Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications

Yohannes Melese Batisawi

A Thesis Submitted to

Department of Electronics and Communication Engineering,

College of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the Master of Science Degree in Electronics and Communication Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2025

Adama, Ethiopia

Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications

Yohannes Melese Batisawi

Major advisor: **Dr. Avtar Singh**

Co-Advisor: **Mr. Tadesse Hailu**

A Thesis Submitted to the Department of Electronics and Communication Engineering,

College of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the Master of Science Degree in Electronics and Communication Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2025

Adama, Ethiopia

DECLARATION

I hereby declare that this Master Thesis entitled "**Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications**" is my own work. That is, it has not been submitted for the award of any academic degree, diploma, or certificate in any other university. All sources of materials that are used for this thesis have been duly acknowledged through citation.

_____	_____	_____
Name of Student	Signature	Date

RECOMMENDATION

We, the advisors of this thesis, hereby certify that we have read the revised version of the thesis entitled "**Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications**" prepared under our guidance by Yohannes Melese Batisawi submitted in partial fulfillment of the requirements for the degree of Master of Science in Electronics and Communication Engineering. Therefore, we recommend the submission of the revised version of the thesis to the department following the applicable procedures.

_____	_____	_____
Major Advisor name	Signature	Date
_____	_____	_____
Co-advisor name	Signature	Date

APPROVAL PAGE OF M.SC. THESIS

We, the advisors of the thesis entitled "**Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications**" and developed by Yohannes Melese Batisawi, hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis.

Major Advisor	Signature	Date
Co-advisor	Signature	Date

We, the undersigned, members of the Board of Examiners of the thesis by Yohannes Melese Batisawi, have read and evaluated the thesis entitled "**Design and Implementation of FPGA-Based Ensemble Neural Networks for Real-Time Applications**" and examined the candidate during open defense. This is, therefore, to certify that the thesis is accepted for partial fulfillment of the requirement of the degree of Master of Science in Electronics and Communication Engineering.

Chairperson	Signature	Date
Internal Examiner	Signature	Date
External Examiner	Signature	Date

Finally, approval and acceptance of the thesis are contingent upon the submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

Department Head	Signature	Date
College Dean	Signature	Date
Office of Postgraduate Studies, Dean	Signature	Date

ACKNOWLEDGMENT

With thoughtful gratitude I acknowledge the many blessings and support that have made this work possible. I am deeply thankful to the almighty God for granting me perseverance, insight, and resilience at every turn of this journey.

I extend my heartfelt appreciation to my advisors, Dr. Avtar Singh and Mr. Tadesse Hailu. Dr. Avtar's constant encouragement and valuable insights have inspired me to reach higher standards. Mr. Hailu's generous guidance and constructive feedback have shaped this research in countless ways. Their combined wisdom and sincerity have been indispensable.

My sincere thanks go to my family for their unwavering love. To my parents, thank you for teaching me the importance of dedication and faith. Your steady confidence in my abilities never wavered, and your sacrifices have carried me forward. To my sisters, thank you for your light hearted reminders of joy and balance. Your support at home and in spirit has been a source of strength.

I am especially grateful to my friends, whose compassion and laughter have brightened the toughest days. Your words of encouragement lifted my spirits and reminded me why this work matters. In particular I wish to acknowledge Abinet Bekele. Her thoughtful advice, listening ear, and persistent belief in me have been a blessing I could always count on.

Finally, I wish to recognize all the faculty, staff, and classmates who offered their time and expertise, asked insightful questions, or simply shared a moment of solidarity along the way. Your generosity, collaboration, and kindness have enriched my experience and made this achievement truly meaningful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acronyms

NN	Neural Network
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
FPGA	Field-Programmable Gate Array
LUT	Look-Up Table
BRAM	Block RAM
SVM	Support Vector Machine
k-NN	k-Nearest Neighbors
IoT	Internet of Things
VHDL	VHSIC Hardware Description Language

ABSTRACT

The demand for deploying neural networks (NNs), particularly Convolutional Neural Networks (CNNs), in real-time, resource-constrained environments is rapidly increasing. However, deploying CNNs on IoT devices, robotics, and autonomous systems presents significant challenges due to their high computational and memory requirements. This thesis presents the design and implementation of a general, modular, and reconfigurable FPGA-based framework for ensemble CNN inference. The system integrates multiple lightweight CNNs using voting and weighted averaging ensemble strategies to improve predictive accuracy and robustness. Hardware-specific optimizations, including L1 unstructured pruning, fixed-point quantization (to 16-bit and 24-bit formats), and pipelined module design are employed to reduce resource usage and enhance performance. The architecture is implemented in Verilog and synthesized using the Xilinx Vivado Design Suite, targeting energy-efficient real-time inference on embedded FPGAs. Evaluation is conducted on two digit recognition tasks: the culturally significant Geez (Amharic) Handwritten Digit Dataset and the widely benchmarked MNIST dataset. The proposed system achieves ensemble inference accuracies of 93.53% on Geez and 99.11% on MNIST, with per-image inference latency as low as 31 μs and throughput exceeding 32,000 frames per second at 50 MHz. Dynamic power consumption is constrained to as low as 0.414 W for a single model and 1.267 W for a three-model ensemble, making the design highly efficient for edge deployment. Benchmarking against CPU and GPU implementations as well as other FPGA accelerators shows superior energy efficiency and real-time performance. The results demonstrate that the proposed FPGA framework effectively balances accuracy, latency, and resource efficiency, offering a scalable solution for deploying ensemble neural networks in constrained environments.

Keywords: FPGA, Ensemble Modeling, Verilog, Model Quantization, Parallel Processing, Neural Network Inference

CHAPTER 1 INTRODUCTION

1.1 Background of the Study

Neural networks (NNs) are computational models inspired by the human brain. They solve complex problems by learning and adapting to data. They mimic the way neurons process and transmit information, enabling machines to learn patterns from data without explicit programming. NNs have revolutionized modern technology, addressing societal challenges such as early disease detection, autonomous driving, and natural disaster prediction.

Particularly, Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) have excelled in various domains. ANNs, with their interconnected nodes and layers, are widely used for tasks such as financial forecasting and speech recognition. CNNs, a specialized type of ANN designed for image and video data, have significantly improved fields like medical imaging and facial recognition. By extracting hierarchical representations from complex, high-dimensional data, CNNs and ANNs outperform traditional machine learning techniques, such as decision trees, support vector machines (SVMs), or k-nearest neighbors (k-NN), bringing transformative changes to diverse applications (Jouppi et al., 2017).

Neural network architectures, such as VGG-16 (Visual Geometry Group), ResNet-50 (Residual Networks), and Transformer-based models, have achieved remarkable advancements but come with significant computational and memory demands, making it difficult to deploy effectively. Such demands stem from the millions of parameters and deep-layer stacks that require advanced hardware and significant energy resources for training and inference (Liu et al., 2017). Yet these same improvements in model accuracy and functionality have escalated energy consumption and deployment costs. Meeting the real-time performance requirements in resource-constrained environments, such as autonomous systems and IoT devices, remains a critical challenge.

Addressing these computational and memory demands requires hardware accelerators capable of high throughput. While modern CPUs provide multiple cores and vectorized instructions, their architecture is not optimized for the massive parallelism inherent in neural networks. Graphics processing units (GPUs), which excel at both training and inference, deliver significantly higher performance but suffer from high power consumption, substantial heat generation, and elevated cost. These drawbacks make GPUs less practical for edge and

embedded deployments, where processing must occur locally and energy efficiency is paramount (Lu, 2024).

Field-programmable gate arrays (FPGAs) offer a promising alternative for neural network deployment, delivering fine-grained parallelism, reconfigurability, and much lower power consumption than GPUs. These strengths help overcome high computation and memory demands while improving energy efficiency, making FPGAs ideal for real-time inference on edge and embedded platforms. Moreover, their capacity for reconfiguration allows rapid adaptation to new algorithms, ensuring continued relevance in rapidly evolving contexts (Shawahna et al., 2019).

In addition to hardware innovations, algorithmic techniques such as model ensembling have enhanced neural network performance and reliability. By combining predictions from multiple smaller models, ensembles improve accuracy, robustness, and generalization through model diversity, which mitigates overfitting and reduces sensitivity to noise. However, these benefits come with increased computational and memory demands during inference. Compared to a single large network, an ensemble of smaller, efficient models can achieve superior performance when working collaboratively.

1.2 Statement of the Problem

Neural networks have transformed fields like image classification, speech recognition, and natural language processing. Yet as architectures have grown deeper (e.g., VGG-16, ResNet, Transformer models), they now contain millions of parameters. This scale increases computational complexity, memory footprint, and power consumption, making real-time deployment in resource-limited settings especially difficult.

To address these constraints, hardware accelerators are often used. Graphics processing units (GPUs) offer high throughput for both training and inference, but their substantial power draw, cooling requirements, and cost prevent practical use in edge or embedded deployments. Consequently, alternatives that maintain performance while reducing energy and cost overheads are needed.

Field-programmable gate arrays (FPGAs) offer hardware-level parallelism, reconfigurability, and energy efficiency. These attributes make FPGAs well suited for deploying lightweight neural networks, but large models often exceed their on-chip memory and resource limits.

Excessive data movement between on-chip buffers and external memory also introduces latency and increases power consumption. To mitigate these constraints, FPGA frameworks must optimize resource allocation and fully utilize parallel processing. Even so, scaling to larger architectures remains challenging due to limited FPGA resources and the complexity of deep networks. Combining small, efficient models in an ensemble with FPGA parallelism provides a promising solution. Such frameworks can maintain high accuracy while balancing power efficiency and real-time response.

The challenge lies in achieving a balance between accuracy, power efficiency, latency, and scalability when deploying advanced NN models. While existing hardware accelerators struggle with the increasing complexity of modern NN architectures, FPGAs offer unique capabilities to address these issues. This study focuses on combining the efficiency of FPGA's parallel processing capabilities with the compact and collaborative nature of ensemble models. By leveraging these strengths, the research aims to develop a scalable, energy-efficient framework tailored for real-time NN inference, ensuring optimal performance in resource-constrained environments.

1.3 Objectives

1.3.1 General Objective

To design and implement an FPGA-based hardware implementation that integrates ensemble Neural Network models, focusing on optimizing for energy efficiency, computational performance, and accuracy in real-time applications.

1.3.2 Specific Objectives

- Select the best suitable ensemble technique and use pre-trained Neural Network models with the appropriate modifications.
- Implement hardware-specific model compression techniques to reduce resource usage while maintaining model accuracy.
- Develop a resource-efficient hardware architecture model on FPGA to support ensemble Neural Network models, focusing on real-time applications.
- Optimize the inference latency and throughput ensuring optimal performance in real-time ensemble models.
- Evaluate the performance of the proposed system using benchmarks on accuracy, power consumption, and latency in real-world application scenarios.

1.4 Significance of the Study

By employing FPGA-based ensembles of lightweight neural networks, this research delivers effective deployment strategies for real-time inference in contexts where CPUs and GPUs prove inefficient. The proposed framework enables low-latency, energy-efficient performance for applications such as autonomous systems, IoT devices, and robotics.

Building on ensemble modeling, this research develops techniques to tailor ensemble architectures for FPGA implementation, addressing memory constraints, computational load, and inference latency. Additionally, it advances AI hardware design by combining architecture-level optimizations, such as model pruning, and dynamic reconfiguration strategies. As a result, these contributions guide the development of scalable, energy-efficient systems for real-time deep learning on constrained platforms.

Ultimately, this research offers transformative insights into deploying NN models in constrained environments, paving the way for advancements in critical domains like autonomous navigation, industrial automation, and real-time data analysis.

1.5 The Scope of the Study

This study focuses on the design and implementation of ensemble neural network models for inference on FPGAs. The application of ensemble methods to ANNs and CNNs are investigated, given their efficiency and suitability for various real-world applications. More complex architectures, such as recurrent neural networks (RNNs) and transformer-based models, are excluded due to their computational intensity and increased resource demands, which lay outside the intended scope of this work.

The main emphasis remains on inference. Targeted modifications to the training process, performed entirely in software, aligned individual models with the constraints and requirements of the FPGA-based inference stage. An in-depth exploration of alternative training methodologies was kept beyond the primary scope of the study.

1.6 Research Contribution

The core contribution of this thesis is the design and implementation of a novel, FPGA-based hardware framework specifically tailored for deploying ensemble Neural Network models in real-time, resource-constrained environments. The key contributions are:

- **Novel FPGA Framework:** Development of a modular, reconfigurable FPGA architecture that effectively integrates multiple lightweight neural networks for ensemble inference, enabling flexible adaptation to varying application demands.
- **Optimized Ensemble Integration:** Implementation of hardware-level ensemble techniques such as voting and weighted averaging, which improve accuracy and robustness without incurring prohibitive computational or memory overhead.
- **Resource-Efficient Hardware Optimizations:** Integration of advanced hardware optimization techniques, including L1 unstructured pruning and fixed-point quantization, significantly reducing resource utilization, latency, and power consumption.
- **Demonstrated Real-Time Efficiency:** Comprehensive evaluation demonstrating the framework's capability to achieve high accuracy, low latency, and high throughput, surpassing traditional CPU/GPU implementations and existing FPGA accelerators, validated using benchmark datasets including Geez (Amharic) and MNIST.

This research thus provides a scalable, energy-efficient solution that bridges the gap between computational performance and resource constraints, significantly advancing the practical deployment of neural network ensembles on edge platforms.

1.7 Thesis Organization

The thesis is structured as follows:

- Chapter 2: Literature Review surveys existing research on NN architectures, hardware accelerators, and ensemble modeling, identifying the gap in integrated FPGA-ensemble solutions.
- Chapter 3: Methodology and System Design details the design of the FPGA-based framework, including ensemble techniques, model compression, and hardware implementation using Verilog and Xilinx Vivado.
- Chapter 4: Results and Discussion presents experimental results, including accuracy improvements and hardware metrics, alongside trade-offs and limitations.
- Chapter 5: Conclusion and Recommendation summarizes findings and suggests future work.

CHAPTER 2 LITERATURE REVIEW

2.1 Overview

This chapter surveys prior work on hardware-accelerated neural networks and related optimizations. Section 2.2 presents neural network fundamentals, covering architectures, training, inference, and applications. Section 2.3 reviews hardware platforms for inference, including CPUs, GPUs, and specialized accelerators. Section 2.4 examines FPGA technology and existing FPGA-based neural network implementations. Section 2.5 describes model compression methods such as pruning and quantization. Section 2.6 overviews ensemble modeling approaches. Section 2.7 lists common tools and frameworks used in FPGA and neural network development. Section 2.8 identifies gaps in current research and motivates a combined FPGA-ensemble approach

2.2 Neural Networks and Their Applications

Artificial neural networks (ANNs) are composed of interconnected layers of artificial neurons that adjust connection weights during training (Ian Goodfellow et al., 2016). Convolutional neural networks (CNNs) are a type of ANN designed for grid-like data, such as images. CNNs use convolutional layers to apply filters, pooling layers to reduce dimensionality, and fully connected layers for classification or regression (Rawat & Wang, 2017).

2.2.1 Neural Network Architecture

An artificial neuron (also called a node or unit) is the basic computational building block of a neural network. Each neuron receives one or more input signals, computes a weighted sum plus a bias, and passes the result through a nonlinear activation function before sending an output to the next layer (Ian Goodfellow et al., 2016). Mathematically:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$a = \phi(z)$$

Here, $x_1, x_2, \ldots, x_n$ are the input features, $w_1, w_2, \ldots, w_n$ are the corresponding weights, b is the bias term, z is the linear combination of inputs and weights, and $\phi(z)$ is the activation function that introduces non-linearity into the neuron's output (Ian Goodfellow et al., 2016).

In addition to the basic structure of artificial neurons, CNNs extend the neural network architecture to efficiently process grid-like data structures. Unlike fully connected networks,

CNNs utilize convolutional layers to extract local patterns from the grid data using learnable filters. The operation of a convolutional layer is defined by a kernel or filter that slides over the input and computes dot products at each spatial location (Rawat & Wang, 2017). Mathematically, the 2D convolution operation for an input image I and a filter K is represented as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$$

where S(i, j) is the output feature map, and the summation is performed over the dimensions of the kernel K (Lecun et al., n.d.).

Following the convolution operation, pooling layers are used to progressively reduce the spatial dimensions (width and height) of the feature maps, thereby lowering computational cost and introducing translational invariance. The most common type is max pooling, which selects the maximum value within a local region of the feature map. The max pooling operation over a region R is given by:

$$P(i,j) = \max_{(m,n)\in\mathcal{R}} S\,(i+m, j+n)$$

where P(i, j) is the pooled output and R defines the receptive field (e.g., 2×2 window) (Ian Goodfellow et al., 2016).

2.2.2 Learning and Inference in Neural Networks

The operation of neural networks, including Convolutional Neural Networks (CNNs), can be broadly categorized into two distinct phases: learning (training) and inference. Each plays a critical role in the development and deployment of intelligent systems, yet they differ significantly in terms of computational complexity, resource requirements, and execution time.

2.2.2.1 ***Learning (Training) Process***

The training process is the phase during which a neural network learns to map inputs to outputs by optimizing its internal parameters (weights and biases). Training involves feeding labelled data into the network, computing the loss (error) between the predicted output and the ground truth, and then adjusting the model parameters to minimize this loss. This adjustment is performed using backpropagation and gradient descent (Ian Goodfellow et al., 2016).

2.2.2.2 ***Inference Process***

Inference is the deployment phase where the trained model is used to make predictions on new, unseen data. In this phase, the model performs only a forward pass through the network. The

input is passed through the layers convolutional, pooling, and fully connected to produce an output, such as a classification label or probability distribution (LeCun et al., 2015).

$$\hat{y} = f(x; \theta^*)$$

Here, x is the input data and θ^* represents the learned (frozen) weights and biases from the training phase. Since there is no backpropagation or parameter updating during inference, this phase is significantly faster and more resource-efficient than training.

Table 2.1 Training and Inference comparison in Neural Networks

Aspect	Training	Inference
Operation	Forward + Backward Pass	Forward Pass Only
Time Complexity	High (due to iterative optimization)	Low (single pass per input)
Resource Demand	Requires GPUs/TPUs, large memory	Can run on lightweight devices
Goal	Minimize error via learning	Predict outcomes using trained model
Variability	Stochastic (due to optimization steps)	Deterministic (fixed model parameters)

2.2.2.3 *Applications of Neural Network Models*

Deep learning models such as AlexNet, VGG-16, and ResNet have transformed the landscape of computer vision, natural language processing, and speech recognition by achieving state-of-the-art accuracy across a broad range of tasks. These models leverage deep, hierarchical architectures to learn complex representations from data, allowing them to outperform traditional machine learning techniques in domains such as image classification, object detection, and semantic segmentation.

AlexNet, one of the earlier deep convolutional networks, requires over 1.4 billion floating-point operations and approximately 244 MB of memory just to store weights (Jouppi et al., 2017). The deeper and more accurate VGG-16 pushes these requirements further, with over 30.8 billion operations and 552 MB of weight storage, making it challenging to

deploy such models on traditional CPUs or resource-constrained platforms like embedded systems (Umuroglu et al., 2017).

During inference, these trained models perform only the forward pass of the computation generating predictions without adjusting internal parameters. While this process is less computationally demanding than training, it still requires substantial resources when models are large. As neural networks grow in depth and width to boost accuracy, their inference time increases, often resulting in unacceptable latency for real-time applications. This trend is particularly problematic in edge computing environments, where models must respond quickly within strict latency, power, and thermal constraints.

2.3 Hardware Inference Time Implementation for NN

The deployment of neural networks (NNs) for inference generating predictions from trained models requires hardware platforms that balance computational efficiency, latency, power consumption, and flexibility. As NNs grow in complexity, traditional processors like CPUs and GPUs face limitations in meeting the demands of real-time, edge, and resource-constrained applications. This has encouraged the development of specialized hardware, including ASICs, FPGAs, and emerging paradigms like analog and photonic computing. This review synthesizes advancements, challenges, and trade-offs across these platforms, emphasizing strategies for optimizing inference time while addressing energy and latency constraints.

2.3.1 Hardware Platforms for NN Inference

Traditional processors like Central Processing Units (CPUs) offer flexibility but are generally ill-suited for the highly parallelizable nature of neural network computations, which primarily involve large matrix multiplications and convolutions (Jouppi et al., 2017). CPUs remain widely used due to their flexibility but suffer from inherent serial processing limitations. Multi-core CPUs with vector extensions (e.g., AVX-512) improve parallelism, yet they lag in energy efficiency. Studies show CPU implementations consume 10–100× more energy per inference than specialized accelerators, making them unsuitable for latency-sensitive or edge applications (Sze et al., 2017; E. Wang et al., 2019).

Graphics Processing Units (GPUs) have become the widely adopted standard for accelerating both training and inference due to their massively parallel architecture, featuring thousands of smaller cores optimized for floating-point operations (Lu, 2024). GPUs excel in batch processing through massive parallelism, with frameworks like CUDA and cuDNN optimizing tensor operations. However, their high-power consumption (300+ W for data center GPUs) and

reliance on batch processing limit suitability for single-instance, low-latency tasks (Reuther et al., 2019). Memory bandwidth constraints further hinder performance as model sizes grow. Also, Enterprise-grade GPUs represent a significant investment and may be physically too large for embedded systems.

Application-Specific Integrated Circuits (ASICs) like Google's TPU achieve peak efficiency by adapting circuits to NN operations. TPUs employ systolic arrays for matrix multiplication, delivering 15-30× higher throughput and 30-80× better performance-per-watt than CPUs/GPUs. Edge-focused ASICs, such as Apple's Neural Engine and Intel's Movidius, optimize for low-power inference but lack flexibility, risking obsolescence as algorithms evolve (Sze et al., 2017). This rigidity poses a challenge in the rapidly advancing field of neural networks, where new models and techniques are constantly emerging.

Field-Programmable Gate Arrays (FPGAs) present an attractive alternative for neural network inference acceleration, offering a balance between the flexibility of CPUs and the performance and efficiency of ASICs (F. Liu et al., 2024; Shawahna et al., 2019). FPGAs consist of configurable logic blocks (CLBs), memory blocks (BRAMs), and digital signal processing (DSP) blocks that can be programmed to implement custom hardware architectures tailored to specific neural network models. This reconfigurability allows designers to create highly parallel and pipelined data paths optimized for the exact computational structure and data types of a given network, potentially achieving lower latency and better power efficiency compared to GPUs for certain applications (Umuroglu et al., 2017).

Studies consistently show FPGAs as a middle-ground in flexibility and efficiency: they are more efficient than general-purpose CPUs/GPUs but still reprogrammable unlike fixed-function ASICs (Nechi et al., 2023). For example, in a comparative study accelerating a CNN, an FPGA design achieved lower latency and higher energy efficiency than both an ARM Cortex-A53 CPU and an NVIDIA TX2 GPU at similar accuracy (Giasemis et al., 2025). The researchers demonstrated that a Xilinx Alveo U250 FPGA could slightly *outperform* an NVIDIA RTX 3090 GPU in inference throughput while using only ~60% of the power, translating to roughly a 2× gain in performance-per-Watt**.** Even a smaller Alveo U50 card, though slightly slower than the GPU, consumed so much less power that it delivered ~3× better energy per inference.

These results underscore the efficiency advantage of FPGAs in well-matched workloads. The drawback is that FPGAs typically run at lower clock frequencies and may have lower absolute

throughput on very large batch workloads than top GPUs/TPUs; they also require specialized design effort or high-level synthesis tools to program. Nonetheless, for latency-critical applications (e.g. real-time video analytics, high-frequency trading, or scientific triggers), FPGAs offer deterministic and ultra-low-latency execution.

Implementing neural networks on hardware, regardless of the platform, involves mapping the network's computational graph onto the available hardware resources. This typically involves optimizing the execution of layers like convolution, pooling, and fully connected layers. Key considerations include maximizing parallelism, efficiently managing data movement between memory and processing units, and selecting appropriate data representations (e.g., fixed-point vs. floating-point, reduced precision) to balance performance, power, and accuracy (Gysel, 2016). Techniques such as tiling, loop unrolling, and exploiting data locality are crucial for effective hardware utilization and minimizing off-chip memory access, which is often a bottleneck in achieving high inference speed (Zhang et al., 2015). The trade-offs between computational throughput, memory bandwidth, and resource utilization are central to achieving optimal inference time on any hardware platform.

2.4 FPGA & FPGA-Based NN Implementations

2.4.1 FPGA

FPGAs are integrated circuits that can be electrically reprogrammed post-manufacturing to implement arbitrary digital logic functions. An FPGA consists of a two-dimensional array of programmable logic blocks interconnected by a programmable routing network, along with I/O interfaces at the periphery for off-chip communication(Hauck & DeHon, 2010). The logic blocks (often called configurable logic blocks, CLBs) typically contain look-up tables (LUTs) to implement boolean functions and flip-flops for storage, allowing realization of both combinational and sequential circuits. The programmable interconnect fabric provides flexible routing paths between logic blocks, enabling the FPGA to realize complex wiring of a given design (Shantharama et al., 2020). This reconfigurable logic + routing infrastructure makes FPGAs extremely flexible and general-purpose, albeit usually slower and less area- or power-efficient than fixed-function ASICs of the same technology generation(Hauck & DeHon, 2010).

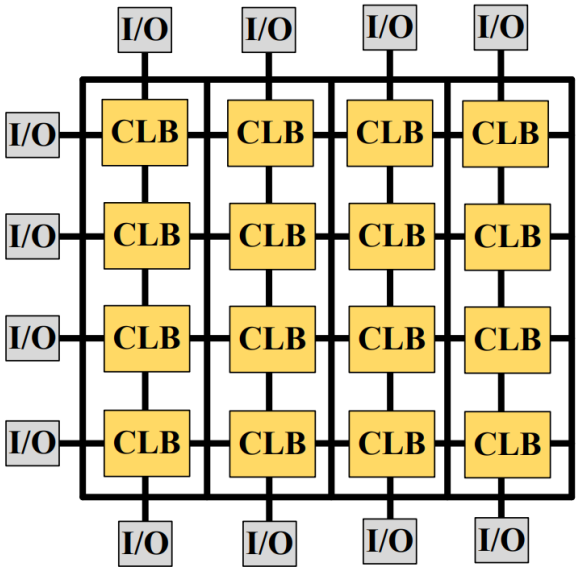


Figure 2.1 Overview of FPGA architecture: CLBs with I/O blocks

The concept of programmable logic emerged in the late 1970s with Programmable Logic Devices (PLDs), but modern FPGAs trace their origins to Ross Freeman's innovations at Xilinx in the early 1980s. The first commercial FPGA, the XC2064, was introduced by Xilinx in 1985, containing just 64 configurable logic blocks (Trimberger, 2015). From these modest beginnings, FPGAs have undergone dramatic evolution in capacity, performance, and functionality.

By the mid-1990s, FPGAs began incorporating embedded memory blocks, multipliers, and eventually complete processor cores, transforming them from simple glue logic devices into platforms capable of implementing entire systems. Modern FPGAs are heterogeneous in architecture, incorporating dedicated hard blocks alongside the generic logic fabric. Notably, most devices include arrays of embedded block RAM (BRAM) for on-chip data storage and DSP slices (dedicated multipliers/accumulators) for efficient arithmetic operations. These hard blocks (memory, multipliers, adders, etc.) are optimized for specific functions (e.g. signal processing) and can deliver high performance for those tasks. For example, a DSP slice might implement a 25×18-bit multiplier–accumulator optimized for common digital signal processing or matrix multiplication operations. The inclusion of such blocks has been driven by the need to improve FPGA efficiency and narrow the gap with ASIC performance for computationally intensive applications. I/O blocks reside at the edges of the chip to handle voltage-level translation and connectivity to external pins, linking the internal reconfigurable fabric with the outside world

Table 2.2 Key architectural components of FPGA

Component	Function	Representative resource in modern devices
CLB/LUT	Boolean & small FSM logic	6-input LUTs, fast carry chains
Programmable interconnect	Hierarchical routing fabric	Switch boxes & segmented channels
BRAM / UltraRAM	On-chip data storage (dual-ported)	Tens of MiB distributed across fabric
DSP slices	Hardened MAC / SIMD arithmetic	25×18/27×18 multiplier accumulator, pre-adder
High-speed transceivers	Off-chip 28-112 Gb s^{-1} serial links	PCIe, Ethernet, DDR/HBM controllers
Hard processor system (HPS)	Embedded ARM/RISC-V cores + peripherals	Zynq-7000, Versal ACAP

Early applications of FPGAs included prototyping ASIC designs and implementing custom controllers or signal processing functions in domains like telecommunications and industrial control. By the 2000s, vendors began integrating the aforementioned BRAM and DSP hard blocks, which enabled efficient implementation of DSP algorithms (e.g. FIR filters, FFTs) and high-speed arithmetic on FPGAs.

This broadened their use into areas such as high-performance digital signal processing, software-defined radio, and image/video processing. Today's state-of-the-art FPGAs often even embed hardened CPU cores and high-speed transceivers, further expanding their general-purpose applicability. Crucially, the reconfigurability of FPGAs the ability to redefine the hardware's functionality by downloading a new configuration bitstream remains their defining feature. This flexibility allows adapting the hardware architecture to the problem at hand, which has opened the door for FPGAs to accelerate emerging workloads like machine learning. Indeed, with their abundant fine-grained parallelism, customizable data paths, and energy-efficient operation, FPGAs have **"**emerged as a promising platform**"** for modern compute-intensive tasks such as neural network processing (Yan et al., 2024).

FPGAs are programmed using Hardware Description Languages (HDLs) such as VHDL and Verilog, or increasingly through High-Level Synthesis (HLS) tools that allow design specification in languages like C/C++ or OpenCL. Traditional RTL development

(Verilog/VHDL) is still the performance gold standard, but high-level-synthesis (HLS) and OpenCL enable software engineers to target FPGAs. Vendor tool-chains (AMD Vivado/Vitis, Intel Quartus Prime) transform behavioral code into placed-and-routed bitstreams, performing logic synthesis, timing closure and bitstream generation automatically. The same flow now supports partial reconfiguration, letting a running design swap bitstreams on-the-fly for time-multiplexed acceleration.

Table 2-3 FPGA product families for different vendors

Vendor	Low-cost / IoT	Mid-range	High-performance	SoC / Heterogeneous
AMD-Xilinx	Spartan-7, Artix-7	Kintex-7, Kintex U+, ACAP Versal-AI Edge	Virtex-7, Virtex U+, Versal-HPC	Zynq-7000, Zynq U+, Versal-AI Core
Intel	MAX 10	Cyclone 10, Agilex E-series	Arria 10, Stratix 10, Agilex F-series	SoC Cyclone-V / Arria-V
Lattice	iCE40-Ultra / Nexus	ECP5, Mach-XO5	—	CrossLink-NX (embedded ML)
Microchip	IGLOO-2	PolarFire, RTG4 (rad-tolerant)	—	PolarFire-SoC (RISC-V + FPGA)

2.4.2 FPGA-Based NN Implementation

FPGA-based neural network accelerators typically target inference-time deployment (feed-forward evaluation of trained models) rather than training. This is because training involves iterative weight updates and massive memory transfers, which are challenging to implement efficiently on FPGAs limited on-chip memory and relatively lower clock speeds (Yan et al., 2024).

Parallelism and Pipelining: NN inference is composed of many independent arithmetic operations (e.g. multiply-accumulates across neurons or convolution kernels), which can be spatially replicated on the FPGA. Designers frequently unroll loops and instantiate multiple processing elements to compute many outputs in parallel, for example, processing several convolution channels or neurons simultaneous. At the network level, one can also pipeline different layers of the network. In a layer-pipelined design, while one layer is computing on the current data, the next layer can start processing the previous layer's results, so that multiple

layers operate concurrently in a streaming fashion (Ji et al., 2023). This deep pipelining, combined with parallel computation within each layer, significantly boosts throughput. An FPGA implementation of a CNN can thus achieve high frame rates and low per-image latency by leveraging channel parallelism (parallel processing of many feature map channels) and layer pipelining (overlapping the execution of successive network layers) (Ji et al., 2023).

In practice, the degree of parallelism is often limited by the FPGA's resources (LUTs, DSPs, BRAM), so designers must choose a parallelization strategy (e.g. partial unrolling or tiling) that fits the device while meeting throughput requirements. Still, even moderate levels of parallelism can deliver large speedups: for instance, an OpenCL-based CNN on a modest Xilinx Zynq-7000 FPGA achieved a 10× speedup over a CPU implementation for MNIST and CIFAR-10 inference (Gdaim & Mtibaa, 2025).

Quantization and Low-Precision Arithmetic: Unlike GPUs (which excel at floating-point), FPGAs are more efficient with integer or fixed-point arithmetic, and they allow custom bit-widths. By quantizing neural network parameters to, say, 8-bit or 4-bit integers (and in extreme cases 1-bit binary nets), one can drastically reduce memory usage and utilize the FPGA's LUTs/DSPs more efficiently (each DSP slice can often perform several low-bit operations in a single cycle). Many FPGA NN accelerators therefore use fixed-point math and carefully quantized networks to strike a balance between accuracy and performance (Yan et al., 2024). A recent work on a binary-weight ResNet-18 accelerator on Xilinx hardware achieved nearly 5,000 frames per second throughput using 1–2-bit weights on an Alveo FPGA card (Peng et al., 2021).

Memory Hierarchy and Dataflow: FPGAs provide fast on-chip memories (BRAM or UltraRAM) that can be used as caches or buffers for weights and intermediate results. For instance, a block of coefficients or a tile of an activation map might be stored on-chip and reused for many computations (e.g. sliding a window across the image for convolution) before being written back to external memory. On-chip memory is crucial for reducing latency fetching data from off-chip DRAM not only is slower but also consumes more energy than on-chip reads (Yan et al., 2024).

One study implemented a real-time object detection CNN on an Intel Stratix 10 FPGA, achieving 2,167 frames per second (batch-1 throughput) with only 2.13 ms latency end-to-end (Anupreetham et al., 2024). This design leveraged a fully pipelined dataflow and ran at 400 MHz, outperforming prior FPGA results by 5.3× in throughput and 5× lower latency at

comparable accuracy (Anupreetham et al., 2024). High-end accelerator cards like Xilinx Alveo have been shown to sustain very high throughput as well – e.g. over 5,000 images/**s** on an Alveo U280 for a quantized ResNet-18 model (Peng et al., 2021) highlighting that FPGAs can rival or exceed GPU inference performance for suitably optimized networks.

Table 2.3 Performance Highlights of FPGA-Based Neural Network Implementations

Model & board	Technique	Throughput / latency	Energy-eff.
VGG-16 on Arria-10 GX1150 (Anupreetham et al., 2024)	8-bit quant., loop-tiling	645 GOP s^{-1}, 47.9 ms	30 GOP s^{-1} W^{-1} (21 W)
VGG-16-SVD on Zynq-ZC706 (Qiu et al., 2016)	16-bit fixed-point, data-arrange	4.45 fps (batch 1)	5.7 W
ResNet-18 binary on Alveo U280(AMD, 2021)	1-bit weights, FINN2	> 5,000 img s^{-1}	2.3× GPU perf/W
YOLO v3 on Xilinx DPU (U280) (AMD, 2021)	Dedicated DPU, INT8	21–36 fps (1-batch)	7× T4 GPU perf/W

2.5 Model Compression Techniques

Model compression aims to reduce model size and accelerate inference while preserving accuracy. Two primary compression approaches are pruning and quantization, which can dramatically shrink model footprints and inference costs.

2.5.1 Pruning

Unstructured pruning (fine-grained): Unstructured pruning removes individual weights deemed less important (often those with small magnitudes). It yields irregular sparsity a high proportion of zero-valued weights scattered arbitrarily across weight tensors. This fine-grained removal can significantly reduce parameter count (e.g. a 9× reduction in AlexNet parameters was demonstrated by Zhao et al., 2019) with minimal accuracy loss after retraining. The memory footprint drops proportionally to pruned weights, though some storage overhead is needed for indices or masks. A key drawback is that unstructured sparsity is hard for conventional hardware to exploit the remaining nonzero weights are irregularly distributed, preventing straightforward speedup on dense linear algebra routines. As a result, unstructured pruning alone typically requires specialized software or hardware support to realize acceleration (Cheng et al., 2023). Without sparse-aware libraries or hardware, the compute

units still traverse zero weights, yielding little to no speed gain. Research prototypes and accelerators have been developed to leverage unstructured sparsity: for example, (Han et al., 2016) EIE ASIC demonstrated up to 13–189× speedups by skipping zero operations in a pruned network. In general, however, unstructured pruning is most effective for reducing model size (for storage or transmission) and can improve energy efficiency, but its impact on runtime depends on the availability of sparse computation engines (Cheng et al., 2023).

Structured pruning (coarse-grained): Structured pruning removes entire groups of weights such as neurons, convolution filters, or attention heads, thereby eliminating whole feature maps or channels. This produces a smaller, dense sub-network with fewer layers or smaller layer dimensions. Examples include filter pruning in CNNs (dropping selected convolution kernels) or layer pruning in transformer models. Because the pruned network remains regular (just with reduced width or depth), it can run faster on standard hardware without custom libraries (Cheng et al., 2023). Pruning entire channels often simplifies memory access patterns and is friendly to high throughput processors.

2.5.2 Quantization

Quantization reduces model size and accelerates computations by representing network weights and activations with lower precision numbers (fewer bits than the standard 32-bit floating point). Instead of 32-bit floats, one might use 8-bit integers (INT8), 4-bit integers, or even 1-bit binary values for parameters. There are two major paradigms: uniform precision quantization, where a single bit-width is applied uniformly to all layers, and mixed-precision quantization, where different layers (or even different network parts) use different bit-widths.

Uniform quantization: In uniform quantization, the entire network uses the same precision (for example, all weights and activations are quantized to 8 bits). Using a uniform bit-width means the arithmetic units (multipliers/adders) can be identical and densely packed, maximizing parallel compute throughput (Park et al., 2021). Uniform quantization is attractive for its simplicity and generality hardware designers can optimize a single low-precision data path and reuse it for every layer. However, the one-size-fits-all bit-width may be suboptimal for accuracy. Different layers have different sensitivity to quantization errors. Binary networks (1-bit weights) or ternary networks (2-bit) can greatly accelerate inference on specialized hardware (e.g. by turning multiplies into bit-wise operations) but suffer significant drops in accuracy on complex tasks (Zhao et al., 2019).

Mixed-precision quantization: Mixed-precision approaches allocate different bit-widths to different layers or network components, recognizing that not all parts of a network require the same precision. The goal is to achieve better accuracy than a uniform low-bit model, while still gaining efficiency by using lower precision where possible (Qin et al., 2025). Mixed-precision quantization is especially relevant when hardware supports multiple precisions efficiently. Recent research and hardware trends are indeed moving in this direction: emergent DNN accelerators support 1–8 bit arithmetic in flexible ways to improve computation efficiency (K. Wang et al., 2019).

Quantization strategies and impact: Implementing quantization can be done post-hoc or during training. Post-training quantization converts a trained model's weights (and possibly activations) to lower bit representations (e.g. by rounding or clustering values) after training is complete. This is fast and doesn't require retraining, but can lead to some accuracy loss, especially at very low precisions, since the model wasn't optimized for the quantized weights. In contrast, quantization-aware training (QAT) inserts fake quantization operations into the training process, simulating low-bit behavior so that the model learns to compensate for quantization errors (Jacob et al., 2018).

2.6 Ensemble Modeling

Traditional ensemble methods: Bagging (bootstrap aggregating) and boosting are classic ensemble techniques that improve performance by combining multiple "weak" learners. Bagging (Breiman, 1996) trains multiple models in parallel on different random subsets of the training data (typically sampling with replacement). The process reduces variance by averaging out the noise in different models, bagging yields a more stable and generally better model than any single one (Du et al., 2025). In contrast, boosting (Freund & Schapire, 1997) builds ensembles sequentially: each new model is trained to address the errors of the current ensemble so far. In boosting, models are added one after another, with each model focusing more on instances that previous models' mis-predicted. Boosting tends to reduce bias by combining many weak learners that are tuned to complement each other, it can achieve a strong overall predictor (Du et al., 2025).

Ensembles in neural networks: Deep neural networks can be ensembled in similar ways, though practical considerations differ because NNs are often large and costly to train. The most common approach for neural network ensembles is akin to bagging: train multiple neural networks independently (with different random initializations, shuffling, or even different

architectures/hyperparameters) and then combine their outputs. Even if each network is a high-capacity model, differences in initialization and training make them converge to different local minima in weight space, and thus they will have different prediction errors (Huang et al., 2017). Hansen & Salamon's, classic work noted that an ensemble of similar neural networks can reduce generalization error if the individual models' errors are at least partially uncorrelated a condition often met by training with different random seeds or data splits.

However, a straightforward neural network ensemble (training M separate networks) comes at a steep cost: training deep networks is computationally expensive, and doing it M times multiplies the cost linearly (Huang et al., 2017). There are also hybrid strategies like stacking, where one trains different models and then learns a meta-learner (often a shallow model) to combine their outputs. Stacking can achieve very high accuracy by weighting models in an optimal way based on their performance (Du et al., 2025).

Inference-time ensemble strategies: Regardless of how an ensemble is trained, the question at inference is how to combine the predictions of multiple models. The simplest methods are majority voting (for classification) and averaging (for regression or probabilistic outputs). In majority voting, each model in the ensemble votes on the predicted class and the majority (or plurality) class is taken as the final output. This can be seen in random forests where each tree votes on the class. Alternatively, one can average the predicted class probabilities or confidence scores from each neural network and then choose the class with highest average probability effectively a form of soft voting that often performs better than hard majority voting because it accounts for the strength of each prediction. Caruana et al., notably used weighted averaging to win a competition by selecting and weighting models from a large ensemble pool

2.7 Tools and Frameworks

Prominent FPGA platforms include Intel Quartus Prime, Microchip Libero, and Lattice Diamond, each offering unique strengths in various applications and environments. Among these, Xilinx Vivado Design Suite stands out due to its extensive feature set, comprehensive support for complex FPGA families, and emphasis on high productivity and abstraction levels.

The Xilinx Vivado Design Suite, launched in 2012 as a significant enhancement over its predecessor, the Xilinx ISE, addresses the evolving demands of FPGA-based system designs. Vivado, developed in C++, is compatible with major operating systems like Windows and

Linux, and it effectively caters to modern FPGA families, including the 7-series, UltraScale, and UltraScale+ architectures.

Vivado's Integrated Design Environment (IDE) encapsulates a range of sophisticated tools critical for system-level design, implementation, and verification. The IP Integrator (IPI) is a graphical tool designed to facilitate the plug-and-play integration of intellectual property cores, thus significantly simplifying the complex interconnections typically encountered in advanced FPGA designs. Additionally, Vivado's High-Level Synthesis (HLS) capability is particularly notable for its capacity to convert high-level software algorithms directly into hardware designs, reducing reliance on manual RTL coding and expediting the design process significantly.

Vivado employs a comprehensive design flow that includes distinct stages of simulation, synthesis, and implementation. The simulation phase encompasses behavioral simulations at the RTL level, followed by detailed post-synthesis and post-implementation simulations to progressively ensure design accuracy and integrity at each stage. Vivado Simulator provides advanced capabilities, supporting mixed-language simulations and robust verification methodologies, such as Universal Verification Methodology (UVM) and comprehensive code coverage analysis.

During synthesis, Vivado transforms hardware description language (HDL) designs into device-specific netlists, incorporating strategies like incremental compilation and detailed performance and resource utilization reports. Implementation leverages advanced algorithms, including machine learning techniques, to optimize placement and routing. This process ensures improved Quality-of-Results (QoR) and accelerates the design closure phase. The culmination of the implementation stage is the generation of a bitstream, the final configuration file that programs the FPGA device.

HDL languages play a critical role in FPGA development, with the two dominant languages being VHDL and Verilog. VHDL, known for its strong type-checking and verbose syntax, is widely used in academia and industries requiring rigorous design verification. In contrast, Verilog, the chosen language for this research, is known for its simplicity, readability, and widespread industry adoption, particularly for large-scale digital systems. Verilog's concise syntax and efficient representation of hardware constructs make it ideal for developing complex neural network architectures, enabling rapid prototyping and streamlined debugging processes.

In addition to development tools, choosing an appropriate FPGA platform is crucial for real-world deployment. One example is the Defense-Grade Zynq-7000 XQ7Z100 SoC, which combines a dual-core ARM Cortex-A9 processor with Kintex-7 equivalent programmable logic. It is designed for demanding real-time and resource-limited applications. The device features ruggedized packaging, an extended operating range (–55°C to +125°C), and comprehensive radiation testing.

The Zynq-7000 platform incorporates robust security mechanisms, employing RSA authentication, AES encryption, and SHA hashing for secure boot processes and logic configuration. Additionally, it provides extensive peripheral support such as Ethernet, USB 2.0 OTG, CAN bus, SPI, UARTs, and I2C interfaces, enhancing its adaptability and functionality across diverse application scenarios. Its programmable logic section boasts 444K logic cells, 277,400 Look-Up Tables (LUTs), 554,800 flip-flops, 3,020 KB Block RAM, and 2,020 DSP slices, all optimized for high-performance signal processing tasks.

2.8 Gaps in Existing Research

Despite the progress in both algorithms and hardware tools, a significant gap remains in effectively deploying ensemble neural network models in resource-constrained and real-time environments. Ensemble methods (which combine multiple neural networks for a collective decision) are well-known to boost accuracy and robustness of predictions. However, this comes at the cost of much higher inference-time computation and latency. In a straightforward ensemble, multiple models must be stored and executed, which “greatly increases computational cost for a modest reduction in error” compared to a single model (Kondratyuk et al., 2020). For example, Ahmed et al. note that a deep ensemble incurs high computation and hardware overhead (latency and energy) because it requires the storage and processing of multiple models, rendering such methods “not suitable for battery-powered edge devices with limited computing and memory resources.” (Ahmed et al., 2024).

This challenge highlights a research gap: how to retain the accuracy gains of ensembles without the prohibitive overhead. Current literature only sporadically addresses this, often focusing on either algorithmic accuracy or efficiency in isolation. On one hand, the field of model compression has matured, providing techniques like network pruning and quantization that dramatically reduce the size and speed of individual models. On the other hand, hardware acceleration research provides ways to speed up inference using parallelism and specialized architectures (e.g. FPGAs, TPUs). Yet, integrated approaches that combine ensemble diversity,

aggressive model compression, and hardware-aware acceleration are lacking. Few studies have examined ensembles of neural networks that are themselves heavily compressed (sparse, quantized models) running on parallel hardware, and how to optimize the joint design (Weng et al., 2025). In fact, mapping multiple compressed models onto an FPGA with minimal resource and latency overhead presents non-trivial challenges.

A promising solution direction and the focus of this thesis is to combine lightweight neural networks with compression techniques and exploit FPGA parallelism to enable efficient ensemble inference. By using smaller CNN/ANN models (or otherwise efficient architectures) as ensemble members, and applying pruning and quantization to each, the memory and computation per model can be greatly reduced. Each model may sacrifice some individual accuracy, but together the ensemble can still outperform a single large model in robustness and overall accuracy. Crucially, FPGAs can use the advantage of spatial parallelism to run multiple small networks concurrently or pipeline their execution, mitigating the latency penalty of having multiple models (Kondratyuk et al., 2020). In other words, instead of one large network, several tiny networks can be deployed in parallel on the FPGA fabric an approach shown to potentially “increase throughput” and efficiency in certain cases (Weng et al., 2025).

Pruning removes redundant calculations and quantization shrinks data widths, which not only fits models into on-chip FPGA resources but also speeds up arithmetic operations (Aarrestad et al., 2021). By jointly considering these techniques, an ensemble of compressed models can approach the inference cost of a single model while delivering compatible accuracy. This multi-faceted strategy addresses the gap in current research: rather than treating ensemble accuracy and hardware efficiency as separate issues, it merges them. Early evidence is encouraging for example, Google researchers found that an ensemble of smaller models can sometimes outperform a single large model in accuracy and require fewer total FLOPs, offering a more flexible trade-off between speed and accuracy (Kondratyuk et al., 2020). However, comprehensive studies that implement such solutions on real hardware are sparse. Thus, this thesis aims to fill that void by demonstrating an integrated approach that unites ensemble learning, model compression, and FPGA-based implementation. In doing so, it tackles the twin goals of high robustness (through ensemble diversity) and low latency (through compression and parallel hardware execution), directly addressing the identified gap in existing research.

CHAPTER 3 METHODOLOGY AND SYSTEM DESIGN

3.1 Overview

This chapter details the comprehensive methodology used to design and implement a general Field-Programmable Gate Array (FPGA) based framework for integrating ensemble neural network models. The primary focus is on optimizing for energy efficiency, computational performance and accuracy in real-time applications. Building upon its foundational application in Geez Digit Recognition, this chapter explicitly integrates the widely recognized MNIST dataset as a second, distinct application. This strategic integration serves to thoroughly benchmark and validate the framework's generality, adaptability and comparative performance across diverse, yet structurally similar, digit recognition tasks. The discussion will highlight how the proposed hardware architecture, built upon reusable Verilog modules and managed within the Xilinx Vivado Design Suite, remains consistent and reusable. This consistency demonstrates its inherent capacity to support varied ensemble configurations and different datasets without requiring application-specific hardware redesigns, thus solidifying its claim as a "general framework" for ensemble Neural Network (NN) inference on FPGAs.

The framework can effectively handle two distinct digit recognition tasks-Geez, a culturally specific and less common dataset, and MNIST, a globally recognized and widely used benchmark-it inherently. This demonstrates its generalizability beyond a single role application. This generalizability is paramount for a "framework", as it implies reusability and scalability to other similar problems, such as other character recognition tasks or simple image classification. This foresight in design thinking, moving from a specific problem (Geez) to a general solution (a versatile FPGA framework), significantly improves the long-term impact and applicability of the research. The framework is thus positioned not just as a solution for Geez digits but as a versatile platform applicable to a broader class of problems, thereby increasing its academic and practical value. This elevates the research's contribution to the field of general AI hardware acceleration, rather than limiting it to a specific application.

3.2 Selection of Ensemble Techniques and Models

3.2.1 Ensemble Technique Selection

This section reiterates the selection of voting and weighted averaging as the primary ensemble techniques employed within the framework. These techniques were preferred over more computationally intensive alternatives. While weighted averaging is a form of stacking, the

framework's approach of using multiple smaller models helps to compensate for the computational demands typically associated with stacking. Voting and weighted averaging offer a robust balance between accuracy improvement and hardware efficiency. They enable straightforward parallel inference of individual models followed by simple, low-overhead combination logic that can be efficiently implemented on the FPGA fabric, thereby aligning with the framework's optimization goals for energy efficiency and computational performance.

This approach aligns directly with the growing trend of "TinyML" and efficient AI, where deploying models on edge devices, such as IoT sensors, autonomous systems and other embedded systems, necessitates severe resource optimization. The chosen ensemble techniques are thus not just academically interesting but also practically viable and highly relevant for real-world low-power, low-latency deployments.

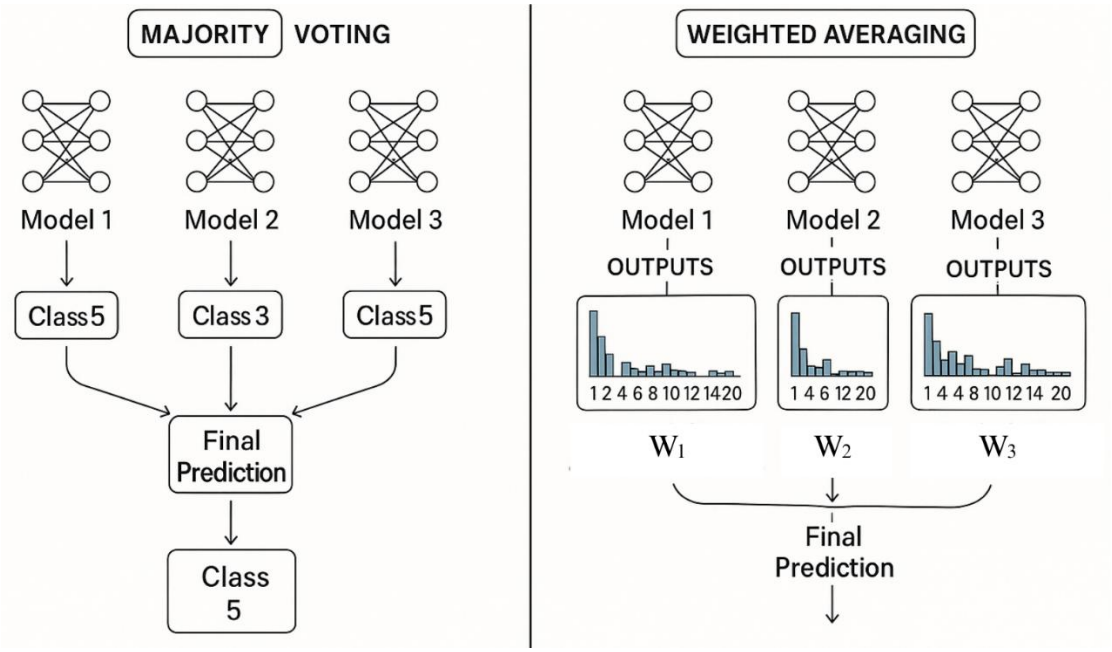


Figure 3.1 Voting and Weighted Ensemble of models

3.2.2 Model Selection and Adaptation

The foundational application for this framework is Geez Handwritten Digit Recognition. This task uses the Geez Handwritten Digit Dataset, which comprises 46,755 training samples and 5,197 test samples, covering 20 distinct digit classes (1–10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 1000). The original work by Ali Nur et al., (2022) proposed a deep Convolutional Neural Network (CNN) for this task.

Figure 3.2 Geez digits (Ali Nur et al., 2022)

However, this original model was deemed impractical for FPGA deployment due to its significantly high parameter count, which translated into excessive memory and computational demands beyond typical FPGA capabilities.

To address these limitations, a re-engineered, lightweight CNN architecture, specifically named SimpleCNN, was developed. This architecture retains the core feature extraction capabilities essential for digit recognition while drastically reducing resource requirements. Its modest parameter count (approximately 43,364) and reduced multiply-accumulate operations (around 1.37 million) make it inherently suitable for efficient FPGA implementation. Three distinct instances of this SimpleCNN were trained for the Geez dataset, forming the basis of the initial ensemble. For the Geez models, both 24-bit and 16-bit quantization were used during evaluation to assess the impact of precision on accuracy and resource use.

To show the framework's benchmarking and comparison capabilities, the widely recognized MNIST dataset (LeCun & Cortes, 1998) is integrated as a second application. MNIST consists of 70,000 grayscale images of handwritten digits (0-9), with 60,000 images designated for training and 10,000 for testing. Each image is standardized to a size of 28x28 pixels. Its simplicity, widespread use as a standard benchmark in machine learning, and structural similarity to the Geez digit recognition task makes it an ideal candidate for demonstrating the generality and comparative performance of the FPGA framework. For the MNIST models, only 16-bit quantization was applied during evaluation.

The strategic choice of MNIST is not merely to add another dataset but to provide a globally recognized benchmark against which the framework's performance, efficiency and accuracy can be universally understood, compared and validated. This allows for a more robust and widely interpretable validation of the "general framework" claim. Crucially, a different lightweight CNN architecture, named SimpleCNNMNIST, was developed for the MNIST dataset. This decision explicitly highlights the framework's generalizability: the primary hardware architecture is designed to efficiently process any model conforming to the SimpleCNN or SimpleCNNMNIST structure and its quantized representation, regardless of the specific dataset it was trained on (Geez or MNIST). Three distinct instances of SimpleCNNMNIST were trained for MNIST, mirroring the approach taken for Geez, to form

its respective ensemble. As per the design specification, the underlying utility and other hardware modules remain identical; only the training data, the specific model architecture (SimpleCNN vs. SimpleCNNMNIST) and the resulting trained model parameters differ between the Geez and MNIST applications.

3.2.2.1 *Architecture Overview*

The SimpleCNN architecture is designed for classifying grayscale images of handwritten Geez digits, resized to 32 × 32. Its lightweight design ensures efficient deployment on FPGAs while providing robust feature extraction capabilities. Below is a detailed layer-by-layer breakdown of the architecture:

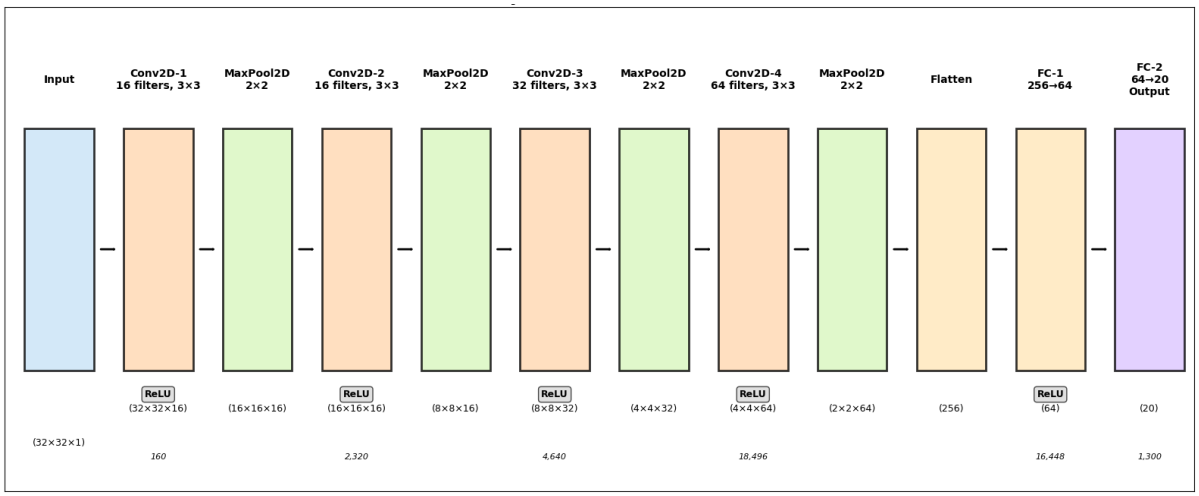


Figure 3.3 Architecture of SimpleCNN

The SimpleCNN follows a progressive feature extraction strategy, with early layers (Conv1, Conv2) focusing on low-level features and deeper layers (Conv3, Conv4) capturing complex patterns. The doubling of filters (16 → 32 → 64) reflects the need for intricate feature representation. Max-pooling reduces computational complexity and introduces robustness to translations, crucial for handwritten Geez digits with varied styles.

The SimpleCNN is parameter-efficient, with 43,364 trainable parameters, making it ideal for ensemble deployment on FPGAs. The table below details the parameters for each layer, applicable to all three models.

Table 3.1 Trainable weight and bias parameters of SimpleCNN

Layer	Parameter Type	Shape	Number of Parameters	Percentage
Con1	Weights	16 × 1 × 3 × 3	144	**0.37%**
	Biases	16	16	

Conv2	Weights	16 × 16 × 3 × 3	2,304	**5.35%**
	Biases	16	16	
Conv3	Weights	32 × 16 × 3 × 3	4,608	**10.70%**
	Biases	32	32	
Conv4	Weights	64 × 32 × 3 × 3	18,432	**42.65%**
	Biases	64	64	
FC1	Weights	256 × 64	16,384	**37.93%**
	Biases	64	64	
FC2	Weights	64 × 20	1,280	**3.00%**
	Biases	20	20	
Total			43,364	**100%**

The SimpleCNNMNIST architecture is designed for classifying grayscale images of handwritten digits from the MNIST dataset, resized to 32 × 32. Its lightweight and efficient design ensures effective feature extraction while maintaining computational simplicity, making it suitable for deployment on resource-constrained platforms such as FPGAs. Below is a detailed layer-by-layer breakdown of the architecture:

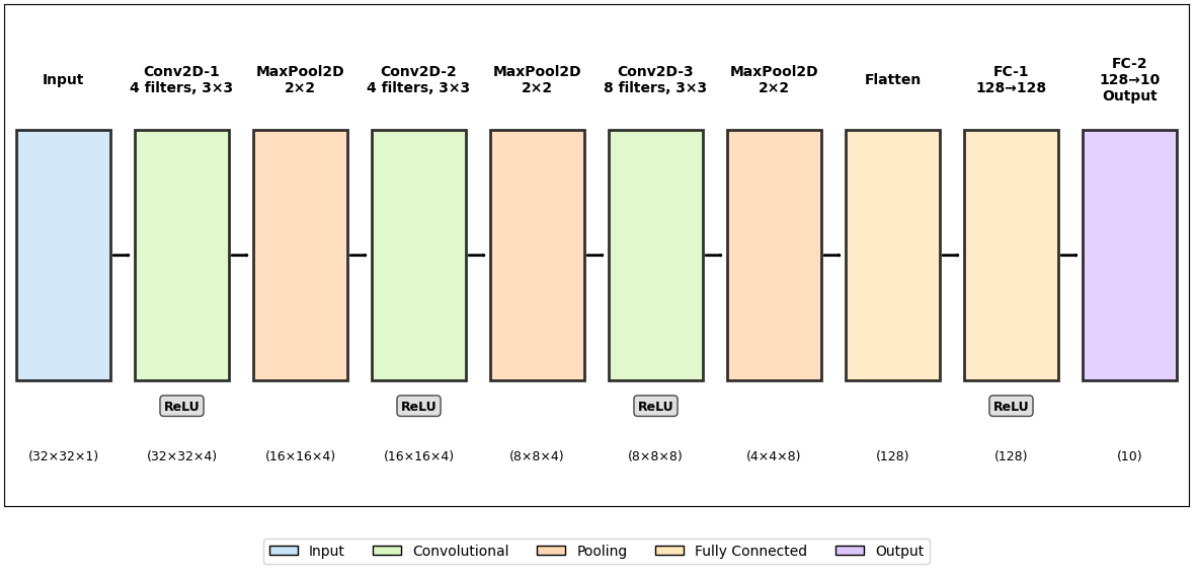


Figure 3.4 Architecture of SimpleCNNMNIST

The SimpleCNNMNIST employs a streamlined feature extraction strategy, with early convolutional layers (Conv1, Conv2) capturing low-level features such as edges and textures, and the deeper layer (Conv3) extracting more complex patterns. The use of a minimal number

of filters ($4 \rightarrow 4 \rightarrow 8$) ensures efficiency while still providing sufficient feature representation for the relatively simple MNIST digit classification task. Max-pooling layers reduce spatial dimensions, lowering computational complexity and introducing robustness to small translations, which is essential for handling variations in handwritten digits. The fully connected layers (FC1, FC2) consolidate the extracted features into a compact representation for final classification into one of the 10-digit classes.

The SimpleCNNMNIST is highly parameter-efficient, with only 18,286 trainable parameters, making it ideal for lightweight applications and ensemble deployments. The table below details the parameters for each layer, applicable to all three models.

Table 3.2 Trainable weight and bias parameters of SimpleCNNMNIST

Layer	Parameter Type	Shape	Number of Parameters	Percentage
Conv1	Weights	$4 \times 1 \times 3 \times 3$	36	0.20%
	Biases	4	4	0.02%
Conv2	Weights	$4 \times 4 \times 3 \times 3$	144	0.79%
	Biases	4	4	0.02%
Conv3	Weights	$8 \times 4 \times 3 \times 3$	288	1.57%
	Biases	8	8	0.04%
FC1	Weights	$128 \times (8 \times 4 \times 4)$	16,384	89.57%
	Biases	128	128	0.70%
FC2	Weights	10×128	1,280	7.00%
	Biases	10	10	0.05%
Total			**18,286**	**100%**

3.2.2.2 *Model Diversity and Ensemble Synergy*

The effectiveness of an ensemble relies on the diversity of its constituent models to produce uncorrelated errors, enabling collective performance to surpass that of any single model. The three SimpleCNN models share an identical architecture but were trained with slight variations to introduce diversity in their feature representations, enhancing the ensemble's robustness and accuracy. The following strategies were employed to introduce diversity during training:

1. **Random Weight Initialization**: All models used Kaiming normal initialization for layers with ReLU activation to ensure consistent weight initialization. Models 1 and 2 shared the same random seed (1009) for reproducibility, while Model 3 used a different seed (42), which may have led to a different convergence path.

2. **Regularization Techniques**: To encourage diverse learning behavior, regularization was applied differently across models. Models 1 and 2 used dropout in the fully connected layer with rates of 0.1 and 0.25 respectively, while Model 3 did not use dropout. These variations influenced each model's ability to generalize and contributed to differing predictive performance.

3. **Data Augmentation Techniques**: All models used similar data augmentation strategies during training random rotations (up to 10°), resized crops (scale 0.8–1.0), and slight color jitter to introduce variation in digit appearance. The test dataset preprocessing was identical across models. As a result, differences in data exposure were minimal, with most variation stemming from random seed effects during augmentation.

4. **Training Duration**: The models were trained for varying durations: Models 1 and 2 for 25 epochs, and Model 3 for 20 epochs. The shorter training time for Model 3 may have led to less refined feature learning, contributing to differences in generalization and error patterns across the models.

5. **Batch Size**: All models used a consistent batch size of 64 for both training and test data loaders, with shuffling enabled for training and disabled for testing.

For example, Model 1 might misclassify a digit due to its lower dropout rate, but Model 2, with higher dropout, could correct this by generalizing better to certain patterns, and Model 3, with a different seed and shorter training, might offer a complementary perspective based on its unique optimization path. This synergy, driven by the limited but intentional training variations, aims to enhance the ensemble's accuracy and robustness for recognizing handwritten Geez digits. All three models use the Adam optimizer with an initial learning rate of 0.002.

The three SimpleCNNMNIST models (Model 1, Model 2, and Model 3) also share an identical architecture but were trained with intentional variations to introduce diversity in their feature representations. These variations enhance the ensemble's robustness and accuracy for classifying handwritten digits from the MNIST dataset. Below are the strategies used to introduce the diversity in the modeling.

1. **Random Weight Initialization:** Each model used a different random seed (Model 1: 42, Model 2: 253, Model 3: 599), which introduced variability in weight initialization and learning paths. Models 2 and 3 explicitly applied Kaiming initialization, while Model 1 used PyTorch's default, adding further diversity in their training dynamics.
2. **Data Augmentation Techniques**: All models used data augmentation to introduce variation during training, with slight differences in parameters. Models 1 and 2 applied random rotations up to 10°, while Model 3 used up to 15°, potentially improving its robustness to rotation. All models resized images to 32×32 and normalized them consistently, with identical preprocessing for the test set. Variation in random seeds (42, 253, 599) also contributed to differences in the augmented data each model encountered.
3. **Training Duration**: Although all models were set to train for 100 epochs, the available outputs are truncated, indicating evaluation at different stages. Despite the shared training duration, differences in random seeds and augmentation strategies likely led to variations in convergence behavior and final performance.
4. **Batch Size**: Batch size varied slightly across models, influencing the stochasticity of training. Models 1 and 2 used a batch size of 128, while Model 3 used 64, leading to more variable gradient updates and potentially more diverse optimization in Model 3. Although Model 2 initially defined a loader with batch size 64, it was overridden. Shuffling was applied during training for all models and disabled during testing to ensure consistent evaluation.
5. **Validation Split**: The models used different validation splits, impacting training and evaluation. Model 1 used a 20% split, resulting in a larger validation set, while Models 2 and 3 used 15%, allowing for more training data. This affected both the amount of data each model trained on and the reliability of validation metrics, with Model 1 likely gaining more stable generalization estimates and Models 2 and 3 benefiting from greater training exposure.
6. **Device**: Models 1 and 3 were trained on a CUDA-enabled GPU, while Model 2 was trained on a CPU. This hardware difference may have impacted training speed and numerical precision, potentially influencing optimization dynamics and contributing to variations in the learned weights.

All three models use the Adam optimizer with an initial learning rate of 0.002 and a StepLR scheduler (step_size=8, gamma=0.5) to halve the learning rate every 8 epochs.

3.3 Model Optimization

3.3.1 Pruning

L1 unstructured pruning is employed, targeting individual weights in convolutional and fully connected layers based on their magnitude. This method, known as L1 unstructured pruning, uses the L1 norm to identify and remove the smallest weights, setting them to zero. By doing so, it creates a sparser network, eliminating less critical connections. This sparsity reduces memory usage and computational complexity, making it particularly advantageous for hardware-constrained environments.

The pruning process is integrated into the model loading phase, where an optional parameter specifies the fraction of weights to remove. PyTorch's prune.l1_unstructured method is applied to the weights of relevant layers, targeting those with the smallest L1 norm values. Varying pruning levels can be applied across different models to introduce diversity, thereby enhancing robustness when combining them in an ensemble by leveraging their varied error profiles.

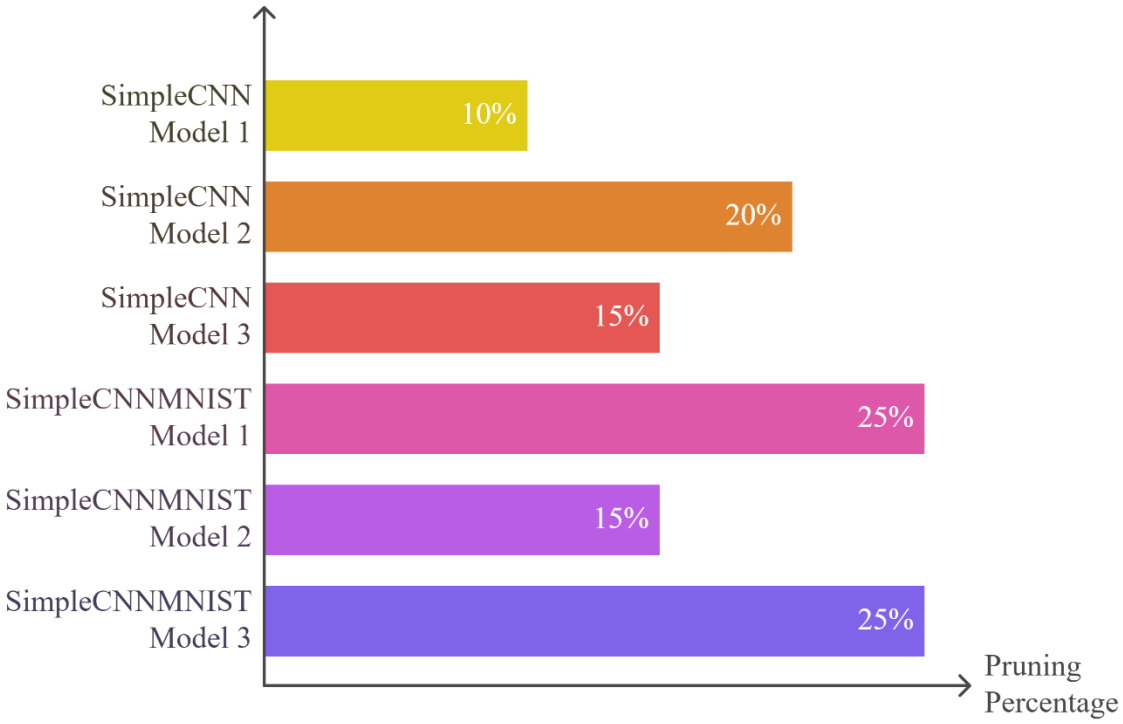


Figure 3.5 Pruning Percentages used on the six CNN models

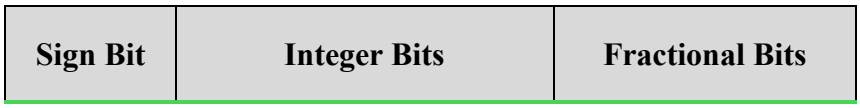
3.3.2 Quantization

The weights, biases, and input data of the SimpleCNN models are quantized to both 24-bit and 16-bit and SimpleCNNMNIST models are converted to 16-bit fixed-point representation. This

process involves scaling and rounding floating-point values to fit the fixed-point format, ensuring compatibility with hardware constraints while maintaining sufficient precision for the task.

3.4 Fixed-Point Arithmetic Implementation

Fixed-point arithmetic is a critical technique for representing weights and biases in neural network models deployed on FPGA-based frameworks. This approach optimizes resource utilization, addresses the limited on-chip memory of FPGAs, and enhances energy efficiency.



Representation of Fixed-Point Arithmetic

3.4.1 Representation and Bit-Width Selection

In this implementation, fixed-point formats of 24-bit and 16-bit are used. The 24-bit format consists of 1 sign bit, 11 bits for the integer part, and 12 bits for the fractional part. The 16-bit format includes 1 sign bit, 7 bits for the integer part, and 8 bits for the fractional part.

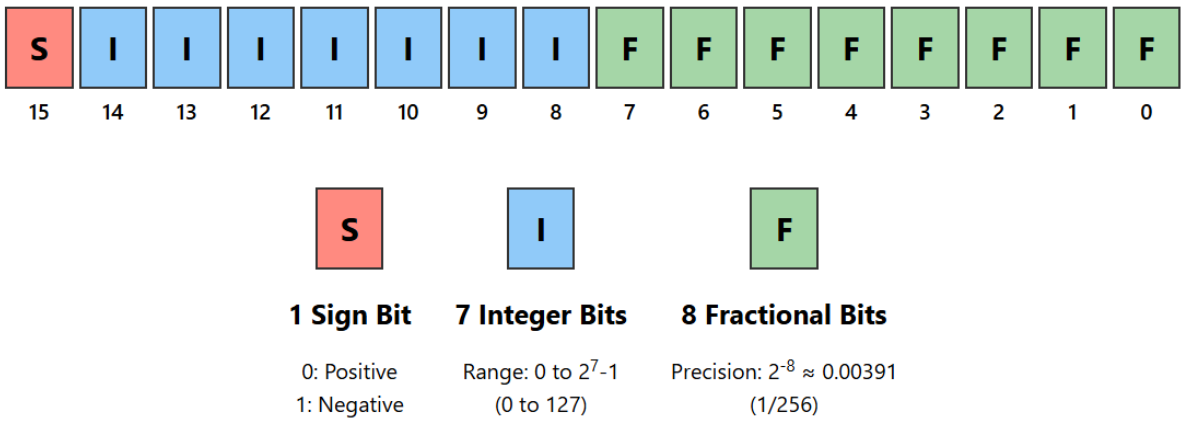


Figure 3.6 16-Bit [1-7-8] Fixed point representation

3.4.2 Implementation Process

The conversion from floating-point to fixed-point is a structured process, implemented statically to simplify hardware design. Here's how it works:

1. **Sign Extraction**: The sign of the floating-point value is identified and assigned to the sign bit (0 for positive, 1 for negative).
2. **Integer Part**: Extract the absolute value's integer portion. It must fit within 7 bits (range: 0 to 127). If it exceeds this, it is clamped to 127 to prevent overflow.
3. **Fractional Part**: The fractional portion is multiplied by (2^8) (since there are 8 fractional bits), rounded to the nearest integer, and represented in binary.

4. **Negative Number Handling**: For negative values, the two's complement is computed by inverting the bits and adding one, ensuring compatibility with arithmetic operations.

3.5 Implementation Using Verilog and Xilinx Vivado

This section details the hardware implementation of an ensemble Convolutional Neural Network (CNN) model on a Field-Programmable Gate Array (FPGA) using Verilog and the Xilinx Vivado Design Suite. The approach follows a bottom-up methodology, starting with the individual components of the ANN/CNN and building up to a complete, testable framework. The implementation is designed to support real-time inference for the trained ensemble model, leveraging the FPGA's parallelism and reconfigurability.

The first step involves breaking down the CNN model into its fundamental layers and operations, which are implemented as modular units in Verilog. These components are then integrated into a cohesive system, tested, and optimized for performance and resource efficiency. The implementation is structured around several key components that work together to realize an efficient and modular CNN-based system in hardware.

At the foundation are the distinct layers of the Convolutional Neural Network (CNN), each designed as an independent Verilog module. These include the convolution layers, pooling layers, and fully connected (dense) layers. By modularizing these layers, the design becomes more manageable, scalable, and reusable across different architectures. Each layer is custom-made to handle specific tasks: convolution layers perform feature extraction using learned kernels, pooling layers reduce spatial dimensions and enhance translational invariance, and dense layers handle high-level reasoning and classification tasks.

To introduce non-linearity and enable complex pattern learning, the design incorporates activation functions such as ReLU (Rectified Linear Unit) and Softmax. ReLU is applied after convolution and dense layers to zero out negative values, introducing sparsity and enabling efficient hardware usage. The Softmax function, typically used in the final classification stage, converts the output scores into probabilities, aiding in decision-making when classifying input data.

All arithmetic operations in the system are performed using quantized fixed-point arithmetic rather than floating-point. This approach significantly reduces the hardware complexity, resource utilization, and power consumption while maintaining sufficient precision for neural

network inference. The fixed-point representation is carefully chosen to balance numeric range and precision, ensuring accurate computation without overflow or underflow.

A notable feature of the design is the ensemble framework. Instead of relying on a single CNN model, the system supports running multiple CNNs in parallel. This ensemble approach enhances prediction robustness and accuracy. Mechanisms such as majority voting or weighted averaging are employed to combine the outputs of the individual models, allowing the ensemble to leverage the strengths of each member.

The entire implementation is developed and validated using the Xilinx Vivado toolchain. Vivado provides a comprehensive environment for hardware design, simulation, synthesis, and timing analysis. It ensures that the final design meets all performance and power constraints, and is suitable for deployment on FPGA platforms. This toolchain also facilitates integration, debugging, and optimization, making it an essential part of the development workflow.

The bottom-up approach begins with the design and implementation of individual CNN layers and operations in Verilog. Each layer is crafted as a reusable module, parameterized to accommodate variations in the network architecture. These modules are then connected to form the complete CNN, with additional logic to support the ensemble configuration. The use of fixed-point arithmetic ensures compatibility with the quantized data format established earlier, while parallelism is exploited to enhance throughput.

The framework is tested on the trained ensemble model, with inputs processed through the FPGA and outputs validated against expected results. This ensures that the hardware implementation aligns with the software model's performance while achieving real-time efficiency.

3.5.1 Quantized Addition and Multiplication

In the context of Field-Programmable Gate Arrays (FPGAs) for ensemble Convolutional Neural Networks (CNNs), efficient arithmetic operations are paramount to achieving energy efficiency, computational performance, and real-time inference under resource constraints. This subsection presents the design and implementation of quantized addition, multiplication, and multiply accumulate (MAC) operations. While the examples and discussions use a 24-bit and 16-bit fixed-point format for clarity, it is important to note that the implementation is not limited to this specific bit width. Instead, it incorporates a generic parameter WIDTH, allowing

the bit width to be adjusted according to the precision needs of the application. This flexibility ensures that the framework can be tailored to balance resource utilization and computational accuracy, making it suitable for a wide range of applications from low-power embedded systems to high-precision scientific computations.

These operations, implemented in Verilog, use two's complement representation for signed numbers. They include overflow detection and saturation logic to maintain numerical stability while optimizing the use of hardware resources.

3.5.1.1 *Quantized Addition*

The quantized addition module, ***quant_add***, facilitates the signed addition of two fixed-point numbers, parameterized by WIDTH, targeting FPGA deployment. The design employs combinational logic for addition and overflow handling, with the result registered synchronously to support pipelined architectures.

In two's-complement arithmetic, addition is performed as standard binary addition, but overflow detection is essential to prevent wrap-around errors that could compromise neural network accuracy. Overflow occurs when two inputs with the same sign yield a sum with a different sign. The module detects this condition and applies saturation logic, clamping the result to the maximum positive or minimum negative value, thereby maintaining numerical integrity without the overhead of floating-point arithmetic.

```
Algorithm quant_add(clk, ce, in1, in2) → sum:
    Inputs: clk, ce, in1, in2 (WIDTH bits)
    Output: sum (WIDTH bits, registered)

    wires:
        sum_temp = in1 + in2
        sum_comb

        if (MSB(in1) equals MSB(in2)) and (MSB(sum_temp) not equal MSB(in1)) then
            if MSB(in1) is 1 then
                sum_comb = MIN_NEGATIVE
            else
                sum_comb = MAX_POSITIVE
        else
            sum_comb = sum_temp          // no overflow
        end if

    On rising edge of clk:
        if ce is high then
            sum ← sum_comb
        end if
```

The ***quant_add*** module accepts two inputs, in1 and in2, each of width WIDTH, and produces a registered output, sum, of the same width.

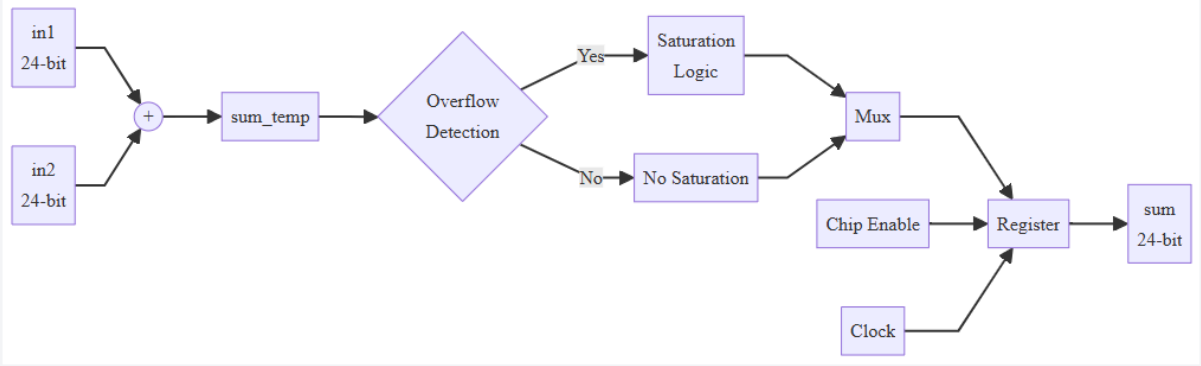


Figure 3.7 quan_add module with 24-bit width

3.5.1.2 Quantized Multiplication

The ***quant_mul*** module performs signed multiplication of two fixed-point numbers, parameterized by *WIDTH* and *FRAC*, ensuring precision and efficiency by aligning the result with the fractional bits and applying saturation on overflow.

Multiplying two fixed-point numbers with *FRAC* fractional bits produces a product of width 2 × *WIDTH*. The result is extracted to maintain the *WIDTH*-bit format, aligning with the fractional representation. Overflow is detected by comparing the upper bits of the full product to the sign bit, with saturation applied to preserve accuracy. This fixed-point approach reduces power consumption compared to floating-point multiplication, supporting the energy efficiency objective.

```
Algorithm quant_mul(clk, ce, op_a, op_b) → result:

Inputs:
    clk, ce
    op_a, op_b : WIDTH-bit signed inputs

Output:
    result : WIDTH-bit registered output

Wires:
    prod       = signed(op_a) * signed(op_b)
    temp       = prod[FRAC + WIDTH - 1 : FRAC]
    sign_bit   = prod[FRAC + WIDTH - 1]
    overflow   = OR(prod[2*WIDTH-1 : FRAC+WIDTH]
                    XOR {sign_bit, ...})
    max_val    = 2^(WIDTH-1) - 1
    min_val    = -2^(WIDTH-1)
    result_next = overflow ? (sign_bit ? min_val : max_val) : temp

On rising edge of clk:
    if ce:
        result ← result_next
```

The ***quant_mul*** module processes inputs *op_a* and *op_b* to produce a *WIDTH*-bit output, result.

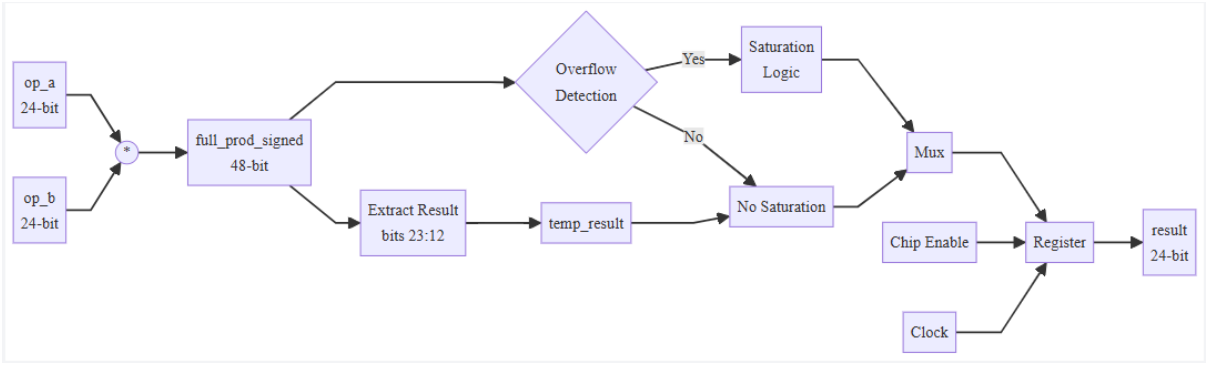


Figure 3.8 quan_mul module with 24-bit width

This design optimizes FPGA resources and supports real-time performance through synchronous operation, with *WIDTH* and *FRAC* parameters allowing customization for different precision requirements.

3.5.1.2 *Quantized Multiply-Accumulate (MAC) Operation*

The ***mul_acc*** module integrates multiplication and addition into a single operation, essential for weighted sums in neural network layers. It enhances computational efficiency by streamlining the data path, with parameters *WIDTH* and *FRAC* allowing flexibility in bit width and fractional precision.

The MAC operation multiplies two inputs, extracts the aligned result, and adds it to a third input (e.g., a bias). Overflow detection and saturation are applied to the addition step, ensuring stability. This combined operation reduces latency and resource usage, aligning with the performance objectives.

```
Algorithm mul_acc(clk, reset, ce, op_a, op_b, op_c) → result:
    Inputs:
        clk, reset, ce
        op_a, op_b, op_c : WIDTH-bit signed fixed-point

    wires:
        mul_result = (op_a * op_b)[FRAC + WIDTH - 1 down to FRAC]
        sum_temp = mul_result + op_c

        sum_comb:
            if overflow occurs (as defined previously in quant_add):
                clamp sum_temp to max/min limit based on sign
            else:
                sum_temp

    On rising edge of clk or reset:
        if reset:
            result ← 0
        else if ce:
            result ← sum_comb
```

The ***mul_acc*** module processes inputs *op_a*, *op_b*, and *op_c* to produce a WIDTH-bit result

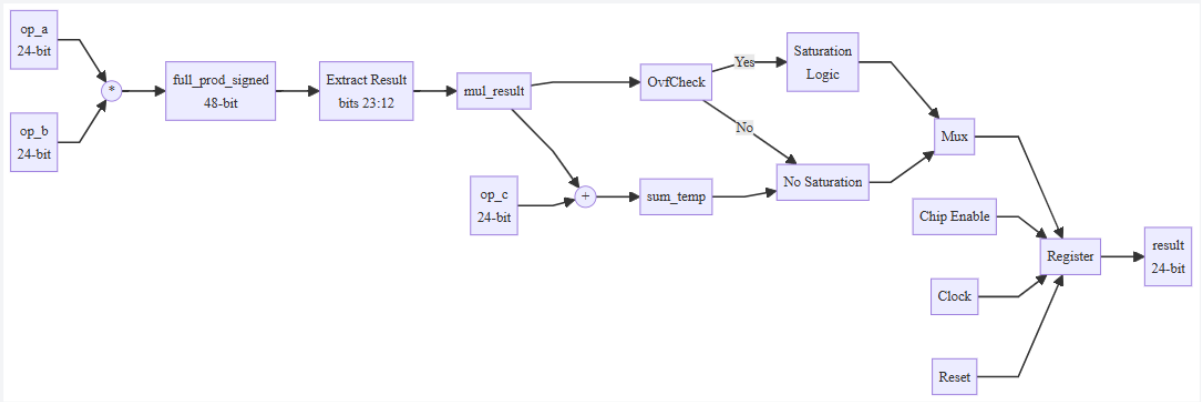


Figure 3.9 mul_acc module with 24-bit width

3.5.2 Common Utility Modules

In the FPGA-based framework for ensemble Convolutional Neural Networks (CNNs), utility modules play a critical role in supporting efficient data handling, memory management, and signal processing. This subsection details the design and implementation of three key modules: the shift register, Block RAM (BRAM) memory unit, and comparison module. These modules are parameterized for flexibility, optimized for FPGA deployment, and designed to meet the objectives of energy efficiency, computational performance, and real-time inference under resource constraints.

3.5.2.1 *Shift Register*

The shift register module, ***shift_reg***, is essential for sequential data processing in CNNs, such as sliding windows in convolution and pooling operations. It is parameterized by WIDTH (bit width of each element) and SIZE (number of stages), enabling adaptability to various layer configurations.

The shift register employs a hybrid design combining a flip-flop (FF) for the first stage and Shift Register LUTs (SRLs) for subsequent stages. The FF ensures reliable reset behavior, while SRLs leverage FPGA LUT resources to minimize flip-flop usage, enhancing energy efficiency. Data shifts through the stages on each clock cycle when enabled by the chip enable (ce) and valid input signals, reducing unnecessary operations and power consumption.

Table 3.3 Parameter description of the shift_reg module

Parameter	Description	Usage

WIDTH	Bit width of each element in the shift register	Configures the data width to match CNN data requirements (e.g., 8, 16, 32 bits)
SIZE	Number of stages in the shift register	Determines delay length for operations such as sliding windows in convolution (e.g., 3, 5, 7 for different kernel sizes)

The algorithm of the module is as described below.

```
Algorithm shift_reg(clk, reset, ce, valid_input, d) → q:

Inputs:
    clk, reset, ce, valid_input
    d : WIDTH-bit input

Output:
    q : WIDTH-bit registered output (delayed by SIZE cycles)

Registers:
    first_stage           : First stage with reset
    srl_stages[0:SIZE-2]  : Shift stages without reset

On rising edge of clk or reset:
    if reset:
        first_stage ← 0
    else if ce & valid_input:
        first_stage ← d

On rising edge of clk:
    if ce & valid_input:
        for i in 0 to SIZE-2:
            srl_stages[i] ← (i == 0) ? first_stage : srl_stages[i-1]

Output:
    q ← (SIZE == 1) ? first_stage : srl_stages[SIZE-2]
```

The ***shift_reg*** module accepts a WIDTH-bit input d and produces a WIDTH-bit output q, delayed by SIZE clock cycles. The first stage uses a flip-flop with asynchronous reset, while remaining stages are implemented as SRLs to optimize resource usage. The output is assigned based on the SIZE parameter, ensuring flexibility for single-stage or multi-stage configurations.

3.5.2.2 *BRAM Memory Unit*

The ***simple_bram*** module provides a parameterized Block RAM memory unit for storing CNN weights, biases, and intermediate data. It is parameterized by WIDTH (data width) and DEPTH (number of memory locations), offering scalability for diverse memory requirements.

The module is designed as a single-port, synchronous BRAM, utilizing FPGA dedicated memory resources for high-density, low-power storage. It supports write enable (we) for controlled updates and separate read and write addresses for flexible data access, critical for CNN dataflow optimization.

Table 3.4 Parameter description of the simple_bram module

Parameter	Description
WIDTH	Data width in bits
DEPTH	Number of memory locations
ADDR_WIDTH	Calculated as $\log_2$(DEPTH) – width of address bus

The memory operates synchronously with the clock signal (clk). On each rising edge of the clock: If write enable (we) is high, data from data_in is written to the memory location specified by write_addr. Simultaneously, data from the memory location specified by read_addr is read and output on data_out.

```
Algorithm simple_bram(clk, we, write_addr, read_addr, data_in) → data_out:
    Inputs:
        clk, we
        write_addr, read_addr : ADDR_WIDTH-bit addresses
        data_in : WIDTH-bit input
    Output:
        data_out : WIDTH-bit output

    Memory:
        mem : Array of WIDTH-bit registers [0 to DEPTH-1]

    On rising edge of clk:
        if we:
            mem[write_addr] ← data_in
        data_out ← mem[read_addr]
```

The module manages a WIDTH-bit input data_in and output data_out, with memory operations synchronized to the clock. The design infers BRAM on the FPGA, minimizing power consumption and maximizing storage efficiency.

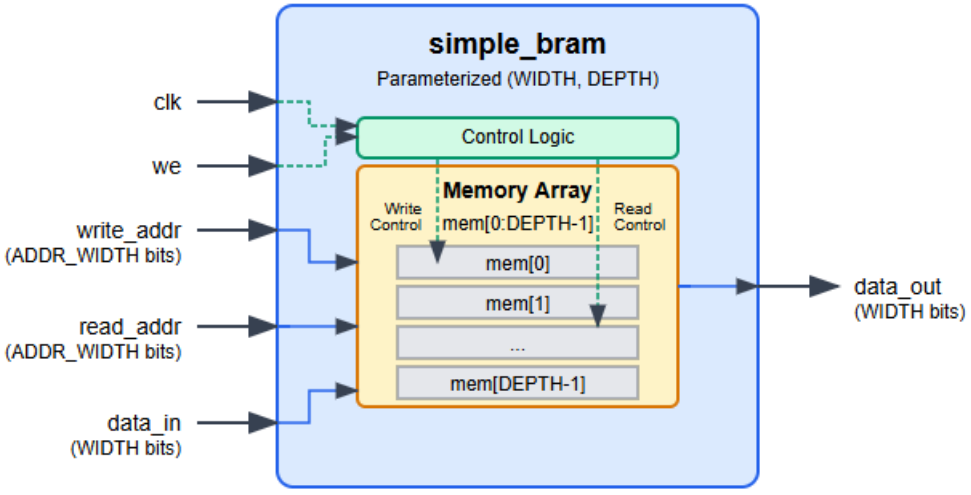


Figure 3.10 Block diagram of simple_bram module

3.5.2.3 *Comparison Module*

The ***comp2*** module performs a signed comparison of two fixed-point numbers, selecting the maximum value, which is vital for operations like max-pooling. It is parameterized by WIDTH for flexibility in bit width. The module uses combinational logic to compare two inputs, in1 and in2, based on their signs and magnitudes, with the result registered synchronously to support pipelining. A valid output signal ensures data integrity in streaming applications, aligning with real-time performance goals.

```
Algorithm comp2(clk, reset, ce, valid_input, in1, in2) → out_val, valid_output:
    Inputs:
        clk, reset, ce, valid_input
        in1, in2 : WIDTH-bit signed inputs
    Outputs:
        out_val : WIDTH-bit registered output
        valid_output : Registered valid signal

    Wires:
        max_val :
            if in1[MSB] = in2[MSB]:
                (in1 > in2) ? in1 : in2
            else:
                (in1[MSB] = 0) ? in1 : in2

    On rising edge of clk or reset:
        if reset:
            out_val ← 0
            valid_output ← 0
        else if ce:
            out_val ← max_val
            valid_output ← valid_input
```

The ***comp2*** module processes WIDTH-bit inputs in1 and in2, producing a registered WIDTH-bit output out_val. The combinational comparison ensures minimal latency, while synchronous registration supports high-throughput designs.

3.5.3 Convolutional Layers

The FPGA-based implementation described here emphasizes optimization for energy efficiency, high computational performance, and real-time inference, specifically addressing resource-constrained environments. This section details two critical Verilog modules: ***conv2d*** for single-channel convolutions and ***convmd*** for multi-channel convolutions, clearly delineating their algorithmic structure, design rationale, and performance benefits. These modules employ quantized arithmetic discussed previously and integrate utility modules, forming a cohesive, efficient framework.

3.5.3.1 *Single-Channel Convolution*

The ***conv2d*** module executes 2D convolutions on single-channel input activation maps, using square kernels with configurable dimensions. Parameters including image size, kernel size, stride, data width, and fractional bit precision offer adaptability for various CNN architectures. Inputs include standard clock, reset, chip enable signals, a streaming pixel input, flattened kernel weights, and a validity indicator. Outputs provide convolution results per kernel window, a validity signal indicating correct results, and a completion signal denoting when convolution processing is finished.

Table 3.5 Parameter description of the conv2d module

Parameter	**Description**
IMG_SIZE	Dimension of the square input activation map (IMG_SIZE × IMG_SIZE pixels)
KERNEL_SIZE	Dimension of the square convolution kernel (KERNEL_SIZE × KERNEL_SIZE weights)
STRIDE	Sliding step size for the convolution kernel (same for horizontal and vertical)
WIDTH	Total bit width for data representation (activations, weights, and results)
FRAC	Number of fractional bits in the fixed-point representation

This module employs a streaming dataflow approach, where activations sequentially enter the processing chain, significantly reducing on-chip memory demands and power usage. Multiply-accumulate (MAC) operations are chained, utilizing FPGA parallelism to maximize throughput. Shift registers align data across rows, strategically employing FPGA LUT-based shift registers (SRLs) rather than memory-intensive solutions, optimizing both power consumption and FPGA resource allocation.

```
Algorithm conv2d(clk, reset, ce, activation, kernel, valid_input) → conv_out, conv_valid,
conv_done:
```

```
  Inputs:
    - clk, reset, ce, valid_input
    - activation: single input pixel (WIDTH bits)
    - kernel: flattened array of KERNEL_SIZE × KERNEL_SIZE weights

  Outputs:
    - conv_out: convolution result
    - conv_valid: high when conv_out is valid
    - conv_done: high when full image convolution is complete

  Internal:
    - Line buffer stores KERNEL_SIZE rows of IMG_SIZE pixels (sliding window)
    - Kernel buffer holds weights loaded once at IDLE
    - MAC (mul_acc) computes sum of element-wise products
    - FSM: IDLE → COMPUTE → OUTPUT

  Operation:
    1. Reset clears buffers, counters, and state.
    2. On each valid input (when ce & valid_input):
       - Shift activation into the bottom row of the line buffer
       - At IDLE, load kernel weights into buffer
       - Transition to COMPUTE state

    3. In COMPUTE:
       - For each kernel weight:
         - Select corresponding activation from line buffer (with boundary check and zero
padding)
         - Feed activation and weight to MAC
         - Accumulate output product in sum_reg
       - After all weights processed, move to OUTPUT

    4. In OUTPUT:
       - Set conv_out to accumulated sum
       - Assert conv_valid
       - Count valid outputs
       - If all output positions are covered, assert conv_done
       - Return to IDLE
```

Control logic based on cycle counting precisely manages convolution result validity and the completion of image processing, ensuring synchronization with subsequent CNN layers. Parameterization supports flexibility, allowing adjustments to kernel size, image dimensions, and data precision to accommodate diverse CNN implementations. The chained MAC design substantially minimizes data movement, improving computational efficiency.

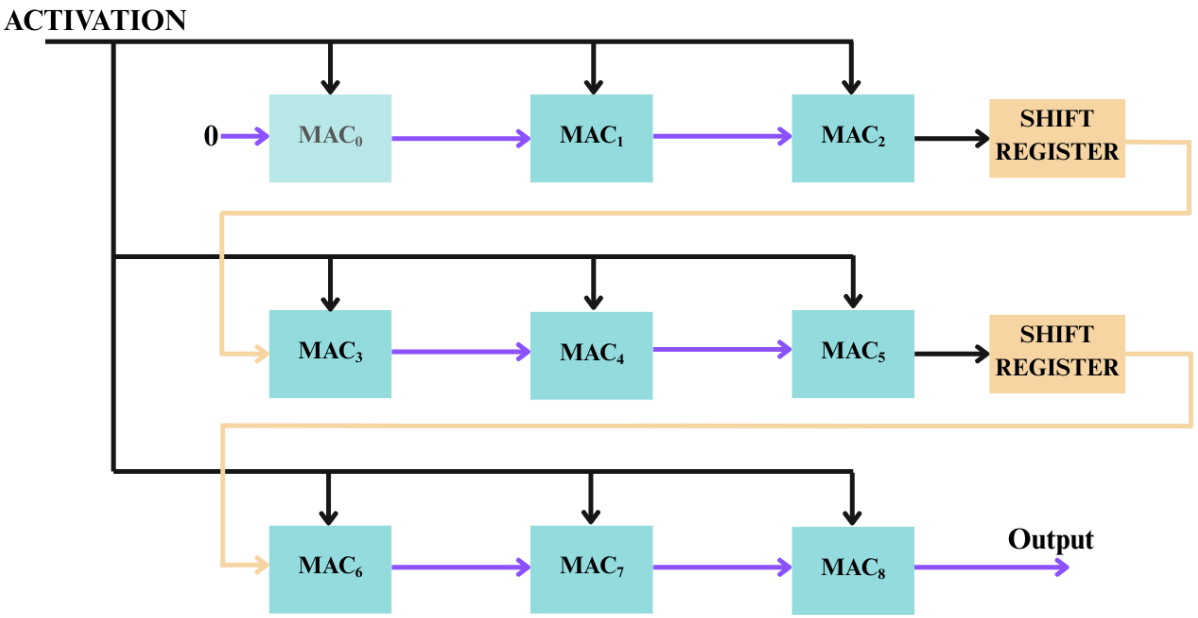


Figure 3.11 Convolution operation for 3x3 Kernel

The conv2d module implements a streaming convolution approach that efficiently utilizes FPGA resources by combining line buffers, shift registers, and a chained multiply-accumulate (MAC) structure. Incoming activation pixels are sequentially streamed and stored across multiple rows in a line buffer, simulating a sliding convolution window without requiring full image storage. Shift registers are used to realign activation data across rows and columns by introducing cycle-based delays, eliminating the need for large memory blocks and enabling precise spatial alignment of inputs for each kernel position. The MAC unit operates iteratively, multiplying each activation with its corresponding kernel weight and accumulating the results over several clock cycles. Control logic orchestrates the computation by sequencing kernel accesses, managing partial sum accumulation, and signaling when a valid result is ready or when the entire image has been processed. This design allows for high-throughput, low-latency convolution operations tailored to fixed-point arithmetic, making it ideal for power- and memory-efficient CNN implementations on FPGAs.

3.5.3.2 *Multi-Channel Convolution*

The ***convmd*** module expands the capabilities of ***conv2d*** to accommodate multi-channel input and output scenarios, vital for processing data from complex sources such as RGB images or deep CNN layers. Parameters from ***conv2d*** are inherited, with additional inputs specifying the number of input and output channels. Inputs include concatenated multi-channel activations, kernel weights for each channel pair, biases for output channels, and an optional ReLU

activation flag, along with standard control signals and validity indicators. Outputs provide results for all output channels with synchronized control signals.

This design extensively employs parallelization by instantiating multiple ***conv2d*** modules concurrently, one per input-output channel pair, significantly enhancing throughput and computational efficiency by utilizing FPGA parallel processing capabilities. Partial convolution results from each input-output pair are efficiently summed using quantized arithmetic through a chain of ***quant_add*** modules, maintaining an efficient pipeline flow. Bias addition and conditional combinational ReLU activation introduce non-linearity with minimal latency, further optimizing computational speed.

```
Algorithm convmd(clk, reset, ce, valid_input, activation, kernel, bias, relu_act) →
conv_out, conv_valid, conv_done:
  Inputs:
    - clk, reset, ce, valid_input
    - activation : C_in × WIDTH-bit input vector (per input channel)
    - kernel     : C_out × C_in × KERNEL_SIZE² × WIDTH-bit weights
    - bias       : C_out × WIDTH-bit bias vector
    - relu_act   : flag to enable ReLU activation

  Outputs:
    - conv_out   : C_out × WIDTH-bit output vector
    - conv_valid : signal when output is valid
    - conv_done  : signal when all output pixels are computed

  Internal Logic:
    → 1. For each output channel `c_out` ∈ [0, C_out):
         → For each input channel `c_in` ∈ [0, C_in):
             - Instantiate `conv2d` to process one (c_in, c_out) pair:
               activation  = activation[c_in]
               kernel      = kernel slice for c_out, c_in
               output      = partial_conv_out[c_out][c_in]

    → 2. For each c_out:
         → Sum partial outputs from all input channels:
             if C_in == 1:
               sum_partials = partial_conv_out[c_out][0]
             else:
               sum_chain[0] = partial_conv_out[c_out][0]
               for i = 1 to C_in-1:
                   sum_chain[i] = quant_add(sum_chain[i-1], partial_conv_out[c_out][i])
               sum_partials = sum_chain[C_in-1]

    → 3. Add bias:
         biased_sum = quant_add(sum_partials, bias[c_out])

    → 4. Apply ReLU if enabled:
         if relu_act:
             conv_out[c_out] = (biased_sum[WIDTH-1] == 0) ? biased_sum : 0
         else:
             conv_out[c_out] = biased_sum

  Control:
    → All conv2d instances share the same control signals.
    → conv_valid = partial_conv_valid[0][0]
    → conv_done  = partial_conv_done[0][0]
```

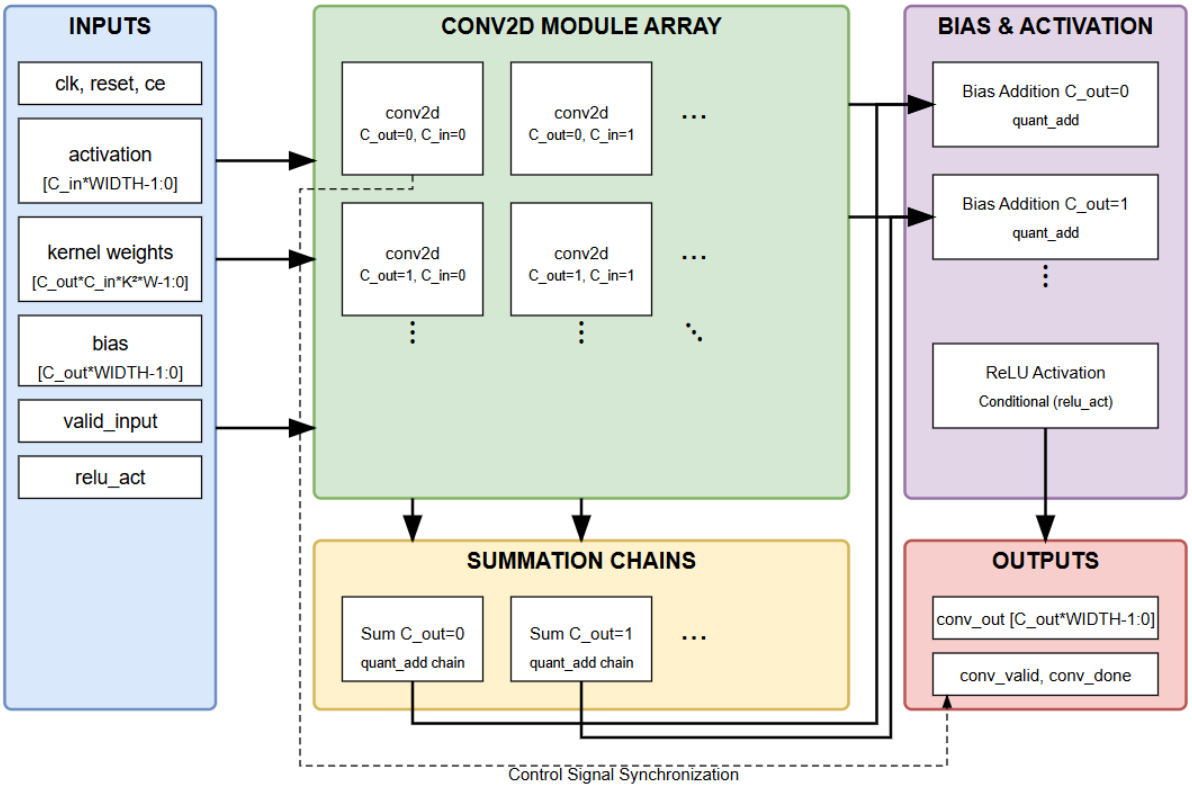


Figure 3.12 Module Architecture of convmd

3.5.4 Padding Modules

To ensure correct spatial alignment during convolution operations, padding is a critical preprocessing step. In convolutional neural networks (CNNs), especially when using valid or same padding strategies, input activation maps must be extended by a border of zeros around their edges. This is essential to preserve spatial dimensions or to center the convolution kernel appropriately over the input. The FPGA-based implementation discussed in the previous section introduces a streaming convolution architecture. To maintain this streaming model while handling boundary conditions correctly, dedicated padding modules are required.

The ***pad_stream*** and ***pad_stream_md*** modules were developed to support both single-channel and multi-channel input maps respectively. These modules integrate seamlessly with the ***conv2d*** and ***convmd*** modules, offering pipelined padding functionality that complements the dataflow-oriented convolution design.

3.5.4.1 *Single-Channel Streaming Padding*

The ***pad_stream*** module implements zero-padding for a single input channel in a streaming fashion. It receives one activation pixel per clock cycle and transforms the input image of size IMG_SIZE × IMG_SIZE into a padded image of size (IMG_SIZE + 2) × (IMG_SIZE + 2).

This is done by surrounding the input image with a border of zero-valued pixels. The module operates in two distinct phases: write and read.

During the write phase, incoming pixels are written row by row into a circular buffer (implemented using a dual-port BRAM). A write pointer tracks the location in memory, and writing continues until all pixels of the original image have been stored. Once the image is fully written, the module transitions to the read phase.

In the read phase, the module emits one pixel per clock cycle, reconstructing the padded image row by row. For boundary pixels (top row, bottom row, leftmost column, and rightmost column), it directly outputs a zero. For inner pixels, it calculates the appropriate read address in the buffer and retrieves the pixel data. This calculation uses modular arithmetic to compute the buffer index based on the desired padded row and column positions.

```
Algorithm pad_stream(clk, reset, ce, input_act, valid_input) → padded_act, padded_ce, pad_end:

  Inputs:
    - clk, reset, ce, valid_input
    - input_act: single input pixel (WIDTH bits)
  Outputs:
    - padded_act: padded pixel output
    - padded_ce: high when padded_act is valid
    - pad_end: high when padding is complete
  Internal:
    - BRAM buffer of size IMG_SIZE * 3
    - write_ptr: next write location in BRAM
    - row, col: position in padded image
    - out_count: number of padded pixels outputted
  Operation:
    1. On reset:
       - Clear padded_act, padded_ce, pad_end, out_count, row, col, write_ptr
    2. On posedge clk:
       - If ce && (valid_input || out_count >= IMG_TOTAL):
         - If out_count < PAD_TOTAL:
           - If row or col is on border (0 or PADDED_SIZE-1):
             - padded_act = 0
           - Else:
             - padded_act = BRAM[next_read_ptr]
           - padded_ce = 1
           - If out_count < IMG_TOTAL:
             - Write input_act to BRAM[write_ptr]
             - write_ptr = (write_ptr + 1) % BUFFER_SIZE
           - Update row, col for next position
           - out_count = out_count + 1
         - Else:
           - pad_end = 1
           - padded_ce = 0
       - Else:
         - padded_ce = 0
         - pad_end = 0
```

Internally, the module tracks its position within the padded output using a pair of counters (row and col) and a global counter out_count. It outputs valid padded pixels on padded_act with an associated enable signal padded_ce. Once the full padded image has been emitted, it asserts the pad_end signal to inform downstream modules (like the convolution core) that the padding stage is complete.

This module is fully synchronous with the system clock and uses chip enable (ce) and a data-valid signal (valid_input) to control when it accepts inputs. By streaming the padded image linearly without requiring full buffering of the input in memory, it minimizes latency and avoids costly memory overhead, in line with the overall low-resource FPGA design philosophy.

3.5.4.2 *Multi-Channel Streaming Padding*

The pad_stream_md module generalizes the functionality of pad_stream to support multiple input channels concurrently, such as those produced by earlier convolutional layers in a CNN. This module takes a vector of activations each element corresponding to a separate input channel and performs independent padding on each one in parallel.

```
Algorithm pad_stream_md(clk, reset, ce, pixels_in, valid_input) → padded_act, padded_ce,
pad_end:
```

```
  Inputs:
    - clk, reset, ce, valid_input
    - pixels_in: C_in * WIDTH bits (pixels for all channels)
  Outputs:
    - padded_act: C_in * WIDTH bits (padded pixels)
    - padded_ce: high when padded_act is valid
    - pad_end: high when padding is complete
  Operation:
    1. For each channel c (0 to C_in-1):
       - input_act[c] = pixels_in[c*WIDTH +: WIDTH]
       - Run pad_stream[c] with input_act[c]
       - Collect padded_act[c] from pad_stream[c]
    2. Concatenate padded_act[0] to padded_act[C_in-1] into padded_act
    3. Set padded_ce = pad_stream[0].padded_ce
    4. Set pad_end = pad_stream[0].pad_end
```

Internally, pad_stream_md instantiates one pad_stream module per channel. Each instance receives the corresponding slice of the multi-channel input vector and processes it independently. This parallel instantiation strategy ensures that all channels are padded simultaneously and uniformly, maintaining channel-wise alignment required for convolution.

The outputs from the individual pad_stream instances are then reassembled into a single vector padded_act, which matches the original format of the input activation map. Although each pad_stream operates independently, they are inherently synchronized because they share the same clock, chip enable, and control signals. Therefore, the module propagates the padded_ce and pad_end signals from the first pad_stream instance, assuming all instances transition states simultaneously.

This approach ensures that the padded multi-channel activation map is fully compatible with the convmd module, which expects pixel-wise data from multiple channels at once. The use of parallel padding avoids bottlenecks and allows full utilization of downstream convolutional processing resources.

The padding modules are tightly integrated with the convolution cores, both logically and temporally. Rather than pre-padding entire images in external memory, the system pads each activation in real-time, on-chip, as it arrives. This streaming approach to padding aligns with the FPGA-optimized convolution architecture. By capturing the padding logic within dedicated hardware modules, the overall system design remains modular and composable. Padding becomes an isolated, reusable hardware function that can be easily paired with any streaming convolution engine. It supports various CNN models with different input sizes and channel depths, all while preserving the efficiency and scalability needed for real-time inference on resource-constrained platforms.

3.5.5 Pooling Layers

This section details the FPGA-based implementation of pooling layers, specifically focusing on max-pooling, through two Verilog modules: ***max_pool*** for single-channel pooling and ***max_pool_md*** for multi-channel pooling. The pooling operations follow a structured and parameterized approach to ensure flexibility, scalability, and efficient resource utilization.

3.5.5.1 *Single-Channel Pooling*

The ***max_pool*** module performs max-pooling operations on single-channel activation maps. It processes input pixels in a streaming fashion, sequentially evaluating pixel values within specified pooling windows to select the maximum value. Parameters including image size (IMG_SIZE), pooling window dimension (POOL_DIM), stride (STRIDE), and data bit width (WIDTH) ensure the module can adapt to varying CNN configurations.

Table 3.6 Parameter description of the max_pool module

Parameter	**Description**
IMG_SIZE	Width and height of the input image (square image assumed)
POOL_DIM	Dimension of the pooling window (POOL_DIM × POOL_DIM)
STRIDE	Step size for the pooling window in both horizontal and vertical directions
WIDTH	Bit width for pixel values (fixed-point representation)

Inputs to the ***max_pool*** module include standard clock, reset, and chip enable signals, along with streaming input pixels and a validity indicator. Outputs comprise the maximum pooled value for each window, a valid signal indicating correct output, and a completion signal indicating the end of processing for the current input image.

```
Algorithm max_pool(clk, reset, ce, pixel_in, valid_input):
    Inputs:
        pixel_in: single pixel streamed per clock (row-major)
        valid_input: indicates valid pixel input

    Outputs:
        max_out: maximum pixel in each POOL_DIM × POOL_DIM window
        pool_valid: indicates max_out is valid
        pool_end: indicates entire image processed

    Counters (on rising edge of clk):
        Increment pixel position counters (col, row, total pixels processed).

    Pooling Logic:
        Compare adjacent pixels horizontally (comp_adj) → max_adj
        Delay max_adj by one row (using shift_reg) → max_upper
        Compare vertically (comp_up): max_adj vs max_upper → pool_result

    Output Logic (on clk rising edge):
        If pooling window complete:
            max_out ← pool_result
            pool_valid ← 1
        Else:
            pool_valid ← 0

    Completion Logic:
        Set pool_end = 1 when all pixels processed, else 0
```

Internal counters track row and column positions to manage pixel flow and determine pooling window boundaries. Shift registers delay pixel values to align adjacent and vertically neighboring pixels required for pooling comparisons. Combinational comparison logic, utilizing comp2 modules, identifies the maximum pixel value within each pooling window.

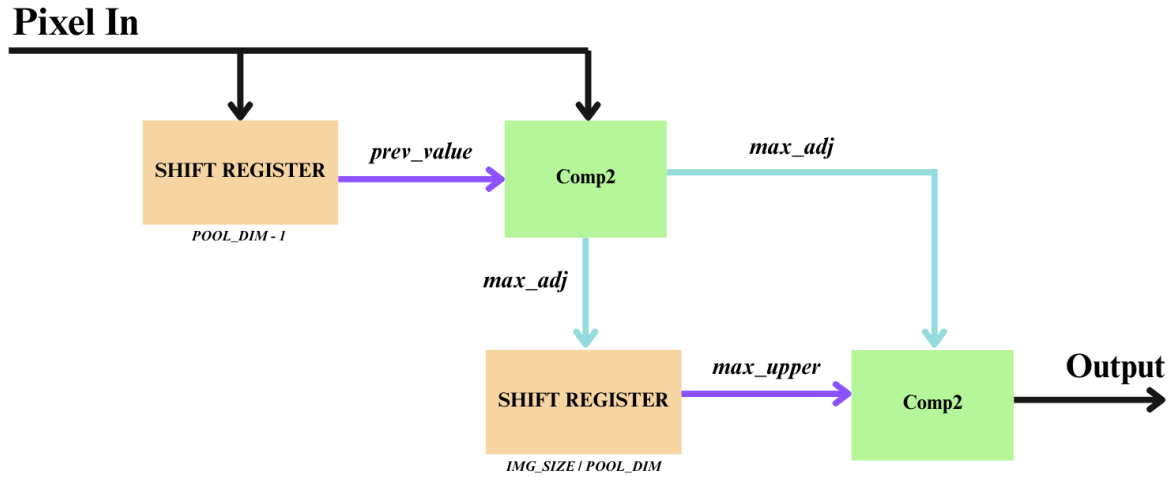


Figure 3.13 2x2 Max pooling window logic

Precise control logic generates validity and completion signals, ensuring synchronization with downstream CNN operations. The implementation's streamlined dataflow significantly reduces memory access overhead, contributing to energy efficiency and supporting real-time inference requirements.

3.5.5.2 *Multi-Channel Pooling*

The ***max_pool_md*** module extends the ***max_pool*** capabilities to handle multi-channel inputs simultaneously, crucial for modern CNN architectures processing complex input data. Parameters from ***max_pool*** are inherited, with additional parameters specifying the number of input channels (C_in).

The module inputs include concatenated pixels across multiple channels, standard control signals, and validity indicators. Outputs provide concatenated maximum pooled results across all channels, along with synchronized validity and completion signals.

```
Algorithm max_pool_md(clk, reset, ce, pixels_in, valid_input):
    Inputs:
        pixels_in: streamed pixels, C_in channels in parallel
        valid_input: indicates valid pixel input

    Outputs:
        max_out: pooled results, C_in channels
        pool_valid, pool_end: status signals

    For each input channel c in 0 to C_in-1 (in parallel):
        Run max_pool on pixels_in[c] → max_out[c]

    Combine max_out from all channels into a single output vector

    pool_valid and pool_end are taken from the first max_pool instance (all instances
synchronized)
```

Internally, the module instantiates multiple parallel ***max_pool*** instances, each processing an individual channel independently. Results from these parallel operations are combined into a unified output vector, facilitating synchronized control signals for seamless downstream integration. The parallelized approach utilized in the ***max_pool_md*** module maximizes FPGA resource utilization, significantly increasing computational throughput and enabling real-time pooling operations across multiple channels.

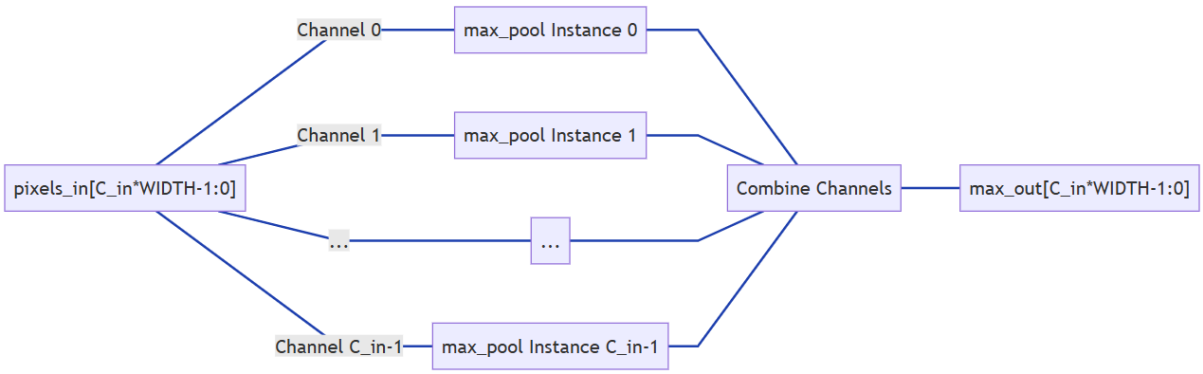


Figure 3.14 multi-channel data flow of max pooling

The pooling layer implementation described here prioritizes energy efficiency, computational performance, and adaptability. Fixed-point arithmetic reduces power consumption, while the streaming dataflow approach minimizes memory usage. Parallel computation across multiple channels further enhances processing speed, crucial for real-time inference scenarios. The modular, parameterized architecture facilitates scalability and ease of integration within various CNN frameworks, effectively addressing resource constraints and performance objectives inherent in FPGA-based CNN deployments.

3.5.6 Dense Layers

Dense (fully connected) layers form an essential component of convolutional neural networks (CNNs), particularly in the classification stages. These layers take flattened feature maps from convolutional and pooling layers and produce output predictions. This FPGA-based implementation introduces two essential Verilog modules: the ***neuron*** module for individual neuron computations and the ***dense_layer*** module for multi-neuron dense layer computations. Both modules use parameterized, quantized arithmetic to ensure flexibility, scalability, and resource efficiency.

3.5.6.1 *Neuron Module*

The ***neuron*** module implements a single artificial neuron, calculating a weighted sum of inputs and producing an output value. It performs a dot product between a stream of input values and a corresponding set of weights, producing a single scalar output. Key parameters such as IN_SIZE, WIDTH, FRAC, and MUL_LATENCY govern the number of inputs, fixed-point representation, and multiplication pipeline latency.

The internal logic includes a looped accumulation of input-weight products using a single mul_acc instance. Control logic ensures alignment between data arrival and multiplier latency, managing the accumulation process sequentially across input indices. Once all inputs have been processed, the result is output and synchronized with the rest of the system.

```
Algorithm neuron(clk, reset, ce, valid_input, in_data, index, weights) → neuron_out,
valid_output, neuron_end:
```

```
Inputs:
  - clk, reset, ce, valid_input
  - in_data      : single input value (WIDTH bits)
  - index        : index of the current input (log2(IN_SIZE) bits)
  - weights      : flattened vector of IN_SIZE weights

Outputs:
  - neuron_out   : accumulated dot-product result
  - valid_output : high when output is valid
  - neuron_end   : high one cycle after valid_output

Internal:
  - weights_arr[0:IN_SIZE-1] ← extracted from flattened weights
  - current_input, current_weight → inputs to mul_acc
  - product ← result of current_input × current_weight
  - sum_reg ← accumulated sum
  - count ← current accumulation step
  - accumulating ← state flag for active computation

Logic:
  → 1. On reset:
       - Clear count, sum_reg, flags, output registers

  → 2. On ce:
       - If valid_input and not already accumulating:
           - Begin accumulation (accumulating = 1)
           - Reset count and sum_reg

       - If accumulating:
           - For count < IN_SIZE:
               - Load current_input = in_data
               - Load current_weight = weights_arr[index]
               - After MUL_LATENCY cycles:
                   sum_reg += product
               - Increment count
           - Else:
               - Output result: neuron_out = sum_reg
               - Set valid_output = 1
               - End accumulation

       - If not accumulating:
           - valid_output = 0

       - neuron_end = valid_output (delayed flag)
```

This approach avoids simultaneous multiplication of all input-weight pairs, trading off area efficiency for lower resource usage by time-multiplexing the multiplier hardware. It ensures accurate and deterministic accumulation by carefully managing timing and control signals.

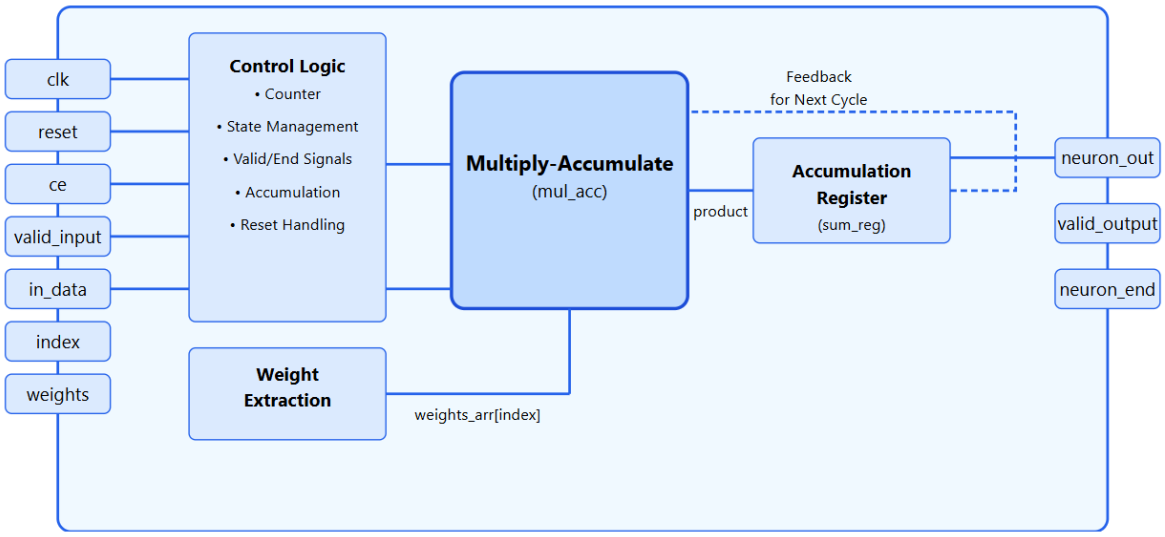


Figure 3.15 Architecture of neuron module

3.5.6.2 *Dense Layer Module*

The ***dense_layer*** module aggregates multiple ***neuron*** modules to form a fully connected layer, essential for classification tasks within CNNs. Parameters extend those of the neuron module, adding the number of neurons (OUT_SIZE). Inputs include standard control signals, flattened input vectors, weights, biases, and an optional ReLU activation flag. Outputs provide concatenated neuron outputs, along with synchronized validity and completion signals.

```
Algorithm dense_layer(clk, reset, ce, valid_input, in_data, weights, biases, relu_act) →
out_data, valid_out, done:

  Inputs:
    - clk, reset, ce, valid_input
    - in_data   : flattened input vector of IN_SIZE elements
    - weights   : flattened matrix (OUT_SIZE × IN_SIZE) weights
    - biases    : flattened bias vector of OUT_SIZE elements
    - relu_act  : ReLU activation enable flag

  Outputs:
    - out_data  : concatenated activated outputs from all neurons
    - valid_out : asserted when output is ready
    - done      : asserted one cycle after valid_out

  Internal:
    → 1. On reset:
         - Clear input index and processing state

    → 2. On ce:
         - If valid_input and not processing:
             - Start processing
             - Reset input index
         - If processing and input_idx < IN_SIZE - 1:
             - Increment input index
         - If input_idx reaches IN_SIZE - 1:
             - Stop processing

    → 3. For each output neuron j ∈ [0, OUT_SIZE):
         - Instantiate a `neuron` module:
             - Feed current_input = in_data[input_idx]
             - Slice weights for neuron j: weights[j]
             - Use common index and valid_input logic
             - Get neuron_out, valid_output, neuron_end
```

```
        - Add bias: biased_sum = neuron_out + biases[j]
        - Apply activation:
            activated_out = (relu_act) ?
                            ((biased_sum ≥ 0) ? biased_sum : 0) :
                            biased_sum
        - Assign activated_out to corresponding slice of out_data[j]

    → 4. Output control:
        - valid_out  ← valid_output from first neuron
        - done       ← delayed version of valid_out
```

Each neuron receives the same input vector but a different slice of the weight matrix. The output from each neuron is post-processed in two stages:

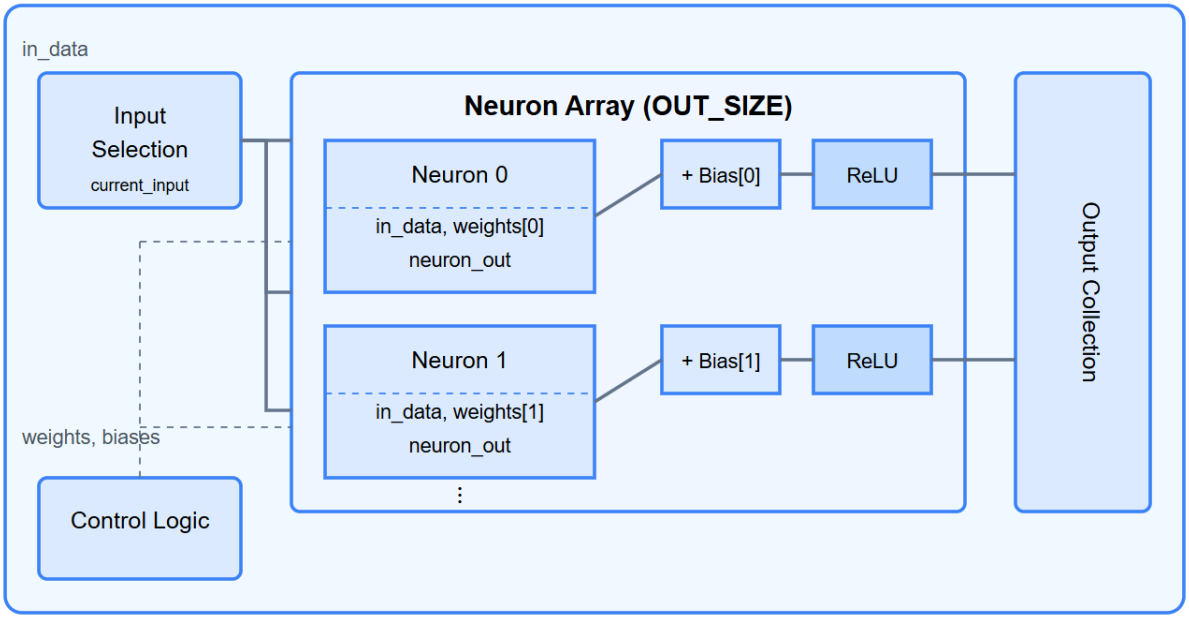


Figure 3.16 Architecture of dense_layer module

While the ***dense_layer*** module processes a vector input in a single clock cycle, the ***dense_flatten*** module is designed to stream and flatten multi-channel, spatial feature maps (e.g., convolutional output tensors) into a serialized format that feeds directly into a dense computation pipeline. It serves as a functional variant of the ***dense_layer*** module, adapted to work with image-like data stored across multiple channels and spatial locations.

In the ***dense_flatten*** implementation inputs are not pre-flattened vectors, but rather spatial maps across C_IN channels and IMG_SIZE × IMG_SIZE pixels. Internally, the module uses BRAMs to buffer input activations, storing them one frame at a time. Once a frame is fully loaded, a

scalar is read from each BRAM sequentially (flattening on-the-fly) and streamed into each neuron.

```
Algorithm dense_flatten(clk, reset, ce, valid_input, input_act, weights, biases,
relu_act)  → out_data, valid_out, done
```

```
  Inputs:
    - clk, reset, ce
    - valid_input     : asserted when input_act is valid
    - input_act       : C_IN × WIDTH-bit vector per input frame (IMG_SIZE × IMG_SIZE)
    - weights         : (IN_SIZE × OUT_SIZE) × WIDTH-bit weight matrix
    - biases          : OUT_SIZE × WIDTH-bit bias vector
    - relu_act        : enable flag for ReLU activation

  Outputs:
    - out_data        : OUT_SIZE × WIDTH-bit output vector
    - valid_out       : asserted when output is valid
    - done            : asserted one cycle after valid_out

  Internal Memory (BRAMs):
    → Create C_IN BRAM blocks of size DEPTH = IMG_SIZE²
    → WRITE mode:
         - For each cycle with valid_input:
             - Write one row (C_IN pixels) to BRAM[0:C_IN-1][write_addr]
             - Increment write_addr
             - Set input_complete when write_addr reaches DEPTH-1

    → READ mode (once input_complete is set):
         - Sequentially output one scalar at a time:
             - current_scalar ← BRAM[ch_cnt][read_addr]
             - Global index = data_count
             - For each channel, increment ch_cnt and read_addr
             - After full scan (C_IN × DEPTH), clear input_complete

  Neuron Array (OUT_SIZE neurons):
    → For each neuron[n] ∈ [0, OUT_SIZE):
         - Input: scalar = current_scalar, index = data_count
         - Weights: weights[n]
         - neuron_out[n] ← dot(input_flattened, weights[n])

         → Post-processing:
             - Add bias: biased_r ← neuron_out[n] + bias[n]
             - ReLU: if relu_act, set to 0 if negative
             - Assign to out_data[n]

  Output Control:
    → valid_out = all neuron_valid[n] asserted (after ReLU latency)
    → done = valid_out (delayed one cycle)
```

A global index counter (data_count) tracks the position of each scalar within the flattened input, allowing neurons to correctly apply their weights. The module operates in two distinct phases: Write Phase and Read Phase. The module transitions between these phases to efficiently process multi-channel image data through a dense neural network layer.

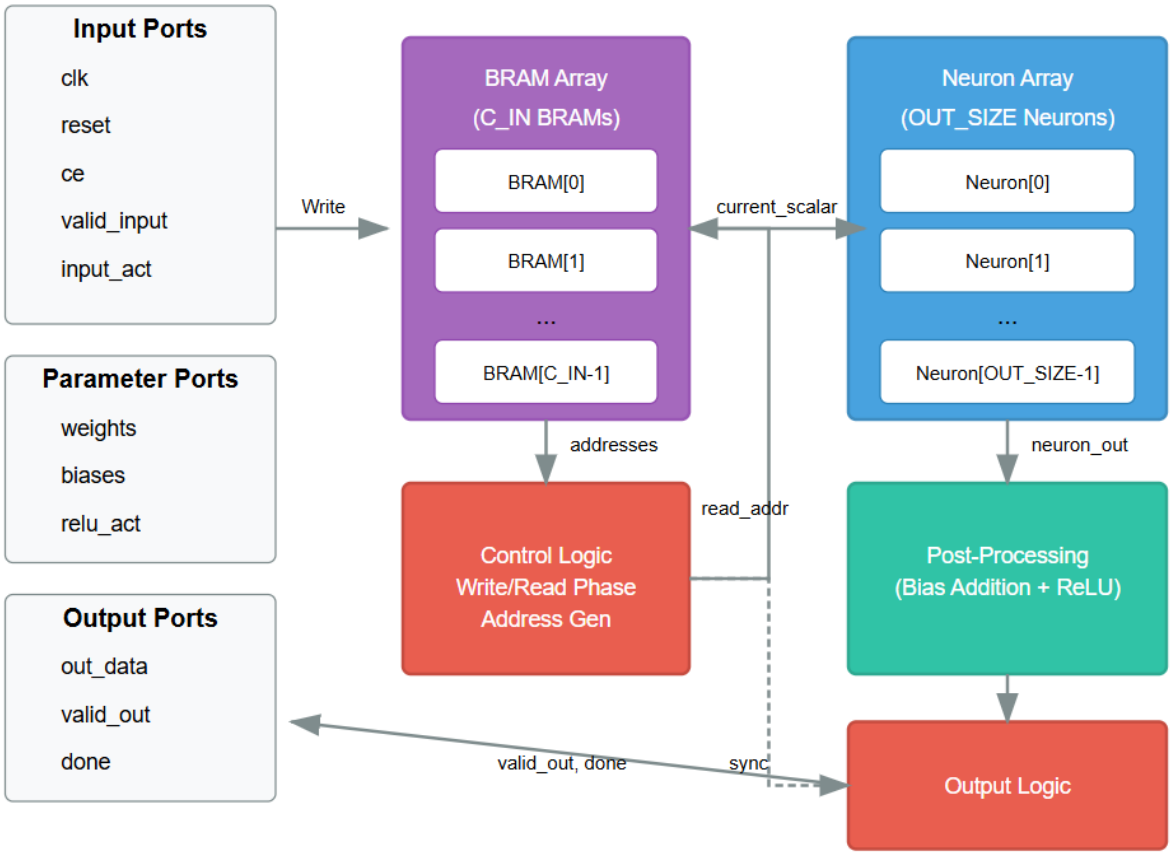


Figure 3.17 Block Diagram of the dense_flatten Module

Together, the ***neuron***, ***dense_layer***, and ***dense_flatten*** modules offer a robust framework for implementing dense computations on FPGAs. The ***dense_layer*** module is optimized for already-flattened input vectors, making it ideal for later stages of CNN inference. In contrast, the ***dense_flatten*** module is a powerful variant designed to flatten and process convolutional outputs in a streaming manner, bridging the gap between feature extraction and classification within FPGA-accelerated CNN architectures.

By adopting quantized arithmetic and modular, parameter-driven design, these implementations strike a balance between performance, resource efficiency, and adaptability making them suitable for real-time, low-power AI inference applications.

3.6 FPGA Architecture Design for Ensemble Models

The primary goal of the FPGA architecture design is to establish a flexible, scalable, and generalizable framework capable of implementing various neural network models efficiently. This generalizability significantly simplifies the deployment process, enabling the seamless construction of different model representations by assembling modular building blocks developed in the earlier phases of this research. The modular approach reduces complexity and accelerates the process of implementing different architectures.

3.6.1 Single model representation

Creating a neural network model using this FPGA framework is straightforward. Each neural network component, such as convolution, pooling, padding, and dense layers, is represented as a well-defined Verilog module. These modules can be conveniently instantiated and interconnected, significantly easing the creation of any neural network architecture. For example, the SimpleCNNMNIST model, structured specifically for the MNIST digit recognition task, can be readily instantiated by chaining together the pre-developed padding (pad_stream_md), convolution (convmd), pooling (max_pool_md), and dense layer (dense_flatten and dense_layer) modules. This modular design ensures that assembling a complete CNN architecture is similar to connecting building blocks, showing reusability and reducing potential for errors or inconsistencies.

To illustrate the ease and practicality of this process, consider the first stage of the SimpleCNNMNIST model. Initially, the input pixels are padded using the pad_stream_md module, which aligns data for proper convolutional processing. Following padding, the convmd convolution module processes the padded activations, applying kernel weights and biases while leveraging built-in multiply-accumulate logic optimized for fixed-point arithmetic. Subsequently, the resulting activations flow into the max_pool_md pooling module, which performs spatial downsampling. This systematic propagation continues similarly through subsequent stages, clearly defined by module interconnections and guided by precise handshake signals.

Each module communicates its operational status using well-structured handshake signals, such as valid_input, output_valid, and output_end. These signals ensure coherent data propagation throughout the pipeline, significantly simplifying timing control and debugging. For instance, as the padding module completes its operations, it outputs a valid signal (padded_ce) to indicate that its output data is ready and valid for the convolution module, which in turn produces a corresponding valid signal (conv_valid) once it finishes processing. Each subsequent module then follows suit, making the entire process of data flow explicit and easily traceable.

Table 3.7 SimpleCNNMNIST model timing and control Signals

Module	Input Valid Signal	Output Valid Signal	Completion Signal
Padding (Stage 1)	valid_input	padded_ce1	pad_end1

Convolution (Stage 1)	padded_ce1	conv_valid1	conv_done1
Max Pooling (Stage 1)	conv_valid1	pool_valid1	pool_end1
Padding (Stage 2)	pool_valid1	padded_ce2	pad_end2
Convolution (Stage 2)	padded_ce2	conv_valid2	conv_done2
Max Pooling (Stage 2)	conv_valid2	pool_valid2	pool_end2
Padding (Stage 3)	pool_valid2	padded_ce3	pad_end3
Convolution (Stage 3)	padded_ce3	conv_valid3	conv_done3
Max Pooling (Stage 3)	conv_valid3	pool_valid3	pool_end3
Dense Flatten	pool_valid3	dense1_valid	dense1_end
Dense Layer	dense1_valid	dense2_valid	dense2_end
Total Network	valid_input	output_valid	output_end

This handshake mechanism is clearly demonstrated in the SimpleCNNMNIST timing and control signals table. Each neural network layer module utilizes specific input and output valid signals, creating a highly predictable and reliable data propagation process. The padding modules begin operations upon receiving the valid_input signal and provide their outputs with padded_ce signals. Similarly, the convolution modules commence operations upon receiving these padded data signals and subsequently emit valid convolution outputs indicated by conv_valid. Pooling modules further propagate signals downstream, using similar validation schemes. The final dense layers terminate this structured communication, outputting fully validated results.

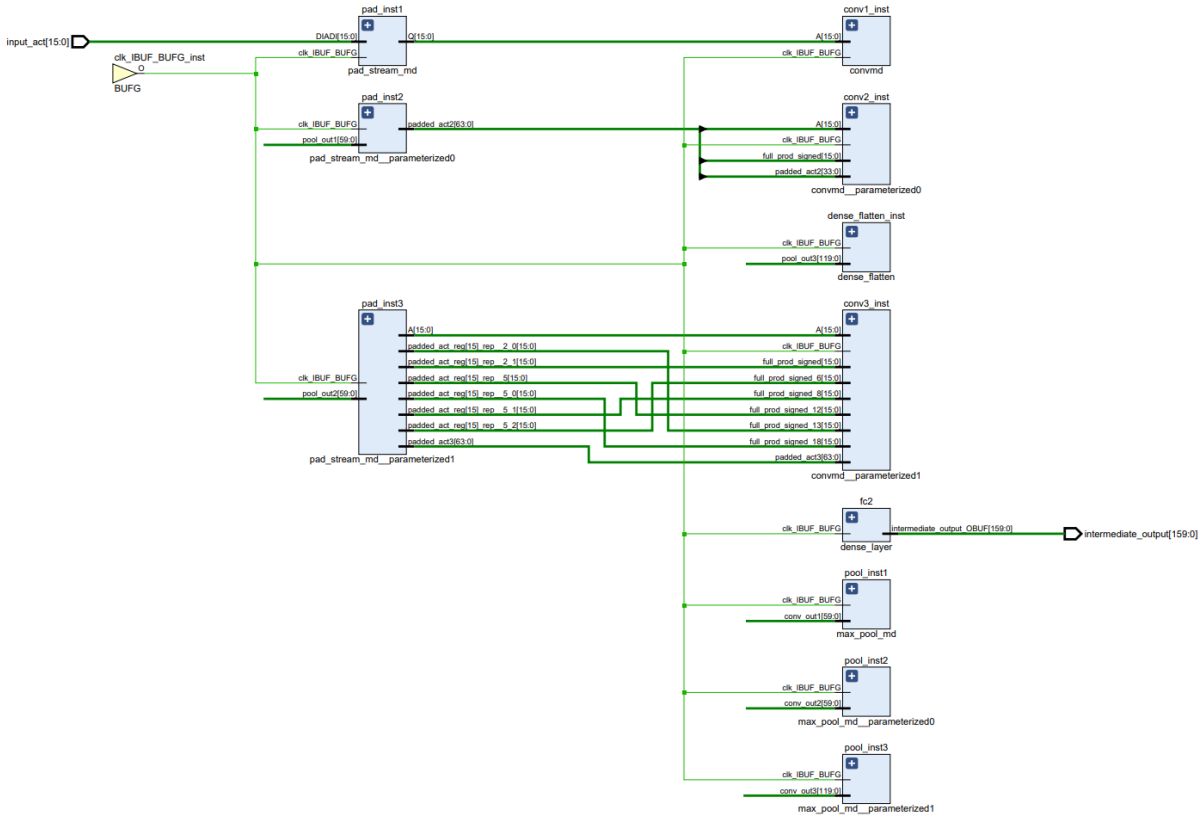


Figure 3.18 Dataflow diagram of the designed SimpleCNNMNIST model

The single model representation is designed to be highly generalizable. While the SimpleCNNMNIST model serves as a specific example, the modular components can be reconfigured to support different architectures or datasets. The SimpleCNN architecture is also designed using this approach.

Verilog modules named ***mnist_cnn1***, ***mnist_cnn2***, ***mnist_cnn3*** are created for the SimpleCNNMNIST models and ***cnn_model1***, ***cnn_model2***, ***cnn_model3*** are created for SimpleCNN. Each module will specify the architecture used and with weight and bias values represented in the quantized bit. The weight and biases will be assigned to the specific module by passing them to the modules as an input but internally stored for usage.

3.6.2 Ensemble of models

The architecture is built around modular Verilog components created and control mechanisms, enabling efficient inference for tasks like MNIST digit recognition. The FPGA architecture supports two primary ensembling techniques: parallel and sequential, each with distinct implementation strategies and trade-offs.

3.6.2.1 *Parallel Ensembling*

Parallel ensembling involves executing multiple neural network models simultaneously on the same input data, leveraging the FPGA's parallel processing capabilities to minimize inference latency. In this approach the individual models will be created as a separate hardware component. Each models process the inputs concurrently, producing individual predictions.

For MNIST models ***mnist_ensemble*** module is created and the inputs will be passed to the ***mnist_cnn1***, ***mnist_cnn2***, ***mnist_cnn3*** parallelly. Similarly, for the SimpleCNN models an ***ensemble_model*** is created and similarly the inputs will be processed parallelly by the three SimpleCNN model modules ***cnn_model1***, ***cnn_model2***, ***cnn_model.***

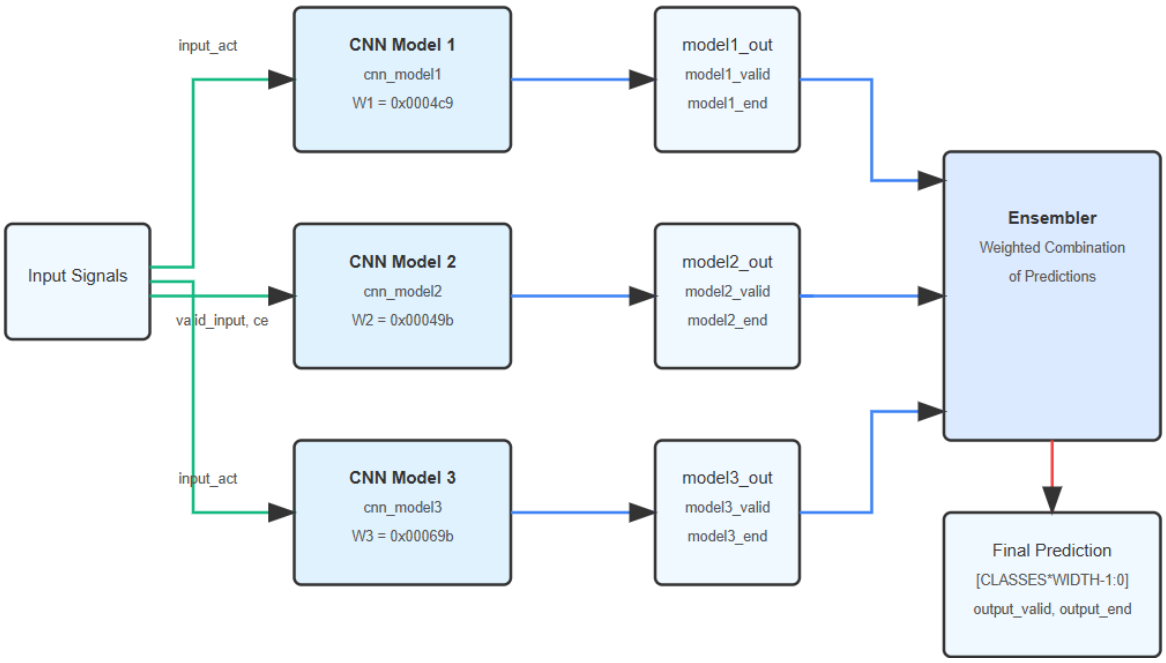


Figure 3.19 SimpleCNN parallel ensemble model architecture

This approach maximizes throughput by processing all models in parallel, with the inference time determined by the slowest model.

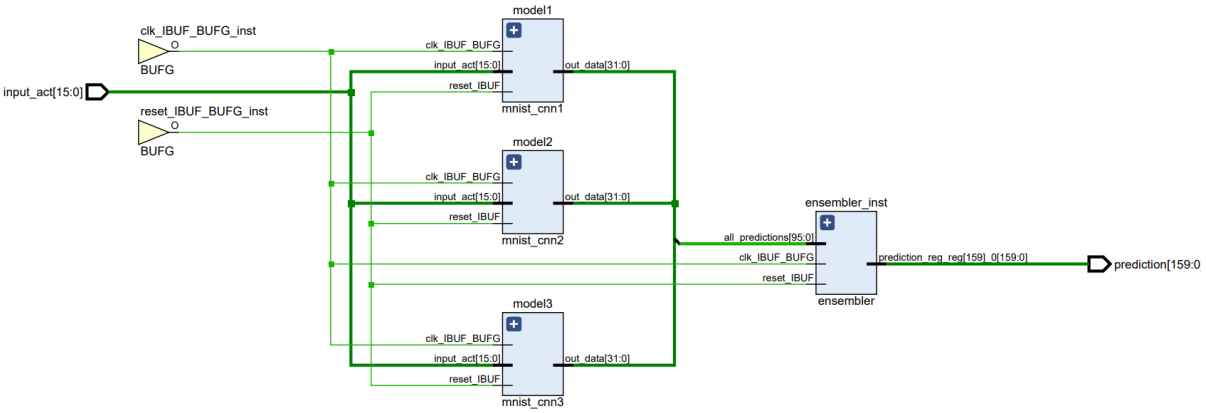


Figure 3.20 Dataflow diagram of parallel ensemble of SimpleCNNMNIST model

3.6.2.2 *Sequential Ensembling*

Sequential ensembling reuses a single hardware instance to process multiple models one after another, significantly reducing resource consumption. For this approach, a general module based on the architecture SimpleCNN and SimpleCNNMNIST is created. The module won't have specified weight and bias values assigned to it, but the top-level determines the weight and bias values to be assigned to the specific modules, enabling them to process different models with the same architecture at different times. In this approach, the input won't be directly passed to the modules in parallel; rather, a buffer will be used to store the incoming pixel values or input for later access.

To achieve this, the system employs a structured method that optimizes resource usage while ensuring each model processes the same input data sequentially. The input data, such as pixel values from an image, is first captured and stored in a buffer as it arrives. This stored input is then replayed to the general module multiple times once for each model eliminating the need for simultaneous input feeds. A control mechanism, typically a state machine, arranges the sequence of operations. It begins by storing the input in the buffer, then proceeds to process each model one at a time. For each model, the top-level dynamically assigns a unique set of weights and biases to the general module, effectively transforming it to represent a different model during each processing phase.

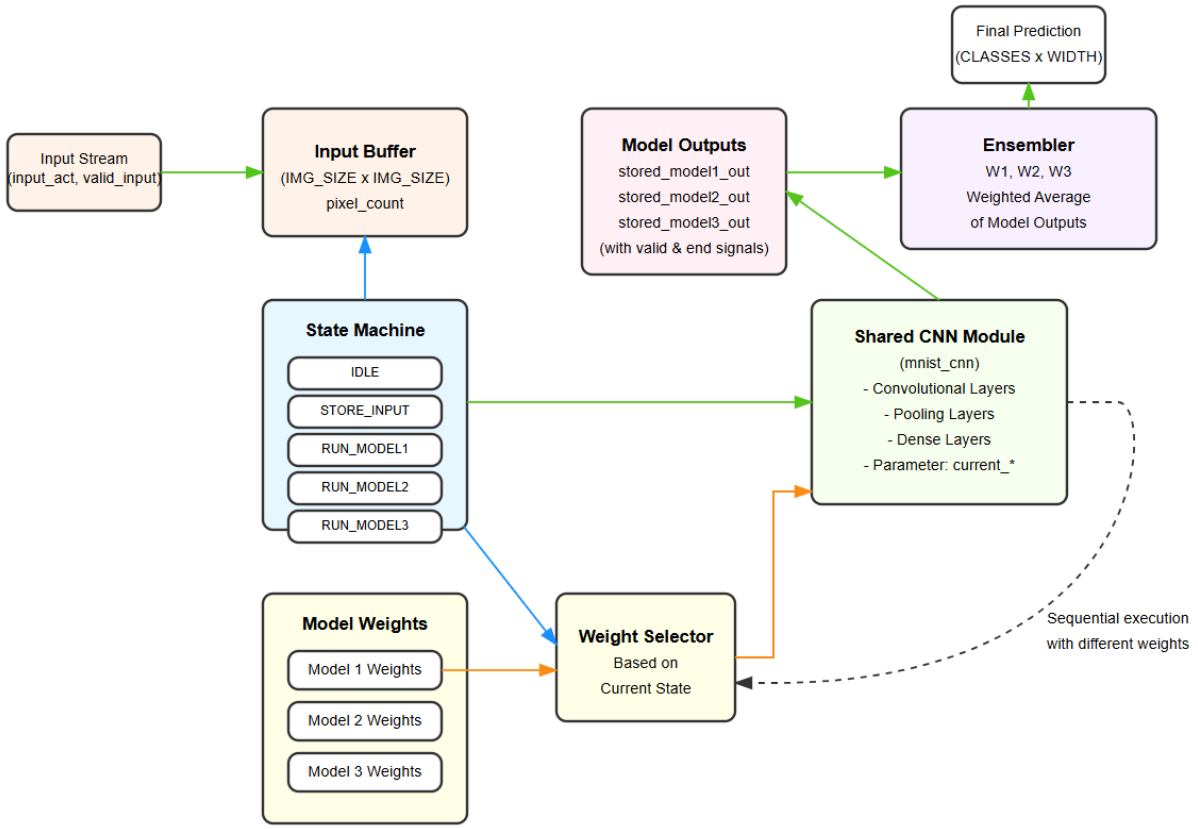


Figure 3.21 SimpleCNN sequential ensemble model architecture

Once a model finishes processing the input, its output is saved, and the system moves on to the next model, reusing the same hardware instance with a new set of weights and biases. After all models have been processed, the individual outputs are combined using a weighted approach to produce a final prediction. This sequential processing, while potentially increasing latency compared to parallel methods, drastically reduces the need for multiple hardware instances, making it highly efficient in terms of resource utilization. The use of a buffer ensures that the input data remains consistent across all models, and the dynamic assignment of parameters allows the same architecture to adapt to different models without requiring additional hardware resources.

This approach is particularly advantageous in environments where hardware resources are limited, offering a practical trade-off between processing time and resource efficiency. By carefully managing the flow of data and the assignment of model parameters, it achieves the goal of running multiple models on a single hardware instance effectively.

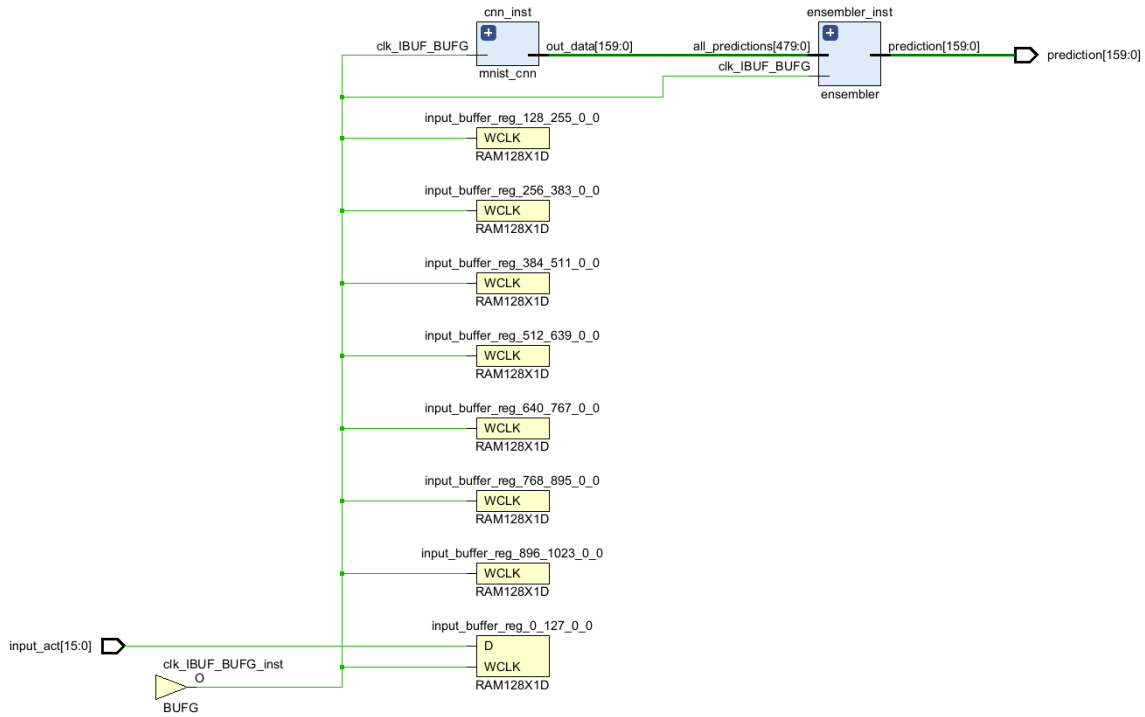


Figure 3.22 Dataflow diagram of sequential ensemble of SimpleCNNMNIST model

3.6.3 Weight and Bias Extraction

The trained convolutional neural network (CNN) models are initially developed and optimized using standard software-based deep learning frameworks. Once the training process is complete and the desired performance metrics are achieved, the model parameters, including weights and biases, are extracted from the trained neural network in their native floating-point representation. These extracted parameters are then carefully quantized into a fixed-point representation. This conversion from floating-point to fixed-point involves determining an appropriate bit length and fractional part. Following quantization, the fixed-point parameters are transformed into a format compatible with the FPGA implementation, typically represented as hexadecimal values for easy integration. These formatted values are embedded directly into the FPGA design as initialized parameters within the Verilog modules. Consequently, when the FPGA performs inference, these embedded weights and biases are readily accessible within its internal logic blocks and memory, allowing the hardware to execute inference operations directly without runtime parameter loading. This systematic approach ensures a seamless transfer from a trained software model to an FPGA-based inference system optimized for low latency and high energy efficiency.

CHAPTER 4 RESULTS AND DISCUSSIONS

4.1 Experimental Setup

The experimental setup for this thesis focuses on evaluating the performance of an FPGA-based ensemble neural network framework. The setup encompasses the dataset, preprocessing techniques, and the integrated use of software and hardware tools to develop and deploy the framework. The following description incorporates detailed explanations of PyTorch, Xilinx Vivado Design Suite, Testbench, and Synthesis, highlighting their roles in the experimental pipeline.

PyTorch is widely-used open-source deep learning framework, was employed for the design, training, and initial validation of the neural network models. Specifically, the SimpleCNN and SimpleCNNMNIST architectures, designed for the digit recognition task were implemented using PyTorch. Training was conducted on standard computing hardware, using PyTorch's automatic differentiation and optimization capabilities to fine-tune model parameters. Additionally, PyTorch simplified model optimization through techniques such as L1 unstructured pruning.

Xilinx Vivado Design Suite: Vivado provides a comprehensive environment for synthesizing, implementing, and programming Xilinx FPGAs. Vivado's synthesis tools converted these Verilog descriptions into gate-level netlists, optimizing the design for the target FPGA's architecture, which includes Configurable Logic Blocks (CLBs), Block RAMs (BRAMs), and Digital Signal Processing (DSP) slices. The implementation phase involved mapping, placing, and routing the netlist onto the FPGA fabric, ensuring efficient resource utilization and meeting timing constraints. Vivado's ability to optimize for power, performance, and area was crucial for achieving the low-latency and energy-efficient inference required for real-time applications.

Testbench Simulation: To verify the correctness of the FPGA implementations, testbenches were developed using Verilog. In this study, testbenches were designed to replicate the inference process by feeding preprocessed input data from the Geez and MNIST dataset into the FPGA design and comparing the hardware outputs with those generated by the software models in PyTorch.

Synthesis: In this study, synthesis was performed using Vivado's synthesis tools, which translated the Verilog code into a netlist comprising logic gates, flip-flops, and other FPGA resources. This process optimized the design for performance metrics such as clock frequency,

power consumption, and resource utilization, ensuring that the neural network operations could be executed efficiently on the FPGA.

4.2 Results

4.2.1 Model Training Results

4.2.1.1 *SimpleCNN Model Training Results*

After training Model 1 for 25 epochs the training accuracy is 86.81% and loss is 0.462.

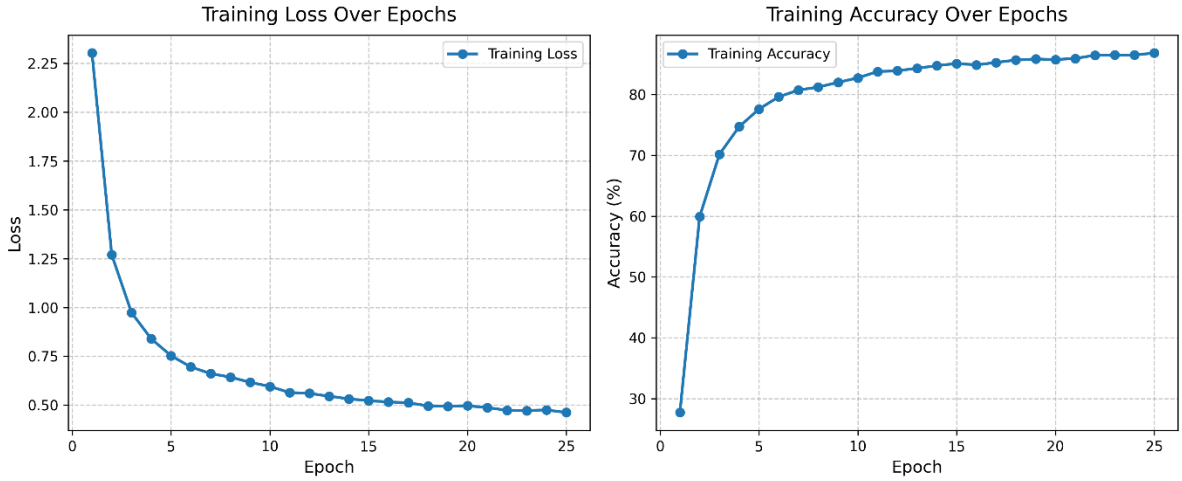


Figure 4.1 Training loss and accuracy of SimpleCNN model 1 over epochs

For Model 2, with the same number of epochs, 25, the training accuracy is 86.07% and loss is 0.498.

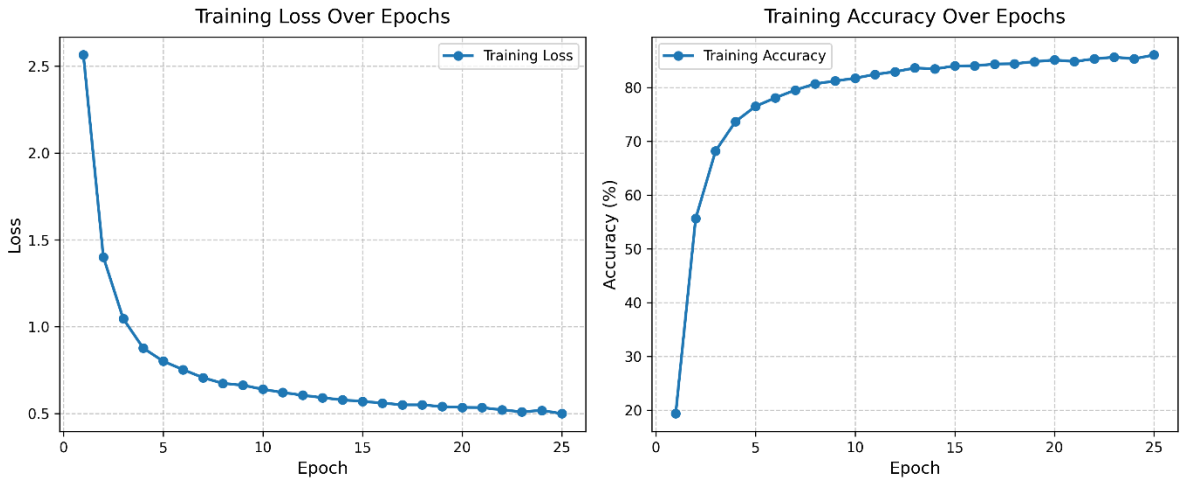


Figure 4.2 Training loss and accuracy of SimpleCNN model 2 over epochs

After training Model 3, for 20 epochs the training accuracy is 86.74% and the loss is 0.453.

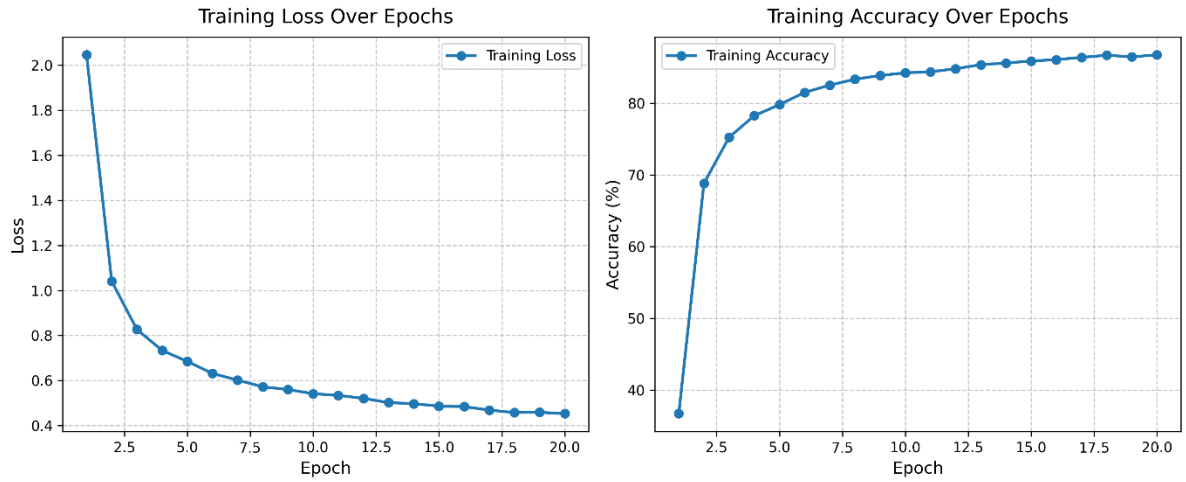


Figure 4.3 Training loss and accuracy of SimpleCNN model 3 over epochs

On the test set, the accuracy of Model 1 is 90.30%, Model 2 is 89.74%, and Model 3 is 89.11%. After applying L1 pruning to the three models, the initial accuracy changed.

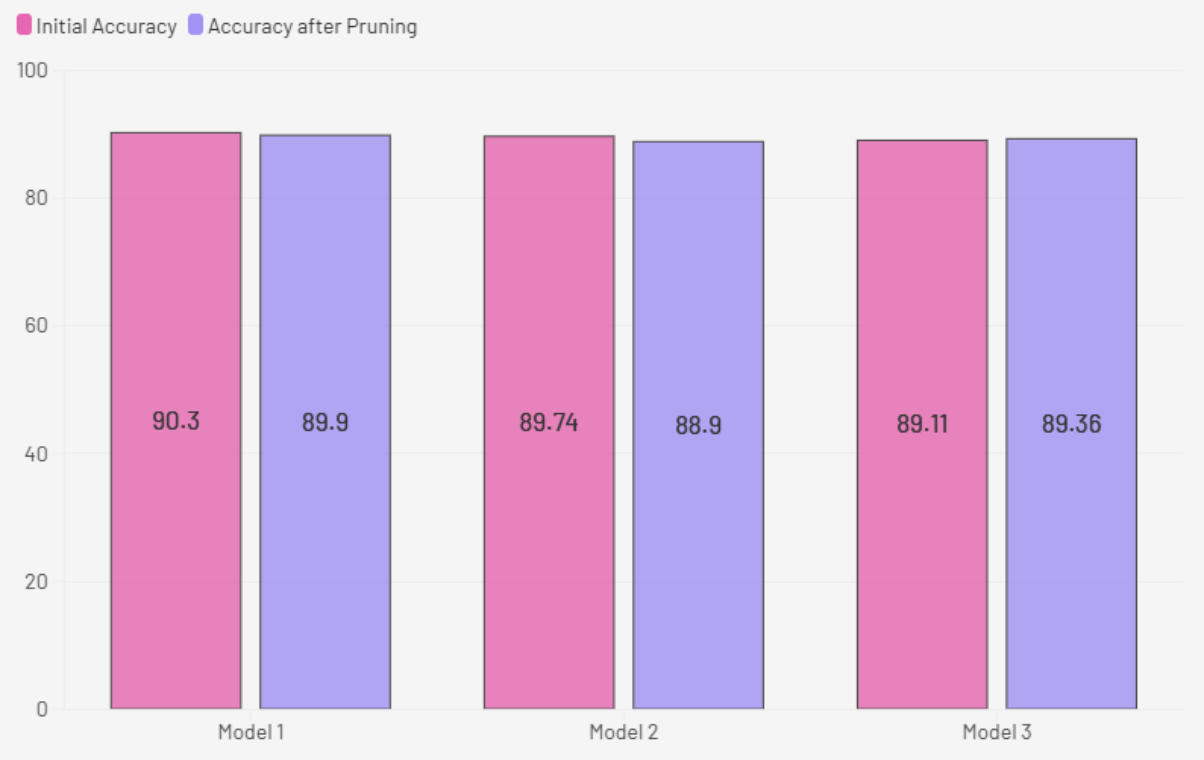


Figure 4.4 Three SimpleCNN models accuracy after applying pruning

The voting prediction and weighted ensemble were applied to the pruned models. The voting applied used equal voting power of 1:1:1. For the weighted average, equal weighting of 1/3 = 0.3334 was first given to all three models, and another set of weights was found after checking the best rates: 0.2792 for Model 1, 0.3283 for Model 2, and 0.3925 for Model 3.

Table 4.1 Accuracy results for different ensembling techniques on SimpleCNN models

Weighting Method	Model Weights (M1, M2, M3)	Accuracy (%)	Wrong Classifications
Equal Weight (Avg)	0.33, 0.33, 0.33	93.48	339/5197
Custom Weighted Average	0.28, 0.33, 0.39	93.59	333/5197
Equal Weight Voting	Voting-based	92.20	405/5197

The next step is to apply quantization to the models. Based on the approach mentioned in Chapter 3, the model weights and parameters are converted to 16-bit and 24-bit. For the 24-bit representation, 12 bits are allocated for the fractional part, and for the 16-bit representation, 8 bits are allocated for the fractional part. The best scoring model, which is the custom weighted average model, was tested again after quantization.

Table 4.2 SimpleCNN Ensemble model performance after applying quantization

Quantization	Weighting Method	Model Weights (M1, M2, M3)	Accuracy (%)	Wrong Classifications
24-bit	Weighted Average	0.33, 0.33, 0.33	93.44	341/5197
	Custom Weights	0.28, 0.33, 0.39	93.53	336/5197
	Equal Voting (1:1:1)	Voting-based	92.15	408/5197
16-bit	Weighted Average	0.33, 0.33, 0.33	88.11	618/5197
	Custom Weights	0.28, 0.33, 0.39	88.19	614/5197
	Equal Voting (1:1:1)	Voting-based	86.45	686/5197

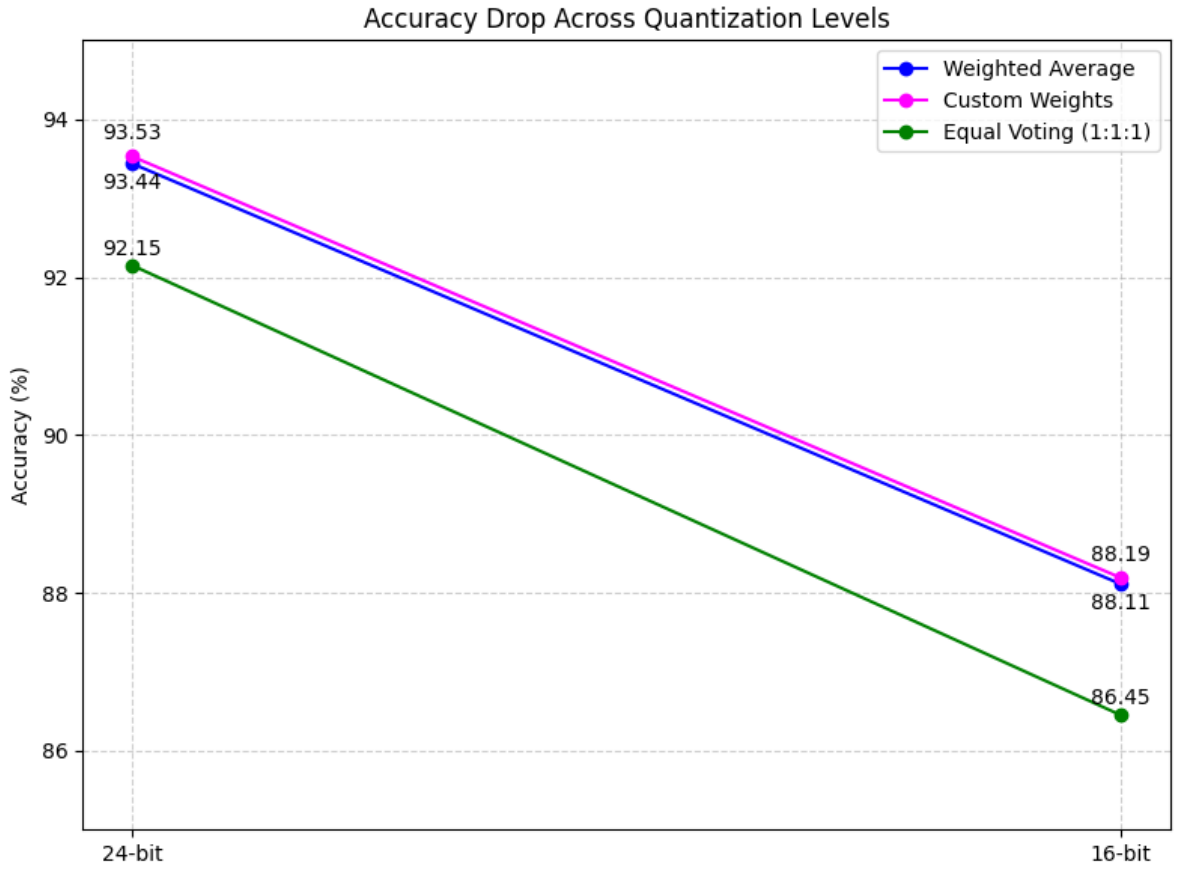


Figure 4.5 24-bit and 16-bit quantization effect on the SimpleCNN models performance

The after applying the quantization the best scoring model is custom weight 24-bit ensemble model. The figure below shows some of the wrongly classified images from the 336 images.

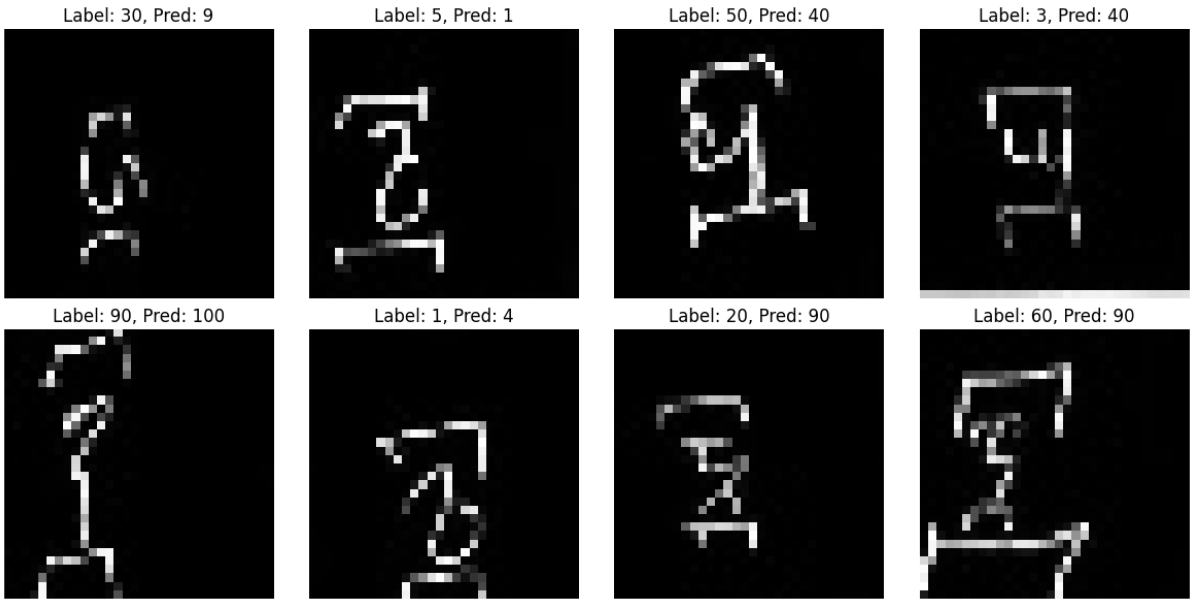


Figure 4.6 Wrong classification by 24-bit Custom Weights SimpleCNN Ensemble Models

4.2.1.1 SimpleCNNMNIST Model Training Results

After training for 25 epochs, the model achieved a training accuracy of 98.96% and a training loss of 0.0308. The validation accuracy was 98.38%, with a validation loss of 0.0576.

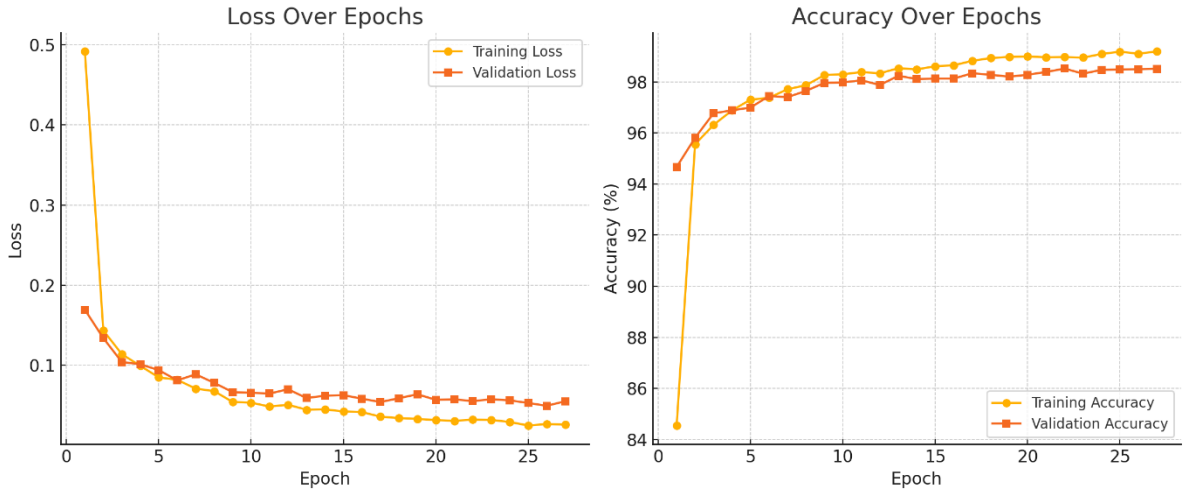


Figure 4.7 Training and validation loss and accuracy of SimpleCNNMNIST model 1 over epochs

After 17 epochs, the second model achieved a training accuracy of 98.88% and a training loss of 0.0345. The validation accuracy was 98.47%, with a validation loss of 0.0506.

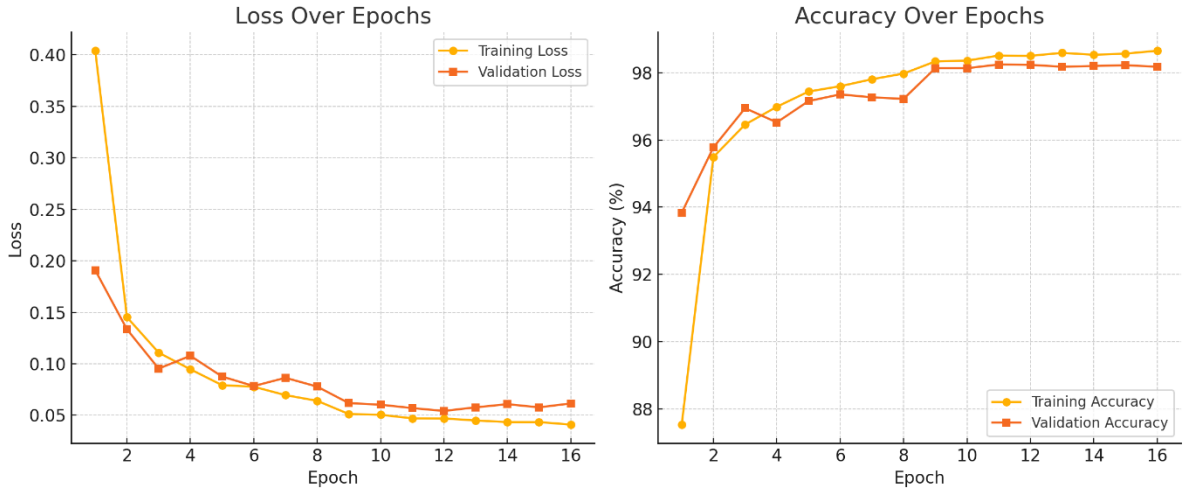


Figure 4.8 Training and validation loss and accuracy of SimpleCNNMNIST model 2 over epochs

For the third model, after training it for 43 epochs, the model reached a training accuracy of 99.23% with a corresponding loss of 0.0241. On the validation set, it achieved an accuracy of 98.70% and a loss of 0.0508.

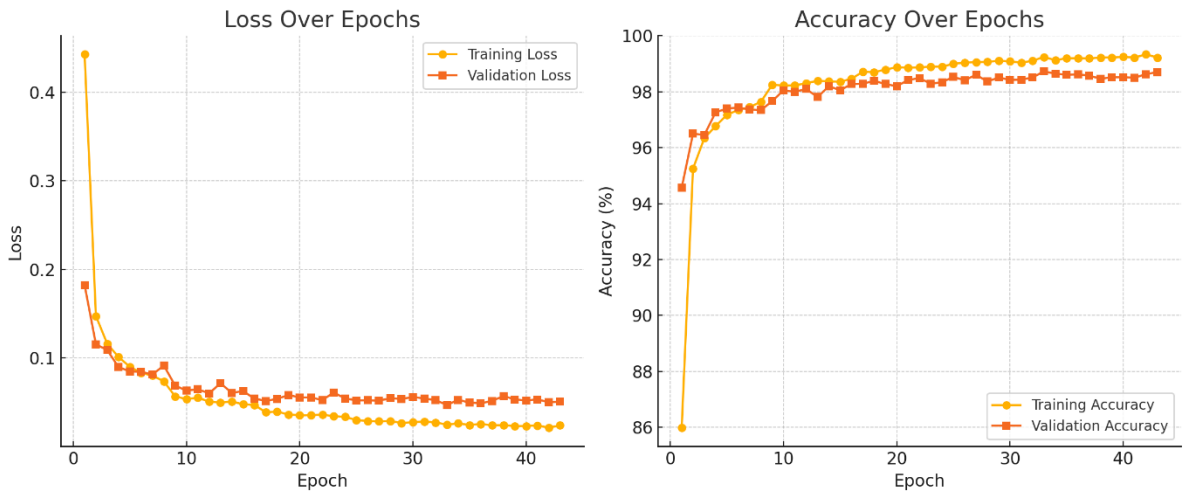


Figure 4.9 Training and validation loss and accuracy of SimpleCNNMNIST model 3 over epochs

Here is the performance of individual SimpleCNNMNIST model on the test set.

Table 4.3 SimpleCNNMNIST models accuracy and wrong classification of test set

Model	Accuracy(%)	Wrong Classification
Model 1	98.82%	118
Model 2	98.84%	116
Model 3	98.95%	105

After applying the pruning, the models are combined with three types and ensemble techniques. And the result is as follows:

Table 4.4 Ensemble SimpleCNNMNIST models accuracy and wrong classification of test set after applying pruning

Weighting Method	**Model Weights (M1, M2, M3)**	**Accuracy (%)**	**Wrong Classifications**
Weighted Average	0.33, 0.33, 0.33	99.17	83
Custom Weighted Average	0.27, 0.22, 0.51	99.21	79
Equal Weight Voting	Voting-based	98.61	139

After applying 16-bit quantization the ensemble model accuracy and number of wrong classifications is

Table 4.5 Ensemble SimpleCNNMNIST models accuracy and wrong classification of test set after applying 16-bit quantization

Weighting Method	**Model Weights (M1, M2, M3)**	**Accuracy (%)**	**Wrong Classifications**
Weighted Average	0.33, 0.33, 0.33	99.06	94
Custom Weighted Average	0.46, 0.30, 0.23	99.11	89
Equal Weight Voting	Voting-based	99.07	93

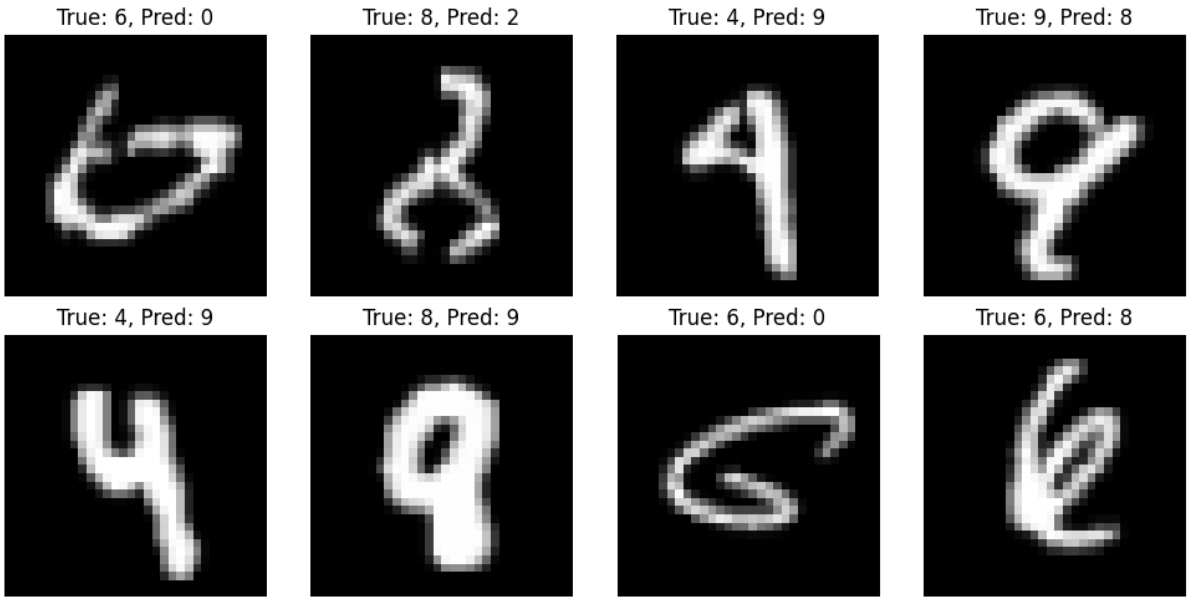


Figure 4.10 Wrong classification by 16-bit Custom Weights Ensemble SimpleCNNMNIST Models

4.2.2 Simulation Results

In the testbench, all of the modules were tested and passed their correctness is cross-checked with the expected results with assertions. For the NN models in the hardware the input images designed in the quantization bit format and passed to the models through the testbench.

4.2.3 Hardware Performance Metrics

4.2.3.1 Power Consumption

Power consumption was measured using Vivado's power analysis tools, reporting both dynamic power (attributed to switching activity) and static power (due to leakage). Each module was synthesized and implemented individually to isolate its power footprint, with simulations based on a clock frequency of 50Hz.

Table 4.6 Power Consumption of Individual Modules

Module	Parameters used	Quantization Bits	Dynamic Power (W)	Device Static Power (W)	Total Power (W)
quant_add		24	0.016	0.168	0.184
		16	0.011	0.168	0.179
quant_mul		24	0.023	0.168	0.191
		16	0.012	0.168	0.18
mul_acc		24	0.023	0.168	0.191
		16	0.016	0.168	0.184
shift_reg	SIZE = 3	24	0.002	0.168	0.17
		16	0.002	0.168	0.17

simple_bram	DEPTH = 4	24	0.016	0.168	0.184
		16	0.011	0.168	0.179
conv2d	IMG_SIZE = 32 KERNEL_SIZE = 3 STRIDE = 1	24	0.033	0.168	0.201
		16	0.022	0.168	0.19
max_pool	IMG_SIZE = 32 POOL_DIM = 2 STRIDE = 2	24	0.004	0.168	0.172
		16	0.003	0.168	0.171
max_pool_md	IMG_SIZE = 32 POOL_DIM = 2 STRIDE = 2 C_in = 3	24	0.007	0.168	0.175
		16	0.005	0.168	0.173
neuron	IN_SIZE = 4	24	0.010	0.168	0.178
		16	0.008	0.168	0.176
dense_layer	IN_SIZE = 4 OUT_SIZE = 2	24	0.029	0.168	0.197
		16	0.019	0.168	0.187

While the above table shows the individual modules power usage. The synthesis is also performed for both single and parallel ensemble SimpleCNNMNIST model

Table 4.7 On-Chip power report of single and ensemble SimpleCNNMNIST model

Component	**Single Model**		**Parallel Ensemble Model**		**Sequential Ensemble Model**	
	Power (W)	**Percentage (%)**	**Power (W)**	**Percentage (%)**	**Power (W)**	**Percentage (%)**
Dynamic	**0.414**	**71%**	**1.267**	**88%**	**0.474**	**73%**
Clocks	0.030	7%	0.083	7%	0.039	8%
Signals	0.084	20%	0.255	20%	0.161	34%
Logic	0.086	21%	0.264	21%	0.201	42%
BRAM	0.003	1%	0.008	1%	0.002	1%
DSP	0.212	51%	0.657	52%	0.069	15%
I/O	<0.001	0%	<0.001	0%	<0.002	0%
Static	**0.169**	**29%**	**0.176**	**12%**	**0.171**	**27%**

The total power consumption by a single model is 0.414 Watt while the Dynamic power taking 71% of it. The DSP component takes the higher (51%) of the dynamic power. Most of the consumption by the components is around 3x the single model usage. The power consumption of the single model is 0.414 Watts, with dynamic power being the dominant factor, accounting for 71% of the total. Within the dynamic power, the DSP component exhibits the highest consumption, representing 51%. When considering the utilization of ensemble model such components, the overall power consumption escalates to approximately three times the power usage of the single model.

4.2.3.2 Resource Utilization

Resource utilization directly impacts factors such as cost, power consumption, and the scalability of the design for more complex models or larger ensembles. This section details the utilization of key resources such as Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and Digital Signal Processing (DSP) slices.

Table 4.8 Resource Utilization of Individual modules

Module	Parameters used	Quantization Bits	LUT	FF	BRAM	DSP
quant_add		24	27	24	0	0
		16	25	16	0	0
quant_mul		24	16	24	0	2
		16	12	16	0	1
mul_acc		24	36	24	0	2
		16	24	16	0	1
shift_reg	SIZE = 3	24	13	48	0	0
		16	9	32	0	0
simple_bram	DEPTH = 4	24	0	0	0.5	0
		16	0	0	0.5	0
conv2d	IMG_SIZE = 32 KERNEL_SIZE = 3 STRIDE = 1	24	322	317	1	18
		16	249	259	0	9
max_pool	IMG_SIZE = 32 POOL_DIM = 2 STRIDE = 2	24	96	164	0	0
		16	68	116	0	0
max_pool_md	IMG_SIZE = 32 POOL_DIM = 2 STRIDE = 2 C_in = 3	24	275	466	0	0
		16	191	322	0	0
neuron	IN_SIZE = 4	24	102	61	0	2
		16	70	40	0	1
dense_layer	IN_SIZE = 4 OUT_SIZE = 2	24	273	119	0	4
		16	185	77	0	2

The FPGA resource utilization results highlight how each module consume critical hardware resources, namely look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), and DSP slices. The arithmetic units for fixed-point operations are particularly lightweight: for example, a 24-bit fixed-point adder (quant_add) uses only 27 LUTs and 24 FFs without any DSP or BRAM, and its 16-bit version further reduces to 25 LUTs and 16 FFs. Multipliers are more demanding, but quantization significantly lowers their cost. A 24-bit multiplier (quant_mul) requires 2 DSP slices (each Xilinx DSP can handle a 18×25-bit multiplication) and 16 LUTs, whereas a 16-bit multiplier fits into a single DSP with 12 LUTs. This roughly 50% reduction

in DSP usage from 24-bit to 16-bit is consistent across the design; for instance, the multiply-accumulate unit (mul_acc) drops from 2 DSPs at 24-bit to 1 DSP at 16-bit. Similarly, the shift registers module uses fewer flip-flops when the data width is halved (48 FFs at 24-bit vs. 32 FFs at 16-bit). These module-level figures demonstrate the benefits of quantization in conserving hardware resources without altering the network's architecture. The only moderate BRAM usage appears in the simple_bram module for caching streaming data (about 0.5 of a 36Kb Block RAM for both 16-bit and 24-bit widths), indicating that most model parameters and activations are stored in distributed memory (LUTs as RAM) or registers rather than large on-chip RAM blocks.

Table 4.9 Resource Utilization of a single SimpleCNNMNIST model

Name	Slice LUTs 277400	Slice Registers 554800	F7 Muxes 138700	F8 Muxes 69350	Slice 69350	LUT as Logic 277400	LUT as Memory 108200	Block RAM Tile 755	DSPs 2020
mnist_cnn1	26032	19892	1263	120	8650	24864	1168	2.5	606
conv1_inst	1374	801	0	0	460	1246	128	0	36
conv2_inst	4035	3105	0	0	1270	3779	256	0	144
conv3_inst	7873	6177	0	0	2473	7361	512	0	288
dense_flatten_inst	9557	7675	1188	120	3412	9461	96	0	128
fc2	632	421	75	0	228	632	0	0	10
pad_inst1	217	57	0	0	69	217	0	0.5	0
pad_inst2	797	177	0	0	254	797	0	0	0
pad_inst3	854	357	0	0	299	806	48	0	0
pool_inst1	124	269	0	0	90	92	0	0	0
pool_inst2	215	265	0	0	85	183	32	0	0
pool_inst3	408	521	0	0	157	344	64	0	0

At the full model level, the SimpleCNNMNIST instance occupies a modest fraction of the FPGA's available logic and memory resources. The entire single-model design utilizes approximately 26,032 LUTs and 19,892 FFs, which is only about 9% of the 277,400 LUTs and 3.6% of the 554,800 flip-flops available on the target device. BRAM usage is minimal (roughly 2.5 BRAM tiles out of 755, <1%) since the network weights are relatively small and largely stored in fixed registers or LUTs. The most significant resource demand is in DSP slices due to convolutional and fully-connected layer computations. The single CNN model consumes 606 DSP slices, around 30% of the 2020 DSPs on the FPGA. This DSP utilization aligns with the network's arithmetic intensity: each convolution layer instantiation contributes a substantial number of multipliers. In fact, the resource breakdown shows an increasing DSP count in

deeper layers, Conv1 uses 36 DSPs, Conv2 uses 144 DSPs, and Conv3 uses 288 DSPs – reflecting the growing number of filter operations at each successive layer. The final dense layers also add to this total (the combined flattening and first fully-connected layer uses 128 DSPs), though the last classification layer with only 10 outputs is negligible by comparison (10 DSPs). This analysis confirms that DSP slices are the critical resource for the design, whereas logic utilization (LUT/FF) and memory remain within comfortable limits. The concentration of LUTs in certain modules (e.g. the dense/flatten layer uses ~9.5k LUTs on its own) suggests that the wiring and summation logic for connecting the convolution outputs to the fully-connected layer is a non-trivial portion of the design. Still, the overall resource footprint of a single model is modest relative to a modern FPGA's capacity, which is encouraging for scaling or integrating additional functionality.

Table 4.10 Resource Utilization of parallel ensemble SimpleCNNMNIST model

Name	Slice LUTs (277400)	Slice Registers (554800)	F7 Muxes (138700)	F8 Muxes (69350)	Slices (69350)	LUT as Logic (277400)	LUT as Memory (108200)	Block RAM Tile (755)	DSPs (2020)
mnist_ensemble	78797	58994	4013	360	25808	75293	3504	7.5	1821
ensembler_inst	167	200	0	0	150	167	0	0	3
model1	26062	19816	1263	120	8499	24894	1168	2.5	606
model2	26299	19833	1352	120	8600	25131	1168	2.5	606
model3	26266	19835	1398	120	8605	25098	1168	2.5	606

The resource demands grow when moving from a single model to an ensemble of models, though the exact impact depends on the ensemble integration strategy. In a sequential ensemble configuration, where the same hardware is time-multiplexed across multiple models, the resource utilization remains essentially the same as for one model – only a small overhead is added for control logic and intermediate result storage. The parallel ensemble that instantiates multiple models' side by side incurs a roughly linear increase in resource usage. In the custom 3-model parallel ensemble examined, the total LUT and DSP usage is roughly three times that of a single model (each CNN replicated fully). This puts the parallel ensemble at roughly 78k LUTs and about 1,818 DSPs in the FPGA – nearly 90% of the available DSP budget. Such a design pushes the device close to its DSP capacity, underscoring that the DSP count (and by extension, the arithmetic intensity of the CNNs) is the primary limiting factor for scaling up the ensemble size. The current design illustrates that a moderate-size FPGA can host multiple CNN models concurrently, provided each model is streamlined in its resource usage. The

efficient use of resources in each module aided by quantization to 16-bit and by the simple architectural structures – makes the ensemble implementation feasible without exhausting LUTs or memory. Overall, the resource utilization analysis shows a design that is well-balanced: it uses a significant portion of the FPGA's parallel computation capability (DSPs) to achieve high throughput, while keeping within reasonable bounds of logic and memory usage, thus leaving some headroom for future scalability or additional processing modules.

4.2.3.3 Design Timing

Table 4.11 Design Timing Summary of SimpleCNNMNIST models

Metric	Single Model	Sequential Ensemble Model	Parallel Ensemble Model
Worst Negative Slack (WNS)	2.538 ns	2.970 ns	2.416 ns
Total Negative Slack (TNS)	0.000 ns	0.000 ns	0.000 ns
Worst Hold Slack (WHS)	0.031 ns	0.025 ns	0.035 ns
Total Hold Slack (THS)	0.000 ns	0.000 ns	0.000 ns
Worst Pulse Width Slack (WPWS)	9.232 ns	9.232 ns	—
Total Pulse Width Negative Slack	0.000 ns	0.000 ns	0.000 ns
Number of Failing Endpoints	0	0	0

The timing analysis of the SimpleCNNMNIST FPGA implementation indicates that the design meets timing requirements with comfortable margins, even as the architecture scales from a single model to an ensemble. The post-place-and-route timing report shows no timing violations for any of the configurations (single, sequential ensemble, or parallel ensemble), with all paths meeting the target clock period. The worst-case setup slack (WNS) is positive in all cases, meaning the design could actually run at a higher frequency than the baseline constraint. Specifically, the single-model design has a WNS of +2.538 ns, the sequential ensemble +2.970 ns, and the parallel ensemble +2.416 ns. The total negative slack (TNS) is 0 in all cases, confirming that there are no failing paths and every timing endpoint meets or exceeds the required setup time. The hold-time analysis is similarly positive: the worst hold slack (WHS) is on the order of only a few hundredths of a nanosecond (e.g. 0.025–0.035 ns), and no hold violations occur (THS = 0). Such tiny hold slack values suggest the design is very

close to optimal balancing of pipeline stages without over-constraining, and any minor hold adjustments (like small delay insertions) have been successfully handled by the tools.

4.3 Analysis of the Results

The performance of the FPGA-based ensemble neural network system was evaluated using several key metrics: accuracy, power consumption, resource utilization, latency, and throughput. These metrics are critical for assessing the system's ability to meet the thesis objectives of optimizing energy efficiency, computational performance, and accuracy in real-time applications. This section analyzes the trends, trade-offs, and implications derived from the results presented in section 4.2, providing insights into the system's overall effectiveness and its suitability for resource-constrained environments.

4.3.1 Model Accuracy and Performance

The accuracy of the models, both individually and in ensemble configurations, serves as a primary indicator of the system's predictive performance. The results demonstrate that the ensemble approach significantly enhances accuracy while maintaining robustness, particularly when combined with optimization techniques such as pruning and quantization.

4.3.1.1 Ensemble Improvement

For the SimpleCNN models (Geez digits), the ensemble approach improved accuracy from 90.30% (single model) to 93.53% (custom weighted ensemble). Similarly, for SimpleCNNMNIST (MNIST digits), accuracy increased from 98.95% (single model) to 99.21% (custom weighted ensemble). This improvement stems from the diversity introduced through training variations, such as different random seeds, dropout rates, and training durations. By leveraging uncorrelated errors among the models, the ensemble effectively reduces misclassifications, enhancing overall robustness.

4.3.1.2 Pruning Impact

Moderate L1 unstructured pruning, which removes less significant weights, had a minimal impact on accuracy. For instance, the SimpleCNN models-maintained accuracies above 89% even after pruning.

4.3.1.3 Quantization Effects

Quantization to 24-bit and 16-bit fixed-point representations introduced a trade-off between precision and resource efficiency. For SimpleCNN, the 24-bit quantized ensemble achieved

93.53% accuracy, while the 16-bit version dropped to 88.19%. In contrast, SimpleCNNMNIST showed a smaller accuracy reduction, from 99.21% (pre-quantization) to 99.11% (16-bit). This discrepancy indicates that the more complex Geez digit recognition task (with 20 classes) is more sensitive to precision loss than the simpler MNIST task (10 classes).

4.3.2 Hardware Efficiency

The hardware efficiency of the system was assessed through power consumption and resource utilization metrics, which are critical for deployment in resource-constrained environments. The analysis reveals that the system achieves a balance between performance and efficiency.

However, it requires significant FPGA resources, including LUTs, DSP slices, and BRAM, which may limit scalability for larger ensembles or more complex models. The architecture mitigates this by optimizing resource allocation and using pipelining to overlap layer computations within each model.

4.3.2.1 Power Consumption

The power consumption results illustrate the trade-offs between single-model efficiency and ensemble performance in the FPGA implementation. For a single SimpleCNN model, the on-chip power draw is measured at approximately 0.414 W at 50MHz. This power figure is impressively low in absolute terms, reflecting the efficiency of using a compact, quantized CNN on an FPGA. Breaking down the power usage, the dynamic power the portion consumed by active switching of transistors – accounts for about 71% of the total (roughly 0.29 W), while the remaining 29% (about 0.12 W) is static power due to leakage and baseline FPGA circuitry.

The dominance of dynamic power indicates that the FPGA resources are actively doing work and not sitting idle; it also suggests that any further power optimization would need to focus on reducing switching activity or voltage, since the static component is relatively fixed for a given device. Among the various FPGA components, the DSP slices contribute the largest share of dynamic power. Notably, the DSP circuitry alone constitutes about 51% of the total dynamic power in the single-model design. In other words, roughly half of the active energy is spent in the multiplier/accumulator units performing convolution and dense layer computations.

The Block RAMs, LUTs, and routing also consume dynamic power, but their contributions are smaller by comparison (each of those categories was reported to be a much smaller percentage of the dynamic power). The fact that DSPs dominate power consumption is actually a favourable outcome: DSP blocks on modern FPGAs are more power-efficient for arithmetic

operations than equivalent logic implementations, meaning the design is leveraging the most power-optimized resources for its heavy computations.

When moving from a single model to an ensemble of models, the power usage scales up in ways that reflect the underlying hardware sharing (or replication) strategy. For the *parallel ensemble* where three CNNs run simultaneously on separate hardware, the power consumption was observed to be approximately three times that of the single model – on the order of 1.2 W total. This near-linear scaling makes sense: with three times as many DSPs and logic elements switching on each clock cycle, dynamic power roughly triples. The static power component, however, does not triple – the FPGA's baseline leakage stays roughly the same (in fact, it can increase slightly if more circuitry is powered up, but it's relatively minor).

This linear scaling highlights an important consideration: while parallel ensembling yields higher throughput and potentially higher accuracy (through majority voting or averaging), it comes at a cost of proportionally higher power draw. In power- or thermally-constrained environments, this could be a limiting factor. However, even ~1.2 W total is quite modest for an FPGA design many high-end FPGAs can consume tens of watts, so the ensemble remains within a very low power envelope.

For the *sequential ensemble* approach, where only one model is active on the hardware at a time (the models run one after the other on the same circuitry), the instantaneous power consumption at any given moment remains equivalent to the single-model power (about 0.414 W dynamic). The advantage of sequential ensembling is that it does not increase the peak power or require additional parallel hardware. The trade-off is that it increases the total energy per image (since the FPGA has to compute three forward passes serially). Energy is the product of power and time: if one model takes time T to process an input at 0.414 W, then running three models sequentially will take ~3T time at the same 0.414 W, consuming roughly 3× the *energy* for that input. But the peak power remains 0.414 W, which might be crucial for battery-powered or heat-sensitive deployments.

Comparing these FPGA power figures to other platforms highlights one of the key benefits of FPGAs: high energy efficiency for specialized tasks. Drawing under half a watt for a CNN inference is orders of magnitude more efficient than a general-purpose CPU executing the same neural network in software. CPUs often consume tens of watts and take substantially longer to compute each image, resulting in far higher energy per inference. Even GPUs, which are optimized for parallel arithmetic, tend to run at higher power (tens to hundreds of watts) and

only achieve good energy efficiency when batching many inferences together. The FPGA design, by contrast, is batch-1 oriented and still maintains low power consumption.

4.3.2.2 Resource Utilization

From a broader perspective, the hardware resource utilization of the system reflects a careful compromise between accommodating multiple neural network models and maintaining the feasibility of implementation on the given FPGA. The analysis of LUT, FF, BRAM, and DSP usage across both the Geez-digit CNN (SimpleCNN) and the MNIST-digit CNN (SimpleCNNMNIST) confirms that the design leverages the FPGA's strengths (massive parallel arithmetic and configurable logic) while approaching, but not exceeding, critical resource limits.

Both CNN architectures were deliberately constrained in size (with relatively few layers and filters) so that even an ensemble of three models could fit in the device. The SimpleCNNMNIST model, for instance, used about 9% of available LUTs and 30% of DSPs in single-model form. The SimpleCNN for Geez digits are a slightly larger network (having an extra convolution layer and more feature maps), and it correspondingly consumes more resources – though still within the FPGA's capacity. When three instances of SimpleCNN were implemented in parallel (for the Geez ensemble), the resource utilization scaled up roughly threefold, coming very close to the device limits in some categories (e.g. saturating nearly all DSP blocks). High utilization is generally positive for efficiency (no silicon area is wasted), but if taken too far it can hurt timing closure or power. In this case, despite the high utilization, timing remained acceptable (no violations), and power stayed within reasonable bounds, indicating that the resource usage was balanced and the FPGA toolflow handled the design well.

Another important aspect of resource utilization is how the design handles the ensemble at a system level. The architecture is inherently modular: each CNN model instance is a block that can be instantiated once (for a single model or sequential ensemble) or multiple times (for a parallel ensemble). The *sequential ensemble* reuses the same block, which is an extremely resource-efficient approach. The only added resource overhead for sequential ensembling is some control logic and perhaps small buffers to hold intermediate votes or results between model runs, which are negligible compared to the CNN itself. This means that in scenarios where throughput can be sacrificed, one could deploy an ensemble on a much smaller FPGA by using the sequential approach. For instance, even a low-cost FPGA that could barely fit one

model could still host an ensemble (running one after another) with the same accuracy benefits, at the cost of proportionally higher latency. In contrast, the *parallel ensemble* demands replicating the full resource usage for each additional model.

4.3.2.3 Inference Latency and Throughput

Inference latency and throughput are critical metrics for evaluating real-time performance. Latency refers to the time taken for a single input to propagate through the neural network accelerator and produce an output, while throughput denotes how many inferences can be completed per unit time (often measured in frames per second, FPS). This section compares the simulation-based inference latencies of the SimpleCNNMNIST designs against the theoretical limits predicted by static timing analysis, and discusses the implications for operating frequency and real-time suitability.

The post-implementation simulation (functional timing simulation without hardware delays) was conducted with a 10 ns clock period (100 MHz). The single SimpleCNNMNIST model produced an end-to-end inference latency of approximately 15,495 ns (~15.5 µs) per image. The parallel ensemble version (which runs three SimpleCNNMNIST models concurrently on the same input) achieved a similar latency of about 15,605 ns (~15.6 µs). This negligible 0.7% overhead in the parallel ensemble is expected, as the three CNN models execute in parallel, adding only minor combining latency. In contrast, a sequential ensemble (where two models process the input one after the other) would incur roughly triple the latency of a single model. These results indicate that parallelization at the model level can preserve single-model latency (by doing work concurrently), whereas sequential execution linearly increases latency with the number of models in the ensemble.

Theoretical Timing Limits (WNS and Fmax): To understand the maximum speed the designs can safely run; the Worst Negative Slack (WNS) from timing analysis will be examined. WNS represents the margin by which the design meets or fails timing at the target clock period. Synthesis and static timing were performed with a 20 ns clock period (50 MHz); all designs reported a positive WNS, meaning they met the 50 MHz timing with time to spare. The single CNN model had WNS = 2.538 ns, the sequential ensemble WNS = 2.970 ns, and the parallel ensemble WNS = 2.416 ns.

These slack values can be used to estimate the *theoretical maximum frequency* (Fmax) for each design.

$$F_{\max} \approx \frac{1}{T_{\text{clk}} - \text{WNS}}$$

Using this formula, the single model's critical path (20 ns – 2.538 ns = 17.462 ns) corresponds to Fmax ≈ 57.3 MHz. Likewise, the sequential ensemble can theoretically clock up to ~58.7 MHz, and the parallel ensemble up to ~56.9 MHz, based on their slack margins.

Latency and Throughput at 50 MHz vs. Maximum Frequency: Given the above results, we can compare the real hardware latency (at 50 MHz or at Fmax) with the optimistic simulation latencies (at 100 MHz). If the single CNN model is run at the original 50 MHz clock used in synthesis, its 15.5 µs simulation latency would double to ~31 µs (since the clock period is twice as long), yielding a throughput of about 32,000 FPS.

$$1{,}549.5 \text{ cycles} \times 20 \text{ ns} = 30{,}990 \text{ ns} = 30.99 \text{ µs}$$

$$\text{FPS} = \frac{1}{30.99 \times 10^{-6}} \approx 32{,}260 \text{ FPS}$$

However, because the design has positive slack, it can be clocked higher than 50 MHz. At the estimated Fmax of ~57.3 MHz, the single model's inference latency would be about 27 µs, corresponding to roughly 37,000 FPS throughput. This is the theoretical minimum latency (and maximum throughput) attainable without violating timing. By comparison, the 100 MHz simulation was *idealized* – it suggests a 15.5 µs latency (≈64,500 FPS), which is much lower than what the real hardware can achieve. In fact, the simulation clock was approximately 1.75× faster than the single model's Fmax, meaning the simulation-based latency was roughly half of the hardware-constrained latency. A similar gap is seen for the parallel ensemble: at 56.9 MHz Fmax its latency would be ~27.4 µs (~36,400 FPS), versus 15.6 µs (≈64,100 FPS) in 100 MHz simulation. The sequential ensemble, if run at ~58.7 MHz, would have an inference time around 81–82 µs for three models in series (~18,900 FPS).

4.3.3 Scalability and Flexibility

The design and implementation of the FPGA-based ensemble CNN system exhibit scalability and flexibility at multiple levels, which are key for extending the approach to larger networks, different tasks, or bigger ensemble sizes. At the architectural level, the framework was built in a modular fashion: each convolutional layer, pooling layer, and dense layer is a reusable Verilog module with parameterized dimensions (such as image size, number of channels, kernel size, etc.). This modular design means that scaling up the network for instance, increasing the

number of layers or the number of neurons per layer can be achieved by instantiating additional modules or widening their parameters, as long as resources permit. The study demonstrated this flexibility by implementing two different CNN architectures (SimpleCNN for Geez digits, and SimpleCNNMNIST for MNIST) under the same framework with minimal changes to the hardware design flow.

The SimpleCNN has four convolution layers and was designed for 20-class Geez digit recognition, whereas SimpleCNNMNIST has a slightly different topology optimized for the simpler 10-class MNIST task. Despite these differences, both models were integrated into the *same* FPGA framework seamlessly. This confirms that the hardware is general enough to support various ANN/CNN configurations. The only modifications needed were at the level of model parameter inputs and minor control logic adjustments; the core dataflow (streaming through pad, conv, pool, and dense units) remained consistent. Such flexibility is valuable because it suggests that the framework can readily be adapted to new problems: as long as a neural network can be expressed in terms of the provided module types (convolutions, pools, etc.) and fits in the FPGA resources, it can be deployed without redesigning the entire hardware. In essence, the design acts as a template for a family of CNNs rather than a single fixed network implementation.

The study explored *pruning* and *quantization* as means to shrink the models. Additional pruning (removing more redundant weights) allows a larger model architecture to be squeezed into the same resource footprint by reducing the number of active multiplications. There is a trade-off, as aggressive pruning might hurt accuracy, but if done carefully (e.g., structured pruning that removes entire filters or channels) it could free up resources to allocate elsewhere in the network. The flexibility of the design in terms of precision is already shown: both 24-bit and 16-bit versions were implemented and evaluated, illustrating that the hardware was designed to handle different fixed-point precisions with minor changes.

In summary, the system demonstrates strong scalability and flexibility, grounded in its modular architecture and resource-conscious design. It supports different CNN architectures (proving generality across two datasets), adjustable precision, and multiple ensemble configurations. It can be scaled up to larger networks or more ensemble members, although careful consideration must be given to the FPGA's resource ceiling.

4.3.4 Error Analysis

Looking first at the single-model performance, certain patterns emerge in the errors on both the Geez and MNIST digit recognition tasks. For the simpler MNIST dataset (10 classes of handwritten Arabic digits), errors are exceedingly rare (the single model was already about 98.95% accurate), but when they do occur, they tend to involve digits that are similar in appearance. Common confusions include, for example, mistaking a sloppy *"4"* for a *"9"*, or vice versa, as those two digits share a similar triangle-loop structure when hand-written.

In the case of the Amharic (Geez) digit recognition (20 classes), the error patterns are a bit more complex, reflecting the greater difficulty of the task. The Geez numerals have more classes and some of them are visually intricate or share elements (for instance, the symbols for 100 and 1000 have some common components in their writing). The single SimpleCNN model achieved about 90.30% accuracy on the Geez test set, meaning nearly 10% of the test examples were misclassified. Investigating these errors, certain digits were consistently harder to recognize and there are also wrongly classified images in both train and test sets. These erroneous images include empty spaces, non-digits signs, and incorrectly labelled images.

The introduction of ensemble methods was motivated by exactly these kinds of errors; the aim was that different models might not make the same mistakes on the same inputs. The results confirm that ensembling significantly reduces the overall error rates: for Amharic digits, accuracy jumped from 90.3% to 93.59% with the custom-weighted ensemble, and for MNIST from 98.95% to 99.21% even with pruning.

In practice, the models were trained with slight variations (different random initializations, different training augmentation or hyperparameters) so that their decision boundaries wouldn't be identical. The benefit is clearly seen in error patterns: the misclassifications that remain in the ensemble output tend to be those that are genuinely challenging or ambiguous cases where all models struggled. In MNIST, after ensembling, the error rate is so low (0.79%) that only a handful of images are wrong out of 10,000. In Geez, after ensembling with pruning, around 6.4% of test examples are still misclassified, but this is a substantial reduction from 9.7% (single model). The remaining errors often involve the most difficult classes as mentioned where apparently all three models in the ensemble had trouble.

Another aspect of the error analysis is examining the impact of the quantization and pruning on the types of errors. It can be observed that for the Amharic digit model, quantization from 24-bit to 16-bit caused a noticeable drop in accuracy (from ~93.5% to ~88.2% for the ensemble

in one experiment). This implies that some errors were introduced or exacerbated by the reduced numerical precision. A closer look revealed that many of the new errors in the 16-bit model were on those classes that were already borderline the reduced precision likely made the model's feature representations or decision boundaries a bit less exact, so inputs that were previously just on the correct side of a classification threshold might have flipped to the wrong side.

4.3.5 Benchmarking

To contextualize the performance of the proposed FPGA-based ensemble CNN system, it is useful to compare the results with both similar FPGA implementations in the literature and with more conventional platforms like CPUs and GPUs. Such benchmarking helps illustrate the advantages and any trade-offs involved in the approach. Across the board, the design demonstrates competitive or superior performance-per-cost metrics, especially in terms of energy efficiency and throughput given the low power budget.

In terms of accuracy, the SimpleCNN ensemble model achieves state-of-the-art levels for the tasks at hand, although using a much more compact model. For the Geez handwritten digit dataset, the ensemble reached ~93.6% accuracy, which is only a few percentage points away of the best reported results that use much larger CNN models. (Ali Nur et al., 2022) report about 96.21% accuracy with a deep CNN on the same Geez digit dataset. The ensemble model accuracy is slightly lower, but it's important to note that the referenced 96% model is significantly heavier in computation and was evaluated in a software context without the resource constraints of FPGA deployment.

The performance of the framework is further validated on the MNIST dataset, a widely adopted benchmark for handwritten digit recognition. The best-performing configuration, operating at its maximum safe frequency, is detailed below.

Single Model (SimpleCNNMNIST)

Throughput: 32,260 FPS

Latency: 30.99 μs

Resource Usage: 26,032 LUTs / 19,892 FFs / 606 DSPs / 2.5 BRAMs

Power Usage: 0.414W

Accuracy: 98.39%

Parallel Ensemble (3x SimpleCNNMNIST)

Throughput: 32,054 FPS

Latency: 30.99 μs

Resource Usage: 78,797 LUTs / 58,994 FFs / 1818 DSPs / 7.5 BRAMs

Power Usage: 1.267W

Accuracy: 99.11%

The ensemble implementation demonstrates that a considerable increase in accuracy can be achieved with minimal additional latency by leveraging the parallel processing capabilities of the FPGA. Although the power usage increases nearly threefold when compared to the single-model version, the throughput remains constant, making the power-to-accuracy ratio highly favorable for real-time applications.

To further highlight the efficiency of the proposed design, comparisons are drawn with two FPGA-based accelerators reported by (Liang et al., 2024)shown in the table below:

Table 4.12 Reported Accuracy, power consumption and Latency Liang et al.

Platform/Approach	Accuracy	Power Consumption (w)	Latency (Time)
PIPELINE accelerator	99.01%	2.193 w	1.07 ms
UNROLL accelerator	99.01%	2.029 w	16.02 ms

Compared to these designs, the proposed system achieves comparable or superior accuracy with substantially lower latency and power consumption. Specifically, the FPGA ensemble design reaches 99.11% accuracy with only 30.99 microseconds of latency and 1.267 W of power usage, representing a 96.9% reduction in latency and a 42.2% reduction in power compared to the PIPELINE accelerator. Against the UNROLL accelerator, the latency is reduced by over 98%.

From an energy-efficiency perspective, the proposed system demonstrates remarkable improvements. While the designs reported by Liang et al. are optimized for high throughput using loop pipelining and unrolling strategies, they are constrained by relatively high latency and power profiles, which may limit their application in power-sensitive environments such as edge devices or wearable platforms. In contrast, the modular and quantized architecture presented in this thesis targets real-time, per-sample inference with high responsiveness and minimal power draw, aligning well with the requirements of embedded and ultra-low-power AI applications.

Furthermore, comparative insights from other academic studies reinforce these findings. For instance, Qiu et al. (2016) demonstrated that their FPGA implementation of a VGG-like network achieved only 4.45 FPS on Zynq ZC706 at 5.7 W power, emphasizing the challenge of running large CNNs on embedded FPGAs. Meanwhile, Peng et al. (2021) reported >5000 FPS throughput on an Alveo U280 using binary-weighted ResNet-18, with highly specialized hardware support. In this context, the proposed system achieves six times higher throughput than high-end platforms running quantized models, using only a fraction of the power and without requiring binarization.

It is also noteworthy that the system is designed for streaming, low-batch inference, which represents a performance bottleneck for GPU accelerators. GPUs rely on large batch sizes to amortize overheads, but in interactive or streaming inference scenarios, such batching is often infeasible. CPUs, while more versatile, cannot match the FPGA's parallelism and deterministic timing. In comparative academic benchmarks, CPUs have been shown to take tens of milliseconds to execute a LeNet-style inference, even on highly optimized software stacks (Reuther et al., 2019; Liu et al., 2024). GPUs reduce this latency to under a millisecond, but typically operate at 200–300 W power. The proposed FPGA framework achieves inference latencies an order of magnitude lower than GPUs and two orders of magnitude lower than CPUs, while operating under 1.3 W in ensemble mode.

In conclusion, the benchmarking results highlight the energy-efficient, high-performance nature of the developed FPGA framework. It not only delivers competitive accuracy on real-world tasks but does so with significantly better latency and power efficiency than comparable solutions. The combination of ensemble inference, model compression, and fixed-point arithmetic within a highly parallelized FPGA architecture offers a compelling path forward for scalable and responsive neural network deployment in real-time, resource-constrained environments.

4.4 Implementation Challenges and Bottlenecks

During synthesis, achieving timing closure for deeper pipelines required iterative floor planning and manual constraint adjustments. Some critical paths in the convolution modules initially violated timing at the target 50 MHz clock. Pushing deeper pipelining stages helped but increased register use.

Finally, testbench validation exposed latency mismatches between simulation and post-implementation results. Small irregularities in clock domain crossings required additional

synchronization registers. These challenges underscore the need to iteratively refine both RTL and Vivado constraints to meet the low-latency, high-throughput goals.

CHAPTER 5 CONCLUSION AND RECOMMENDATION

5.1 Summary of Findings

This thesis implements a flexible FPGA-based framework for ensemble neural network models and evaluates it on a Xilinx FPGA. The performance was evaluated using two different digit classification problems: recognition of handwritten Geez script digits and the widely used MNIST dataset. The experimental results demonstrated that our FPGA accelerator can achieve low-latency inference and high energy efficiency for small CNN models. For instance, the deployed CNN achieved millisecond-range inference times with accuracy above 99% on the MNIST benchmark, while consuming only a few watts of power. When comparing these results to conventional processors, the advantages are clear, the FPGA design drastically reduces energy consumption relative to a GPU or CPU for the same task, while delivering comparable (or faster) response times. This outcome is in line with contemporary research which shows FPGAs offering the best energy-per-operation and lowest latency among general-purpose computing devices for neural network inference. Therefore, it can be concluded that FPGA-based implementation is a viable and often superior solution for deploying lightweight CNNs in resource-constrained settings.

A key strength of the developed framework is its generalizability and reconfigurable architecture. Unlike fixed-function accelerators or application-specific ICs, this design was conceived to be modular and not tied to a single CNN model or dataset. The hardware architecture is composed of parameterized modules (for convolution, pooling, fully-connected layers, etc.) that can be re-used or re-scaled to implement different network topologies. This means that, with minor adjustments, the same FPGA design can support multiple NN models beyond the ones demonstrated in this thesis. This claim is validated by deploying two distinct CNN instances on the framework: one trained on Geez digit data and another (LeNet-5 variant) trained on MNIST. The success of both implementations confirms that the platform is not limited to MNIST, but rather MNIST serves as a convenient benchmark example of a lightweight CNN that our system can handle. In practice, a developer could retrain or replace the neural network (for example, using a different character set or an entirely different image classification task) and then map it onto our FPGA accelerator with minimal hardware changes.

This level of flexibility is a known advantage of FPGAs in deep learning applications, and this study capitalizes on it by ensuring the design remains reconfigurable and extensible. The modular approach also simplifies future upgrades. For instance, substituting a more advanced

convolution unit or adding support for a larger input image size can be done by updating the corresponding module without redesigning the entire system.

Overall, the thesis has demonstrated that an FPGA-based CNN inference engine can meet the dual objectives of high performance (in terms of throughput and latency) and low power consumption for small-scale image recognition tasks. The combination of algorithmic optimization (through network simplification and quantization) and hardware parallelism allowed us to reach energy efficiency levels unattainable by CPU or GPU only solutions, albeit with a cost of some accuracy. This work thereby contributes to the growing body of evidence that heterogeneous computing with FPGAs is a promising route for deploying AI algorithms at the edge, where energy and real-time constraints are paramount.

5.2 Future Work

Based on the findings, several directions to further improve and generalize the proposed framework are recommended:

- Integrate training or on-device learning: In its present form, the framework accelerates inference only; training of the CNN is done offline on a CPU/GPU. An interesting extension would be to incorporate support for on-chip training or online learning (even if at a limited scale). This could involve accelerating the backward pass or adopting incremental learning approaches, enabling the system to adapt to new data in the field. Although training is far more computationally intensive, partial reconfiguration of the FPGA or hybrid CPU-FPGA strategies could be investigated to extend the framework's capabilities beyond inference.
- Optimize and automate the design flow: Adapting the accelerator to a new CNN currently requires some manual reconfiguration (e.g., adjusting layer parameters in the hardware description). To enhance usability, a software toolflow or compiler could be developed to automatically map an arbitrary trained CNN onto the FPGA accelerator. High-level synthesis (HLS) tools or FPGA ML frameworks (such as Xilinx FINN or DNN-to-RTL compilers) could be leveraged to generate the hardware configuration from a network description. This would further demonstrate the generality of the approach and reduce the engineering effort for deploying different models. Automating the mapping process would make the framework more accessible for rapid prototyping of CNNs on FPGAs.

- Extending to multi-FPGA or SoC implementations: The current design was implemented on a single FPGA (Zynq SoC). For future work, one could investigate a multi-FPGA setup or an updated SoC with a higher-end FPGA or additional accelerator cores. A multi-FPGA system, as shown by researchers, can attain even higher absolute throughput while maintaining energy efficiency.

In conclusion, the FPGA-based CNN acceleration framework developed in this thesis proved to be effective for the targeted application of handwritten digit recognition, delivering low-latency and energy-efficient performance that meets the demands of real-time applications. More importantly, the design principles; modularity, reconfigurability, and hardware/software co-design ensure that this solution can be extended to a variety of NN models and applications with relative ease. By validating the framework on both a culturally significant use-case (Geez digit recognition) and a standard benchmark (MNIST), its broad applicability is illustrated. A continued development along the recommended directions will further enhance the system's capability and flexibility. Ultimately, this work serves as a step toward general-purpose CNN acceleration on FPGA platforms, contributing to the deployment of efficient AI inference engines in everyday devices and domains where traditional high-power computing is not feasible. The evidence collected and the comparative analysis with CPU/GPU approaches strongly support the adoption of FPGA-based accelerators for energy-conscious AI solutions at the network edge.

REFERENCES

Aarrestad, T., Loncar, V., Ghielmetti, N., Pierini, M., Summers, S., Ngadiuba, J., Petersson, C., Linander, H., Iiyama, Y., Di Guglielmo, G., Duarte, J., Harris, P., Rankin, D., Jindariani, S., Pedro, K., Tran, N., Liu, M., Kreinar, E., Wu, Z., & Hoang, D. (2021). Fast convolutional neural networks on FPGAs with hls4ml. *Machine Learning: Science and Technology*, *2*(4). https://doi.org/10.1088/2632-2153/ac0ea1

Ahmed, S. T., Hefenbrock, M., & Tahoori, M. B. (2024). *Tiny Deep Ensemble: Uncertainty Estimation in Edge AI Accelerators via Ensembling Normalization Layers with Shared Weights*. http://arxiv.org/abs/2405.05286

Ali Nur, M., Abebe, M., & Rajendran, R. S. (2022). Handwritten Geez Digit Recognition Using Deep Learning. *Applied Computational Intelligence and Soft Computing*, *2022*. https://doi.org/10.1155/2022/8515810

AMD. (2021). *U280 Performance - Vitis AI Library User Guide (UG1354)*. https://docs.amd.com/r/1.3-English/ug1354-xilinx-ai-sdk/U280-Performance

Anupreetham, A., Ibrahim, M., Hall, M., Boutros, A., Kuzhively, A., Mohanty, A., Nurvitadhi, E., Betz, V., Cao, Y., & Seo, J.-S. (2024). High Throughput FPGA-Based Object Detection via Algorithm-Hardware Co-Design. *ACM Trans. Reconfigurable Technol. Syst.*, *17*(1). https://doi.org/10.1145/3634919

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123–140. https://doi.org/10.1007/BF00058655

Caruana, R., Crew, G., & Ksikes, A. (2004). *Ensemble Selection from Libraries of Models*.

Cheng, H., Zhang, M., & Shi, J. Q. (2023). *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*. http://arxiv.org/abs/2308.06767

Du, K. L., Zhang, R., Jiang, B., Zeng, J., & Lu, J. (2025). Foundations and Innovations in Data Fusion and Ensemble Learning for Effective Consensus. In *Mathematics* (Vol. 13, Issue 4). Multidisciplinary Digital Publishing Institute (MDPI). https://doi.org/10.3390/math13040587

Freund, Y., & Schapire, R. E. (1997). A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139. https://doi.org/https://doi.org/10.1006/jcss.1997.1504

Gdaim, S., & Mtibaa, A. (2025). Accelerating convolutional neural networks on FPGA platforms: a high-performance design methodology using OpenCL. *Journal of Real-Time Image Processing*, *22*(2), 67. https://doi.org/10.1007/s11554-025-01647-5

Giasemis, F. I., Lončar, V., Granado, B., & Gligorov, V. V. (2025). *Comparative Analysis of FPGA and GPU Performance for Machine Learning-Based Track Reconstruction at LHCb*. http://arxiv.org/abs/2502.02304

Gysel, P. (2016). *Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks*. http://arxiv.org/abs/1605.06402

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., & Dally, W. J. (2016). EIE: efficient inference engine on compressed deep neural network. *Proceedings of the 43rd International Symposium on Computer Architecture*, 243–254. https://doi.org/10.1109/ISCA.2016.30

Hansen, L. K., & Salamon, P. (1990). Neural Network Ensembles. *Pattern Analysis and Machine Intelligence, IEEE Transactions On*, *12*, 993–1001. https://doi.org/10.1109/34.58871

Hauck, S., & DeHon, A. (2010). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*.

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017). *SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE*. www.kaggle.com

Ian Goodfellow, Yoshua Bengio, & Aaron Courville. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2018). *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*.

Ji, M., Al-Ars, Z., Hofstee, P., Chang, Y., & Zhang, B. (2023). FPQNet: Fully Pipelined and Quantized CNN for Ultra-Low Latency Image Classification on FPGAs Using OpenCAPI. *Electronics (Switzerland)*, *12*(19). https://doi.org/10.3390/electronics12194085

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P. L., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., … Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. *Proceedings - International Symposium on Computer Architecture*, *Part F128643*, 1–12. https://doi.org/10.1145/3079856.3080246

Kondratyuk, D., Tan, M., Brown, M., & Gong, B. (2020). *When Ensembling Smaller Models is More Efficient than Single Large Models*. http://arxiv.org/abs/2005.00570

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., & Cortes, C. (1998). The MNIST database of handwritten digits. *CiNii Research*. http://yann.lecun.com/exdb/mnist/

Lecun, Y., Eon Bottou, L., Bengio, Y., & Abstract|, P. H. (n.d.). *Gradient-Based Learning Applied to Document Recognition*.

Liang, Y., Tan, J., Xie, Z., Chen, Z., Lin, D., & Yang, Z. (2024). Research on Convolutional Neural Network Inference Acceleration and Performance Optimization for Edge Intelligence. *Sensors*, *24*(1). https://doi.org/10.3390/s24010240

Liu, Z., Dou, Y., Jiang, J., Xu, J., Li, S., Zhou, Y., & Xu, Y. (2017). Throughput-optimized FPGA accelerator for deep convolutional neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, *10*(3). https://doi.org/10.1145/3079758

Lu, H. (2024). FPGA-Based Vit Inference Accelerator Optimization. In *Highlights in Science, Engineering and Technology ACME* (Vol. 2024).

Nechi, A., Groth, L., Mulhem, S., Merchant, F., Buchty, R., & Berekovic, M. (2023). FPGA-based Deep Learning Inference Accelerators: Where Are We Standing? *ACM Transactions on Reconfigurable Technology and Systems*, *16*(4). https://doi.org/10.1145/3613963

Park, S., Kwon, B., Sim, K., Lim, J., Kim, T.-H., & Choi, J. (2021). *Uniform-Precision Neural Network Quantization via Neural Channel Expansion*. https://openreview.net/forum?id=oGq4d9TbyIA

Peng, H., Zhou, S., Weitze, S., Li, J., Islam, S., Geng, T., Li, A., Zhang, W., Song, M., Xie, M., Liu, H., & Ding, C. (2021). *Binary Complex Neural Network Acceleration on FPGA*. http://arxiv.org/abs/2108.04811

Qin, T., Li, Z., Zhao, J., Yan, Y., & Du, Y. (2025). Mixed precision quantization based on information entropy. *Scientific Reports*, *15*(1). https://doi.org/10.1038/s41598-025-91684-8

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., & Yang, H. (2016). Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 26–35. https://doi.org/10.1145/2847263.2847265

Rawat, W., & Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. In *Neural Computation* (Vol. 29, Issue 9, pp. 2352–2449). MIT Press Journals. https://doi.org/10.1162/NECO_a_00990

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., & Kepner, J. (2019). Survey and Benchmarking of Machine Learning Accelerators. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–9. https://doi.org/10.1109/HPEC.2019.8916327

Shantharama, P., Thyagaturu, A. S., & Reisslein, M. (2020). Hardware-Accelerated Platforms and Infrastructures for Network Functions: A Survey of Enabling Technologies and Research Studies. In *IEEE Access* (Vol. 8, pp. 132021–132085). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ACCESS.2020.3008250

Shawahna, A., Sait, S. M., & El-Maleh, A. (2019). FPGA-Based accelerators of deep learning networks for learning and classification: A review. In *IEEE Access* (Vol. 7, pp. 7823–

7859). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ACCESS.2018.2890150

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. (2017). *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. http://arxiv.org/abs/1703.09039

Trimberger, S. M. (2015). Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, *103*(3), 318–331. https://doi.org/10.1109/JPROC.2015.2392104

Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., & Vissers, K. (2017). FINN: A framework for fast, scalable binarized neural network inference. *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 65–74. https://doi.org/10.1145/3020078.3021744

Wang, E., Davis, J. J., Zhao, R., Ng, H. C., Niu, X., Luk, W., Cheung, P. Y. K., & Constantinides, G. A. (2019). Deep neural network approximation for custom hardware: Where We've Been, Where We're going. *ACM Computing Surveys*, *52*(2). https://doi.org/10.1145/3309551

Wang, K., Liu, Z., Lin, Y., Lin, J., & Han, S. (2019). *HAQ: Hardware-Aware Automated Quantization with Mixed Precision*.

Weng, O., Andronic, M., Zuberi, D., Chen, J., Geniesse, C., Constantinides, G. A., Tran, N., Fraser, N. J., Duarte, J. M., & Kastner, R. (2025). Greater than the Sum of its LUTs: Scaling Up LUT-based Neural Networks with AmigoLUT. *FPGA 2025 - Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 25–35. https://doi.org/10.1145/3706628.3708874

Yan, F., Koch, A., & Sinnen, O. (2024). *A survey on FPGA-based accelerator for ML models*. http://arxiv.org/abs/2412.15666

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based accelerator design for deep convolutional neural networks. *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 161–170. https://doi.org/10.1145/2684746.2689060

Zhao, Y., Shumailov, I., Mullins, R., & Anderson, R. (2019). *TO COMPRESS OR NOT TO COMPRESS: UNDERSTANDING THE INTERACTIONS BETWEEN ADVERSARIAL ATTACKS AND NEURAL NETWORK COMPRESSION*.

Research Fund Acknowledgement

This research is funded by Adama Science and Technology University under the grant number ASTU/SM-R/1126/25, Adama, Ethiopia.