POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Improving the Efficiency of Deep Learning Model Implementations Using Hardware Aware Design Techniques

MOBIN VAZIRI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2024

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Improving the Efficiency of Deep Learning Model Implementations Using Hardware Aware Design Techniques

présenté par **Mobin VAZIRI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

François-Raymond BOYER, président
Pierre LANGLOIS, membre et directeur de recherche
Shervin VAKILI, membre et codirecteur de recherche
François LEDUC-PRIMEAU, membre

DEDICATION

Dedicated to my family,
and to the one who gives meaning to infinity…

ACKNOWLEDGEMENTS

I am profoundly thankful to Prof. Pierre Langlois, my supervisor, for his unwavering support, guidance, and the vast knowledge he has generously imparted throughout my research. Prof. Langlois significant expertise in the field has been instrumental in developing my research work, guiding me in setting research objectives, developing solutions, and experimenting with new ideas. His encouragement and inspiration have been invaluable, and I am sincerely grateful for his mentorship.

My heartfelt thanks go to my co-supervisor, Prof. Shervin Vakili, whose unwavering dedication to excellence and meticulous attention to detail have significantly shaped the quality of my work. His guidance and constructive feedback have been crucial to my success, and I consider myself fortunate to have such a knowledgeable advisor. Prof. Vakili's contributions to my research have been immense, and I genuinely appreciate his support.

I sincerely thank NSERC for their financial support and the GR2M lab for providing the necessary processing resources. Their support was essential to the completion of this work.

Lastly, I want to express my sincere appreciation to my family for their selflessness, unwavering belief in me, and constant encouragement throughout my journey. I hope I have made them proud.

RÉSUMÉ

Avec la croissance rapide de la génération de données et la complexité croissante des tâches en apprentissage automatique, cryptographie et traitement en temps réel, le besoin de matériel performant et efficace augmente. Les CPU traditionnels, bien que polyvalents, peinent à répondre aux exigences de traitement parallèle, entraînant une consommation d'énergie élevée, une latence accrue et un débit réduit. Bien que les GPU gèrent mieux les grandes quantités de données et les algorithmes complexes, leur forte consommation énergétique et leur manque d'extensibilité limitent leur efficacité dans des environnements contraints comme le calcul en périphérie et les appareils IoT. Ainsi, la conception de circuits dédiés sur FPGA et ASIC s'avère essentielle pour atteindre performance, efficacité énergétique et adaptabilité.

Les FPGA, offrant flexibilité post-fabrication et parallélisme, équilibrent performance et adaptabilité, tandis que les ASIC, optimisés pour des tâches spécifiques, offrent une efficacité énergétique supérieure mais manquent de reconfigurabilité. La demande croissante pour l'IA en temps réel a encouragé des techniques de compression de modèles comme l'élagage et la quantification, qui réduisent les besoins en calcul et en mémoire des réseaux neuronaux, rendant ces modèles adaptés aux dispositifs limités. Par ailleurs, les réseaux de neurones artificiels (ANN) remplacent de plus en plus les méthodes de calcul traditionnelles dans des domaines comme la cryptographie et les systèmes de contrôle, offrant des alternatives plus efficaces pour des applications en temps réel, telles que la communication sécurisée.

Cette thèse présente trois contributions : *DyRecMul: a novel low-cost approximate multiplier for FPGAs*, pour une multiplication efficace dans les accélérateurs d'IA ; *Optimized Deep Learning Architectures for Efficient Automatic Modulation Recognition*, qui réduit la complexité des modèles de deep learning pour les systèmes de communication ; et *HENNC: Efficient FPGA Core Generation for ANN-Based Chaotic Oscillators Using High-Level Synthesis*, qui utilise des modèles ANN pour des applications cryptographiques sur du matériel économe en énergie. Ces contributions offrent des solutions innovantes pour des systèmes de calcul avancés sur des plateformes matérielles contraintes.

ABSTRACT

As data generation surges and computational tasks in areas like machine learning, cryptography, and real-time processing grow more complex, the need for hardware that balances performance and efficiency has increased. Traditional CPUs, though versatile, struggle with the parallel processing demands of these tasks, leading to higher energy consumption, increased latency, and reduced throughput. GPUs offer significant improvements in handling large datasets and complex algorithms, but their high energy use and scalability issues limit their effectiveness in resource-constrained environments, such as edge computing or IoT devices. To achieve real-time performance, enhanced energy efficiency, and adaptability, designing and implementing dedicated computational circuits on FPGAs and ASICs has proven effective and often essential. FPGAs combine post-fabrication flexibility with scalable parallelism, offering a balanced trade-off between performance and adaptability. In contrast, ASICs, while highly optimized for specific tasks, provide superior performance and energy efficiency but lack reconfigurability. Furthermore, the growing demand for real-time AI has also spurred the development of model compression techniques, such as pruning and quantization, which reduce the computational and memory requirements of neural networks, making them suitable for devices with limited processing power. Pruning removes unnecessary parameters, streamlining models without significantly affecting accuracy, while quantization reduces the precision of weights and activations to improve efficiency on hardware like FPGAs and ASICs. Additionally, Artificial Neural Networks (ANNs) are increasingly replacing traditional computational methods in fields such as cryptography and control systems, providing more efficient alternatives to resource-intensive numerical methods for simulating complex systems. This shift is particularly beneficial for real-time applications, such as secure communication, where ANN-based models offer faster, more efficient performance.

This thesis builds on these advancements in hardware efficiency and presents three key contributions: "DyRecMul: a novel low-cost approximate multiplier for FPGAs", which tackles the challenge of hardware-efficient multiplication for AI accelerators; "Optimized Deep Learning Architectures for Efficient Automatic Modulation Recognition", which applies pruning and quantization to reduce the complexity of deep learning models for real-time communication systems; and "HENNC: Efficient FPGA Core Generation for ANN-Based Chaotic Oscillators Using High-Level Synthesis", which replaces traditional numerical simulations with ANN-based models for cryptographic applications, enabling secure systems on low-cost, energy-efficient hardware. These contributions provide innovative solutions for deploying advanced computational systems on constrained hardware platforms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
AMR	Automatic Modulation Recognition
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CFGLUT5	Configurable Lookup Tables
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DL	Deep Learning
DSP	Digital Signal Processing
FPGA	Field-Programmable Gate Array
FLOP	Floating-Point Operation
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
LUT	Lookup Table
MAC	Multiply-Accumulate
ML	Machine Learning
NLP	Natural Language Processing
PRNG	Pseudo-Random Number Generator
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
RK	Runge-Kutta
SNR	Signal-to-Noise Ratio
TPU	Tensor Processing Unit

CHAPTER 1 INTRODUCTION

With the rapid evolution of Machine Learning (ML) and Deep Learning (DL) applications, the demand for specialized hardware that combines high performance with energy efficiency has surged in modern computing systems. An increasing range of application domains, including Computer Vision [1], Natural Language Processing (NLP) [2], Speech Processing [3], and Healthcare and Biomedical fields [4], are increasingly dependent on Artificial Neural Networks (ANNs). These networks often contain millions of parameters and require efficient, real-time processing of large datasets. While traditional processors like Central Processing Units (CPUs) are versatile, they often struggle with the high demands of ML and DL tasks, which benefit significantly from massive parallel computing to achieve high throughput and low latency.

CPUs are effective at sequentially handling diverse tasks but face limitations with the large-scale computations of ML and DL models. Graphics Processing Units (GPUs) have emerged as a popular choice for accelerating these computations, providing extensive parallelism that enables faster execution of core neural network operations, such as matrix multiplications and convolutions, outperforming CPUs in speed and efficiency for these workloads. While GPUs deliver high throughput, they can be power-inefficient when scaled for real-time edge applications [5]. To overcome these limitations, reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), has proven effective in achieving the enhanced efficiency necessary for real-time ML and DL inference.

FPGAs are particularly valued for their flexibility, as they can be reprogrammed after fabrication to implement various digital circuits. This reconfigurability makes FPGAs ideal for real-time applications where tasks may evolve or require custom optimization, such as in ML/DL inference and signal processing algorithms. By enabling the design of custom datapaths and control logic, FPGAs offer a unique balance between the high performance of dedicated hardware and the adaptability of software-defined solutions. For instance, in ML inference, an FPGA can be configured to efficiently handle specific neural network models or operations, often achieving lower latency and greater energy efficiency than GPUs. However, FPGAs have limitations compared to GPUs or ASICs, especially in terms of raw computational power. Additionally, designing optimized configurations for FPGAs can be complex and time-consuming, often requiring expertise in Hardware Description Languages (HDLs) like VHDL or Verilog. Despite these challenges, FPGAs are an effective solution when reprogrammability and high efficiency through customized computing circuits are prioritized.

ASICs, unlike FPGAs, are fixed-function circuits designed for specific tasks. Once manufactured, they cannot be reprogrammed; however, this fixed nature allows them to achieve the highest levels of performance and energy efficiency for their intended function. Properly designed ASIC hardware accelerators for ML/DL computing can often outperform CPUs, GPUs, and even FPGA-based accelerators in terms of speed and power consumption. This makes ASICs ideal for large-scale deployments of deep learning inference when the task is well-defined and remains stable over time. Examples of DL accelerators primarily designed for ASIC implementation include Google's Tensor Processing Units (TPUs) [6], NVIDIA's Deep Learning Accelerator (NVDLA) [7], MIT's Eyeriss [8], and Berkeley's Gemmini [9], all of which provide significant inference performance with enhanced energy efficiency.

In both FPGA and ASIC implementations, constraints in logic and memory resources, as well as limited energy budgets, restrict the maximum achievable performance. Consequently, these devices often face challenges in real-time computing for large ML and DL models, which can contain millions or even billions of parameters. To address this, model compression techniques have become essential, enabling these devices to handle large models more efficiently while maintaining high performance.

Model compression techniques are crucial for reducing the computational and memory demands of deep learning models, enabling their deployment on devices with limited processing power and energy resources. One of the most widely adopted model compression techniques is pruning, which reduces the size of a neural network by removing less important weights and connections. Neural networks often contain a large number of redundant parameters, especially in deep architectures where certain neurons or filters contribute little to the overall accuracy. Pruning techniques identify these redundancies and eliminate them, effectively shrinking the parameters and operations without significantly affecting its accuracy [10]. Pruning can be performed at different granularities, such as unstructured weight pruning, where individual weights are removed, or structured pruning, where entire filters, channels, or layers are pruned. Structured pruning is particularly advantageous for hardware deployment because it results in a regularized model that is easier to map onto hardware resources [11]. By reducing the number of active neurons or filters, pruning not only saves memory but also reduces the number of computations required during inference, making it especially valuable for real-time Artificial Intelligence (AI) applications where low latency and power efficiency are crucial.

Another important model compression technique is quantization, which reduces the precision of the weights and activations in a neural network. Instead of using single-precision 32-bit floating-point representations, quantization reduces these values to lower-precision formats, such as 8-bit integers. This reduction in precision significantly decreases the amount of memory needed to store the model and reduces the cost and latency of arithmetic circuits in DL hardware accelerators. There are two main quantization approaches: Post-Training Quantization, which applies quantization after the model is trained, and Quantization-Aware Training, where quantization is incorporated during the training process, allowing the model to adapt and maintain accuracy [12]. Modern hardware, such as GPUs and FPGAs, are optimized to perform low-precision computations more efficiently, further enhancing the performance gains from quantization. Quantized models have become the standard for deploying deep learning on mobile and embedded devices, where power and memory constraints are strict [13].

ANNs themselves can sometimes serve as a form of hardware optimization by replacing traditional computational methods. Thanks to the Universal Approximation Theorem [14], which states that neural networks can approximate any continuous function to a desired accuracy, ANNs can effectively substitute complex algorithms with simpler, more hardware-friendly models. This approach reduces the computational load and enables more efficient processing in certain applications. Beyond their primary use in machine learning, ANNs are increasingly being applied to replace traditional computational methods, particularly in solving complex differential equations. For example, chaotic systems, which are often modeled using differential equations solved by computationally intensive methods like the Runge-Kutta (RK) algorithm [15], can instead be approximated using ANNs [16]. ANNs can efficiently replicate the behavior of these systems. This substitution results in significantly reduced computational and hardware requirements compared to traditional numerical methods. As a result, ANNs offer a faster and more efficient alternative for solving differential equations in real-time applications, enabling more optimized hardware utilization without sacrificing accuracy.

The discussions above on hardware efficiency, model compression, and the replacement of traditional computational methods with ANNs establish the foundation for the contributions presented in this thesis. Specifically, this thesis introduces the following three contributions in the field of optimized machine learning computing:

- DyRecMul: a fast and low-cost approximate multiplier for FPGAs using dynamic reconfiguration, addresses the need for hardware-efficient multiplication in machine learning, where the need for rapid matrix multiplication is paramount. By leveraging approximate computing, this work proposes a new multiplier that significantly reduces the hardware cost compared to exact multipliers, offering a scalable solution for FPGA-based AI accelerators [17].

- The second contribution presents an accuracy-aware, low-complexity deep learning model designed for Automatic Modulation Recognition (AMR), a task of relevance in certain wireless communication scenarios where modulation classification is required in real-time. To reduce model complexity for real-time applications, pruning and quantization techniques are applied, enabling the deployment of highly accurate models on edge devices without the excessive power and memory demands of traditional architectures. [18].
- Finally, HENNC, a hardware engine for ANN-based chaotic oscillators, is introduced as a fast, computer-aided platform that automatically generates efficient hardware cores for these oscillators. HENNC facilitates the deployment of ANNs as replacements for traditional numerical methods for realizing chaotic systems, with a particular emphasis on cryptographic applications. This work demonstrates the feasibility of implementing secure communication systems on low-cost, energy-efficient hardware by replacing computationally intensive numerical methods with cost-effective ANN-based models [19].

The thesis is organized as follows:

- Chapter 2 introduces key concepts such as ANNs, chaotic systems, FPGAs, pruning, and quantization, serving as a foundation for the methods explored in the following chapters.
- Chapter 3 explains DyRecMul, an FPGA-based approximate multiplier optimized for machine learning, achieving significant hardware efficiency with minimal accuracy loss.
- Chapter 4 describes pruning and quantization techniques in DL models for AMR and introduces a new model, balancing accuracy and parameters.
- Chapter 5 covers HENNC, showing how ANNs replace traditional numerical methods for efficient cryptographic system simulation on hardware.
- Chapter 6 highlights the contributions to hardware efficiency and model optimization, promoting scalable and resource-efficient solutions in modern computing.
- Chapter 7 summarizes the results, emphasizing the impact of the proposed methods and suggesting future work directions.

CHAPTER 2 BACKGROUND

This chapter provides the essential background for this thesis. Section 2.1 covers neuron models and ANNs, followed by an overview of Deep Neural Networks in Section 2.2. Model compression techniques for hardware optimization are presented in Section 2.3, while Section 2.4 details FPGA architecture. Section 2.5 introduces chaotic systems and their applications in cryptography. The chapter concludes with a summary in Section 2.6, integrating these concepts as a foundation for the research.

2.1 Neuron Model and Artificial Neural Networks

ANNs are modeled after the concept of biological neurons described in [20]. A neuron consists of inputs, weights, a bias, and an activation function. The output y of a neuron is computed by summing the weighted inputs and adding a bias term, then passing the result through a non-linear activation function:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.1}$$

where n is the number of inputs to the neuron, x_i represents the input, w_i the weights, and b the bias term. The activation function f introduces non-linearity, enabling the network to learn complex patterns.

ANNs are foundational to modern deep learning and typically consist of three types of layers: input, hidden, and output. These layers enable the network to model complex data relationships by processing inputs through weighted sums, activation functions, and passing results to subsequent layers. Connectivity patterns vary by architecture, from fully connected to more specialized structures. The core operation driving this process is the Multiply-Accumulate (MAC) operation, which involves multiplying inputs by weights and summing the results. Each MAC counts as two Floating-Point Operations (FLOPs): one for multiplication and one for addition. For instance, in a fully connected NN with L layers, the total number of MACs can be computed as:

$$\#MACs = \sum_{i=2}^{L} n_i \times n_{i-1} \tag{2.2}$$

where n_i is the number of neurons in the i^{th} layer.

2.2 Deep Neural Networks

Deep Neural Networks (DNNs) are a subset of ANNs that incorporate multiple hidden layers, allowing the network to learn more abstract representations of data. While traditional ANNs typically consist of a few hidden layers, DNNs can have hundreds or even thousands of layers. This depth enables DNNs to learn hierarchical features, making them particularly effective in tasks like image recognition and NLP. Convolutional Neural Networks (CNNs) are a specialized type of DNN designed for tasks involving grid-like data structures, such as images. Unlike fully connected networks, CNNs take advantage of the spatial structure of data, significantly reducing the number of parameters by using local connections and shared weights [21]. The key operation in a CNN is the convolution, where a two-dimensional filter (or kernel) is applied to local regions of the input data. This operation produces a feature map that highlights specific patterns in the data. The convolution operation is given by:

$$Y[c_{out}, i, j] = \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{m=0}^{K_h-1} \sum_{n=0}^{K_w-1} X[c_{in}, i \cdot S_h + m - P_h, j \cdot S_w + n - P_w] \cdot W[c_{out}, c_{in}, m, n] \quad (2.3)$$

where X is the input feature map, W is the filter, Y is the output feature map, C_{in} and C_{out} are the number of input and output channels, K_h and K_w are the height and width of the filter, and S_h, S_w, P_h, and P_w are the strides and padding along height and width, respectively.

CNNs consist of several types of layers, including convolutional layers, pooling layers (which reduce the spatial dimensions of the feature maps), and fully connected layers, typically used at the final stages of classification tasks. In each convolutional layer, multiple filters are applied to the input data to extract a diverse set of features that capture different patterns, such as edges, textures, and shapes. These learned features are progressively abstracted as the network deepens. The number of MACs in a convolutional layer can be calculated as:

$$\#MACs = H_o \cdot W_o \cdot K_h \cdot K_w \cdot C_{in} \cdot C_{out}, \quad (2.4)$$

where H_o and W_o are output feature size and can be obtained as:

$$H_o = \left\lfloor \frac{\text{Input Height} - K_h + 2P_h}{s_h} \right\rfloor + 1, \quad W_o = \left\lfloor \frac{\text{Input Width} - K_w + 2P_w}{s_w} \right\rfloor + 1. \quad (2.5)$$

2.3 Model Compression Techniques

Model compression techniques reduce the computational and memory needs of DNNs without significantly affecting accuracy. Two of the most popular methods for achieving this are pruning and quantization, each focusing on different aspects of the network. This section discusses both methods and compares them.

2.3.1 Pruning

Pruning is a key model compression technique designed to reduce the number of parameters in DNNs while maintaining model accuracy. This is critical in deploying models on resource-constrained environments, where computational and storage efficiency are paramount. Pruning strategies can be broadly classified into unstructured, structured, and semi-structured methods.

Unstructured pruning involves removing individual weights from the neural network based on some importance criteria, typically targeting weights with low magnitudes [10]. The objective of unstructured pruning is to minimize the loss function $\mathcal{L}(w)$, subject to a sparsity constraint on the weights:

$$\min_{w} \mathcal{L}(w; D) = \min_{w} \frac{1}{N} \sum_{i=1}^{N} \ell(w; (x_i, y_i)), \tag{2.6}$$

$$\text{s.t.} \quad \|w\|_0 \leq k$$

where w represents the network weights, and $D = \{(x_i, y_i)\}_{i=1}^{N}$ is the dataset consisting of input data x_i and corresponding labels y_i for N samples. The pruning process removes weights with small magnitudes, thereby simplifying the model but typically requiring specialized hardware to exploit sparsity for speed improvements [22].

Structured pruning eliminates entire filters, channels, or even layers, resulting in a more compact and hardware-friendly network. Unlike unstructured pruning, it preserves the regular structure of the model. For a given prune ratio and a neural network with $P = \{p_1, p_2, \dots, p_L\}$, where p_i denotes the set of channels, filters, neurons, or attention heads in layer i, the goal of structured pruning is to find $P' = \{p'_1, p'_2, \dots, p'_L\}$, minimizing performance loss while enhancing speed under the defined prune ratio. Here, $p'_i \subseteq p_i$ for $i \in \{1, \dots, L\}$. Structured pruning facilitates hardware acceleration by removing entire elements, such as filters, channels, or layers, leading to a reduced model size and improved computational efficiency.

While structured pruning focuses on maintaining regularity in the network, semi-structured pruning introduces a balance between unstructured and structured pruning. It applies pre-defined sparsity patterns within layers, keeping some level of structure while still allowing for more fine-grained pruning than structured methods. This makes semi-structured pruning more adaptable to hardware acceleration while retaining the flexibility of unstructured pruning. One example of semi-structured pruning is the 2:4 (*2-out-of-4*) sparsity pattern, where two weights out of every four consecutive weights are pruned. This pattern allows semi-structured pruning to take advantage of specialized hardware accelerators like NVIDIA Ampere GPU's sparse tensor cores, which are optimized for specific sparsity patterns [23]. Semi-structured pruning offers the flexibility to achieve high pruning rates like unstructured methods but without the irregularity that can hinder practical speedups. It strikes a balance, making it effective for both accuracy preservation and computational efficiency.

2.3.2 Quantization

Quantization is a model compression technique that reduces the precision of the weights and activations in a neural network from full precision, typically 32-bit floating-point numbers, to lower precision formats such as 16-bit, 8-bit, or even 4-bit integers. By reducing the bit-width of the data used in DNNs, quantization significantly lowers memory usage and computational complexity, making DNNs more efficient for deployment on hardware-constrained applications such as mobile devices and edge computing platforms. This efficiency gain often comes with minimal impact on model accuracy when appropriately applied, making quantization a highly effective technique for real-time inference applications [12].

The process of uniform quantization can be described as mapping a real-valued vector x to a discrete grid of integers using the following transformation for each element:

$$x_{\text{int}} = \text{clip}\left(\left\lfloor \frac{x}{s} \right\rceil + z, L, U\right), \tag{2.7}$$

where s is the scaling factor, z is the zero point, and $\lfloor \cdot \rceil$ denotes the round-to-nearest-integer operator. The clip function ensures that the quantized values lie within the quantization range $[L, U]$, where L and U depend on whether the quantization is signed or unsigned [12].

Quantization methods can be categorized into two main approaches: post-training quantization (PTQ) and quantization-aware training (QAT).

PTQ is a method where a pre-trained model is quantized without further training. PTQ typically converts the weights and activations of a full-precision model into lower precision after the model has been fully trained. This method is fast and straightforward, requiring minimal computational resources as it avoids retraining. However, PTQ can lead to a notable drop in accuracy, especially for larger, more complex models, or when aggressive quantization (such as 4-bit) is applied. Despite these challenges, PTQ remains widely used since it does not require additional training [12].

QAT integrates quantization directly into the training process. During QAT, the model is trained with simulated low-precision operations, allowing it to adapt to the quantization noise introduced by the lower bit-width representations. QAT generally results in better performance and accuracy retention compared to PTQ because the model can learn to compensate for the reduced precision. The downside of QAT is that it requires additional training time and computational resources, which may not always be feasible in real-world applications, particularly for large-scale models. Nevertheless, QAT is preferred when high accuracy is crucial in low-precision deployments [24].

2.4 FPGA Architecture

The architecture of an FPGA consists of four primary components: Logic Elements (LEs), programmable interconnects, Input/Output (I/O) blocks, and embedded memory. These components work together to provide a flexible and reconfigurable platform for custom digital logic designs. AMD-Xilinx FPGAs utilize higher-level units called Configurable Logic Blocks (CLBs), each containing multiple LEs or slices. At the core of FPGA architecture, CLBs are configured to implement arbitrary logic circuits using programmable Lookup Tables (LUTs), flip-flops, multiplexers, and carry logic. In modern AMD-Xilinx FPGAs, each CLB is divided into slices, with each slice containing four 6-input LUTs and eight flip-flops. The 6-input LUTs can implement combinational logic functions or serve as distributed memory, providing versatility for both computation and localized storage needs.

To connect these CLBs and other on-chip resources, programmable interconnects are used, forming the essential pathways for signal routing. Configured through a switch matrix, these interconnects provide the flexibility to dynamically route signals based on the specific design, helping to minimize delays and ensuring smooth communication across the FPGA. This seamless routing architecture enhances the overall efficiency and adaptability of the device.

Supporting external communication, I/O blocks manage data transfer between the FPGA and external systems. These blocks are configurable to accommodate a wide variety of communication standards, making it possible for the FPGA to interface with numerous external components and systems.

Focusing on the latest AMD-Xilinx FPGA families [25], this architecture includes specialized resources like Digital Signal Processing (DSP) slices and Block RAM (BRAM), which complement the core elements. DSP slices are optimized for high-speed arithmetic operations, such as MAC, making them indispensable for tasks involving heavy computation. Meanwhile, BRAM provides on-chip memory, configurable as single-port or dual-port RAM, allowing high-throughput data access during complex operations. Enhancing flexibility further, modern FPGAs supports partial reconfiguration, allowing specific regions of the FPGA to be updated without interrupting the operation of the entire device. This feature is especially valuable in dynamic environments that demand continuous operation while adapting to changing conditions.

2.5 Chaotic Systems

Chaotic systems are deterministic systems that display highly unpredictable behavior due to their sensitivity to initial conditions, often called the "butterfly effect" [26]. These systems, governed by nonlinear differential equations, are deterministic but exhibit complex and seemingly random behavior over time.

One notable example of a chaotic system is the Lorenz system [26], described by the following set of three-dimensional differential equations:

$$\frac{dx}{dt} = \sigma(y - x), \quad \frac{dy}{dt} = x(\rho - z) - y, \quad \frac{dz}{dt} = xy - \beta z \tag{2.8}$$

where x, y, and z are system state variables, and σ, ρ, and β are system parameters. When $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, the system exhibits chaotic behavior, where small changes in initial conditions lead to vastly different trajectories.

Due to the complexity of chaotic systems, they are typically solved using numerical methods such as Euler's method or higher-order Runge-Kutta methods [16]. These techniques approximate the system's state over time by discretizing time into small steps and iteratively solving the equations, though this can become computationally expensive, particularly over long simulation times or in real-time applications. ANNs offer an efficient alternative by learning the system's dynamics directly, enabling future state predictions based on current inputs without the need for iterative equation solving. This reduces computational demands and allows for faster predictions, beneficial in scenarios requiring rapid or resource-efficient processing.

Chaotic systems are widely used in generating Pseudo Random Number Generators (PRNGs) [27]. The deterministic yet unpredictable nature of chaos makes them suitable for certain cryptographic applications and simulations. By evolving a chaotic system from a specific seed (initial condition), a sequence of numbers that mimic randomness can be generated. However, due to statistical weaknesses and predictability concerns, additional post-processing techniques are often required to make chaotic PRNGs suitable for secure encryption systems [28].

2.6 Chapter Summary

This chapter established the foundational concepts that drive the thesis. Starting with the principles of neural networks, we examined how ANNs and DNNs model complex data. Model compression techniques, such as pruning and quantization, were introduced to demonstrate efficient optimization for hardware constraints. FPGA architecture was then discussed as an ideal platform for deploying these optimized networks, thanks to its reconfigurable resources. Finally, chaotic systems were explored for their utility in secure computations. Together, these topics form a cohesive framework for efficient, hardware-accelerated neural network deployment in resource-limited applications, setting the stage for the work that follows.

CHAPTER 3 ARTICLE 1: DYRECMUL: FAST AND LOW-COST APPROXIMATE MULTIPLIER FOR FPGAS USING DYNAMIC RECONFIGURATION

Authors: Mobin Vaziri, Shervin Vakili, Amirhossein Zarei, and J.M. Pierre Langlois.

Published in: ACM Transactions on Reconfigurable Technology and Systems (2024).

Submitted: 21 December 2023 **Published:** 02 May 2024

Abstract

Multipliers are widely-used arithmetic operators in digital signal processing and machine learning circuits. Due to their relatively high complexity, they can have high latency and be a significant s ource o f p ower c onsumption. O ne s trategy t o a lleviate t hese limitations is to use approximate computing. This paper thus introduces an original FPGA-based approximate multiplier specifically o ptimized f or m achine l earning c omputations. I t utilizes dynamically reconfigurable L UT p rimitives i n A MD-Xilinx t echnology t o r ealize t he core part of the computations. The paper provides an in-depth analysis of the hardware architecture, implementation outcomes, and accuracy evaluations of the multiplier proposed in INT8 precision. The paper also facilitates the generalization of the proposed approximate multiplier idea to other datatypes, providing analysis and estimations for hardware cost and accuracy as a function of multiplier parameters. Implementation results on an AMD-Xilinx Kintex Ultrascale+ FPGA demonstrate remarkable savings of 64% and 67% in LUT utilization for signed multiplication and multiply-and-accumulation configurations, respectively when compared to the standard Xilinx multiplier core. Accuracy measurements on four popular deep learning benchmarks indicate a minimal average accuracy decrease of less than 0.29% during post-training deployment, with the maximum reduction staying less than 0.33%. The source code of this work is available on GitHub[1].

[1]`https://github.com/INRS-ECCoLe/DyRecMul`

3.1 Introduction

Computational circuits may become a bottleneck in wireless communications, digital signal processing, multimedia, image processing, machine learning, etc., due to their high energy consumption, long delay, and large circuit area utilization. Applications such as neural networks have inherent error resilience, which presents an opportunity to use approximation techniques for enhancing their efficiency [29, 30]. Multiplication is a fundamental operation in computations, and, to enhance its efficiency, various approximate multipliers have been proposed. These multipliers aim to reduce delay, energy consumption, and area [31]. In 1962, Mitchell [32] introduced the first logarithm-based multiplier. This multiplier converted multiplications into addition operations, and it deployed approximation methods to calculate logarithmic expressions. Since then, there have been significant advancements in logarithm-based approximate multipliers [33–36]. Liu et al. used approximate adders to improve accuracy and reduce power consumption compared to conventional logarithm-based multipliers [33]. Given that two-variable multiplication is a nonlinear operation, several works proposed methods to use linearization for approximate multiplication [37,38]. An approximate linearization algorithm, called ApproxLP [37], utilized comparators to separate different sub-domains and allocate a proper linear function to each one instead of a nonlinear multiplication. This method, however, requires additional comparators to increase accuracy and reduce sub-domain size, which can result in longer delays and increased area [31]. Chen et al. introduced an Optimally Approximate Multiplier (OAM) [38] to improve linearization and minimize the number of comparators, resulting in lower delay and improved performance. Other studies suggest utilizing a combination of accurate and approximate multipliers with varying precision levels to achieve the necessary accuracy, but this may come at the cost of higher energy consumption and circuit area [39, 40].

In non-logarithmic multiplication, the approximation can be introduced at four distinct stages: data input, partial product generation, accumulation, and Booth encoding [31]. Truncating input data is often used to reduce bitwidth, and it can be done through methods such as the Dynamic Segment Method (DSM) and Static Segment Method (SSM) [41–44]. DSM keeps a specified number of bits from the most significant '1' in an n-bit operand but requires more hardware resources than SSM. SSM has limited and pre-determined options for truncating input data, resulting in more redundant bits [31,41]. Some works have introduced approximations in the partial product generation stage [45–48]. However, the most beneficial stage to introduce approximation is during partial product accumulation, which typically requires the largest circuit area and computation time.

Compressors can be used for counting ones in tree-based partial product accumulations [31], such as the Wallace tree [49] and Dadda tree [50]. Instead of exact compressors, a variety of approximate compressors have been introduced to decrease circuit area and delay [46, 47, 51–62]. These include the approximate 1-bit half adder, approximate 1-bit full adder, and approximate 4-2 compressors [54, 55, 58, 60]. To further decrease energy consumption, delay, and hardware utilization, high-order approximate compressors with more than five inputs have been proposed [51, 53, 59, 61]. Mahdiani et al. proposed a technique that removes the least significant partial products from the partial product matrix to decrease circuit area and delay, although it comes with some degree of error which can be adjusted by modifying the truncation level [56].

The existing approximate methods, which have been mainly designed to reduce energy consumption and area utilization in ASIC implementations, may not be as effective in FPGAs. This is because FPGA reconfigurable logic fabrics are typically based on fixed-size LUTs. Modern AMD-Xilinx and Intel FPGAs have hardwired DSP multipliers that offer faster and more energy-efficient multiplication than soft implementation on general-purpose LUT fabrics. The DSP-based multipliers are valuable resources that are situated in specific locations, which can result in long routing delays. On the other hand, LUTs are spread out across the chip, making them more easily routable. Moreover, there is a limitation on the number of DSP-based multipliers available. One solution is to deploy approximation methods to enhance the efficiency of LUT-based multiplication, in terms of speed, energy/power consumption, and hardware utilization. Ullah et al. [63, 64] implemented an optimization technique by truncating the least significant partial product of a 4×2 multiplier to reduce LUT utilization. Van Toan et al. [65] designed compact 3-2 and 4-2 compressors for use in different approximate multipliers with varying levels of accuracy. Kumm et al. [66] proposed dynamically reconfigurable FIR filters in Xilinx FPGAs using Configurable Look-up Tables (CFGLUT5s).

This paper introduces DyRecMul, a cost-effective dynamically reconfigurable approximate multiplier for FPGAs. It is optimized for machine learning computation, has a short critical path, and uses a small number of LUTs. DyRecMul utilizes AMD-Xilinx technology's reconfigurable LUT primitives to approximate multiplication without significant accuracy degradation, even in post-training inference. To preserve dynamic range, DyRecMul utilizes a cost-effective encoder that transforms a fixed-point operand into a floating-point format, and a decoder to revert the result back to fixed-point. The paper presents the design details and evaluation results of an INT8 version of DyRecMul since INT8 is a popular datatype in cost-effective machine learning computing. The multiplier introduces negligible accuracy loss while reducing significantly the number of required LUTs in FPGAs compared to the standard exact multiplier. Additionally, DyRecMul boasts very low latency, which allows for a faster clock frequency than that of typical AMD-Xilinx multiplier cores. The key contributions of this paper are summarized below.

- Utilization of dynamically reconfigurable LUTs in AMD-Xilinx FPGAs to make a low-cost and fast short-bitwidth multiplier.
- Internal conversion of fixed-point to a floating-point format to preserve dynamic range that is beneficial for machine learning and deep learning applications.
- For an INT8 case study, illustration of the design detail of a highly optimized encoder circuit to convert INT8 to 8-bit floating-point (FP8) format and a low-cost decoder to convert FP8 to INT8.

3.2 Analytical Description

A precise N-bit signed integer multiplication of two operands, X and W, can be represented by:

$$Z = s_z.\sum_{i=0}^{2N-2} z_i.2^i = \; s_x.\sum_{i=0}^{N-2} x_i.2^i \;\times\; s_w.\sum_{i=0}^{N-2} w_i.2^i \tag{3.1}$$

where x_i, w_i and z_i denote the i^{th} bit of operand X and W and the result Z, respectively. s_x takes the value of -1 when X is negative, and 1 otherwise. The same logic applies to s_z and s_y. In INT8 representation, $N = 8$. When using multipliers in a computing system that only supports a single datatype, the output Z must be expressed in the same format as the input operands. This can be done using techniques like truncation.

The proposed approach utilizes an encoder to convert the first operand, X, to a floating-point representation, $\hat{X}_{float}$, of format:

$$\begin{aligned} &float\left(sign_{BW}, exp_{BW}, mnt_{BW}\right), \\ &\text{sign}_{BW} \in \{0,1\},\ exp_{BW} > 0,\ mnt_{BW} < N-1 \end{aligned} \tag{3.2}$$

where $sign_{BW}$, exp_{BW}, and mnt_{BW} denote the bitwidth of the sign, exponent, and mantissa elements, respectively. In this section, centered on signed multiplication, a single bit is designated to represent the sign, i.e., $sign_{BW} = 1$. To enable covering the entire dynamic range of X, exp_{BW} and mnt_{BW} must fulfill the following:

$$2^{exp_{BW}} + mnt_{BW} > N - 1. \tag{3.3}$$

The mantissa is a segment of mnt_{BW} bits from $|X|$, with the leftmost bit in $|X|$ that contains '1' being the most significant bit. Since the mantissa is shorter than the magnitude bits of X, this conversion involves an approximation. The conversion can be expressed as:

$$\hat{X}_{float} = \left[\hat{X}_{sign},\ \hat{X}_{exp}\ ,\ \hat{X}_{mnt}\ \right] \tag{3.4}$$

$$\begin{cases} \hat{X}_{sign} = x_{N-1}, \\ \hat{X}_{exp} = N - i\ where \begin{cases} 2^i < |X| \leq 2^{i-1},\ X \geq 2^{mnt_{BW}} \\ 0,\ X < 2^{mnt_{BW}}, \end{cases} \\ \hat{X}_{mnt} = Round\ (\frac{|X|}{2^{exp}}) \end{cases}$$

The mantissa $\hat{X}_{mnt}$ is then multiplied by $|W|$ to generate the mantissa of the result. To keep the result in the format of Eq. 3.2, the product is quantized to mnt_{BW} bits using:

$$\hat{Z}_{mnt} = Q\left(\hat{X}_{mnt} \times |W|\,,\ mnt_{BW}\right), \tag{3.5}$$

where Q is the quantization function. The result must be converted back to integer format. For this purpose, the $(N-1)$-bit absolute value of Z is first calculated by:

$$|Z| = 2^{\hat{X}_{exp}} \times \hat{Z}_{mnt} \tag{3.6}$$

The result, Z, in integer format is obtained by applying a two's complement function when Z is negative. The sign of Z is determined by XORing the sign bits of X and W.

$$Z = s_z.\,|Z|\,, \begin{cases} s_z = +1,\ x_{N-1} \bigoplus w_{N-1} = 0 \\ s_z = -1,\ x_{N-1} \bigoplus w_{N-1} = 1 \end{cases} \tag{3.7}$$

3.3 DyRecMul for INT8 Multiplication: Architecture and Components

INT8 quantization is a supported feature in machine learning frameworks such as TensorFlow Lite and PyTorch, as well as hardware toolchains like AMD-Xilinx DNNDK. Moreover, INT8 is a popular datatype in machine learning hardware accelerators and ML-optimized GPUs designed for embedded and edge applications [67], [68]. The INT8 DyRecMul described in this section can be used in place of INT8 multipliers for pre- or post-training inference. This multiplier is intended primarily for single-datatype INT8 architectures, meaning that it calcu-

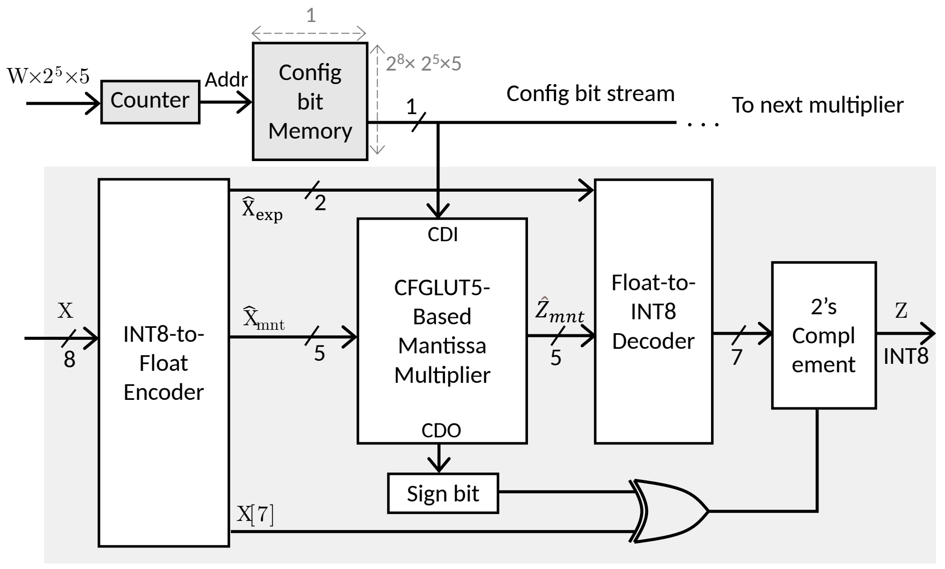
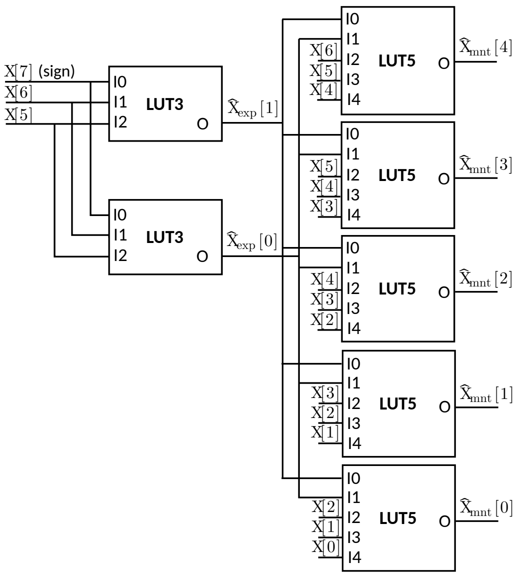


Figure 3.1 DyRecMul architecture for INT8 multiplication of $Z = X \times W$, using $float\,(1, 2, 5)$ internal floating-point format.

lates $Z = X \times W$, where X, W and Z are all INT8. Fig. 3.1 depicts its architecture, which consists of four main components: (1) a cost-effective INT8 to floating-point encoder; (2) an ultra-low-cost mantissa multiplier using dynamically reconfigurable LUTs; (3) a floating-point to INT8 decoder; (4) a two's complement logic. The following subsections describe each component in detail.

3.3.1 Integer to Floating-Point Encoder

The proposed architecture converts its first operand, X, from INT8 to $float\,(1, 2, 5)$ representation, where 1, 2, and 5 indicate the number of allocated bits to the sign, exponent, and mantissa, respectively. This conversion corresponds to Eq. 3.4 in Section 3.2. The mantissas are multiplied and the exponents are added, and this conversion limits the binary multiplication to five bits while maintaining the dynamic range, which is crucial for accurate DL computations. As will be discussed later, DyRecMul deploys a CGFLUT5-based unsigned multiplier in which one operand must be five bits wide to achieve optimal efficiency. When this multiplier is used in a weight stationary DL accelerator, input feature maps are fed as the first operand, X, to the multipliers. Without a floating-point conversion, the feature maps would need to be directly quantized to five bits. That would significantly limit the supported dynamic range, which could cause important accuracy loss for small activation values and ultimately lead to a prohibitive level of inaccuracy in ML inference. More precisely, such a quantization would limit the range of supported values to $[-2^5, 2^5 - 1]$, while $float\,(1, 2, 5)$



X	$\hat{X}_{exp}$
[7:5]	[1:0]
000	00
001	01
01x	10
10x	10
101	01
111	00

$\hat{X}_{exp}$	X	$\hat{X}_{mnt}$
[1:0]	[6:4]	[4]
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{X}_{exp}$	X	$\hat{X}_{mnt}$
[1:0]	[5:3]	[3]
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{X}_{exp}$	X	$\hat{X}_{mnt}$
[1:0]	[4:2]	[2]
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{X}_{exp}$	X	$\hat{X}_{mnt}$
[1:0]	[3:1]	[1]
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

$\hat{X}_{exp}$	X	$\hat{X}_{mnt}$
[1:0]	[2:0]	[0]
00	xx0	0
00	xx1	1
01	x0x	0
01	x1x	1
1x	0xx	0
1x	1xx	1

Figure 3.2 INT8 to $float\,(1,2,5)$ encoder: LUT mapping and corresponding truth tables.

expands the range to $[-2^3 \times 2^5, 2^3 \times 2^5 - 1]$ and demands the same unsigned multiplier size for mantissa multiplication. The experimental results in Section 3.7.3 demonstrate that this conversion greatly helps in maintaining the precision of DL calculations.

The conversion logic from INT8 to $float\,(1,2,5)$ is designed to be low-cost and efficient, requiring only seven LUT5 elements. Fig. 3.2 depicts the truth tables and the corresponding LUT5 allocations for this encoder. The 2-bit exponent is acquired from the sign bit and the two most significant bits, while each mantissa bit is obtained from the exponent and three corresponding input bits. The encoder's critical path consists of two LUT5 units and their corresponding routing circuits.

3.3.2 Dynamically Reconfigurable Mantissa Multiplier

This component implements mantissa multiplication according to Eq. 3.5. The mantissa multiplier which serves as the core component of DyRecMul, is a cost-effective unsigned integer multiplier based on AMD-Xilinx CFGLUT5 primitives. CFGLUT5 offers a distinctive capability allowing the logical function of the LUT5 to be altered during circuit operation. This reconfiguration is achieved by programming new configuration bits into the LUT5, enabling it to adopt a new logic function with a maximum of 5-bit inputs and one output. The design subsystem controls the programming of new configuration bits, and this reconfiguration process does not necessitate external partial or complete reprogramming of the FPGA, ensuring uninterrupted operation. The programming of new configurations is carried out serially through a single-bit CDI input port, while CDI/CDO ports facilitate the cascading of multiple CFGLUT5 units in a chain.

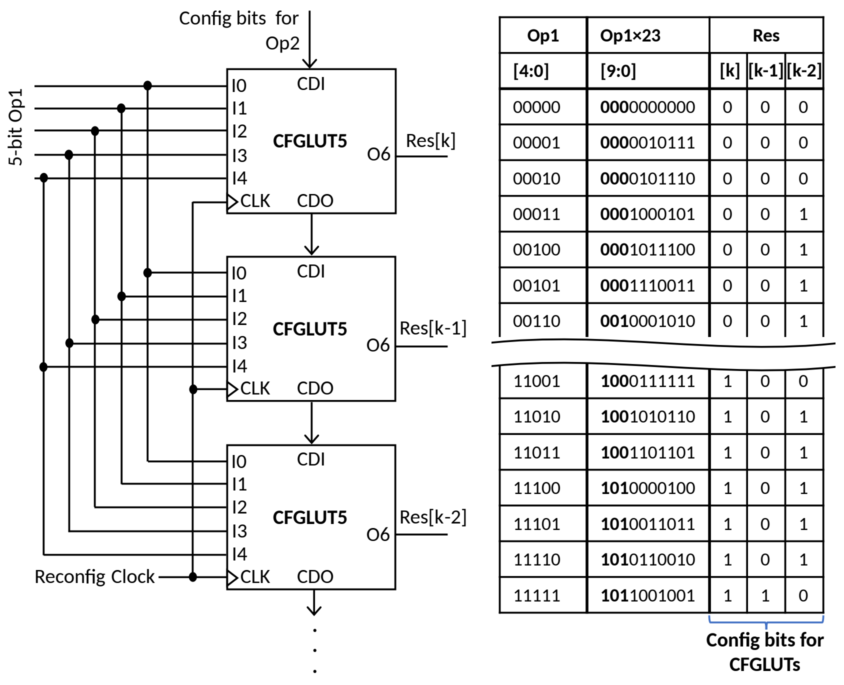
In the mantissa multiplier, the logic circuit for multiplication with the second operand is programmed into CFGLUT5 units. Whenever the second operand, W, undergoes changes, the CFGLUT5 units within the mantissa multiplier are reconfigured to establish a constant multiplication logic with the updated value. The first operand is fed into the 5-bit input port of CFGLUT5s. Hardcoding the multiplication logic for the second operand directly into the LUTs, instead of employing a traditional double-operand multiplier, significantly reduces the hardware cost of the multiplication circuit.

Fig. 3.3 illustrates the LUT mapping and configuration bits for an example 5-bit unsigned multiplier. The output consists of the quantized k most significant bits of the result. This means that only k CFGLUT5s are needed, each generating one output bit. In this example, a value of 23 is assumed for the second operand, W, configuring the CFGLUT5s to realize the logic circuit of a constant multiplier calculating the product of a 5-bit quantized first operand, and 23. Fig. 3.3 depicts the process of calculating the configuration bits for three CFGLUT5s which produce the three most significant bits of the product. Rounding is used in these calculations to minimize quantization error. The configuration bits are loaded into the CFGLUT5s serially through a cascaded CDI and CDO chain. As the bitwidth of the first operand surpasses five, the number of CFGLUT5s required for each output bit grows exponentially. More precisely, the required number of CFGLUT5s is:

$$\#CFGLUT5 = 2^{b_1-5} \times k \tag{3.8}$$

where b_1 and k denote the bitwidth of the first operand, X, and the result, Z, respectively.

As an example, implementing an exact 8×8 unsigned multiplier requires $2^3 \times 16 = 128$ CFGLUT5s. Although the LUT utilization may appear to be high, a more serious obstacle lies in the significant number of reconfiguration bits - a total of 4096 (128×2^5). In order to fully exploit the CFGLUT5s while minimizing their quantity, we limited the first operand, to five bits, resulting in one CFGLUT5 utilization per result bit. Thus, using k parallel CFGLUT5s allows for calculating the k-bit result of a 5-bit multiplication. If the bitwidth is shorter, some CFGLUT5s may remain partially unused, while increasing it to over five bits will exponentially increase the number of CFGLUT5s. Additionally, for the INT8 DyRecMul, we set the result bitwidth k to five, restricting the number of CFGLUT5s to only five units and the number of reconfiguration bits to 160. When this multiplier is used in a weight stationary DL accelerator, weights are translated into the configuration bits in CFGLUT5s, and input feature maps are fed as operands to the multipliers. As there is no interdependence



Op1	Op1×23	Res		
[4:0]	[9:0]	[k]	[k-1]	[k-2]
00000	**0000**000000	0	0	0
00001	**0000**010111	0	0	0
00010	**0000**101110	0	0	0
00011	**0001**000101	0	0	1
00100	**0001**011100	0	0	1
00101	**0001**110011	0	0	1
00110	**0010**001010	0	0	1
11001	**1000**111111	1	0	0
11010	**1001**010110	1	0	1
11011	**1001**101101	1	0	1
11100	**1010**000100	1	0	1
11101	**1010**011011	1	0	1
11110	**1010**110010	1	0	1
11111	**1011**001001	1	1	0

Figure 3.3 5-bit dynamically reconfigurable multiplier with 3-bit result using CFGLUT5 primitives. The second operand, $Op2 = 23$ in this example.

between CFGLUT5 elements, they can operate entirely in parallel, resulting in a short critical path of just one CFGLUT5.

The configuration bits for CFGLUT5s are stored in a shared BRAM-based memory, as illustrated in Fig. 3.1. This memory functions like a read-only memory, storing all configuration bits for every possible 2^N value of the second operand. A counter generates the read address for the configuration memory during reconfiguration. When the second operand, W, changes, this counter is initialized to the address of the first configuration bit corresponding to the new W. The counter then increments with each clock cycle, and during each cycle, it programs one bit into the chain of CFGLUT5s inside the mantissa multiplier through CDI/CDO pins. In the case of the INT8 DyRecMul, this process continues for 160 clock cycles to write all 160 new configuration bits.

When multiple DyRecMuls are utilized in a design, the reconfiguration BRAM can be dual-ported to facilitate the parallel reprogramming of two DyRecMuls. Furthermore, a single reconfiguration memory can be employed for the sequential reprogramming of more than two DyRecMuls, albeit at the expense of a longer reconfiguration time. The reconfiguration time is an important factor in the efficiency of DyRecMul. As previously mentioned, DyRecMul is most suitable for applications where one of the operands does not change frequently. In other words, DyRecMac gains an edge when one of the operands remains unchanged significantly longer than the reconfiguration time. The influence of reconfiguration time on the overall processing time primarily relies on the dataflow and architecture of the DL accelerator. Section 3.6 will review some of the main target applications that can effectively take advantage of DyRecMul.

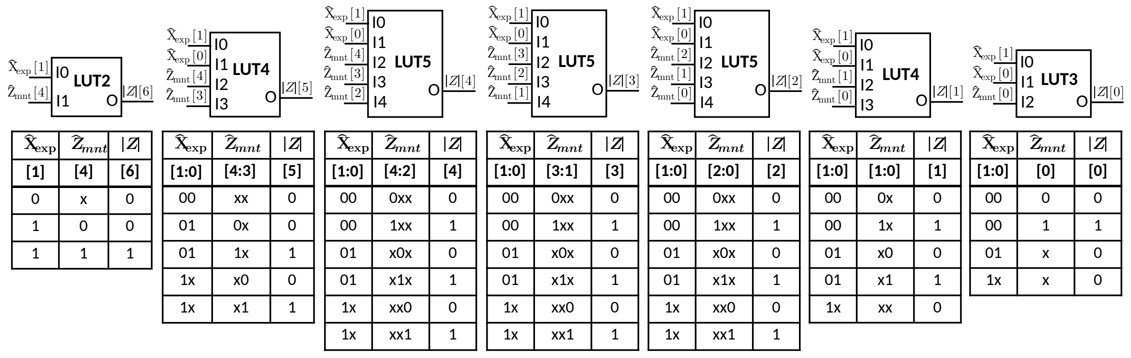
3.3.3 Floating-point to Integer Decoder

In deep learning computing, multiplication operations are often followed by addition or subtraction. However, a significant drawback of floating-point representations is the high hardware cost of their adder/subtractor circuits, which limits overall efficiency. To address this issue and enable subsequent addition or subtraction in integer format, DyRecMul converts the result back to integer form using decoder and two's complement components.

The decoder converts the result of mantissa multiplication along with the exponent of the first operand into a 7-bit unsigned format. The decoder needs seven LUT5 units, with each unit producing one output bit using two exponent bits and a maximum of three mantissa bits. The truth table of each output bit is shown in Fig. 3.4. Since all LUT5 units function in parallel, the critical path is only one LUT5.

3.3.4 Two's Complement Logic

The output of the decoder is the absolute value of the result. As described in Section 3.2, the sign of the product can be obtained by exclusive OR operation between the sign bit of the two operands. If the product is negative, a two's complement function must be applied. Two's complement logic can be realized using eight LUT5 units. The carry signal is expected to be a major source of latency in this multiplier. To create an unsigned version of DyRecMul, we only need to remove the two's complement stage, modify the floating-point format to


$\hat{X}_{\text{exp}}[1]$
$\hat{Z}_{\text{mnt}}[4]$
I0
I1
LUT2
O
$|Z|[6]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[4]$
$\hat{Z}_{\text{mnt}}[3]$
I0
I1
I2
I3
LUT4
O
$|Z|[5]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[4]$
$\hat{Z}_{\text{mnt}}[3]$
$\hat{Z}_{\text{mnt}}[2]$
I0
I1
I2
I3
I4
LUT5
O
$|Z|[4]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[3]$
$\hat{Z}_{\text{mnt}}[2]$
$\hat{Z}_{\text{mnt}}[1]$
I0
I1
I2
I3
I4
LUT5
O
$|Z|[3]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[2]$
$\hat{Z}_{\text{mnt}}[1]$
$\hat{Z}_{\text{mnt}}[0]$
I0
I1
I2
I3
I4
LUT5
O
$|Z|[2]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[1]$
$\hat{Z}_{\text{mnt}}[0]$
I0
I1
I2
I3
LUT4
O
$|Z|[1]$
$\hat{X}_{\text{exp}}[1]$
$\hat{X}_{\text{exp}}[0]$
$\hat{Z}_{\text{mnt}}[0]$
I0
I1
I2
LUT3
O
$|Z|[0]$

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1]	**[4]**	**[6]**
0	x	0
1	0	0
1	1	1

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[4:3]**	**[5]**
00	xx	0
01	0x	0
01	1x	1
1x	x0	0
1x	x1	1

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[4:2]**	**[4]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[3:1]**	**[3]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[2:0]**	**[2]**
00	0xx	0
00	1xx	1
01	x0x	0
01	x1x	1
1x	xx0	0
1x	xx1	1

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[1:0]**	**[1]**
00	0x	0
00	1x	1
01	x0	0
01	x1	1
1x	xx	0

$\hat{X}_{\text{exp}}$	$\hat{Z}_{mnt}$	$\|Z\|$
[1:0]	**[0]**	**[0]**
00	0	0
00	1	1
01	x	0
1x	x	0

Figure 3.4 $float\,(1,2,5)$ to INT8 decoder: LUT mapping and corresponding truth tables.

$float\,(0,2,5)$, and make slight adjustments to the encoder and decoder stages. The results in section 3.7 indicate that the unsigned version is more efficient than the signed version. This is because the two's complement stage, which consumes a significant amount of hardware resources, is present only in the signed DyRecMul.

3.4 Generalized DyRecMul

Section 3.3 detailed the design of an INT8 multiplier with internal $float\,(1,2,5)$ conversion. A crucial question arises regarding the effectiveness of DyRecMul with other data types and floating-point formats. The cost and accuracy of a DyRecMul multiplier depend on four key parameters: mnt_{BW}, the mantissa bitwidth in the floating-point encoding of *Op1*; exp_{BW}, the exponent bitwidth; $m\hat{X}_{exp} = max\left(\hat{X}_{exp}\right)+1$, the maximum value that the exponent may attain plus one; and $\hat{Z}_{BW}$, the number of mantissa product bits generated by the mantissa multiplier unit. Altering these parameters results in different trade-offs between hardware cost and accuracy. The main task in designing a DyRecMul multiplier entails selecting suitable values for these parameters. This section provides an estimation of the hardware cost and accuracy of an $N \times N$ DyRecMul as a function of these four parameters. These estimation functions provide insight into the overall efficiency of the DyRecMul method and facilitate the process of designing DyRecMul multipliers for various data types.

3.4.1 Hardware Cost Estimation

In this section, we provide a conservative estimation of the hardware cost for three crucial components of unsigned DyRecMul—namely, the encoder, mantissa multiplier, and decoder—measured in terms of the number of utilized LUT5 units. To establish a uniform metric, we estimate the number of LUT5s, striving to map all logic onto LUT5 resources, excluding other elements such as F7, F8, and F9 MUX primitives in AMD-Xilinx FPGA technology [69]. Since each LUT6 in modern AMD-Xilinx technology consists of two LUT5 units, the estimations provided in this section can be converted to LUT6 utilization by dividing the estimated number of LUT5s by two. Moreover, the estimations can be readily extended to the signed DyRecMul by applying minor modifications and incorporating the LUT utilization for the two's complement component.

Integer to Floating-Point Encoder

As illustrated in Fig. 3.2 for the INT8 case, the integer to floating-point encoder comprises two components: (1) an *exponent identifier* circuit, and (2) *mantissa extractor*.

The *exponent identifier* examines the $m\hat{X}_{exp}$ most significant bits to determine the exp_{BW} bits of the exponent. This is realized by a circuit that bears similarities to a $m\hat{X}_{exp}$-to-exp_{BW} priority encoder. As long as $m\hat{X}_{exp} \leq 5$, each output bit can be generated by a single LUT5. However, when $m\hat{X}_{exp}$ exceeds 5, a conservative and simplified estimation approach involves dividing the exponent identifier into two steps. The first step employs a logic circuit to convert the $m\hat{X}_{exp} - 1$ most significant bits, retaining only the most significant '1' bit and masking other less significant bits to zero. This step requires almost $m\hat{X}_{exp} - 1$ LUTs. The second step resembles a binary decoder with exp_{BW} output bits. In decoders, each output bit typically depends on a maximum of $2^{exp_{BW}}/2$ input bits. One LUT5 is adequate to process a maximum of five input bits in this stage. Since exp_{BW} is not supposed to be too large in DyRecMul, we can estimate that $\lceil 2^{exp_{\mathrm{BW}}}/10 \rceil$ LUT5s are required for each output bit. In summary, the estimated total number of LUT5s for the exponent identifier can be expressed as:

$$exp_{LUT5} = \begin{cases} exp_{BW}, & m\hat{X}_{exp} \leq 5, \\ \left\lceil \frac{2^{exp_{BW}}}{10} \right\rceil \times exp_{BW} + m\hat{X}_{exp} - 1, & m\hat{X}_{exp} > 5. \end{cases} \tag{3.9}$$

Recalling that $m\hat{X}_{exp} = max\left(\hat{X}_{exp}\right) + 1$, the exponent bitwidth, exp_{BW}, is directly derived from $m\hat{X}_{exp}$ as follows:

$$exp_{BW} = \left\lceil \log_2\left(m\hat{X}_{exp}\right) \right\rceil . \tag{3.10}$$

The *mantissa extractor* functions as a multiplexer, with the exponent acting as its selector and $m\hat{X}_{exp}$ bits of the first operand as its input. Each bit of the mantissa is derived from the exp_{BW}-bit exponent and $m\hat{X}_{exp}$ bits of the input, X. As long as $m\hat{X}_{exp} + exp_{BW} \leq 5$, each output bit can be generated by one LUT5. However, if $m\hat{X}_{exp} + exp_{BW}$ exceeds 5, a multilayer tree of LUTs implements this multiplexer. Every 4×1 multiplexer can be implemented within a single LUT6 or two LUT5. Thus, we conservatively estimate the number of LUT5 in the first layer as $\left\lceil m\hat{X}_{exp}/4 \right\rceil \times 2$. Every four LUTs from the first layer, along with two exponent bits, feed one LUT6 in the second layer. Therefore, we estimate the number of LUT5s in the second layer as $\left\lceil m\hat{X}_{exp}/16 \right\rceil \times 2$. A third layer will be needed only when $m\hat{X}_{exp}$ exceeds 16 and will need more than one LUT when it exceeds 64, which is too large to consider. Hence, for simplicity, we discard the third layer here. Overall, the obtained estimation for LUT5 utilization in the mantissa extractor will be as follows:

$$mnt_{LUT5} = \begin{cases} mnt_{BW}, & m\hat{X}_{exp} < 4, \\ mnt_{BW} \times 2, & m\hat{X}_{exp} = 4, \\ \left(\left\lceil \frac{m\hat{X}_{exp}}{4} \right\rceil + \left\lceil \frac{m\hat{X}_{exp}}{16} \right\rceil \right) \times 2 \times mnt_{BW}, & m\hat{X}_{exp} > 4. \end{cases} \tag{3.11}$$

The total estimated number of LUT5 units required for integer to floating-point encoding is:

$$encoder_{LUT5} = exp_{LUT5} + mnt_{LUT5} \tag{3.12}$$

Floating-Point to Integer Decoder

The decoder converts the product result to integer format. Similar to the mantissa extractor in the encoder, the decoder operates as a multiplexer, with each output bit chosen from one of the $m\hat{X}_{exp}$ mantissa product bits, $\hat{Z}_{exp}$. Consequently, each output bit of the decoder requires a multiplexer, employing the exponent as the selector to choose among $m\hat{X}_{exp}$ bits of the mantissa product. As long as $m\hat{X}_{exp} + exp_{BW} \leq 5$, a single LUT5 is sufficient for implementing the multiplexer of each bit. If it exceeds 5, a similar approximation to Eq. 3.11 is applied. The resulting estimation function for the total number of LUT5s for the decoder is as follows:

$$decoder_{LUT5} = \begin{cases} N, & m\hat{X}_{exp} < 4, \\ N \times 2, & m\hat{X}_{exp} = 4, \\ \left(\left\lceil \frac{m\hat{X}_{exp}}{4} \right\rceil + \left\lceil \frac{m\hat{X}_{exp}}{16} \right\rceil \right) \times N \times 2, & m\hat{X}_{exp} > 4. \end{cases} \tag{3.13}$$

For the signed version, Eq. 3.13 needs to be adjusted by replacing N with $N - 1$.

CFGLUT5-Based Mantissa Multiplier

As detailed in Section 3.3.2, an estimate of the required number of LUT5 units for mantissa multiplication can be obtained as follows:

$$mnt\_multiply_{LUT5} = \begin{cases} \left\lceil \hat{Z}_{BW} \right\rceil, & mnt_{BW} \leq 5, \\ \left\lceil 2^{mnt_{BW}-5} \times \hat{Z}_{BW} \right\rceil, & mnt_{BW} > 5. \end{cases} \tag{3.14}$$

Increasing $\hat{Z}_{\mathrm{BW}}$ enhances accuracy at the expense of a larger mantissa multiplication unit and an extended reconfiguration time.

As an example, for a UINT8 DyRecMul multiplier with $float\,(0,2,5)$ internal floating-point format, and the parameter values of $N = 8$, $exp_{BW} = 2$, $mnt_{BW} = 5$, $m\hat{X}_{exp} = 4$, and $\hat{Z}_{\text{BW}} = 5$, the estimated total number of LUT5s using Eq. 3.12, 3.13, and 3.14 is as follows:

$$total_{LUT5} = encoder_{LUT5} + decoder_{LUT5} + mnt\_multiply_{LUT5} = 12 + 16 + 5 = 33 \quad (3.15)$$

From Eq. 3.15, the estimated number of LUT6 units required for this UINT8 DyRecMul is approximately 17. It is worth noting that the synthesizer tools typically have the capability to reduce LUT utilization through optimization.

3.4.2 Accuracy Considerations

In addition to hardware costs, accuracy is another crucial factor to consider when determining DyRecMul parameter values. Specifically in DL applications, relative error serves as a crucial accuracy metric. This importance arises from the prevalence of small values in a significant portion of features and weights in DL computing. Consequently, a simple binary quantization may cause numerous features to be truncated to zero, leading to a substantial loss of accuracy. Hence, there is a growing tendency to use floating-point datatypes, which offer lower relative errors, in ML computation.

Relative error, RE_i when presenting value i is defined as follows:

$$RE_i = \frac{e_i}{Exact_i}, \quad (3.16)$$

where e_i, absolute error, is:

$$e_i = \left|Exact_i - Approx_i\right|, i \in \mathbb{N}. \quad (3.17)$$

Mean Relative Error (MRE) serves as an illustrative metric for measuring relative errors, and it is calculated as follows in unsigned multiplication:

$$MRE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} \frac{e_i}{Exact_i}, \quad (3.18)$$

where N denotes the input bitwidth. A floating-point representation with an mnt_{BW}-bit mantissa can always preserve the mnt_{BW} significant bits from the leftmost '1' (in positive numbers). In the worst case, where only the most significant mantissa bit is a '1', the maximum possible relative error is equal to $2^n / \left(2^{mnt_{BW}+n} + 2^n\right) = 1/2^{mnt_{BW}} + 1$. Conversely, in integer quantization, even quantizing a single least significant bit can lead to a worst-case relative error of 1.

Two key factors significantly impact the relative error in DyRecMul: (1) the mantissa bitwidth, mnt_{BW}, and (2) the maximum exponent value, $m\hat{X}_{exp} - 1$. A longer mantissa allows for the preservation of more least significant bits, thereby improving resolution. Increasing mnt_{BW} reduces the worst-case relative error. On the other hand, the maximum exponent value, coupled with the mantissa bitwidth, defines the supported dynamic range of the floating-point representation. Specifically, when converting an N-bit integer to a $float\,(sign_{BW}, exp_{BW}, mnt_{BW})$ format, the floating-point representation can encompass maximum $m\hat{X}_{exp} + mnt_{BW} - 1$ bits of the input. If N is greater than $m\hat{X}_{exp} + mnt_{BW} - 1$, then the remaining $N - m\hat{X}_{exp} - mnt_{BW} + 1$ bits would be discarded.

The recommended approach to determine suitable parameter values for an $N \times N$ DyRecMul multiplier begins with identifying the two most crucial parameters: $m\hat{X}_{exp}$ and mnt_{BW}. As highlighted in the above discussions, there is a delicate balance between hardware cost and accuracy when selecting values for these parameters. The hardware cost estimation functions and MRE can be employed to determine the optimal values for these parameters in accordance with the design constraints. The exponent bitwidth, exp_{BW}, is then simply obtained using Eq. 3.10. The only remaining parameter is $\hat{Z}_{\mathrm{BW}}$, the bitwidth of the reconfigurable mantissa multiplier. This parameter significantly influences the hardware complexity of the mantissa multiplier, decoder, and, most importantly, the reconfiguration time. It is advisable to fine-tune this parameter after establishing other parameter values, using a simple exhaustive search. This process involves assessing hardware and reconfiguration costs, conducting an accuracy analysis of several options, and selecting the value that provides the optimal trade-off and meets the specified requirements.

3.5 Reconfiguration Time and Rapidly Reconfigurable DyRecMul

The reconfiguration time of CFGLUT5 units in the mantissa multiplier is a crucial parameter for the practicality of DyRecMul. As illustrated in Fig. 3.1, the reconfiguration bits are stored in a BRAM-based *Config bit Memory*, which functions as look-up tables. When CFGLUT5s need to be updated with a new W value, the new W serves as an address to read a total of $\#CFGLUT5 \times 2^5$ reconfiguration bits to be programmed into the CFGLUT5s, where $\#CFGLUT5$ has been calculated in Eq. 3.8. Since the programming is executed serially through a cascaded chain, the entire reconfiguration of the mantissa multiplier will take $\#CFGLUT5 \times 2^5$ clock cycles. In the case of the INT8 DyRecMul presented in Section 3.3, which utilizes 5 CFGLUT5s, this corresponds to $5 \times 2^5 = 160$ clock cycles. Throughout the reconfiguration time, the multiplier remains non-functional. Therefore, a standard DyRecMul architecture, as discussed thus far, is suitable for applications where the frequency of updating one of the operands is slow enough to ensure that the reconfiguration stall time does not introduce a significant latency overhead in overall processing time.

To mitigate this limitation, this section introduces a Rapidly Reconfigurable version of DyRecMul, termed RR_DyRecMul. RR_DyRecMul employs a ping-pong scheme to conceal the reconfiguration time. As illustrated in Fig. 3.5, the RR_DyRecMul architecture incorporates a second CFGLUT5-based mantissa multiplier. A multiplexer is utilized to select the output of one of the mantissa multipliers at any given time. While the first mantissa multiplier is operational, the second one can be reconfigured with the next W value. The transition from the old W value to the new one can then occur by simply adjusting the multiplexer selector, and activating the second multiplier while putting the first one in reconfiguration mode. In RR_DyRecMul, if one of the operands remains unchanged for at least $\#CFGLUT5 \times 2^5$ clock cycles, the reconfiguration time overhead ideally becomes zero. The cost overhead comprises an additional mantissa multiplier, the LUT cost of which is estimated in Eq. 3.14, as well as a 2-to-1 multiplexer that would require a maximum of $\hat{Z}_{\mathrm{BW}}$ LUTs.

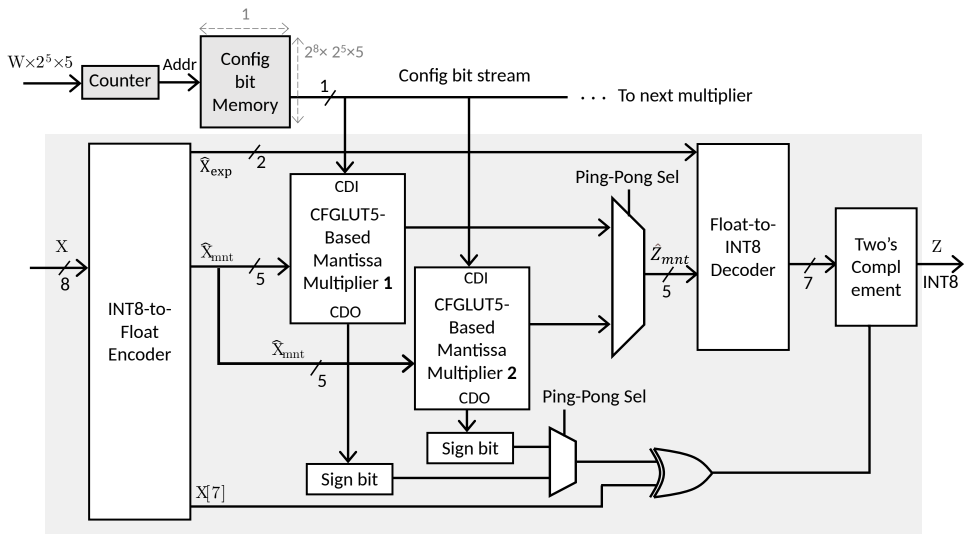


Figure 3.5 Architecture of RR_DyRecMul utilizing a ping-pong scheme to reduce the reconfiguration latency.

3.6 Target Applications

As Section 3.7 will unveil, DyRecMul achieves a significant enhancement in efficiency, particularly with its innovative CFGLUT5-based mantissa multiplier. The reconfiguration time is a crucial factor that may introduce latency overhead, particularly when the hardcoded operand experiences frequent variations. Consequently, DyRecMul emerges as an effective solution for applications in which one operand remains relatively constant. This characteristic aligns well with various ML computations and signal and image processing applications, where the infrequency of changes in one operand is a common property. This section provides an overview of some key applications for the DyRecMul multiplier. The utilization of DyRecMul has the potential to significantly enhance the efficiency of FPGA-based computing in these applications.

3.6.1 Low-cost Hardware Accelerators for Updatable Supervised Learning

Hardware accelerators are essential for achieving the required inference throughput, low latency, and energy efficiency needed for deploying deep learning models across various applications. In an expanding array of applications across domains such as wireless communication and the inference of tiny ML models, where throughput is crucial, a fully parallelized accelerator architecture assigns a dedicated hardware multiplier to each multiplication. When model parameters remain unchanged, employing a constant multiplier can considerably enhance efficiency. However, occasional updating of ML parameter values is a crucial requirement for numerous supervised learning applications. Even in specific ML approaches, like federated learning and incremental learning, the periodic adjustment of model parameters is a fundamental feature. The support for such occasional parameter updates requires non-constant multipliers, which, in turn, incurs significant costs and energy overhead. DyRecMul serves as a solution that bridges the gap between constant and non-constant multipliers, delivering markedly enhanced efficiency compared to non-constant multipliers while accommodating occasional updates.

3.6.2 Deep Reinforcement Learning Hardware

Reinforcement Learning (RL) methods encompass two distinct phases: exploration and exploitation. In the exploration phase, the agent actively seeks the optimal action selection policy for each state within the environment. This policy undergoes iterative refinement until a near-optimal solution is achieved. Subsequently, the RL transitions to the exploitation phase, during which the agent applies its learned policy to make decisions within the environment, aiming to maximize its cumulative reward. In deep reinforcement learning techniques like deep Q-learning, a DNN is utilized to fully approximate Quality Values (Q-values) or state-action policies. Throughout the exploration phase, the DNN undergoes retraining, whereas, in the considerably longer exploitation phase, the DNN's parameter weights remain constant. DyRecMul is a fitting choice for such applications characterized by relatively short periods of coefficient changes and where constant multipliers suffice for the majority of operational time. DyRecMul efficiently addresses the implementation of high-performance DNNs by supporting weight updates during the exploration phase through reconfiguration. Afterward, it seamlessly operates as a cost-effective constant multiplier during the exploitation phase.

3.6.3 DL Hardware Accelerators With Weight-Stationary Dataflows

DL hardware accelerators are increasingly utilized for computing the inference of DL models. These accelerators are commonly designed to handle computations across various types and sizes of DL algorithms. Given that MAC operations form the predominant majority and bottleneck, these accelerators typically feature a core component consisting of a matrix of Processing Elements (PEs) each containing a MAC operator. Weights and features are provided to this PE matrix using a predefined dataflow, wherein matrix multiplications are typically carried out for various DL layers such as multilayer perceptron layers or convolutional layers. A commonly used dataflow in DL accelerators is the weight-stationary scheme, where weights are stored in local memories within the PEs, and input features are sequentially uploaded and circulated across PEs [6]. DyRecMul is an effective solution for weight-stationary dataflows, particularly when dealing with large-sized input features that necessitate infrequent updates of weights. In addition to offering reduced hardware cost and latency compared to standard multipliers, DyRecMul eliminates the need for internal registers to store weights. This is achieved by directly hardcoding and programming the weights into the reconfigurable mantissa multiplier. In scenarios where the input feature size is substantial, the number of clock cycles during which weights must persist on PEs is notably greater than the reconfiguration time calculated in Section 3.5. This long stability period for weights eliminates the need for reconfiguration time overhead in RR_DyRecMul, as it affords sufficient time to program the next weights into the second mantissa multiplier unit while the current weights are still in use. The transition to the new weights can then be seamlessly executed within a single cycle. Even with the baseline DyRecMul, a larger input feature size results in less frequent reconfiguration, leading to a correspondingly lower reconfiguration time overhead.

3.6.4 Updatable Digital Filters

Beyond its applications in ML hardware, DyRecMul can be effectively utilized in digital filters for signal and image processing. This is particularly the case in scenarios where the filter coefficients remain constant but require occasional updates. DyRecMul achieves markedly higher efficiency by hardcoding coefficients into CFGLUT5s compared to non-constant multipliers, all while allowing for periodic coefficient updates. Alternative solutions for achieving updatable filters involve using non-constant multipliers with additional registers to store coefficients or resorting to constant multipliers, demanding the reprogramming of the entire FPGA when weights are updated. The former solution often results in diminished efficiency concerning hardware cost and latency, while the latter necessitates a service interruption for FPGA reprogramming. DyRecMul serves as a solution that bridges the gap between non-constant and constant multiplier methods, offering enhanced efficiency while allowing dynamic coefficient updates.

3.7 Results

We evaluated INT8/UINT8 DyRecMul by conducting error analysis, measuring hardware efficiency, and running accuracy tests on popular benchmark DL models. This section presents the results, discussions, and comparisons.

3.7.1 Error Analysis and Evaluation

One important method for assessing the accuracy of approximate calculations is error measurements and analysis. These assessments gauge the difference between the results of approximate calculations to those of exact calculations to determine any discrepancies [70]. This section presents the accuracy of DyRecMul with five error metrics, including MRE (defined in Eq. 3.18), Error Probability (EP), Mean Absolute Error (MAE), Mean Squared Error (MSE), and Normalized Error Distance (NED). For unsigned arithmetic, these metrics are defined as follows:

$$EP = \frac{1}{2^N} \sum_{i=0}^{2^N-1} E_i, E_i = \begin{cases} 1, & \text{if } e_i \neq 0, \\ 0, & \text{if } e_i = 0, \end{cases} \tag{3.19}$$

$$MAE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} e_i, \tag{3.20}$$

$$MSE = \frac{1}{2^N} \sum_{i=0}^{2^N-1} (e_i)^2, \tag{3.21}$$

$$NED = \frac{1}{2^N} \sum_{i=0}^{2^N-1} \frac{e_i}{max(e)}, \tag{3.22}$$

where N denotes the input bitwidth and e_i has been defined in Eq. 3.17.

Table 3.1 compares DyRecMul with existing approximated multipliers [46, 48, 52, 63–65, 71–74]. Our primary focus lies on the signed version of the DyRecMul, tailored specifically for ML applications. Nonetheless, we present the unsigned version of the multiplier, which is suitable for error-tolerant applications within the domain of image processing, such as image sharpening and smoothing [75]. Our method outperforms [72] by a significant margin in error analysis. The approximate multipliers proposed by [65] and [73] offer reconfigurability based on an approximation factor P representing the number of encoders employed in the partial product generation. DyRecMul yields comparable accuracy compared to this method in the low approximation ($P = 8$) mode. In the mid-level approximation mode ($P = 10$), our method outperforms the other designs and proves its robustness and effectiveness.

Although error analysis can be useful in evaluating the accuracy of approximated circuits, it may not give a complete picture. In Section 3.7.3, we will examine how the approximation in DyRecMul affects the inference accuracy of some benchmark CNN models in order to gain a more comprehensive understanding.

3.7.2 Hardware Implementation Results

DyRecMul was modeled in VHDL, then synthesized and implemented with AMD-Xilinx Vivado ML 2022.2 for a xcku5p Kintex Ultrascale+ FPGA with a speed grade of -3. For comparisons, we also evaluated the default AMD-Xilinx multiplier circuit. To ensure fairness, the synthesis tool was instructed to avoid using DSP resources.

Table 3.2 presents the implementation costs and maximum supported clock frequency of INT8/UINT8 DyRecMul and the Xilinx standard multiplier cores in three configurations: signed multiplier, unsigned multiplier, and signed MAC. The metrics include the utilization of LUTs and CARRY8 primitives, as well as the maximum clock frequency that is supported. Table 3.2 also presents the results of four unsigned approximate multipliers from existing works. These data have been extracted from the reported implementation results on Xilinx Spartan-6 in [65], and normalized based on exact multiplier results. As all the compared

Table 3.1 Error Analysis and Comparisons

Multiplier	**EP**	**MAE**	**MRE**	**MSE**	**NED**
DyRecMul (signed)	0.5157	397	0.0680	96336	0.00005
Danopoulos [72]	0.7480	464	0.1259	5515991	0.01160
Liu [73] ($P = 8$)	-*	-	0.0525	-	-
Liu [73] ($P = 10$)	-	-	0.1991	-	-
Van Toan [65] ($P = 8$)	-	100	0.2299	-	-
Van Toan [65] ($P = 10$)	-	512	0.9320	-	-
DyRecMul (unsigned)	0.7380	336	0.0194	260528	0.1210
Ha [52]	-	3490	0.3676	-	-
Ullah [64] $Approx1$	-	-	0.016	-	-
Ullah [64] $Approx2$	-	-	0.030	-	-
Ullah [64] $Approx3$	-	-	0.027	-	-
Ullah [63] Ca	-	-	0.0029	-	-
Ullah [63] Cc	-	-	0.1293	-	-
Yang [48]	-	220	0.0196	-	-
Venkatachalam [46]	-	101	0.0548	-	-
Liu [71]	-	130	0.0062	-	-
Ansari [74]	-	1530	0.1336	-	-

* Not reported in the reference

designs are combinational multipliers, none of them utilizes flip-flops. Additionally, the utilization of other resources, such as BRAMs and DSPs, is also zero.

The results indicate significant reductions in LUT usage with 64%, 80%, and 67% savings for the signed, unsigned, and MAC cases, respectively, compared to the exact 8×8 multiplier. DyRecMul also archived a higher frequency than all existing exact and approximate multipliers. The ping-pong scheme facilitating rapid reconfiguration in RR_DyRecMul incurs an additional cost of ten LUTs, a still considerably more cost-effective option compared to standard multipliers.

Furthermore, from the discussions in Section 3.3, it can be inferred that the total size of this configuration bit memory in terms of the number of bits is:

$$config\_bit\_memory_{size(bits)} = \hat{Z}_{BW} \times 2^N \times 2^{mnt_{BW}}. \tag{3.23}$$

As each BRAM36K unit can be configured and used as a 32K by 1-bit wide memory, mapping the configuration bit memory onto BRAM36K elements, the total number of required BRAM36K units is calculated as follows:

$$config\_bit\_memory_{\#BRAMs} = \left\lceil \left(\hat{Z}_{BW} \times 2^N \times 2^{mnt_{BW}} \right) / 32768 \right\rceil. \tag{3.24}$$

From Eq. 3.24, we can determine that two BRAM36K units are required for the proposed INT8/UINT8 DyRecMul as $\lceil (5 \times 2^8 \times 2^5)/32768 \rceil = 2$.

In addition to incorporating 8-bit multipliers, we expanded our implementation to include 12-bit and 16-bit multipliers, utilizing distinct internal floating-point formats. This extension enabled us to experimentally assess the efficiency of DyRecMul when applied to larger datatypes. Table 3.3 presents the hardware implementation results for 12-bit and 16-bit multipliers, comparing DyRecMul, RR_DyRecMul, and AMD-Xilinx multiplier cores. For the INT12/UINT12 versions, we allocated three bits to the exponent bitwidth, $exp_{BW} = 3$, and five bits to the mantissa bitwidth, $mnt_{BW} = 5$, resulting in the format $float\,(1/0, 3, 5)$. Consequently, $m\hat{X}_{exp}$ equals eight and seven in these UINT12 and INT12 multipliers, respectively. In the case of INT16/UINT16 multipliers, we assigned four bits and five bits to the exponent and mantissa bitwidths, respectively, yielding the format $float\,(1/0, 4, 5)$. The findings highlight that the hardware efficiency superiority of DyRecMul becomes more pronounced with larger datatypes. Additionally, the results indicate that unsigned multipliers exhibit progressively lower hardware costs compared to their signed counterparts. This disparity is primarily attributed to the two's complement logic in signed DyRecMul, which incurs escalating costs as the input and output bitwidth, N, increases.

Despite considerable utilization of CARRY8 fast carry primitives in standard multipliers, DyRecMul achieves a significantly higher maximum clock frequency thanks to its optimized datapath. DyRecMul yields similar results in both signed multiplier and MAC setups. This is because the two's complement logic at the end of the architecture needs the same amount of LUTs as an adder/subtractor in MAC. In other words, an adder/subtractor can be incorporated into the two's complement LUT-based circuit with almost no cost overhead. Among the three tested setups, DyRecMul achieves the highest performance in the unsigned multiplication setup. This is because the two's complement logic in signed multiplication and the add/sub logic in MAC operations constitute a significant portion of both hardware and latency.

3.7.3 Accuracy in Deep Learning Computation

To ensure the usability of DyRecMul in DNN accelerators, we evaluated its accuracy for the benchmark models ResNet18, ResNet50, VGG19, and DenseNet121 using the CIFAR-10 dataset. For these experiments, we used the AdaPT framework which provides a rapid em-

Table 3.2 Implementation Costs and Performance Comparison for INT8 and UINT8 datatypes

Function	**Multiplier**	**Size**	**#LUT**	**#CARRY8/4**	**Max Freq. (MHz)**
Signed Multiplier	DyRecMul	8×8	25	0	770
	RR_DyRecMul	8×8	35	0	699
	AMD-Xilinx	8×8	69	8	730
	(Exact)	7×7	61	6	660
Signed MAC	DyRecMul	8×8	25	1	769
	AMD-Xilinx	8×8	76	10	571
	(Exact)	7×7	69	8	585
Unsigned Multiplier	DyRecMul	8×8	16	0	950
	RR_DyRecMul	8×8	26	0	847
	AMD-Xilinx	8×8	82	6	684
	(Exact)	7×7	55	6	725
	Venkatachalam [46] *	8×8	108	4	690
	Ha [52] *	8×8	73	4	664
	Ansari [74] *	8×8	63	3	757
	Yang [47] *	8×8	76	3	729
	Ullah [64] $Approx1$†	8×8	64	2	-
	Ullah [64] $Approx2$†	8×8	54	0	-
	Ullah [64] $Approx3$†	8×8	54	0	-
	Ullah [63] Ca†	8×8	67	7	424
	Ullah [63] Cc†	8×8	59	4	452
	Van Toan [65] $P = 8$*	8×8	59	4	759
	Van Toan [65] $P = 10$*	8×8	56	4	849

* Results are obtained from [65] and normalized based on the exact multiplier results.

† The target device was Virtex-7.

Table 3.3 Implementation Costs and Performance Comparison for INT/UINT12 and INT/UINT16 Multipliers

Datatype	**Multiplier**	**Size**	**#LUT**	**#CARRY8**	**Max Freq. (MHz)**
INT12 *	DyRecMul	12×12	62	0	470
	RR_DyRecMul	12×12	70	0	425
	AMD-Xilinx (Exact)	12×12	168	18	525
UINT12 *	DyRecMul	12×12	36	0	615
	RR_DyRecMul	12×12	48	0	580
	AMD-Xilinx (Exact)	12×12	186	15	575
INT16 †	DyRecMul	16×16	104	0	355
	RR_DyRecMul	16×16	114	0	320
	AMD-Xilinx (Exact)	16×16	279	32	355
UINT16 †	DyRecMul	16×16	45	0	510
	RR_DyRecMul	16×16	56	0	475
	AMD-Xilinx (Exact)	16×16	337	28	400

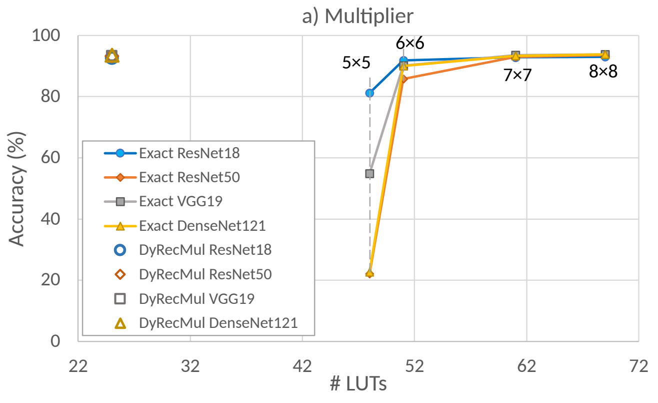
* For INT12/UINT12, $exp_{BW} = 3$ and $mnt_{BW} = 5$.

† For INT12/UINT16, $exp_{BW} = 4$ and $mnt_{BW} = 5$.

Table 3.4 Inference Accuracy and Hardware Metrics Comparison With Previous Studies

Multiplier	**Dataset**	**Accuracy (%)**				**Area (LUTs)**		**Max Freq. (MHz)**	
		ResNet18	**ResNet50**	**VGG19**	**DenseNet121**	**MUL**	**MAC**	**MUL**	**MAC**
Exact (5 × 5)	CIFAR-10	81.20	22.08	34.83	22.72	48	52	740	724
Exact (6 × 6)	CIFAR-10	91.83	85.81	90.17	89.98	51	62	770	700
Exact (7 × 7)	CIFAR-10	92.83	92.95	93.60	93.37	61	69	660	585
Exact (8 × 8)	CIFAR-10	92.98	93.57	93.78	93.85	69	76	730	571
[72] (8 × 8)	CIFAR-10	-	82.70	90.70	-	-	-	-	-
DyRecMul	CIFAR-10	92.72	93.32	93.45	93.54	25	25	770	769

ulation environment to measure the accuracy of new approximate multipliers in the CNNs, LSTMs, and GANs inferences [72]. Fig. 3.6 compares the inference accuracy and the hardware utilization costs offered by an INT8 DyRecMul with those of standard AMD-Xilinx multipliers with different bitwidths. The accuracy is measured for post-training deployment



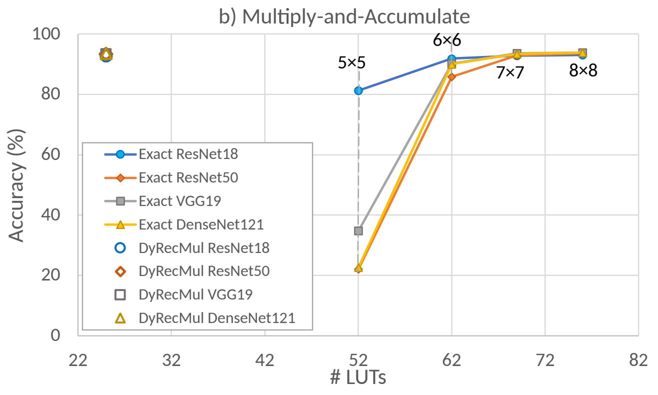


Figure 3.6 Trade-offs between hardware utilization and inference accuracy when running four benchmark CNNs using DyRecMul and standard exact multipliers of different sizes: (a) signed multiplier results, and (b) MAC results.

with no re-training applied. Fig. 3.6 clearly indicates that DyRecMul offers a significantly superior accuracy-hardware cost trade-off compared to the standard AMD-Xilinx multiplier cores. Table 3.4 provides detailed results of INT8 DyRecMul and selected previous works, including inference accuracy, maximum supported clock frequency, and hardware utilization. Based on the findings, DyRecMul provides an average accuracy loss that is only 0.29% lower than that of 8×8 exact multipliers and 0.07% higher than 7×7. The worst-case accuracy distance from the exact 8×8 multiplier is only 0.33%. Table 3.4 also shows that DyRecMul offers higher accuracy compared to Danopoulos [72] INT8 approximate multiplier in the VGG19 test and significantly higher accuracy compared to Danopoulos in ResNet50. Also, as reported in Section 3.7.2, DyRecMul uses fewer LUTs, with a reduction of 64% and 67% in signed and MAC setups, respectively. Furthermore, it can support clock frequencies that are up to 5.5% and 34.6% higher than an exact 8×8 multiplier in signed and MAC setups, respectively.

3.8 Conclusion

This paper introduces DyRecMul, an approximate multiplier meticulously optimized for machine learning computations on FPGAs. Leveraging dynamically reconfigurable logic, this multiplier achieves remarkable hardware efficiency. Additionally, it employs a cost-effective internal floating-point conversion technique to preserve a wide dynamic range, thereby enhancing the precision of machine learning calculations. The results demonstrate that for INT8 precision, DyRecMul requires 64%, 80%, and 67% fewer LUTs compared to the Xilinx standard multiplier core in signed, unsigned, and MAC setups, respectively. Moreover, the maximum supported clock frequency remains notably higher than that of Xilinx multipliers. DyRecMul also provides a substantial advantage over existing approximate multipliers in terms of both hardware utilization and frequency. Additionally, the results indicate that employing this multiplier for post-training DL inference leads to a minimal average accuracy degradation of less than 0.29% compared to exact INT8 multiplication.

CHAPTER 4 ARTICLE 2: ACCURACY-AWARE LOW-COMPLEXITY DEEP LEARNING MODELS FOR AUTOMATIC MODULATION RECOGNITION

Authors: Mobin Vaziri, Shervin Vakili, and J.M. Pierre Langlois.

Published in: IEEE International Conference on Computing, Internet of Things and Microwave Systems (2024).

Submitted: 05 May 2024 | **Accepted:** 26 May 2024

Abstract

This paper explores the design of deep learning models for automatic modulation recognition in wireless communications. The primary goal is to enhance the efficiency and hardware compatibility of CNNs for AMR through hyperparameter tuning and model compression. The paper first examines the effectiveness of applying quantization and pruning on the accuracy and computational cost of two prominent CNN models from the literature. It then introduces a new CNN model that achieves superior accuracy with lower computational complexity compared to previous work. The design flow integrates TensorFlow Lite for pruning and quantization, and NVIDIA TensorRT for benchmarking on GPUs specialized for machine learning computing. Experimental results show significant reductions in model size and computational complexity while maintaining accuracy, rendering the proposed DL models suitable for real-time applications on edge devices.

4.1 Introduction

Deep learning models are transforming wireless communication by tackling challenges that traditional methods struggle to solve. In the physical layer, they streamline complex tasks like signal detection and interference management, learning from data to make real-time adjustments. They also bring new agility to spectrum awareness, helping systems dynamically manage spectrum allocation and classify signals, even in crowded environments [76]. This adaptability is paving the way for more responsive, efficient wireless networks by building on these capabilities. In the conventional communication chain, deep autoencoders have been employed to model the transmitter (TX) and receiver (RX) [77], allowing for a more efficient and accurate representation of the communication process. For channel estimation and modeling, Generative Adversarial Networks (GANs) have been trained to learn the underlying data patterns or distributions, enabling them to generate new, similar data. By using this generated data, data-driven methods, such as deep learning techniques, can approximate the channel without relying on closed-form analytical expressions, ultimately improving communication performance [78].

AMR is instrumental in applications such as cognitive radio, wireless security, signal monitoring, and interference detection [79]. AMR methods can be categorized into Likelihood Theory-based AMR (LB-AMR) and Feature-based AMR (FB-AMR) [80,81]. LB-AMR methods can achieve optimal recognition accuracy but are computationally complex. FB-AMR methods learn features from training samples, they can identify multiple modulation types and have low computational complexity. ANNs, especially those with stacked multi-layer architectures, have shown powerful feature extraction capabilities and outperform traditional LB-AMR and FB-AMR methods. CNNs have been applied to the AMR problem by leveraging their spatial feature extraction power [82, 83], while Recurrent Neural Networks (RNNs) can learn temporal correlation features present in modulation [84]. West et al. proposed a hybrid CNN-Long-Short-Term Memory model consisting of one Long Short-Term Memory (LSTM) and three CNN layers [85]. It includes a skip connection before the LSTM, bypassing two CNN layers and providing a longer time context for extracted features.

This work investigates two top-performing CNN models from the literature for AMR and enhances their processing efficiency and hardware compatibility through model compression techniques, specifically pruning and quantization [10, 12]. Furthermore, leveraging insights from these two models, we introduce a new CNN model that achieves a superior balance between computational efficiency and recognition accuracy. The main contributions can be summarized as follows:

- Hardware-aware model compression techniques to optimize two prominent CNN models proposed for AMR, targeting real-time applications.
- An approach for designing less complex yet high-accuracy DNN models for AMR.
- An end-to-end design flow for model deployment and inference, tailored for data centers and resource-constrained embedded systems.
- A new CNN model that provides an improved tradeoff between accuracy and processing latency.

4.2 Proposed Design Flow

The proposed design flow is outlined in Fig. 4.1. Our initial goal is to reconstruct the model used for AMR in Keras, which is a high-level API on TensorFlow. Next, we apply structured pruning to the model parameters. We use TensorFlow Lite (TFLITE) for PTQ and evaluate the model size and accuracy of the pruned and quantized models. In the inference phase, NVIDIA TensorRT is used to benchmark the optimized models from the previous phase for GPUs in different classes, and we measure the latency. The following subsections provide a detailed explanation of the proposed design flow.

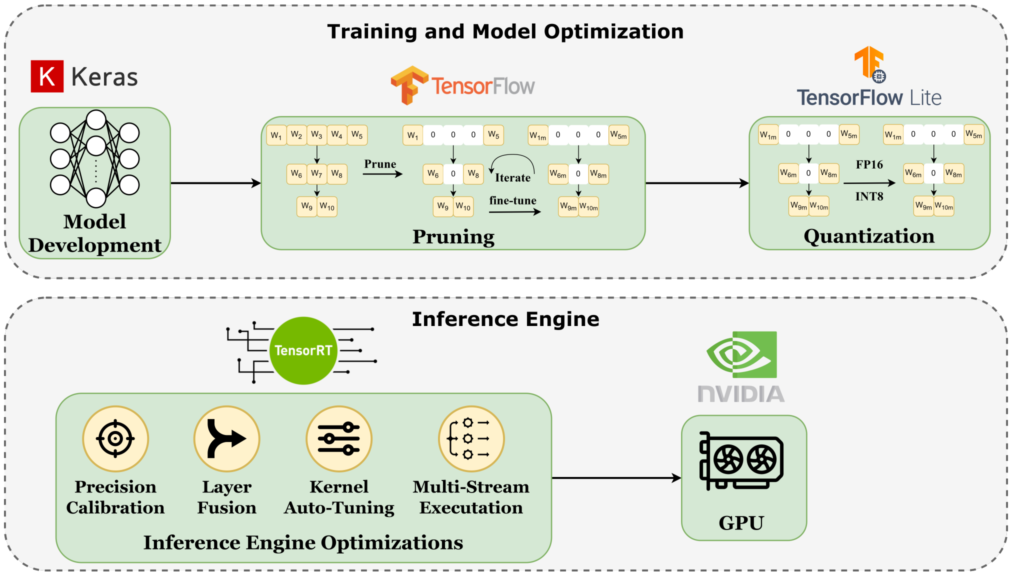


Figure 4.1 Proposed design flow.

4.2.1 Model Development

In this study, we consider two CNN models [82,83]. The model parameters are listed in Table 4.1, and the hyperparameters for training models, inspired by [79], are listed in Table 4.2. These models are trained using the RML2016.10a dataset, where the sample dimensions are 2×128, consisting of 220,000 data samples for Signal-to-Noise Ratios (SNRs) ranging from -20 to +18 dB, encompassing 11 modulation schemes. Based on our training, CNN1 [82], which contains around 1.59 million parameters, achieves an accuracy of 56.24%, whereas CNN2 [83] with approximately 0.86 million parameters achieves an accuracy of 56.6% on the test set, averaged across all SNRs. These results align with the findings reported in previous works [79].

Table 4.1 Summary of The Models' Parameters

Model	FLOPs (M)	Layer Type	# Parameters[†]	Description
CNN1 [82]	6.5	Conv2D	450	50 filters of size (1, 8)
		Conv2D	40,050	50 filters of size (2, 8)
		FC	1,549,056	Fully connected layer with 256 units
		FC	2,827	Output layer with 11 units
CNN2 [83]	78.86	Conv2D	4,352	256 filters of size (2, 8)
		Conv2D	524,416	128 filters of size (2, 8)
		Conv2D	131,136	64 filters of size (2, 8)
		Conv2D	65,600	64 filters of size (2, 8)
		FC	131,200	Fully connected layer with 128 units
		FC	1,419	Output layer with 11 units
Proposed Model	26.26	Conv2D	2,112	64 filters of size (2, 16)
		Conv2D	131,200	128 filters of size (2, 8)
		Conv2D	131,200	128 filters of size (2, 4)
		FC	524,416	Fully connected layer with 128 units
		FC	1,419	Output layer layer with 11 units

[†] Total number of weights and biases.

CNN1 includes two convolutional layers for feature extraction and two Fully Connected (FC) layers for classification. The first FC layer has significant computational complexity,

Table 4.2 Summary of The Training Hyperparameters

Hyperparameter	**Value**
Learning Rate	0.001
Loss Function	Categorical Crossentropy
Optimizer	Adam
Epochs	110
Batch Size	512

containing over 1.55 million parameters, which constitute approximately 97% of the total parameters in that model.

CNN1 requires approximately 92% fewer FLOPs compared to CNN2. However, this comes at the cost of lower model accuracy. CNN2 takes a different approach, utilizing more convolution layers for feature extraction and incorporating pooling layers after each convolution. This design significantly reduces the number of parameters in the FC layers but increases the overall number of operations. Unlike benchmark deep learning models referenced in [86, 87], CNN2 does not increase the number of filters and reduces the filter size in the deeper layers to capture more complex features.

Our proposed CNN model strikes a balance between these two strategies. It features three convolution layers and pooling layers for feature extraction, along with Batch Normalization (BN) layers. This design, with 790,347 parameters, achieves a classification accuracy of 57.82%. This accuracy surpasses that of both CNN1 and CNN2, while maintaining a parameter count similar to CNN2 and significantly lower than CNN1. The application of BN provides two key benefits. First, it enhances model accuracy by normalizing the layer inputs after the activations. This normalization reduces the model's sensitivity to initial weight values and the learning rate, allowing the network to tolerate higher learning rates without the risk of gradients exploding or vanishing. This leads to a faster and more effective training process, thereby increasing the accuracy of the model [12]. Second, it leverages the capability of TensorRT to fuse convolution and BN layers, computing them as a single layer. This fusion results in higher accuracy without incurring latency overhead during inference. This makes our proposed model a promising solution for the AMR problem, balancing accuracy, complexity, and computational efficiency.

4.2.2 Structural Pruning

Structural pruning zeroes out specific model weights at the beginning of the training process. We conducted experiments using three different percentages of sparsity to assess accuracy, employing TensorFlow's constant sparsity feature with a Magnitude-based criterion. The process starts by specifying the desired percentage of sparsity in each layer for the baseline pre-trained model. Then, through fine-tuning, weights with insignificant values are pruned, and awareness of sparsity is imparted. Structural pruning can accelerate inference on supported hardware for several reasons. For instance, if we can entirely remove one or more filters, specialized hardware can skip zeros for both computation and data movement between different memory levels. The results of pruning both models with different sparsity levels, along with the model parameters, are presented in Fig. 4.2.

4.2.3 Post-training Dynamic Range Quantization

TFLITE supports both PTQ and QAT, but with limited data types specialized for hardware accelerators like EdgeTPU. PTQ offers three different methods: FP16 quantization, dynamic range quantization, and integer quantization. Unlike full integer quantization, which quantizes activations (input and output feature maps) to INT8, the other methods do not require representative data for calibration and do not result in significant accuracy degradation. More precisely, in dynamic range quantization and FP16 quantization, only the model pa-

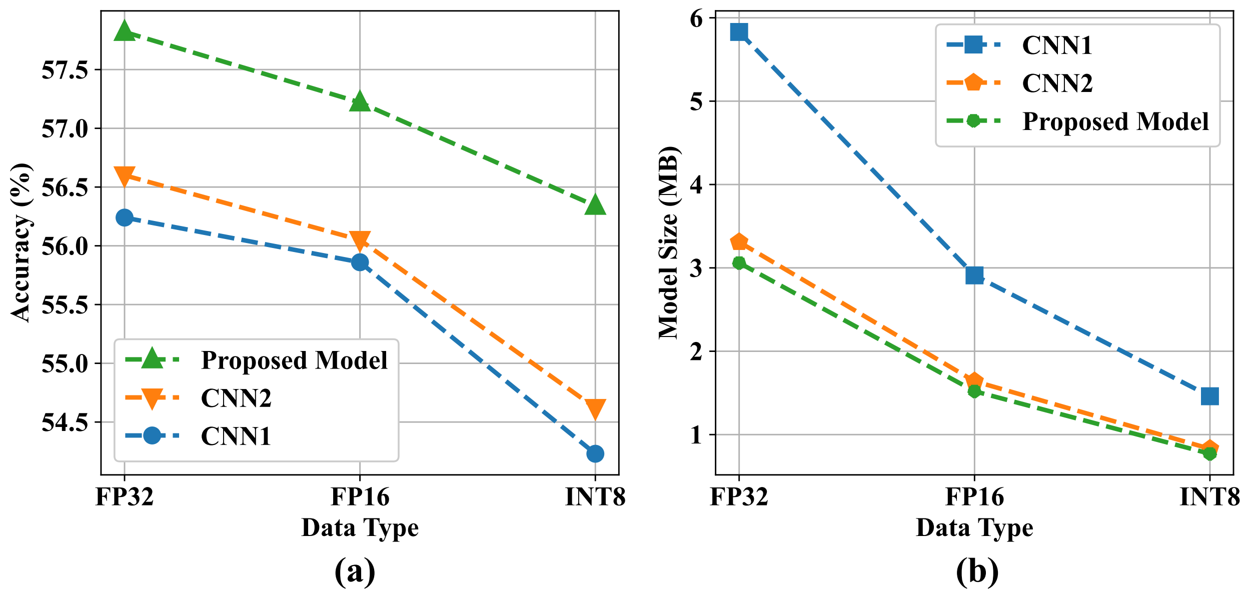


Figure 4.2 Quantizing models with different bitwidths vs (a) accuracy (b) model size.

rameters are quantized, while the activations remain in FP32. Since these two methods do not support retraining, we chose PTQ methods for this study. The accuracy and model sizes of both the baseline models and their quantized versions are depicted in Fig. 4.3.

4.3 Experimental Results

This section presents the results from the baseline and optimized models in AMR inference. For training the base models, we utilize the TPU option available on Google Colaboratory. Subsequently, we deployed TFLITE for both pruning and quantization of the models using different configurations. We evaluated the baseline models on an Intel Xeon CPU on Google Colaboratory using the TensorFlow framework, while the optimized models were evaluated on a GPU using NVIDIA TensorRT. This was done for two primary reasons. Firstly, the optimizations we applied were specifically targeted and supported by TensorRT. Secondly, we aimed to explore the impact of using domain-specific hardware for DNNs, such as NVIDIA® T4 GPU, on accelerating the inference process compared to a CPU. TensorRT can optimize and scale deep learning models, preparing them for deployment in Edge and embedded computing hardware like the NVIDIA Jetson family. In Table 4.3, we outline the main evaluation results. As depicted in Fig. 4.2, pruning models with 75% sparsity results in a significant reduction in accuracy. Given this fact, we chose to maintain the sparsity at 50% and quantize the models using different data types.

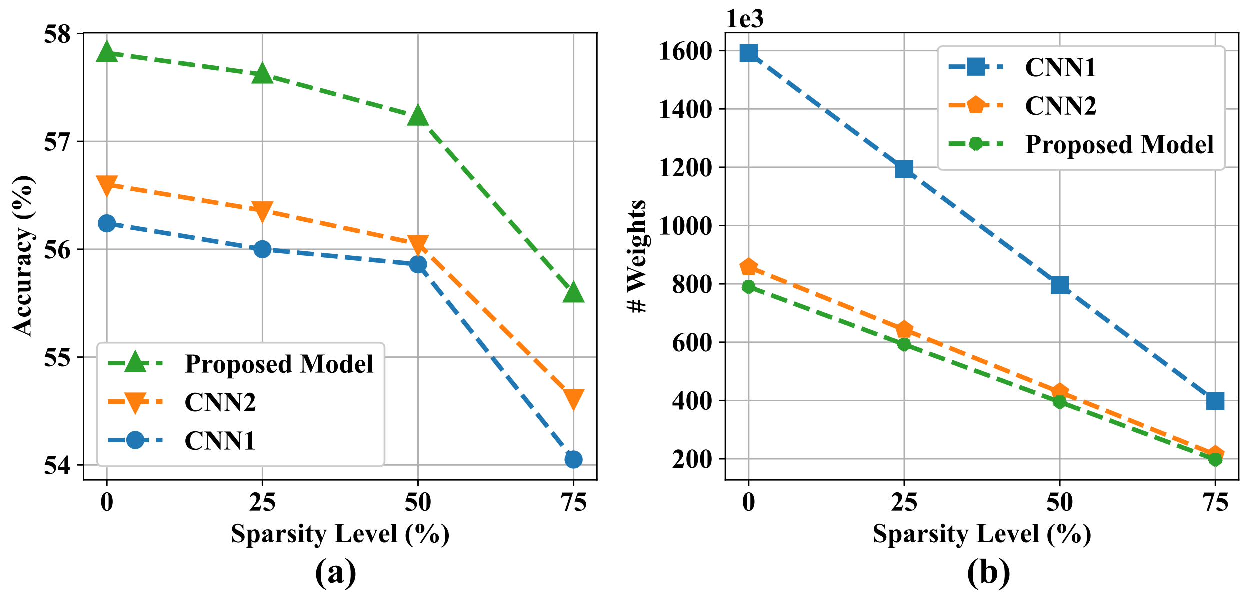


Figure 4.3 Pruning models with different sparsity vs (a) accuracy (b) the number of weights.

Table 4.3 Comparison Between Performance of Baseline Models and Optimized Models

Baseline (CPU Execution)				**Pruned + Quantized Models (GPU Execution)**								
	CNN1 [82]	**CNN2 [83]**	**Proposed**	**CNN1 [82]**			**CNN2 [83]**			**Proposed**		
Datatype	FP32			FP32	FP16	INT8	FP32	FP16	INT8	FP32	FP16	INT8
Accuracy (%)	56.2413	56.6045	57.8227	55.8636	55.8614	54.8386	56.0568	56.0386	54.9341	57.2363	57.2098	56.3409
# Parameters †	1,592,016	857,472	789,888	796,008			428,736			394,944		
Model Size (MB)	5.83	3.31	3.06	3.43	1.89	0.89	1.87	1.04	0.49	1.74	0.97	0.48
Lataency (ms)	50.99	58.49	56.87	0.4001	0.3815	0.3614	1.2119	0.4554	0.4277	0.4877	0.4410	0.4227

† Bias parameter is not included.

As mentioned in Section 4.2.1, our proposed model achieves higher accuracy compared to other models with fewer parameters. However, the results in Table 4.3 demonstrate that the baseline CNN1, which has the lowest number of operations, unsurprisingly achieves the lowest latency during CPU execution. Our proposed model strikes a commendable balance between accuracy and latency with lower FLOPs than CNN2 and higher accuracy than both models.

With optimization and GPU execution, both the previous and proposed models showed significant improvements in inference latency compared to the baseline models executed on CPU. As anticipated, CNN1 exhibited the lowest latency, CNN2 showed the highest latency, and our model achieved the best accuracy with latency comparable to CNN1. Even after quantizing the models to INT8 and applying pruning techniques, our proposed model stands out as the best approach, achieving an inference latency of 0.4227 ms with 50% sparsity and INT8 quantization, while maintaining satisfactory accuracy. However, it is important to note that this latency still surpasses the 0.01 ms requirement for Beyond-5G (B5G) technology [88]. One of the key findings of this study is to illustrate that even with the deployment of DL-optimized GPUs and the implementation of model optimization techniques, there is still a need to develop DNN architectures with reduced computational complexity to meet the criteria of B5G systems.

4.4 Conclusion & Future Work

This study has presented a comprehensive exploration of efficient DL models for AMR in wireless communication. The primary focus was on enhancing the efficiency and hardware compatibility of CNNs for the AMR problem. The effectiveness of hyperparameter tuning and model compression techniques, including pruning and quantization, was thoroughly investigated. The experimental results demonstrate that our proposed CNN model achieves superior accuracy with lower computational complexity compared to previous works. The optimized models show a significant inference speedup compared to running on a CPU. However, despite these optimizations, the latency of the best approach which was our proper model with 0.4227 ms, couldn't meet the mentioned criteria for B5G systems in real-time applications. Looking forward, there are several promising directions for future research. One potential avenue is the design of more efficient DNN model architectures. This could involve further exploration of hyperparameter tuning and model compression techniques, or the development of entirely new architectures specifically tailored for the AMR problem. Another exciting prospect is the design of dedicated hardware accelerators for these models on FPGAs. FPGAs have shown superior performance compared to GPUs in certain applications, and their programmable nature makes them a flexible solution for implementing DL hardware accelerators. This could potentially lead to further reductions in latency and improvements in accuracy, bringing us closer to meeting the stringent requirements of B5G systems.

CHAPTER 5 ARTICLE 3: HENNC: HARDWARE ENGINE FOR ARTIFICIAL NEURAL NETWORK-BASED CHAOTIC OSCILLATORS

Authors: Mobin Vaziri, Shervin Vakili, M. Mehdi Rahimifar, and J.M. Pierre Langlois.

Accepted in: IEEE High Performance Extreme Computing Conference (2024).

Submitted: 10 July 2024 | **Accepted:** 27 August 2024

Abstract

This paper introduces a framework for automatically generating hardware cores for ANN-based chaotic oscillators. The framework starts with training a model to approximate a chaotic system and subsequently conducts design space exploration to identify potential hardware architectures for implementation. From the selected solution, the framework generates synthesizable High-Level Synthesis (HLS) code and a validation testbench tailored for FPGAs. The proposed framework enables a rapid hardware design process for candidate architectures, offering superior hardware cost efficiency and throughput compared to manually designed solutions. The source code is available on GitHub[1].

[1]`https://github.com/INRS-ECCoLe/HENNC`

5.1 Introduction

Chaotic systems have found applications across various fields, including cryptography and secure communications [89]. These systems are particularly effective in image encryption, where their inherent unpredictability ensures robust security against cyber-attacks by scrambling pixels in a highly unpredictable manner [90]. For these applications, a high-throughput PRNG is required for real-time operation [91]. Chaotic oscillators are an appealing choice for PRNGs because of their complex dynamics, ergodicity, and sensitivity to initial conditions. Related studies have shown that, compared to traditional PRNGs such as Linear Feedback Shift Registers (LFSRs) and modern alternatives like the Mersenne Twister (MT), chaotic-based PRNGs demonstrate improved resilience against attacks and enhanced randomness properties [92–94]. While numerical methods such as Forward Euler, Runge-Kutta [95–97], and Heun [98] can produce accurate solutions for even highly complex chaotic systems, their hardware implementation is typically resource-intensive [96].

Previous research has demonstrated the feasibility of using ANNs to approximate chaotic systems [99–102]. Alcin et al. [100] proposed an ANN for a chaotic system on FPGAs. However, the costly implementation of the sigmoid function can significantly impair system performance. Sunny et al. [101] applied Nonlinear Auto-Regressive (NAR) and Nonlinear Auto-Regressive with Exogenous Inputs (NARX) ANNs to model chaotic systems. Nonetheless, NAR and NARX ANNs tend to accumulate feedback errors, which can affect the generated sequences. Al-Musawi et al. [102] introduced an ANN model with fewer hidden neurons; however, this approach resulted in high hardware resource consumption.

This paper presents a comprehensive computer-aided framework for the automated design of efficient hardware architectures for ANN-based chaotic oscillators. By inputting ANN hyper-parameters, users receive a list of Pareto-optimal candidate microarchitectures generated by the framework. The framework then produces the corresponding HLS model for the selected candidate. The main contributions of the paper are:

- A framework for the automated generation of efficient hardware architectures for chaotic oscillators through a rapid design space exploration process. It provides a list of Pareto-optimal candidate architectures, each offering a distinct trade-off between cost and performance.
- Acceleration of the hardware design process through HLS modeling. HLS directives are employed to define the trade-offs between hardware cost and latency.

- Introduction of novel latency and hardware cost estimation functions to speed up the design space exploration.

5.2 ANN-based Chaotic Oscillator

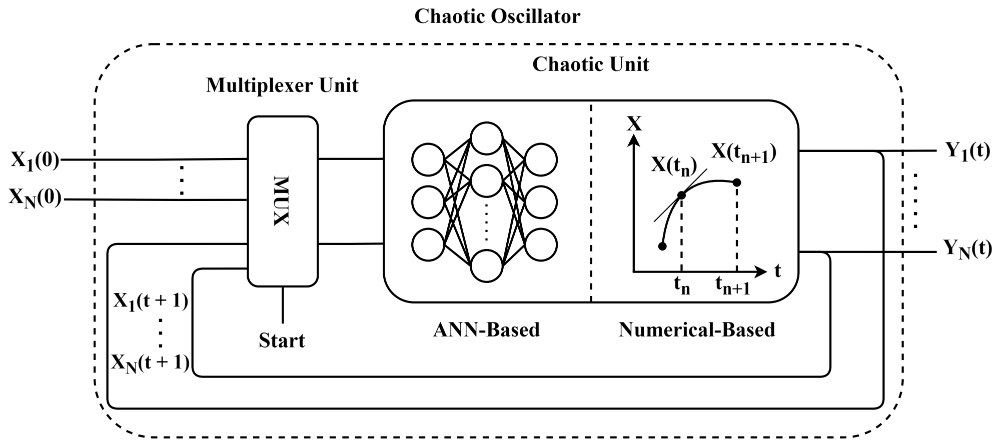


Figure 5.1 Architecture of chaotic oscillators.

Fig. 5.1 illustrates a chaotic oscillator incorporating a Multiplexer (MUX) and a chaotic unit. Solving chaotic systems often involves discretization through methods like the RK algorithm. While RK is a robust analytical method for solving differential equations, it can be computationally expensive [96,97]. ANNs offer a cost-effective alternative for computing non-linear equations, making them suitable for modeling chaotic systems [100]. The performance of such systems largely depends on the implementation approach within the chaotic unit. Key parameters include the number of input and output neurons, which correspond to the dimensionality of the chaotic system, and the number of hidden neurons, which impacts the model's accuracy. Zhang [99] investigated the optimal number of hidden neurons based on the resulting MSE. Their findings indicated that increasing the number of hidden neurons beyond eight did not result in significant improvements in accuracy. This study highlighted the feasibility of using ANNs to effectively model chaotic systems. Further evidence from Yu et al. [103] demonstrated that ANN-based chaotic systems could generate random sequences that successfully pass the stringent randomness tests outlined in the NIST SP 800-22 test suite [104].

Consider a system of N differential equations:

$$\frac{dX_i}{dt} = f_i(X_1, X_2, ..., X_N), \tag{5.1}$$

where X_i are system variables, and t denotes time. The RK 4th-order (RK-4) method is calculated two steps: state variable coefficients (5.2) and variable update (5.3).

$$\begin{aligned}
k_{1i} &= f_i(X_1, X_2, ..., X_N) \\
k_{2i} &= f_i\left(X_1 + \frac{dt}{2} \cdot k_{11}, ..., X_N + \frac{dt}{2} \cdot k_{1N}\right) \\
k_{3i} &= f_i\left(X_1 + \frac{dt}{2} \cdot k_{21}, ..., X_N + \frac{dt}{2} \cdot k_{2N}\right) \\
k_{4i} &= f_i\left(X_1 + dt \cdot k_{31}, ..., X_N + dt \cdot k_{3N}\right)
\end{aligned} \tag{5.2}$$

$$X_{i+1} = X_i + \frac{1}{6} \times (k_{1i} + 2 \times k_{2i} + 2 \times k_{3i} + k_{4i}) \tag{5.3}$$

The multiplications and additions for RK-4 in (5.2) and (5.3) can be divided into static and dynamic terms. Static terms encompass operations that calculate the input arguments in (5.2). Dynamic terms refer to operations present in the selected chaotic system, f_i. The total number of operations is:

$$\begin{aligned}
n_{mul} &= n(mul_{static}) + n(mul_{dynamic}) \\
&= (3 \cdot N^2 + 3 \cdot N) + 4 \cdot n(mul_{dynamic}) \\
n_{add} &= n(add_{static}) + n(add_{dynamic}) \\
&= (3 \cdot N^2 + 4 \cdot N) + 4 \cdot n(add_{dynamic}).
\end{aligned} \tag{5.4}$$

The dynamic terms depend on the selected chaotic system. For instance, for the low-complexity Chen chaotic system [95]:

$$\begin{cases}
\frac{dX_1}{dt} &= a \cdot (X_2 - X_1), \\
\frac{dX_2}{dt} &= (c - a) \cdot X_1 - (X_1 \cdot X_3) + c \cdot X_2, \\
\frac{dX_3}{dt} &= X_1 \cdot X_2 - b \cdot X_3
\end{cases} \tag{5.5}$$

six multiplications and five additions are needed. On the other hand, ANN computation in the layer l and i^{th} neuron is given by:

$$Y_i^{(l)} = \phi\left(\sum_{j=1}^{n_{l-1}} W_{ij}^{(l)} X_j + b_i^{(l)}\right) \tag{5.6}$$

where ϕ is the activation function, n is the number of neurons in the layer, and w and b are the weights and biases of the model, respectively. The number of operations in a general ANN is obtained by:

$$n_{mul} = \sum_{i=2}^{L} n_i \times n_{i-1}, \quad n_{add} = \sum_{i=2}^{L} n_i \times (n_{i-1} + 1). \tag{5.7}$$

Table 5.1 compares the number of operations to implement a minimal chaotic system using the RK-4 method and an ANN proposed by Zhang [99]. ANNs execute a predetermined number of operations based on their structure. However, the computational complexity of the RK-4 method depends on the specific chaotic system it employs. These chaotic systems rely on trigonometric or other non-linear functions, which are computationally expensive. As a result, while achieving minimal error (as shown in Table 5.2) compared to numerical methods, ANNs can provide more stable and efficient solutions.

5.3 Proposed Framework

This section describes the two phases of the HENNC framework, illustrated in Fig. 5.2.

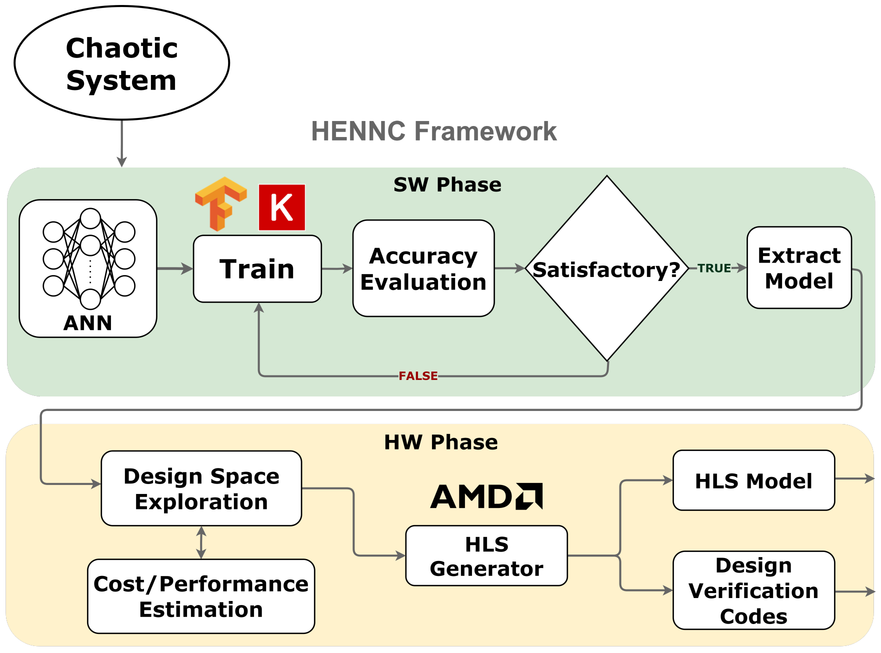


Figure 5.2 HENNC framework design flow

Table 5.1 Number of Operations in an ANN with 8 Neurons in the Hidden Layer, and in RK-4 with Chen Chaotic System

Method	**# Multiplications**	**# Additions**
ANN (3 – 8 – 3) [99]	48	59
RK-4	60	59

Table 5.2 Hyperparameters and Performance of ANN-based Model

Hyperparameters				
Loss Function	Optimizer	Learning Rate		
MSE	Adam	1×10^{-4}		
Performance Metrics				
Activation Function	MSE	MAE	RMSE	R^2
ReLU	0.000308	0.011653	0.017547	0.999990
Tanh	0.006976	0.44344	0.083520	0.999819
Sigmoid	0.044117	0.109703	0.210040	0.998751

5.3.1 Software Phase

The software phase of the HENNC framework begins with generating a chaotic sequence dataset by numerically solving the defined chaotic system using the *Odeint* function from Python's *SciPy* library. We generate 100 k sequences and use 80% for training and 20% for testing. To create the training dataset, we sample the output of the numeric chaotic system at each time step. Since the output at time step t is the input to the system at time step $t + 1$, each labeled data point consists of the output samples from two consecutive time steps. The dataset in the initial phase is used to train the ANN model, which is built with the *Keras* library, and approximates the chaotic system's function through regression. Users specify model hyperparameters, and the model's performance is evaluated with the Mean Absolute Error (MAE), Mean Square Error (MSE), Root Mean Square Error (RMSE), and R-squared (R^2). Table 5.2 presents a hyperparameter set for training the model for Chen's chaotic system. The training process employs early stopping to prevent overfitting, terminating when the model achieves its best possible accuracy on the validation set. At this point, the network parameters are extracted for the hardware phase.

5.3.2 Hardware Phase

Design Space Exploration

Hardware design begins with the search for Pareto-optimal microarchitectures for the ANN model trained in the software phase. The HENNC framework incorporates a highly-configurable template HLS design with various model hyperparameters and a set of HLS directives that trade off parallelism and resource utilization. This flexibility allows the exploration of various ANN core microarchitecture.

HENNC offers users three options: minimum latency, lowest cost, and Pareto-optimal solutions with a parallelism level P. In the lowest cost solution ($P = 0$), the hardware core contains only one adder and one multiplier. Increasing parallelism reduces latency but increases costs. When $P \geq 1$, the number of multipliers and adders is $(2^P \times I)$, with I denoting the number of neurons in the input and output layers. For faster candidate solution characterization, HENNC incorporates a cost and throughput estimation method, described in the next subsection. Fig. 5.3 (a) depicts the architectural design space, with the estimated cost and latency for a 3-16-3 ANN.

The hardware cores support single-precision floating-point computations and, by default, map these operations onto FPGA DSP resources. Users have the option to utilize FPGA LUTs exclusively. Once the user selects the preferred candidate solution, the HENNC framework generates the corresponding chaotic oscillator HLS model.

Estimation functions

Pre-synthesis cost and speed estimation with HLS tools is time-consuming with limited accuracy. To tackle this issue, we developed cost and latency estimation functions with an experimental approach. To estimate latency, we began by measuring the post-synthesis actual latency, in terms of the number of clock cycles, for a range of selected solutions across various ANN sizes and parallelism levels. After normalizing the latency results, we computed the average latency for each parallelism level. By analyzing the average latency values using MATLAB Curve Fitter toolbox, we concluded that a third-degree polynomial approximation provides satisfactory accuracy. Hence, the resulting latency estimation function is as follows:

$$Latency = (I \cdot H) \cdot \left(b_3P^3 + b_2P^2 + b_1P + b_0\right), \tag{5.8}$$

where I is the number of neurons in the input/output layers, H is the number of neurons in the hidden layer, b_3 to b_0 are constant coefficients, and P is the parallelism level. Due

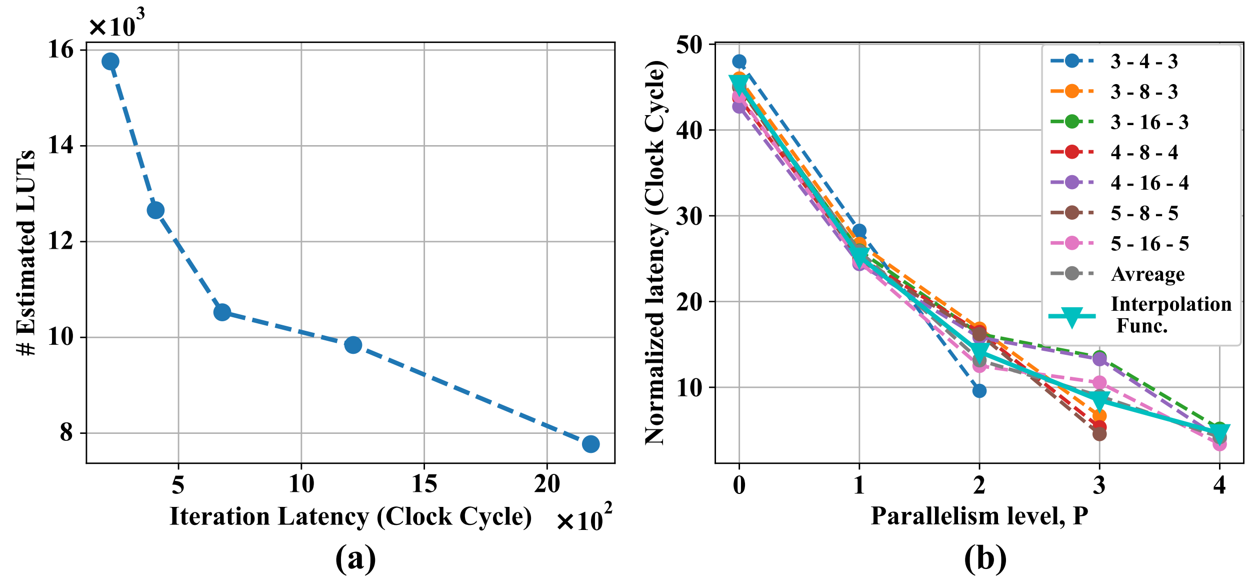


Figure 5.3 (a) Estimated cost and latency for the 3-16-3 ANN. (b) Normalized actual latencies and the interpolation curve when utilizing DSP resources.

to substantial variations in latency with or without DSP utilization, the measurements and interpolation calculations described above are carried out separately for the two scenarios, yielding distinct value sets for b_3 to b_0. Fig. 5.3 (b) presents the actual latency results when utilizing DSP resources, shown in terms of the number of clock cycles normalized by dividing them by $I \times H$. The MATLAB Curve Fitter toolbox was used to determine the interpolation function and the coefficients b_3 to b_0. This process was repeated separately for LUT-based solutions.

For the cost estimation, we generated and synthesized a range of solutions to gain insights into how LUT utilization varies with hyperparameter values and parallelism. LUT usage was measured as we incremented the H and the I while using two levels of parallelism. There is a semi-linear relationship between LUT utilization and both H and I. However, increasing the level of parallelism exerts a non-linear influence on the hardware core control circuit. we opted to express the estimation of LUT usage as a linear function of I and H for each parallelism level (P), employing the following formulation:

$$\#LUT = (c_1 \cdot I \cdot H) + (c_2 \cdot I) + (c_3 \cdot H) + \beta, \tag{5.9}$$

where c_1, c_2, c_3, and β are coefficients determined experimentally for each parallelism level.

These coefficients were established through experimentation for each unique parallelism level,

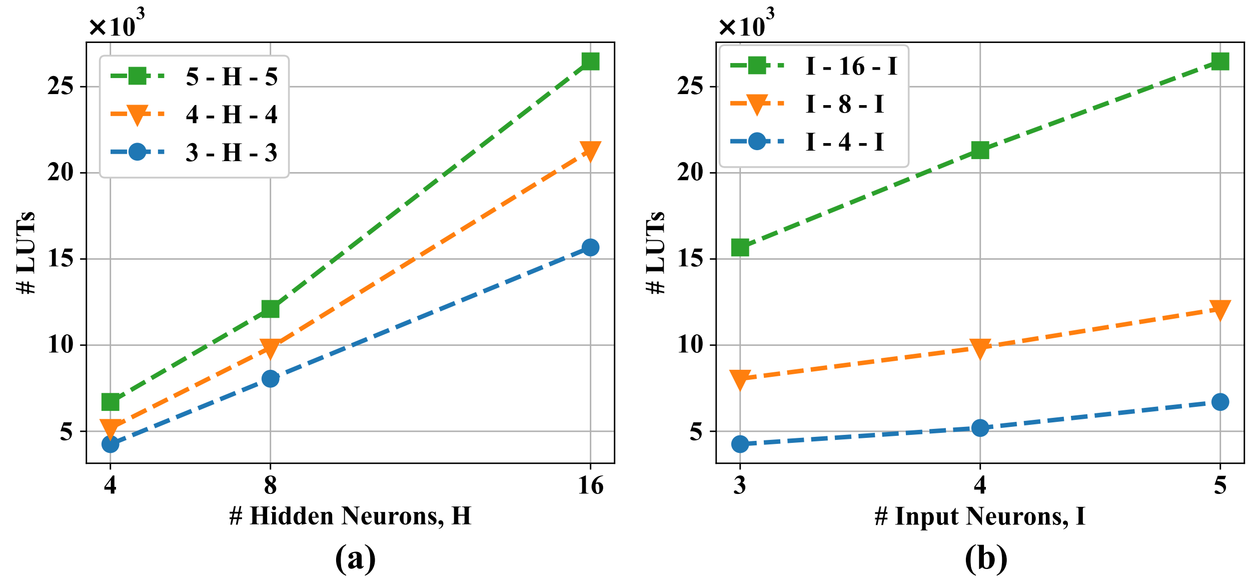


Figure 5.4 Post-synthesis number of LUTs as a function of the number of neurons in the (a) hidden layer, (b) input and output layers.

with a curve fitting tool. For every potential parallelism level, we conducted linear interpolation of LUT vs. H and LUT vs. I results from actual result curves, such as those depicted in Fig. 5.4. Subsequently, we determined the coefficient values by equating Eq. 5.9 with the linear interpolation results. The obtained coefficient values for various parallelism levels were then compiled to create a constant coefficient table. When assessing the LUT cost for a specific candidate architecture, the HENNC framework retrieves the c_1, c_2, c_3, and β values associated with the requested parallelism level from this table. These values are then incorporated into Eq. 5.9 to derive the estimation function with two variables, I and H.

Hardware core generation

In the final step, the HENNC framework generates the HLS design and verification codes for the user-selected solution. Two distinct C++ code files are generated: one for the synthesizable hardware design and another for the testbench to aid in design validation in AMD-Vitis HLS.

5.4 Results

We used AMD-Xilinx Vitis for HLS synthesis, AMD-Xilinx Vivado ML 2023.1 for RTL synthesis, targeting a xcku035 Kintex Ultrascale FPGA with a speed grade of -3. Table 5.3 lists the hardware costs for the three most commonly used ANN models for approximating 3-D chaotic systems. It shows that all HENNC solutions require significantly fewer LUTs and operate at considerably higher clock frequencies compared to [100,102]. In contrast to the architecture proposed by Alcin et al. [100], HENNC's fastest architecture offers reduced latency while consuming nearly 20× fewer LUTs. The other two HENNC solutions have increased latency but lower hardware costs. A primary reason for the substantial decrease in LUT utilization is the adoption of ReLU as an activation function. In contrast, prior works such as [102] employed the exponential function, and [100] utilized numerous floating-point dividers and CORDIC cores to approximate the activation function. For a 3-8-3 ANN, an Intel 12th Gen Intel(R) Core(TM) i7-12700 CPU generates 100 output samples in approximately 3.5 seconds, while the FPGA-based design accomplishes the same task in 31 microseconds.

Fig. 5.5 represents the HENNC design space for different ANN sizes with and without DSPs. The figure illustrates the trade-offs between post-synthesis LUT utilization and the latency offered by each candidate solution. For each ANN size, the smallest and largest solutions in terms of LUT usage represent the top-speed and cost-optimized solutions, respectively. The top-speed hardware leverages maximum parallelism by employing the maximum number of MAC operators. In contrast, the cost-optimized mode typically deploys only a single MAC unit to calculate all neurons.

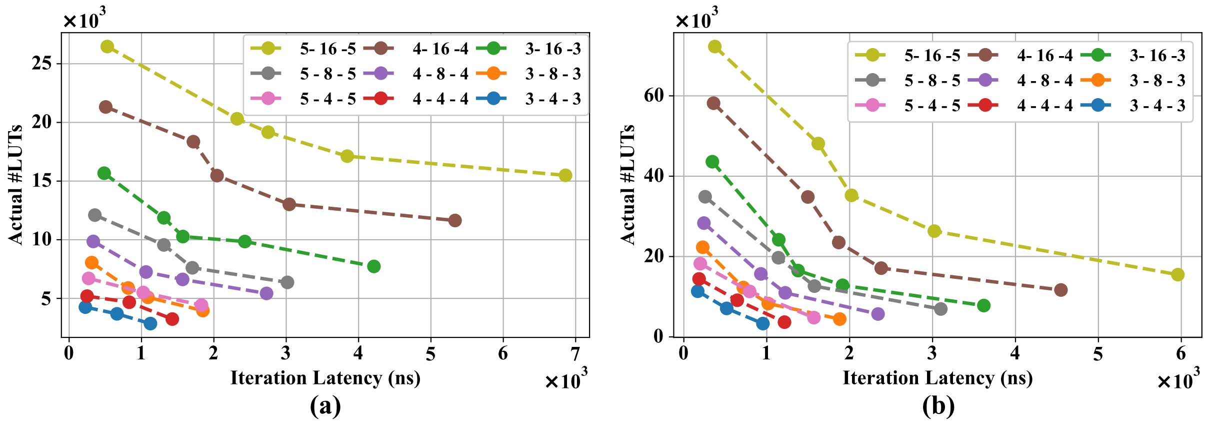


Figure 5.5 Post-synthesis hardware cost and latency of the design space explored by HENNC for different ANN sizes in two modes: (a) with DSP utilization, and (b) without DSP utilization.

Table 5.3 Hardware Costs for HENNC Candidate Designs and Existing Works.

HENNC Estimations								**Post-Synthesis**								
			LUTs		**DSPs**	**Iteration Latency (Clock Cycles)**		**LUTs**		**FFs**		**DSPs**	**fmax (MHz)**		**Iteration Latency (ns)**	
ANN	P	Solutions	With DSP	No DSP	With DSP	With DSP	No DSP	With DSP	No DSP	With DSP	No DSP	With DSP	With DSP	No DSP	With DSP	No DSP
3 – 4 – 3	2	1	4100	10708	44	169	128	4255	11358	8754	14909	44	510	548	224.25	167.44
	1	2	2288	5592	22	302	253	3683	7089	7383	10334	22	510	548	661.05	516.88
	0	3	2215	2896	5	544	500	2853	3314	5633	5735	5	510	548	1123.20	953.68
3 – 8 – 3	3	1	7988	21204	88	203	165	8051	22310	15020	27439	88	510	548	312.01	227.50
	2	2	5744	12352	44	339	256	5885	12279	9981	14210	44	494	387	816.08	717.24
	1	3	5180	8484	22	605	506	5089	8307	7944	10001	22	494	387	1087.83	1017.70
	0	4	4067	4721	5	1089	1001	3962	4440	6388	6485	5	494	386	1848.10	1880.34
3 – 16 – 3	4	1	15764	42196	176	223	290	15671	43604	28723	53538	176	510	548	483.60	343.98
	3	2	12656	25872	88	407	330	11868	24181	19592	27516	88	494	370	1308.96	1144.80
	2	3	10524	17132	44	678	513	10271	16511	14688	18338	44	494	387	1571.56	1377.72
	1	4	9841	13145	22	1211	1013	9846	12844	16818	19396	22	510	548	2427.75	1914.64
	0	5	7771	8425	5	2178	2003	7726	7795	15051	14759	5	510	548	4212.10	3618.16
3 – 4 – 3 [102]	-	1	-	-	-	-	-	21172	-	96	-	162	7	-	-	-
3 – 8 – 3 [100]	-	1	-	-	-	-	-	87207	-	86329	-	8	266	-	543.75	-

5.5 Conclusion

This paper introduced a framework for the rapid and automated generation of hardware cores for ANN-based chaotic oscillators, with a specific focus on FPGA implementation. The framework explores the hardware design space and offers a range of candidate hardware solutions that trade off hardware costs and throughput. The chpater also proposed novel cost and latency estimation functions. The results demonstrate that the proposed framework not only accelerates the hardware design process but also delivers candidate architectures that outperform previous works in terms of efficiency.

CHAPTER 6 GENERAL DISCUSSION

This thesis explores innovative approaches to enhancing hardware efficiency and optimizing deep learning models, particularly for implementations on specialized hardware accelerators using techniques like quantization and pruning. By focusing on applications requiring real-time processing and low power usage, this research addresses critical constraints for deploying AI on embedded and edge devices.

The research begins by introducing DyRecMul, a dynamically reconfigurable FPGA-based approximate multiplier tailored for efficient machine learning computations. Optimized for INT8 precision, DyRecMul achieves substantial reductions in hardware footprint and power consumption without compromising accuracy, essential for AI workloads that require rapid and low-cost computation. By exploiting dynamic reconfiguration, DyRecMul proves its effectiveness in maintaining high accuracy with significantly reduced hardware utilization, making it an efficient solution for machine learning and signal processing tasks on resource-constrained applications. The methodology centers around reconfigurable lookup tables, which allow dynamic configurations based on operational requirements, resulting in a high-performing architecture. Experimental results underscore DyRecMul's capability to act as an alternative to conventional multipliers, providing the necessary performance for low-power AI accelerators, particularly in updatable supervised learning tasks.

Subsequently, the thesis investigates pruning and quantization strategies to optimize deep learning models for AMR, a critical real-time signal processing task in various wireless communication systems. By applying structural pruning and dynamic range quantization, alongside developing a new CNN model that achieves an improved tradeoff between accuracy, model size, and processing latency, the research significantly reduces model complexity and computational demands, essential for devices with limited processing power and memory. The optimized models were benchmarked on GPUs using NVIDIA TensorRT, demonstrating substantial decreases in both computational load and memory usage while maintaining accuracy. This approach is particularly advantageous for AMR in edge devices, where real-time, cost-effective, and energy-efficient wireless signal recognition is crucial. The findings reveal that pruning and quantization are not only effective in reducing the model's footprint but also enable the deployment of complex AI tasks in resource-limited environments, advancing the potential of AI in real-time communications.

Lastly, the thesis presents HENNC, an automated platform for designing efficient hardware engines for Neural Network-based Chaotic Oscillators, tailored to implement chaotic systems for cryptographic applications. Traditionally, chaotic systems require complex numerical methods for accurate simulation; here, they are approximated using neural networks, significantly reducing computational and resource demands with minimal accuracy loss. HENNC provides a fast design and optimization process for the rapid creation of cost-effective and energy-efficient hardware cores for ANN-based chaotic random number generators, thereby advancing the use of neural networks in cryptographic and secure communication domains. Experimental results showed that this framework achieved an enhanced balance between latency and resource usage, making it an efficient and reliable solution for real-world cryptographic implementations. The HENNC approach not only reduced design time but also significantly improved the hardware efficiency of chaotic system models.

The three contributions in this thesis are interconnected by a shared goal of achieving efficiency through specialized hardware and optimized models, forming a progressive framework for hardware-aware AI. DyRecMul, a dynamically reconfigurable FPGA-based INT8 multiplier, provides a core building block that can directly support the quantized computations in subsequent works. When applying pruning and quantization for AMR in the second study, DyRecMul cores can be used to efficiently perform inference on edge devices, particularly as AMR models are reduced to INT8 precision, aligning well with DyRecMul's optimized capabilities. This synergy enables real-time signal recognition on power-limited devices, leveraging both DyRecMul's reconfigurability and the model compression techniques to maximize efficiency.

Furthermore, HENNC, an FPGA-based framework designed for neural network-driven chaotic oscillators, could also benefit from quantization support. By extending HENNC to work with INT8 data types, DyRecMul could be integrated as a high-efficiency computation core within the framework. This integration would streamline HENNC's resource usage on FPGAs, enhancing the practicality of deploying chaotic systems for cryptographic applications on constrained hardware. In this way, DyRecMul not only establishes a hardware foundation for quantized model efficiency but also creates a versatile multiplier that strengthens the deployment of both compressed neural networks and secure chaotic systems in edge and embedded environments. Together, these studies illustrate a unified path from low-level hardware optimizations to application-specific, high-efficiency AI deployments on specialized hardware.

Together, these contributions offer solutions to achieving higher efficiency in machine learning computations. This thesis demonstrates that combining customized hardware microarchitectures with model compression techniques, such as pruning and quantization, enables high accuracy and efficient deployment of AI models, even in resource-constrained environments. The research advances the field by validating that approximate computing and neural network-based solutions can serve as practical alternatives to traditional methods in applications requiring efficiency and adaptability. By contributing to approximate computing, DL model optimization, and automated ANN hardware accelerator optimization, this thesis supports more affordable deployment of AI techniques for real-time edge computing and introduces new insights for future innovations in hardware-aware AI implementations.

CHAPTER 7 CONCLUSION

This thesis has addressed the challenge of deploying high-performing deep learning models on hardware-constrained platforms by introducing specialized methods for model optimization and efficiency enhancement. Through a series of targeted techniques—ranging from reconfigurable hardware multipliers to model compression strategies—the work demonstrates how deep learning architectures can be adapted for real-time applications on specialized hardware, such as FPGAs and GPUs, without sacrificing performance or accuracy. Each contribution not only expands on existing techniques but also provides practical implementations and optimizations that bring advanced AI capabilities within reach for devices with limited computational resources. This conclusion summarizes the key findings and impacts of the work, addresses its limitations, and outlines future research directions to build upon these contributions.

7.1 Summary of Works

This thesis began by developing DyRecMul, a dynamically reconfigurable approximate multiplier tailored for low-power and high-efficiency AI applications on FPGAs. DyRecMul achieved a significant reduction in hardware resources and power consumption compared to standard multipliers, making it suitable for applications in which power constraints are critical. By allowing real-time adaptation through dynamic configuration, DyRecMul aligns well with the resource requirements of embedded AI systems, particularly in tasks like machine learning and digital signal processing.

The work then explored pruning and quantization techniques to enhance the efficiency of deep learning models used in AMR, a key component in wireless communication systems. Through these methods, the thesis demonstrated how deep learning models could be streamlined without compromising on accuracy, enabling efficient deployment on edge devices with limited memory and processing power. By implementing and testing these optimizations on GPUs with TensorRT, the research validated the potential for real-time, low-power AI in real-world communication and security applications.

Finally, the thesis presented HENNC, a Hardware Engine for Neural Network-based Chaotic Oscillators, designed to support cryptographic applications requiring high degrees of randomness and security. By approximating complex chaotic systems with neural networks, HENNC allows for efficient and reliable implementations on low-power hardware. This approach opens new possibilities for secure communication systems on constrained devices, combining the precision of traditional chaotic modeling methods with the efficiency and adaptability of neural networks.

7.2 Limitations

While this research presents notable advancements in efficient model deployment on constrained hardware, a closer analysis of each contribution reveals specific limitations and areas for improvement. DyRecMul leverages approximate computing for efficient multiplication on FPGAs, focusing primarily on INT8 precision. While this precision is widely used in machine learning accelerators, reliance on INT8 limits DyRecMul's versatility across applications requiring higher precision or diverse data types. Additionally, the dynamically reconfigurable approach, although optimized for speed and area, introduces complexities in reconfiguration overhead, which may hinder performance in applications that require frequent operand changes. Evaluating DyRecMul's adaptability to different FPGA architectures and its impact on latency-sensitive applications would further solidify its efficacy.

The optimization of low-complexity deep learning models for AMR through pruning and quantization demonstrated effective reductions in complexity, facilitating deployment on edge devices. However, the approach may not fully account for variations in communication environments, such as high-noise or dynamic channel conditions, which can adversely affect accuracy. Additionally, while quantization and pruning offer efficiency, achieving optimal performance under aggressive compression often requires extensive tuning. More adaptive or context-aware pruning techniques could potentially enhance model robustness, particularly in deployment scenarios with fluctuating resource availability and power constraints.

HENNC innovatively replaces traditional numerical methods for chaotic oscillation with ANN-based models, allowing for energy-efficient, low-cost implementations in cryptographic applications. However, the scalability of this approach to more complex chaotic systems, or systems requiring higher-dimensional representation, may pose challenges due to the growing demands on ANN capacity and computational resources. Furthermore, using neural networks to approximate chaos introduces questions about stability and sensitivity, particularly when the ANN encounters initial conditions it has not been trained on. Addressing these limitations would improve HENNC's applicability in more demanding or unpredictable real-world scenarios.

7.3 Future Research

Future research can expand on these findings in several directions. One avenue is to generalize DyRecMul to support a wider range of data types and applications, enabling broader use across varied AI and signal processing tasks. For pruning and quantization, exploring advanced techniques like mixed-precision quantization or adaptive pruning methods could further optimize model efficiency while preserving accuracy in resource-scarce settings. Additionally, future work on HENNC could investigate the scalability of neural network-based chaotic system models, potentially incorporating reinforcement learning or other adaptive methods to refine approximations dynamically. Furthermore, exploring the integration of these optimized models in IoT and edge devices with specific hardware accelerators could solidify the impact of this research, paving the way for real-time AI solutions in diverse fields, from telecommunications to secure data transmission and beyond.

REFERENCES

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

[2] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language processing (almost) from scratch,” *Journal of machine learning research*, vol. 12, pp. 2493–2537, 2011.

[3] A. Hannun, “Deep speech: Scaling up end-to-end speech recognition,” *arXiv preprint arXiv:1412.5567*, 2014.

[4] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 2015, pp. 234–241.

[5] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, “Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels,” in *2019 IEEE international conference on embedded software and systems (ICESS)*. IEEE, 2019, pp. 1–8.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[7] NVIDIA, “Nvidia deep learning accelerator (nvdla),” https://github.com/nvdla/hw, accessed: 2024-10-17.

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

[9] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, “Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 769–774.

[10] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” *Advances in neural information processing systems*, vol. 28, 2015.

[11] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1389–1397.

[12] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, “A white paper on neural network quantization,” *arXiv preprint arXiv:2106.08295*, 2021.

[13] 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*, 2018, pp. 2704–2713.

[14] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[15] S. Sadoudi, M. S. Azzaz, M. Djeddou, and M. Benssalah, “An fpga real-time implementation of the chen’s chaotic system for securing chaotic communications,” *International Journal of Nonlinear Science*, vol. 7, no. 4, pp. 467–474, 2009.

[16] K. Lamamra, S. Vaidyanathan, A. T. Azar, and C. Ben Salah, “Chaotic system modelling using a neural network with optimized structure,” *Fractional Order Control and Synchronization of Chaotic Systems*, pp. 833–856, 2017.

[17] S. Vakili, M. Vaziri, A. Zarei, and J. P. Langlois, “Dyrecmul: Fast and low-cost approximate multiplier for fpgas using dynamic reconfiguration,” *ACM Transactions on Reconfigurable Technology and Systems*, 2024.

[18] M. Vaziri, S. Vakili, and J. P. Langlois, “Accuracy-aware low-complexity deep learning models for automatic modulation recognition,” in *2024 International Conference on Computing, Internet of Things and Microwave Systems (ICCIMS)*. IEEE, 2024, pp. 1–5.

[19] M. Vaziri, S. Vakili, M. M. Rahimifar, and J. P. Langlois, “HENNC: Hardware Engine for Artificial Neural Network-Based Chaotic Oscillators,” in *Proceedings of the 28th Annual IEEE High Performance Extreme Computing Virtual Conference (HPEC), accepted*, Virtual Conference, September 2024, accepted for publication.

[20] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

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

[22] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[23] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks," *arXiv preprint arXiv:2104.08378*, 2021.

[24] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[25] AMD-Xilinx, "7 series fpgas configurable logic block," available from: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[26] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of atmospheric sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[27] T. Stojanovski and L. Kocarev, "Chaos-based random number generators-part i: analysis [cryptography]," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 281–288, 2001.

[28] M. L. Barakat, A. S. Mansingka, A. G. Radwan, and K. N. Salama, "Generalized hardware post-processing technique for chaos-based pseudorandom number generators," *ETRI journal*, vol. 35, no. 3, pp. 448–458, 2013.

[29] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[30] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488873

[31] Y. Wu, C. Chen, W. Xiao, X. Wang, C. Wen, J. Han, X. Yin, W. Qian, and C. Zhuo, "A survey on approximate multiplier designs for energy efficiency: From algorithms to circuits," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 1, jan 2024. [Online]. Available: https://doi.org/10.1145/3610291

[32] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, Aug 1962.

[33] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, 2018.

[34] M. S. Ansari, B. F. Cockburn, and J. Han, "An improved logarithmic multiplier for energy-efficient neural computing," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 614–625, 2020.

[35] ——, "A hardware-efficient logarithmic multiplier with improved accuracy," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2019, pp. 928–931.

[36] H. Saadat, H. Bokhari, and S. Parameswaran, "Minimally biased multipliers for approximate integer and floating-point multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2623–2635, 2018.

[37] M. Imani, A. Sokolova, R. Garcia, A. Huang, F. Wu, B. Aksanli, and T. Rosing, "Approxlp: Approximate multiplication with linearization and iterative error control," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317774

[38] C. Chen, S. Yang, W. Qian, M. Imani, X. Yin, and C. Zhuo, "Optimally approximated and unbiased floating-point multiplier with runtime configurability," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415702

[39] M. Imani, D. Peroni, and T. Rosing, "Cfpu: Configurable floating point multiplier for energy-efficient computing," in *Proceedings of the 54th Annual Design Automation*

Conference 2017, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3061639.3062210

[40] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "Rmac: Runtime configurable floating point multiplier for approximate computing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3218603.3218621

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

[42] S. Vahdat, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Tosam: An energy-efficient truncation-and rounding-based scalable approximate multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 5, pp. 1161–1173, 2019.

[43] 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*, ser. ICCAD '15. IEEE Press, 2015, p. 418–425.

[44] S. Vahdat, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Letam: A low energy truncation-based approximate multiplier," *Computers & Electrical Engineering*, vol. 63, pp. 1–17, 2017.

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

[46] S. Venkatachalam and S.-B. Ko, "Design of power and area efficient approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1782–1786, 2017.

[47] T. Yang, T. Ukezono, and T. Sato, "A low-power high-speed accuracy-controllable approximate multiplier design," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2018, pp. 605–610.

[48] ——, "Low-power and high-speed approximate multiplier design with a tree compressor," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 89–96.

[49] C. S. Wallace, “A suggestion for a fast multiplier,” *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, Feb 1964.

[50] L. Dadda, “Some schemes for parallel multipliers,” *Alta frequenza*, vol. 34, pp. 349–356, 1965.

[51] D. Esposito, A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra, “Approximate multipliers based on new approximate compressors,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4169–4182, 2018.

[52] M. Ha and S. Lee, “Multipliers with approximate 4–2 compressors and error recovery modules,” *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 6–9, 2018.

[53] C.-W. Tung and S.-H. Huang, “Low-power high-accuracy approximate multiplier using approximate high-order compressors,” in *2019 2nd International Conference on Communication Engineering and Technology (ICCET)*, April 2019, pp. 163–167.

[54] Z. Yang, J. Han, and F. Lombardi, “Approximate compressors for error-resilient multiplier design,” in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 183–186.

[55] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, “Design and analysis of approximate compressors for multiplication,” *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2014.

[56] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, “Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, 2009.

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

[58] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, “Dual-quality 4: 2 compressors for utilizing in dynamic accuracy configurable multipliers,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1352–1361, 2017.

[59] M. Ahmadinejad, M. H. Moaiyeri, and F. Sabetzadeh, “Energy and area efficient imprecise compressors for approximate multiplication at nanoscale,” *AEU-International Journal of Electronics and Communications*, vol. 110, p. 152859, 2019.

[60] A. G. M. Strollo, E. Napoli, D. De Caro, N. Petra, and G. Di Meo, "Comparison and extension of approximate 4-2 compressors for low-power approximate multipliers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3021–3034, 2020.

[61] R. Marimuthu, Y. E. Rezinold, and P. S. Mallick, "Design and analysis of multiplier using approximate 15-4 compressor," *IEEE Access*, vol. 5, pp. 1027–1036, 2016.

[62] X. Wang and W. Qian, "Minac: Minimal-area approximate compressor design based on exact synthesis for approximate multipliers," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2022, pp. 677–681.

[63] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, "Area-optimized low-latency approximate multipliers for fpga-based hardware accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3195996

[64] S. Ullah, S. S. Murthy, and A. Kumar, "Smapproxlib: library of fpga-based approximate multipliers," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3196115

[65] N. Van Toan and J.-G. Lee, "Fpga-based multi-level approximate multipliers for high-performance error-resilient applications," *IEEE Access*, vol. 8, pp. 25 481–25 497, 2020.

[66] M. Kumm, K. Möller, and P. Zipf, "Dynamically reconfigurable fir filter architectures with fast reconfiguration," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, July 2013, pp. 1–8.

[67] Y. Lin, Y. Li, T. Liu, T. Xiao, T. Liu, and J. Zhu, "Towards fully 8-bit integer inference for the transformer model," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, ser. IJCAI'20, 2021.

[68] S. Kim, G. Park, and Y. Yi, "Performance evaluation of int8 quantized inference on mobile gpus," *IEEE Access*, vol. 9, pp. 164 245–164 255, 2021.

[69] Xilinx, "Ultrascale architecture configurable logic block user guide (ug574)," 2017.

[70] Z. Vasicek, "Formal methods for exact analysis of approximate circuits," *IEEE Access*, vol. 7, pp. 177 309–177 331, 2019.

[71] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. Leuven, BEL: European Design and Automation Association, 2014.

[72] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris, and J. Henkel, "Adapt: Fast emulation of approximate dnn accelerators in pytorch," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 2074–2078, June 2023.

[73] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, "Design of approximate radix-4 booth multipliers for error-tolerant computing," *IEEE Transactions on computers*, vol. 66, no. 8, pp. 1435–1441, 2017.

[74] M. S. Ansari, H. Jiang, B. F. Cockburn, and J. Han, "Low-power approximate multipliers using encoded partial products and approximate compressors," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 8, no. 3, pp. 404–416, 2018.

[75] H. Jiang, C. Liu, F. Lombardi, and J. Han, "Low-power approximate unsigned multipliers with configurable error recovery," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 1, pp. 189–202, 2018.

[76] T. Erpek, T. J. O'Shea, Y. E. Sagduyu, Y. Shi, and T. C. Clancy, "Deep learning for wireless communications," *Development and Analysis of Deep Learning Architectures*, pp. 223–266, 2020.

[77] S. Dörner, S. Cammerer, J. Hoydis, and S. t. Brink, "Deep learning based communication over the air," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 132–143, 2018.

[78] H. Ye, G. Y. Li, B.-H. F. Juang, and K. Sivanesan, "Channel agnostic end-to-end learning based communication systems with conditional gan," in *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018, pp. 1–5.

[79] F. Zhang, C. Luo, J. Xu, Y. Luo, and F.-C. Zheng, "Deep learning based automatic modulation recognition: Models, datasets, and challenges," *Digital Signal Processing*, vol. 129, p. 103650, 2022.

[80] B. Dulek, "Online hybrid likelihood based modulation classification using multiple sensors," *IEEE Transactions on Wireless Communications*, vol. 16, no. 8, pp. 4984–5000, 2017.

[81] A. Hazza, M. Shoaib, S. A. Alshebeili, and A. Fahad, “An overview of feature-based methods for digital modulation classification,” in *2013 1st International Conference on Communications, Signal Processing, and their Applications (ICCSPA)*, 2013, pp. 1–6.

[82] T. J. O’Shea, J. Corgan, and T. C. Clancy, “Convolutional radio modulation recognition networks,” in *Engineering Applications of Neural Networks: 17th International Conference, EANN 2016, Aberdeen, UK, September 2-5, 2016, Proceedings 17.* Springer, 2016, pp. 213–226.

[83] K. Tekbıyık, A. R. Ekti, A. Görçin, G. K. Kurt, and C. Keçeci, “Robust and fast automatic modulation classification with cnn under multipath fading channels,” in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, 2020, pp. 1–6.

[84] S. Rajendran, W. Meert, D. Giustiniano, V. Lenders, and S. Pollin, “Deep learning models for wireless signal classification with distributed low-cost spectrum sensors,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 3, pp. 433–445, 2018.

[85] N. E. West and T. O’shea, “Deep architectures for modulation recognition,” in *2017 IEEE international symposium on dynamic spectrum access networks (DySPAN).* IEEE, 2017, pp. 1–6.

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

[87] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[88] K. M. S. Huq, S. A. Busari, J. Rodriguez, V. Frascolla, W. Bazzi, and D. C. Sicker, “Terahertz-enabled wireless system for beyond-5g ultra-fast networks: A brief survey,” *IEEE Network*, vol. 33, no. 4, 2019.

[89] S. H. Strogatz, *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering.* CRC press, 2018.

[90] Z. Hua, Y. Zhou, and H. Huang, “Cosine-transform-based chaotic system for image encryption,” *Information Sciences*, vol. 480, pp. 403–419, 2019.

[91] R. Hamza, Z. Yan, K. Muhammad, P. Bellavista, and F. Titouna, "A privacy-preserving cryptosystem for iot e-healthcare," *Information Sciences*, vol. 527, pp. 493–510, 2020.

[92] S. Kalanadhabhatta, D. Kumar, K. K. Anumandla, S. A. Reddy, and A. Acharyya, "Puf-based secure chaotic random number generator design methodology," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 28, no. 7, pp. 1740–1744, 2020.

[93] L. Skanderova and A. Řehoř, "Comparison of pseudorandom numbers generators and chaotic numbers generators used in differential evolution," in *Nostradamus 2014: Prediction, Modeling and Analysis of Complex Systems.* Springer, 2014, pp. 111–121.

[94] M. N. Aslam, A. Belazi, S. Kharbech, M. Talha, and W. Xiang, "Fourth order mca and chaos-based image encryption scheme," *IEEE Access*, vol. 7, pp. 66 395–66 409, 2019.

[95] M. S. Azzaz, C. Tanougast, S. Sadoudi, R. Fellah, and A. Dandache, "A new auto-switched chaotic system and its fpga implementation," *Communications in Nonlinear Science and Numerical Simulation*, vol. 18, no. 7, pp. 1792–1804, 2013.

[96] K. Rajagopal, M. Tuna, A. Karthikeyan, İ. Koyuncu, P. Duraisamy, and A. Akgul, "Dynamical analysis, sliding mode synchronization of a fractional-order memristor hopfield neural network with parameter uncertainties and its non-fractional-order fpga implementation," *European Physical Journal Special Topics*, vol. 228, pp. 2065–2080, 2019.

[97] M. Alcin, "The runge kutta-4 based 4d hyperchaotic system design for secure communication applications," *Chaos Theory and Applications*, vol. 2, no. 1, pp. 23–30, 2020.

[98] M. Tuna, M. Alçın, İ. Koyuncu, C. B. Fidan, and İ. Pehlivan, "High speed fpga-based chaotic oscillator design," *Microprocessors and Microsystems*, vol. 66, pp. 72–80, 2019.

[99] L. Zhang, "Artificial neural network model design and topology analysis for fpga implementation of lorenz chaotic generator," in *Canadian Conf. on Electrical and Computer Engineering.* IEEE, 2017, pp. 1–4.

[100] M. Alçın, İ. Pehlivan, and İ. Koyuncu, "Hardware design and implementation of a novel ann-based chaotic generator in fpga," *Optik*, vol. 127, no. 13, pp. 5500–5505, 2016.

[101] J. Sunny, J. Schmitz, and L. Zhang, "Artificial neural network modelling of rossler's and chua's chaotic systems," in *IEEE Canadian Conference on Electrical & Computer Engineering.* IEEE, 2018, pp. 1–4.

[102] W. A. Al-Musawi, W. A. Wali, and M. A. A. Al-Ibadi, "New artificial neural network design for chua chaotic system prediction using fpga hardware co-simulation," *International Journal of Electrical and Computer Engineering*, vol. 12, no. 2, p. 1955, 2022.

[103] F. Yu, Z. Zhang, H. Shen, Y. Huang, S. Cai, J. Jin, and S. Du, "Design and fpga implementation of a pseudo-random number generator based on a hopfield neural network under electromagnetic radiation," *Frontiers in Physics*, vol. 9, 2021.

[104] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert *et al.*, *A statistical test suite for random and pseudorandom number generators for cryptographic applications.* US Department of Commerce, Technology Administration, National Institute of . . . , 2001, vol. 22.