Big/little deep neural network for ultra low power inference," in *2015 international conference on hardware/software codesign and system synthesis (codes+ isss)*, pp. 124–132, IEEE, 2015.

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

[95] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute trends across three eras of machine learning," in *2022 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2022.

[96] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in

minutes," in *Proceedings of the 47th international conference on parallel processing*, pp. 1–10, 2018.

[97] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27730–27744, 2022.

[98] G. Hinton, "The forward-forward algorithm: Some preliminary investigations," *arXiv preprint arXiv:2212.13345*, 2022.

[99] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid, "A comprehensive survey on tinyml," *IEEE Access*, 2023.

[100] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*, pp. 38087–38099, PMLR, 2023.

[101] Y. Shang, Z. Yuan, B. Xie, B. Wu, and Y. Yan, "Post-training quantization on diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1972–1981, 2023.

[102] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et al.*, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

[103] S. Fox, S. Rasoulinezhad, J. Faraone, P. Leong, *et al.*, "A block minifloat representation for training deep neural networks," in *International Conference on Learning Representations*, 2020.

[104] K. Zhao, S. Huang, P. Pan, Y. Li, Y. Zhang, Z. Gu, and Y. Xu, "Distribution adaptive int8 quantization for training cnns," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 3483–3491, 2021.

[105] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*, pp. 291–326, Chapman and Hall/CRC, 2022.

[106] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*, pp. 1737–1746, PMLR, 2015.

[107] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer, "8-bit optimizers via block-wise quantization," in *International Conference on Learning Representations*, 2021.