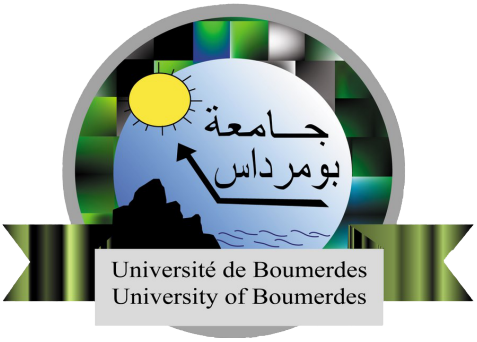

Design and Implementation of a CNN accelerator on SoC FPGA

By

Mabrouki Hammou Yacine

Tamine Abderrahmene

Supervisor: Dr. Walid Touzout

Undergraduate Project

Submitted to the Institute of Electrical and Electronic Engineering UMBB

Department of Basic teaching

Sunday 29th June, 2025

Acknowledgments

Bismillah, all praise and thanks are due to God, the Most Powerful and Most Merciful, who granted us the strength, patience, and determination to complete the work presented in this thesis.

We would like to express our deepest gratitude to our parents, brothers, and friends for their unwavering support, encouragement, and prayers throughout this journey. We also extend sincere appreciation to Wameedh Sc and all the alumni for their insightful guidance and motivation.

A special thanks goes to our supervisor, Dr. Walid Touzout, for his continuous support, mentorship, and invaluable assistance throughout every stage of this work.

Dedications

I, Mabrouki Hammou Yacine, dedicate this thesis to my beloved parents for their endless support, prayers, and encouragement. I also extend this dedication to my dear brothers and to the entire Islamic Ummah.

Last but not least, I express my heartfelt gratitude to all those who have taught, guided, and inspired me throughout my academic journey.

Abstract

Convolutional Neural Networks (CNNs) are widely used in computer vision tasks for their high accuracy and learning capability. However, their computational demands pose challenges for real-time or embedded deployment. Hardware acceleration using System-on-Chip Field Programmable Gate Arrays (SoC FPGAs) offers a promising solution, combining the flexibility of programmable logic with the efficiency of embedded processors.

This project presents the design and implementation of a CNN accelerator on the Xilinx Zynq-7000 (xc7z010) SoC FPGA. The architecture supports matrix convolution and vector multiplication, featuring 8 parallel Processing Elements (PEs). The design was described in VHDL and synthesized using Vivado, while the inference process runs on the ARM Cortex-A9 using bare-metal C. INT8 quantized models were trained in PyTorch for efficient deployment.

The accelerator, operating at 100 MHz, demonstrated low-latency execution. Although the speedup over software-only execution was modest due to memory bottlenecks, the design proved effective in offloading compute-intensive tasks. FPGA resource utilization reached 24% LUTs, 13% FFs, 38% BRAM, and 90% DSPs, fitting well within the xc7z010 device constraints.

Overall, this work highlights the effectiveness of custom CNN accelerators on SoC FPGAs for edge AI applications, offering an efficient trade-off between performance and resource usage.

Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
DSP	Digital Signal Processor/Slice
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GIC	Generic Interrupt Controller
GPIO	General Purpose Input Output
GOP/s	Giga Operations Per Second
IP	Intellectual Property (core)
LUT	Look-Up Table
MAC	Multiply-Accumulate
MNIST	Modified National Institute of Standards and Technology
PE	Processing Element
PL	Programmable Logic
PS	Processing System
PTQ	Post-Training Quantization
QAT	Quantization Aware Training
ReLU	Rectified Linear Unit
SDK	Software Development Kit
SoC	System on Chip
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction and Theoretical Background

1.1 Introduction

Artificial Intelligence (AI) refers to the capability of machines to emulate human cognitive functions such as learning, reasoning, and perception. Within AI, Machine Learning (ML) and, more specifically, deep learning have emerged as powerful approaches for enabling machines to learn from data and make intelligent decisions. Deep learning leverages artificial neural networks to extract hierarchical representations from large datasets, allowing systems to perform complex tasks in areas such as image analysis, natural language processing, and speech recognition.

Convolutional Neural Networks (CNNs) are a specialized class of deep learning models designed for data with grid-like topology, such as images and videos. By exploiting spatial locality through convolutional layers, CNNs have achieved state-of-the-art performance in tasks including image classification, object detection, and semantic segmentation.

While training CNNs is typically performed on high-performance GPUs or computing clusters due to its heavy computational demands, inference—the process of applying a trained model to new data—has different priorities. Inference often needs to be ex-

ecuted with low latency, minimal power consumption, and real-time responsiveness, particularly in embedded or edge computing environments.

To meet these requirements, this project explores the implementation of a CNN inference accelerator on a Field-Programmable Gate Array (FPGA), specifically the Zynq-7000 XC7Z010 SoC. FPGAs offer reconfigurable hardware that combines parallelism, low latency, and energy efficiency—qualities that make them well-suited for accelerating neural network operations. The focus of this work is on deploying a pre-trained model and offloading its compute-intensive convolution and fully connected layers to custom hardware modules, thereby achieving efficient on-device inference without the need for retraining or architectural modification.

1.2 Theoretical Background

1.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are inspired by the structure of the human brain. They consist of layers of artificial neurons that process inputs by applying mathematical formulas—a weighted sum followed by a nonlinear activation. Learning happens by adjusting the strength of these connections (weights) to minimize the difference between predicted outputs and actual results.

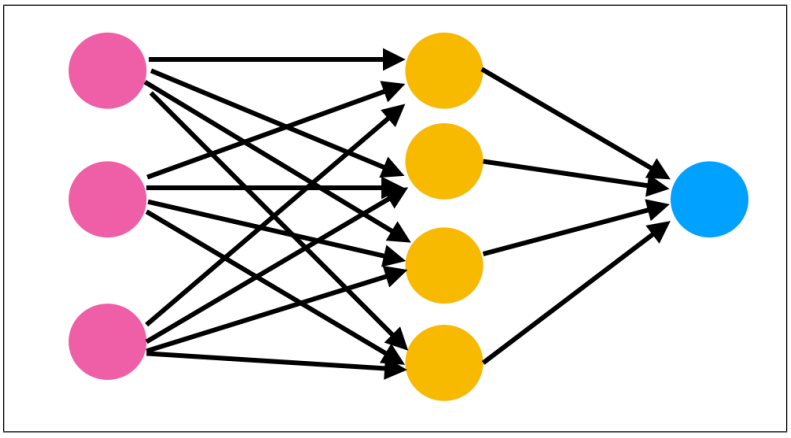

Figure 1.1: Structure of a Deep Neural Network

1.2.2 Convolutional Neural Networks

CNNs extend ANNs by replacing fully connected layers with convolutional layers that apply learnable filters to local receptive fields. A typical CNN architecture comprises:

- **Convolutional layers:** Perform discrete convolutions with multiple kernels, extracting spatial features.
- **Pooling layers:** Downsample feature maps to reduce spatial dimensions and computation.
- **Activation functions:** Introduce nonlinearity (e.g., ReLU) to model complex relationships.
- **Fully connected layers:** Aggregate high-level features for final classification or regression.

Convolutional and pooling operations exhibit high spatial locality and parallelism, making them amenable to hardware acceleration.

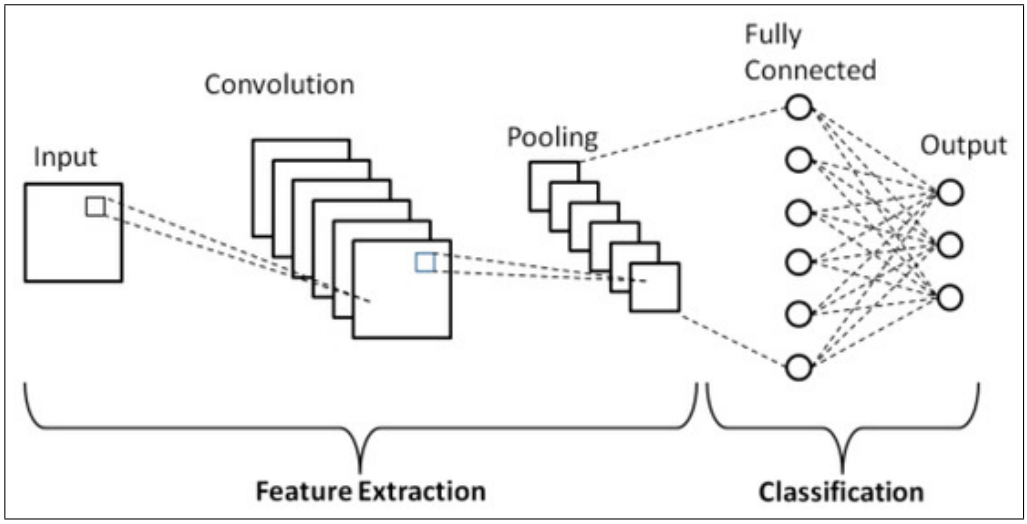

Figure 1.2: Structure of Convolutional Neural Networks

1.2.3 Training vs. Inference

Training involves forward propagation of inputs through the network and backward propagation of errors to update weights. This process requires extensive matrix operations, high-precision arithmetic, and large memory bandwidth, typically executed on GPUs.

Inference uses fixed weights to perform forward propagation only, focusing on efficient execution of convolution and linear operations. Inference precision can often be reduced (e.g., to 8-bit quantization) to improve performance without significant accuracy loss.

1.2.4 FPGA Architecture and Suitability

FPGAs consist of an array of configurable logic blocks (CLBs), DSP slices for arithmetic operations, on-chip Block RAM (BRAM), and programmable interconnects. Key advantages for CNN inference include:

- **Parallelism:** Multiple convolution windows can be processed simultaneously.
- **Low Latency:** Dedicated datapaths eliminate instruction fetch and decode overhead.
- **Reconfigurability:** Hardware can be tailored to specific network topologies and precision.
- **Energy Efficiency:** Fine-grained control over data movement minimizes energy consumption.

However, FPGAs have limited on-chip resources (LUTs, DSPs, BRAM), requiring careful design trade-offs between parallelism, precision, and throughput.

1.2.5 Project Focus

This project implements a custom CNN inference accelerator in VHDL, targeting the Zynq-7000 XC7Z010 FPGA. The design supports 3×3 convolutions and fully connected layers using 8-bit quantized data. By focusing solely on inference, the accelerator avoids the complexity and resource demands of on-device training, enabling a compact, high-throughput solution for edge AI applications.

1.3 Chapter Overview

Chapter 2 presents the hardware architecture of the CNN accelerator, including processing elements, memory layout, and FPGA integration. Chapter 3 describes the software flow, from training and quantization to inference execution on the Zynq platform. Chapter 4 evaluates the system's performance through implementation results, simulation verification, and runtime metrics. Chapter 5 outlines future improvements such as expanded CNN support, better data handling, and new application domains.

Chapter 2

Hardware System Design

2.1 Introduction to the CNN Accelerator

Embedded systems and hardware design present intriguing challenges in accelerating deep learning models, especially within FPGA contexts. Driven by the increasing demand for efficient edge AI solutions, this project implements a custom Convolution Neural Network (CNN) accelerator using VHDL. The target device is the Zynq-7000 XC7Z010 FPGA, emphasizing a fully pipelined, resource-efficient architecture capable of handling CNN and linear layers entirely in hardware.

The accelerator supports CNNs with fixed 3×3 kernels and employs a streamlined data flow from the Zynq Processing System (PS) through Block RAM (BRAM) and into parallel processing elements (PEs). Integration within the Zynq ecosystem leverages AXI BRAM Controller and AXI GPIO interfaces, enabling seamless PS-PL data communication and hardware control.

2.2 Target FPGA Platform

This accelerator is tailored specifically for the Zynq-7000 XC7Z010 FPGA, featuring:

- 2.1 Mb Block RAM (60 BRAM tiles)
- 80 DSP slices

- 17,600 LUTs
- 35,200 flip-flops

Resource constraints require careful optimization in logic simplicity, pipelining efficiency, and memory management techniques, such as compact data packing within BRAM.

2.3 Accelerator Architecture

The custom CNN accelerator architecture is modular and pipelined, efficiently executing convolutions and fully connected operations with quantized 8-bit data.

2.3.1 Processing Elements (PEs)

The accelerator integrates 8 parallel Processing Elements (PEs), each performing multiply-accumulate operations with:

- 9-element input and weight vectors (3×3 convolution windows)
- 8-bit signed integer arithmetic
- A 4-stage pipelined adder tree enabling one-cycle MAC operations

Each PE operates in two modes:

1. **Direct Mode**: For convolution layers.
2. **Accumulation Mode**: For fully connected or multi-channel layers.

Post-processing includes dequantization via bit-shifting and clipping to an 8-bit signed range.

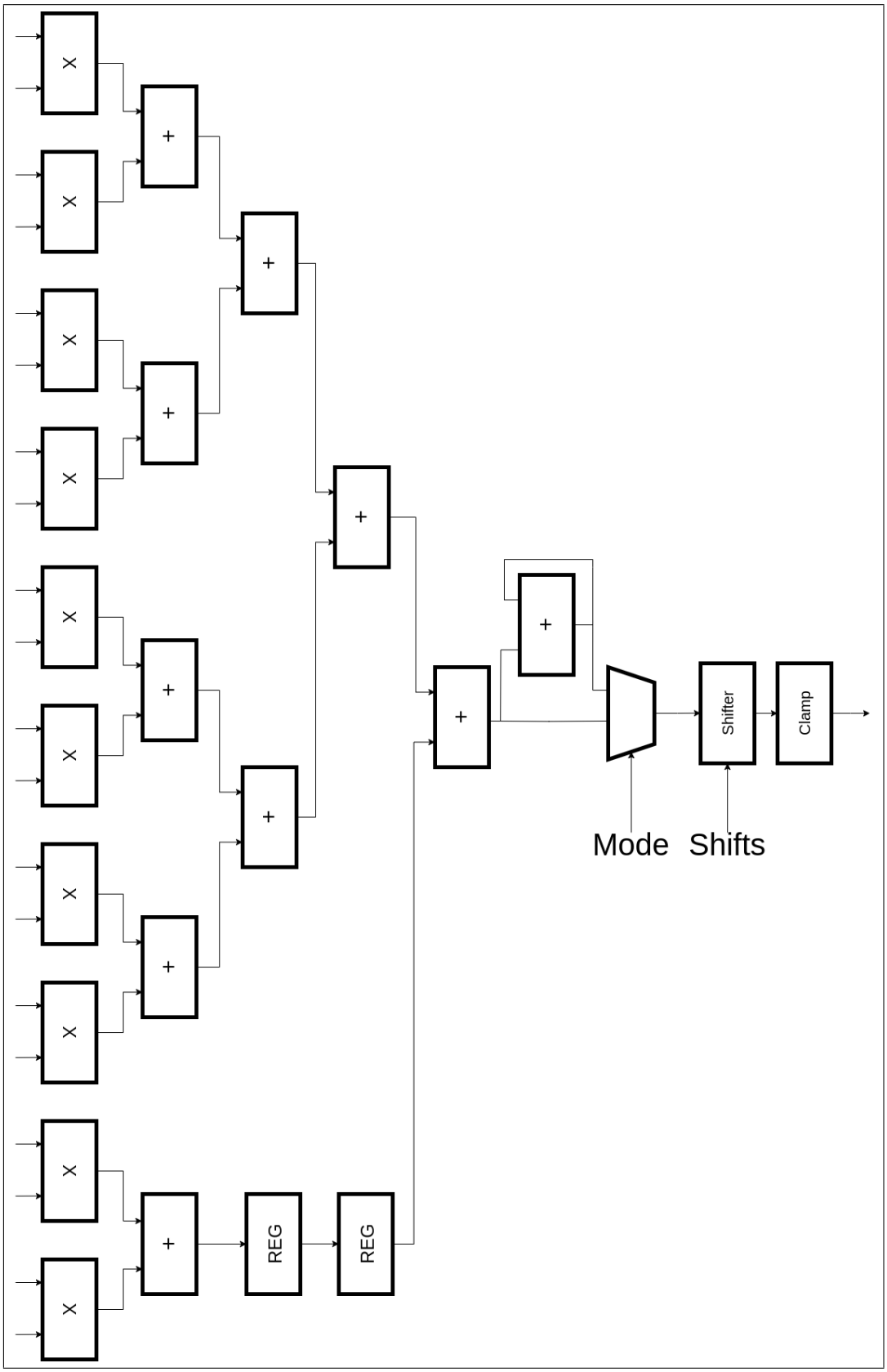

Figure 2.1: Processing element architecture

2.3.2 Memory Organization

Efficient memory management is critical, utilizing 21 BRAM blocks divided into:

Input BRAM Group (3 BRAMs): Stores convolution weights or image data, structured as 72-bit (9-byte) memory words.

Data BRAM Group (18 BRAMs): Supplies image data or dense layer weights, configured as a 576-bit bus to serve all PEs concurrently.

Output BRAMs: Dedicated storage for processed data, optimized for subsequent retrieval by the PS.

2.3.3 Data Flow and Control Mechanism

Data handling involves three sequential stages:

1. **Data Loading**: PS writes data and weights to internal BRAMs.
2. **Internal Computation**: PEs perform computation.
3. **Result Storage**: Results are stored in BRAM, later accessed by PS.

Control is managed via two synchronized Finite State Machines (FSMs), automating data loading and result storage processes, thereby minimizing PS overhead.

2.4 Integration with Zynq Processing System (PS)

Integration with Zynq PS is facilitated through AXI interfaces, enabling efficient hardware-software communication.

2.4.1 AXI BRAM Controller IP

This interface supports direct, memory-mapped communication between PS and PL through AXI BRAM Controllers, configured with 32-bit data widths for optimal PS compatibility. BRAMs internally adjust data widths to match accelerator requirements, ensuring efficient synchronization.

2.4.2 AXI GPIO IP and Interrupt Handling

The accelerator employs two AXI GPIO IP cores to facilitate communication between the Processing System (PS) and Programmable Logic (PL):

- **Control GPIO (13-bit output)**: This GPIO is configured as a 13-bit output interface from the PS to the PL. It transmits control signals used to configure and initiate the accelerator's operation. The mapping is as follows:
  - `axi_gpio(4 downto 0)`: `entemp`, used to select the appropriate output word (BRAM Bank).
  - `axi_gpio(5)`: `start`, initiates a computation cycle.
  - `axi_gpio(6)`: `acc`, enables accumulation mode for multi-layer processing.
  - `axi_gpio(7)`: `rst`, performs a hardware reset on FSMs and processing units.
  - `axi_gpio(11 downto 8)`: `divide`, sets the dequantization factor.
  - `axi_gpio(12)`: `arrange`, controls data arrangement logic and memory routing.

  The GPIO output signals are assigned in VHDL and decoded directly in the `main.vhd` architecture. While only 13 bits are used, the IP is configured specifically for this width, optimizing control bandwidth.

- **Status GPIO (input with interrupt)**: This second AXI GPIO IP is configured as an input to receive the `done` signal from the PL. It is connected to the PS interrupt controller, allowing the PS to be alerted asynchronously when processing is complete. This interrupt-driven approach reduces CPU load and avoids continuous polling.

While this dual-GPIO configuration is not the most resource-optimized approach, it clearly separates control and status paths, simplifying debugging and integration. It also enhances modularity and reliability in the communication protocol between software and hardware domains.

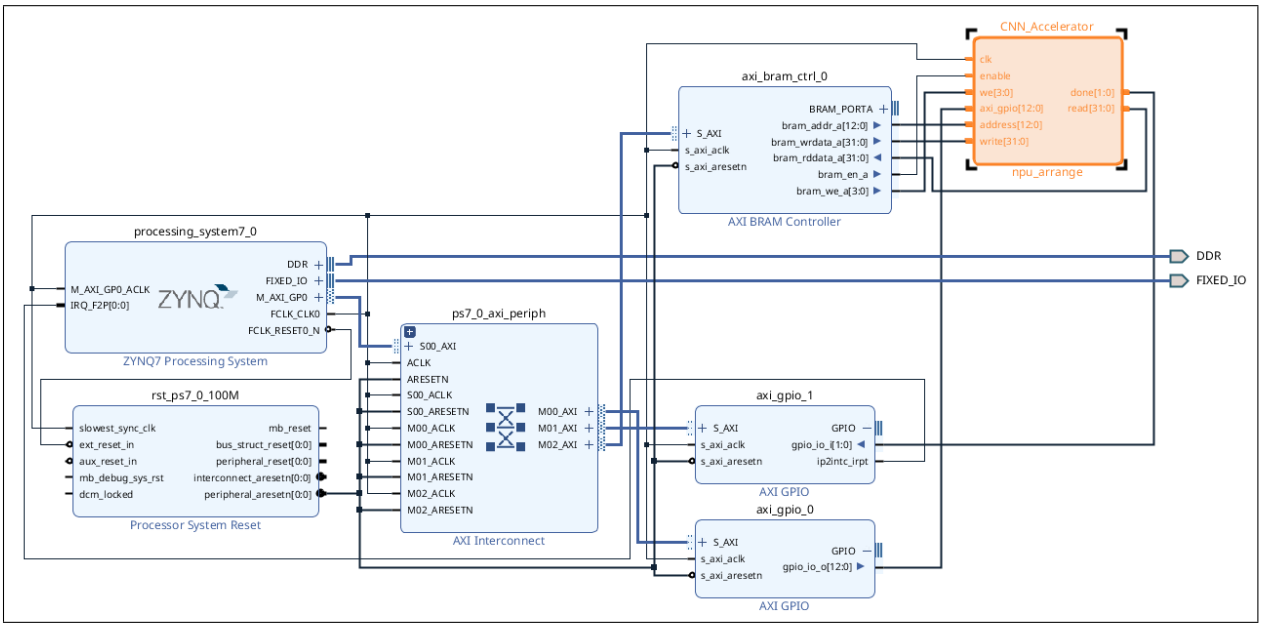

Figure 2.2: System block design

Chapter 3

Software System Design

This chapter details the software design that supports both the training and inference stages of the Convolutional Neural Network (CNN) used in the FPGA-based accelerator system. The chapter is divided into two main sections: the **Training Phase**, which takes place entirely in software, and the **Inference Phase**, where the trained model is deployed and executed on the custom hardware. This dual-phase structure ensures that the software pipeline is aligned with the hardware design, enabling effective performance optimization and modularity.

A key motivation behind the separation of training and inference responsibilities is to allow scalability and adaptability for future enhancements. Training benefits from the flexibility and expressive power of software frameworks, while inference prioritizes speed and deterministic behavior in hardware-constrained environments.

3.1 Training Phase

3.1.1 Framework and Model Architecture

The training was conducted using the PyTorch deep learning framework. The model follows a simple CNN structure:

- A 3×3 convolution layer with stride 3

- A flattening operation
- A fully connected layer without activation functions

```
class SimpleModel(nn.Module):
    def __init__(self, in_channels=1, out_features=10):
        super(SimpleModel, self).__init__()
        self.conv = nn.Conv2d(in_channels, 1, (3,3), stride=3, bias=False)
        self.linear = nn.Linear(100, out_features, bias=False)
    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.shape[0], -1)
        x = self.linear(x)
        return x
```

3.1.2 Dataset and Preprocessing

The MNIST dataset was used. Each image (28×28) underwent the following preprocessing:

- Padding with 1 pixel on all sides
- Normalization to $[-1, 1]$ using mean 0.5 and std 0.5

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Pad(padding=1),
    transforms.Normalize((0.5,), (0.5,))
])
```

3.1.3 Quantization

Post-Training Static Quantization (PTQ) was applied to convert weights and activations to 8-bit integers:

1. Convert model to evaluation mode
2. Run sample data for calibration
3. Replace floating-point operations with integer versions

Quantized models preserved accuracy while reducing memory and computation.

3.1.4 Export Format

Quantized weights were exported as hardcoded C arrays:

- Flattened to 1D 8-bit arrays
- Packed into `uint32_t` words
- Organized by BRAM bank

This format ensured deterministic loading and compatibility with the memory layout of the hardware.

3.2 Inference Phase

3.2.1 Software Stack and Environment

The inference software was written in C and executed on the ARM PS of the Zynq-7000 SoC using:

- Xilinx SDK and drivers
- AXI BRAM Controller for memory access
- AXI GPIO for control
- Interrupt controller (GIC)
- `XTime` for timing

Initialization routines configure GPIO, load memory, and register interrupts.

3.2.2 Memory Handling

Data is loaded to BRAM in bank-address format:

```
for (int address = 0; address < max_address; address++) {
    for (int bank = 0; bank < 18; bank++) {
        BRAM_Write(bank, address, image[(address * 18) + bank]);
    }
}
```

Separate blocks are used for image data, convolution weights, and dense weights.

3.2.3 Accelerator Control

Control is via GPIO bits:

- `start()` – begins computation
- `acc()` – enables accumulation
- `dequantization(scale)` – configures scaling
- `rearrange()` – prepares output

Interrupts indicate stage completion and invoke the `data_ready()` handler.

3.2.4 Result Collection and Prediction

Results are packed in 32-bit words and decoded:

```
int8_t val[10] = {0};
for (int mask = 0; mask < 10; mask++) {
    val[mask] = ((reading[mask/4]) >> ((8 * mask) % 32)) & 0xff;
    if (val[mask] > max) {
        max = val[mask];
```

```
        prediction = mask;
    }
}
```

This minimizes memory bandwidth and simplifies logic.

3.2.5 Performance and Timing

Timing was measured using `XTime`:

```
XTime_GetTime(&starting);
start();
while (counter == 0);
XTime_GetTime(&end);
elapsed_ns = 1.0 * (end - starting) / COUNTS_PER_SECOND;
```

Average inference time was under 300µs, suitable for real-time digit classification.

Chapter 4

Results and Discussion

This chapter presents a comprehensive evaluation of the implemented CNN accelerator, highlighting hardware resource utilization, simulation results, timing measurements, and performance insights. The discussion is organized into two sections: Implementation Results (from Vivado) and Inference Results (from the SDK).

4.1 Implementation Results

The accelerator was synthesized and implemented on the Zynq-7000 XC7Z010 SoC. We analyze resource usage in two configurations:

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
synth_1	constrs_1	Synthesis Out-of-date								2485	2477	23.0	0	72
impl_1	constrs_1	route_design Complete!	1.527	0.000	0.061	0.000	0.000	0.124	0	2485	2477	23.0	0	72

Figure 4.1: Resource utilization of Accelerator

Resource	Utilization	Available	Utilization %
LUT	4228	17600	24.02
LUTRAM	129	6000	2.15
FF	4698	35200	13.35
BRAM	23	60	38.33
DSP	72	80	90.00

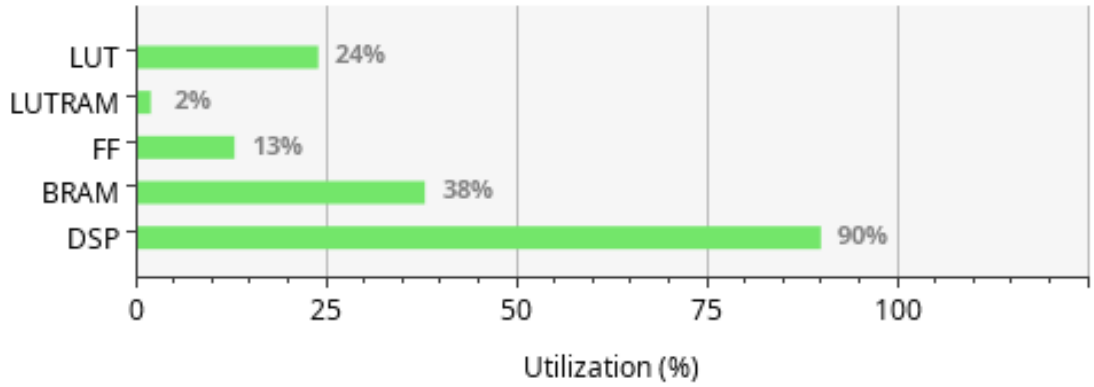

Figure 4.2: Resource utilization of Full system

As shown in Figure 4.2, the full system utilizes approximately 24% of the available LUTs, 13% of flip-flops, 38% of BRAMs, and 90% of DSP slices. The high DSP utilization is expected due to the compute-intensive nature of convolution operations, where each 3×3 convolution requires 9 multiply-accumulate operations. The accelerator design prioritizes DSP efficiency by implementing parallel convolution engines that fully utilize the available arithmetic resources.

4.1.1 Simulation and Functional Verification

The full system was simulated in Vivado. A simulation waveform, shown in Figure 4.3, confirmed correct data transfer, start signal triggering, done signal reception, and BRAM access patterns. The simulation verified:

- Proper initialization of the accelerator

- Correct memory address generation during convolution operations
- Pipeline timing and synchronization
- Accurate result accumulation
- Proper handshaking between the processing system and the accelerator

These simulation results provided confidence in the functional correctness of the accelerator design before proceeding to hardware implementation.

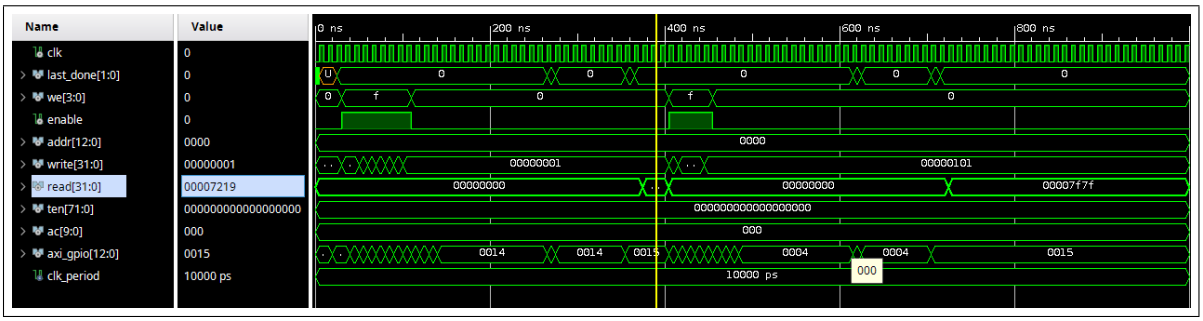

Figure 4.3: Simulation waveform showing correct data transfer and signal behavior.

4.2 Inference Results

4.2.1 UART Output from SDK

The UART terminal output from Xilinx SDK confirmed the flow of initialization, data movement, execution, and prediction. The system successfully recognized input images with the expected digit classification results, validating end-to-end functionality of the accelerator.

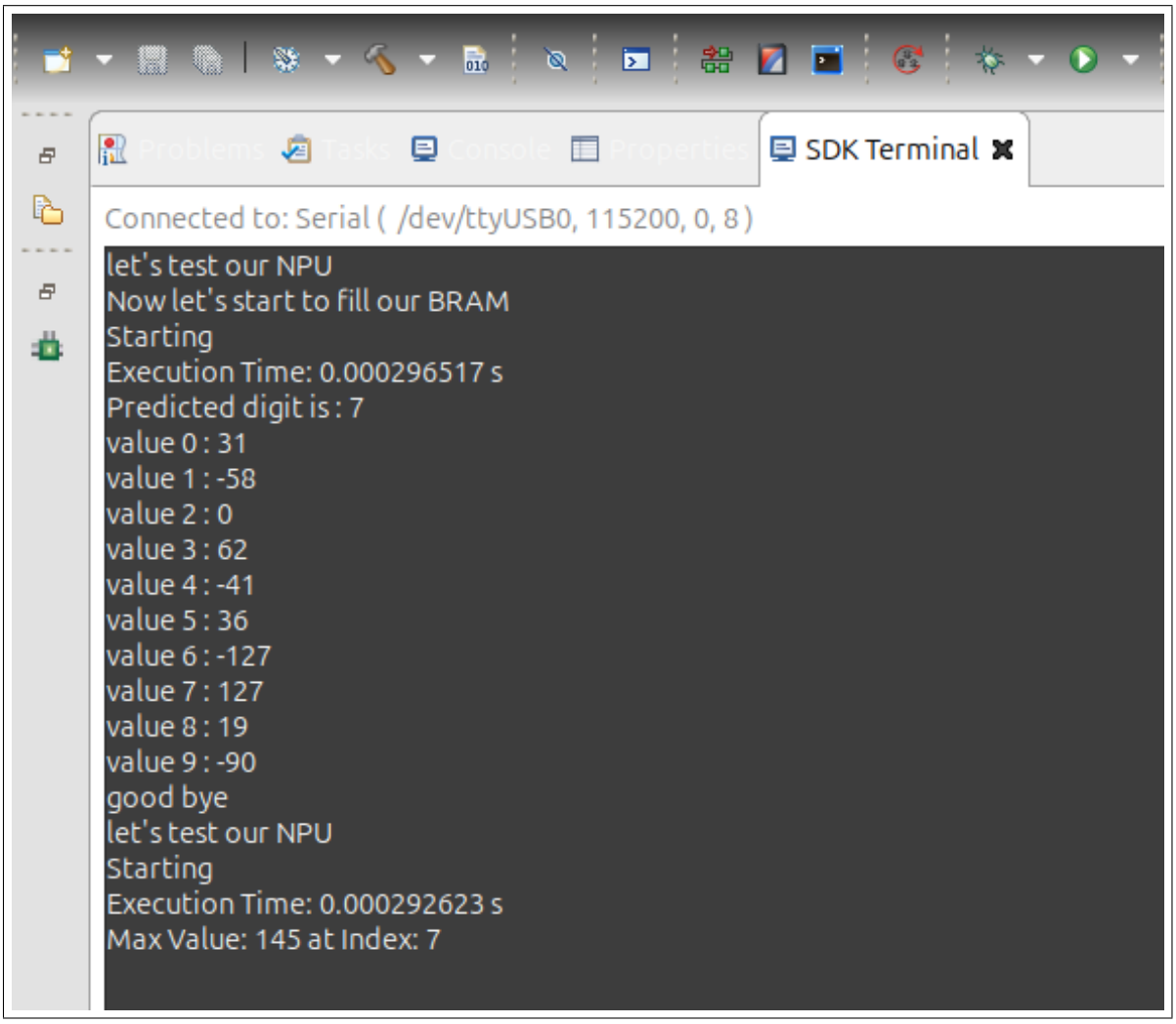

Figure 4.4: UART output result.

Remark: The execution timing breakdown in Figure 4.5 includes both the CNN inference *with* the hardware accelerator (296.517 µs) and *without* the hardware accelerator (292.623 µs), as validated by the UART output above.

4.2.2 Execution Timing Breakdown

Table 4.5 shows the detailed timing breakdown of the CNN inference execution with the hardware accelerator, including the time taken for each stage and its percentage contribution to the total execution time.

Stage	Time (s)	Distribution (%)
Image loading	0.000121386	37.82
Constant data loading	0.000019164	5.97
First processing stage	0.000004920	1.53
Rearrangement	0.000003117	0.97
Linear layer loading	0.000111651	34.78
Second processing stage	0.000003465	1.08
Reading results	0.000002061	0.64
Final linear loading	0.000030447	9.49
Final computation	0.000004191	1.31
Prediction calculation	0.000000933	0.29

Figure 4.5: Execution Timing Breakdown of CNN Inference with Hardware Accelerator

The close timing observed between the software and hardware versions is mainly due to the use of the AXI BRAM controller, which introduces latency. More optimized communication methods, such as AXI Stream, would significantly improve overall throughput.

4.3 Throughput in GOP/s

Each 3×3 convolution involves 9 MAC operations, and the accelerator executes 8 such convolutions in parallel:

- Total operations = $100 \times 8 \times 9 = 7200$
- Total cycles = $100 + 8 = 108$
- Frequency = 100 MHz = 10^8 Hz

Time per execution: $T = \frac{108}{10^8} = 1.08$ microseconds

Throughput: GOP/s = $\frac{7200}{1.08 \times 10^{-6}} \times 10^{-9} \approx 6.66$

Chapter 5

Future Work

This project lays a solid foundation for future enhancements in FPGA-based CNN acceleration. While the initial implementation focused on demonstrating feasibility and executing simple inference tasks on edge devices, several key improvements can significantly increase the system's practical utility and performance.

5.1 Support for Additional CNN Layers

The current accelerator supports only basic convolution and fully connected layers. Future designs should integrate support for additional components commonly found in modern CNNs:

- **ReLU activation:** Introduces non-linearity and improves model expressiveness.
- **Pooling layers:** Such as max pooling, to reduce spatial resolution and computational load.

These enhancements would expand the range of deployable models and enable more complex inference tasks.

5.2 Improved Data Throughput

A key performance limitation in this project was the communication bottleneck introduced by the AXI BRAM controller. Replacing this with an **AXI-Stream interface** would significantly reduce data transfer latency and allow the hardware pipeline to operate at higher throughput, better matching the speed of internal processing elements.

5.3 Runtime Weight Loading

Currently, weights are hardcoded and loaded at compile time. Supporting **runtime weight loading from external memory** (e.g., SD card, DDR) would make the system more flexible, enabling:

- Real-time switching between models.
- Support for larger datasets or multi-task inference.

5.4 Refined Quantization Strategy

The current quantization uses basic 8-bit post-training techniques. Improved strategies, including:

- Quantization-aware training (QAT),
- 16 bit arithmetic support,

can improve numerical stability and maintain accuracy with minimal resource cost.

5.5 Application to Brain Signal Classification

Future iterations will adapt this accelerator for **brain signal classification**, where low latency and edge deployment are critical. Signal preprocessing, custom CNN structures,

and real-time decision support are key areas of exploration. This represents a natural evolution from digit classification to biomedical applications.

5.6 Design Constraints

All enhancements must remain within the constraints of the Zynq-7000 XC7Z010 FPGA:

- Limited LUT, DSP, and BRAM availability.
- Power constraints for edge deployment.
- Real-time processing requirements.

These constraints will guide design trade-offs and optimization choices going forward.

In summary, the current design is a scalable and modular foundation. With targeted improvements in architecture, data handling, and application scope, it has the potential to become a robust tool for real-world embedded AI systems.

Bibliography

1. Pramila P. Shinde and Seema Shah. "A Review of Machine Learning and Deep Learning Applications". In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)* (Aug. 2018).

2. Will Koehrsen. "Deep Neural Network Classifier". In: *Medium* (July 2017)

3. Vincent Camus et al. "Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.4 (Oct. 2019), pp. 697–711. ISSN: 2156-3365.

4. Mirza Cilimkovic. "Neural Networks and Back Propagation Algorithm". In: *Institute of Technology Blanchardstown, Blanchardstown Road North Dublin* 15.1 (2015).

5. Sungju Ryu et al. "BitBlade: Area and Energy-Efficient Precision-Scalable Neural Network Accelerator with Bitwise Summation". In: *Proceedings of the 56th Annual Design Automation Conference 2019* (June 2019), pp. 697–711.

6. Luca Urbinati and Mario R. Casu. "A Reconfigurable Multiplier/Dot-Product Unit for Precision-Scalable Deep Learning Applications". In: *Proceedings of SIE 2022* (2023), pp. 9–14.

7. S. Kala, B. R. Jose, J. Mathew, and S. Nalesh. "High-Performance CNN Accelerator on FPGA Using Unified Winograd-GEMM Architecture". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.12 (Dec. 2019), pp. 2816–2828.

8. Markus Nagel et al. "A White Paper on Neural Network Quantization". In: *Qualcomm AI Research* (June 2021). arXiv preprint: https://arxiv.org/abs/2106.08295.