Liverpool John Moores University

Efficient Hardware Accelerator for Real-Time Facial Computing on Resource-Constrained Edge Devices

By

P. A. P. S. Perera

(23PG1-015)

A dissertation submitted in partial fulfillment for the
MSc in Embedded Systems and IC Design

August 2024

Abstract

The advancement of edge computing has underscored the need for efficient processing solutions capable of handling computationally intensive tasks on resource-constrained devices. This research conveys a comprehensive investigation into the design, development, and evaluation of a hardware accelerator, specifically an FPGA-based, for real-time facial computing. The study addresses the challenges associated with deploying deep learning algorithms, such as convolutional neural networks (CNNs), on embedded systems with limited computational resources. The analysis demonstrates that the Net Engine custom IP replicates software-based outputs with high accuracy and consistency, achieving exceptional performance across various layers and channels of the proposed network (Pnet) model, with accuracy scores nearing 1.000. Notably, the integration of the Net Engine IP resulted in substantial reductions in processing times across critical stages of the CNN. For instance, at a scale factor of the image frame of 1.00, model processing time decreased from 1.5988 seconds to 0.2866 seconds, a reduction of 1.3122 seconds, representing an 82.0% reduction. CNN layer processing time fell from 0.2754 seconds to 0.0420 seconds, representing an 84.7% reduction, and channel processing time dropped from 0.0275 seconds to 0.0042 seconds. Single frame processing time saw a decrease from 0.0084 seconds to 0.0007 seconds, representing a 91.7% reduction. These improvements underscore the Net Engine's capability to enhance real-time performance while operating at a 50MHz clock frequency. Power consumption analysis revealed that the ZYNQ Processing System 7 (PS7) is the primary power consumer, providing insights into the energy efficiency of the design. This research contributes significantly to the field of edge computing by offering a practical and effective solution for deploying facial computing applications on resource-limited devices, thereby advancing the capabilities and operational efficiency of edge AI systems.

Keywords: FPGA, Hardware Acceleration, Facial Computing, Convolutional Neural Networks, Embedded Systems, Real-time Processing, Edge Device

Acknowledgments

I would like to express my sincere gratitude to everyone who provided their advice, recommendations, and support throughout the completion of the project.

First, I would like to express my gratitude to my supervisor, Eng. Kasun K. Herath, for his exceptional guidance, insightful advice, and support. His assistance was crucial in realizing my efforts and bringing this project to the final step. I am also profoundly grateful to my co-supervisor, Dr. Lasith Yasakethu, for his support and valuable feedback.

I would like to express sincere thanks to Eng. Nalaka Samarasinghe, the MSc program coordinator, as well as all the lecturers and faculty members who supported me in succeeding in the completion of the project.

Finally, I am deeply thankful to my family for their unwavering support and encouragement throughout this journey. I also wish to acknowledge my colleagues for their valuable contributions and the time they invested in assisting me with this project.

Thank you all for your generous support and assistance.

P.A.P.S. Perera

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Facial computing covers a spectrum of tasks, from face recognition to emotion detection and facial attribute analysis using facial landmarks, all stimulated by deep learning algorithms renowned for their accuracy and robustness after training. However, implementing these kinds of computationally costly algorithms on resource-constrained edge devices gives massive challenges. Those devices can be mapped from mobile phones to IoT devices and embedded systems, resource-limited computational power, memory, and energy resources. Achieving efficient and real-time facial computing on these platforms demands innovative solutions that optimize both hardware and software components.

Hardware accelerators are a better solution to address the real-time computational requirements of deep learning algorithms on edge devices. The Field-Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and Graphics Processing Units (GPUs) can be listed as well-known examples. These accelerators can significantly enhance processing speed, parallel processing, improve energy efficiency, and enable real-time performance. Nonetheless, each type of accelerator presents its unique performance in terms of flexibility, power consumption, and development complexity.

The selection of this research topic is stimulated by its paramount relevance in the contemporary landscape of technology and AI. Real-time facial computing on resource-constrained edge devices is a critical component of applications ranging from security systems to human-computer interaction. increasing demand for such applications calls for efficient solutions that can operate seamlessly on edge devices with limited resources. By addressing this challenge and designing specialized hardware accelerators that have CNN and max-pooling operation modules, this research seeks to contribute significantly to the development of efficient edge AI applications. The expected contributions include improved user experiences and a practical roadmap for deploying an engine for facial computing at the edge. This research will give the statical and visual evaluation of hardware-based and pure software-based CNN model implementation

1.1 Background and Context

In the modern era of computing, edge devices play a crucial role in enabling real-time data processing directly at the point of data generation. This approach, known as edge computing, significantly reduces latency and conserves bandwidth by processing data locally rather than relying on centralized cloud servers. One of the most demanding applications of edge computing is real-time facial computing, which encompasses tasks such as face recognition, emotion detection, and facial attribute analysis. These tasks are powered by deep learning algorithms that, while highly accurate

and robust, are computationally intensive.

As a technological trend, AI models are becoming an integral part of a valuable feature of modern technology. Therefore, artificial intelligence integrates a wide range of products and solutions. This shift is particularly evident in the IOT sector, where many devices now feature advanced AI-driven predictive capacities. These features enable devices to analyze data at the source and generate informed decisions in real time, enhancing their functionality and efficiency.

Instead of relying only on the software solution, integrating the software with the hardware solution can significantly enhance the performance and efficiency of the entire system. Usually, more computational power required operations can be offloaded from the CPU, and software drivers can be interfaced with the hardware implementation to utilize its computational power and produce an output from the hardware.

1.2 Challenges in Edge-Based Facial Computing

Achieving real-time facial computing on edge devices requires overcoming significant barriers. The high computational demands of deep learning algorithms often exceed the capabilities of resource-limited devices, leading to issues with processing speed, memory usage, and energy consumption. To address these challenges, it is essential to explore solutions that optimize both hardware and software components, ensuring that facial computing applications can operate effectively within the constraints of edge environments.

1.3 Problem Statement

Despite the advancements in edge computing technologies, achieving real-time facial computing on resource-constrained devices remains a challenging problem. Existing solutions may not fully address the performance and efficiency requirements needed for practical applications with statistical and visual evaluation. This research seeks to explore the effectiveness of specialized hardware accelerators in improving processing times and overall efficiency for facial computing tasks on hardware and software-based edge devices.

1.4 Research Objectives

The primary objective of this research is to design and implement a hardware accelerator engine, specifically utilizing FPGA-based architecture, and software driver to driver implemented hardware accelerator engine to archive real-time performance for facial computing tasks on an embedded system.

The research objectives are as follows:

- **Design and Implementation of Efficient Hardware Accelerator**: Design and implement FPGA-based hardware accelerators specifically tailored for real-time facial computing on resource-constrained embedded systems. This includes creating FPGA-based architecture to perform CNN and Max-pooling operations in high-performance and low-latency operations.

- **Optimization of Memory Hierarchy and Data Transfer**: Improve the memory hierarchy and data transfer mechanisms within embedded systems to enhance processing efficiency. And reducing the data transfer bottlenecks to improve overall system efficiency.

- **Effective Offloading of Computational Tasks**: Implement strategies to offload computational tasks from the main processor to the hardware accelerator to ensure real-time performance. And utilize the implemented hardware accelerator by the main processor to perform CNN and Max-pooling tasks.

- **Comprehensive Benchmarking and Evaluation**: Conduct detailed benchmarking and performance evaluations of the hardware accelerators against software-based solutions to assess their effectiveness. This includes the accuracy of the hardware accelerator output concerning the software-based output and processing speed, and comparing the effectiveness of the hardware approach with software implementations.

These objectives are listed to advance the field of edge computing and enhance the capabilities of real-time facial computing on resource-constrained edge devices. By addressing these goals, the research aims to contribute valuable insights and practical solutions to improve the performance and efficiency of facial computing applications in embedded environments.

1.5 Significance of the Study

This study is significant as it addresses a critical component of edge computing—achieving efficient real-time processing on devices with limited resources. By developing and optimizing hardware accelerators for facial computing, this research aims to contribute to the advancement of edge AI applications. The findings are expected to enhance user experiences and provide a practical roadmap for deploying specialized engines for facial computing on resource-constrained devices.

1.6 Structure of the Study

The study is structured into six chapters, each addressing a different aspect of the research on hardware accelerators for real-time facial computing on edge devices.

Chapter 1: Introduction

This chapter introduces the study, providing background and context for edge-based facial computing, outlining the challenges, and discussing the role of hardware accelerators. It also presents the problem statement, research objectives, and the significance of the study.

Chapter 2: Literature Review

This chapter reviews relevant literature, covering existing research and developments in facial computing, edge computing, and hardware accelerators. It identifies gaps in current knowledge and establishes the foundation for the research.

Chapter 3: Methodology

The methodology chapter describes the research design and approach, including, An overview of the system architecture and design.

1. *Hardware Setup*: Details of the hardware components used in the study.
2. *Programmable Logic (PL) Implementation*: Implementation details for programmable logic components.
3. *Net-Engine IP*: In-depth information on the Net-Engine IP, including its input and output ports, register file, floating-point operations, CNN module implementation, and Max-pooling.
4. *Processing System (PS) Implementation*: Details on the processing system, including the net engine driver and neural network implementation.
5. *Memory Map*: Explanation of the memory mapping for the system.
6. *Proposed Network (Pnet in MTCNN)*: Description of the proposed network architecture.
7. *Experimental Method*: Methods used for experimentation and evaluation.

Chapter 4: Results

This chapter presents the findings from the research, including:

1. *First CNN Layer Output Comparison*: Comparisons of the first CNN layer outputs, both numerical and visual.
2. *Final Layer Output Comparison*: Analysis of the final layer outputs.
3. *Model Accuracy Comparison*: Comparison of model accuracy against benchmarks.

4. *Estimated Power Utilization of PL*: Assessment of power utilization for programmable logic components.
5. *Timing Analysis*: Detailed timing analysis, including model processing time, CNN layer processing time, channel processing time, and single-frame processing time.

Chapter 5: Discussion

This chapter discusses the implications of the results, interpreting the findings in the context of existing research and addressing the research questions.

Chapter 6: Conclusion

The final chapter summarizes the study's conclusions and provides recommendations for future research and practical applications.

Chapter 2: Literature Review

The literature on deep learning, edge computing, and related technologies presents a diverse and evolving landscape of research. This critical review assesses key findings from recent studies, focusing on their strengths, limitations, and areas requiring further investigation.

Deep Learning's Role in Edge Computing

Recent scholarly work highlights the growing importance of optimizing deep learning algorithms for edge computing environments. Techniques such as model quantization and pruning have been identified as effective methods for reducing computational and memory requirements while preserving model accuracy (A. M., 2019). These advancements represent a significant step forward in making deep learning feasible on resource-constrained edge devices.

Despite these advancements, challenges remain in scaling optimized models across diverse edge devices. The trade-offs between model complexity and resource constraints often result in suboptimal performance in real-world scenarios. Moreover, the generalizability of these optimization techniques across different deep-learning models and tasks remains underexplored. There is a lack of comprehensive frameworks that integrate various optimization techniques into a unified approach for edge computing. Future research should focus on developing such frameworks to streamline the deployment of deep learning models and enhance their performance across different edge environments.

FPGA-Based Hardware Accelerators

Field-Programmable Gate Arrays (FPGAs) have shown considerable benefits as hardware accelerators, particularly for tasks like facial recognition and emotion detection. Studies demonstrate that FPGAs can improve both processing speed and energy efficiency, making them suitable for real-time applications (A. B. S., 2022).

The complexity of FPGA programming and the need for custom designs present significant barriers to widespread adoption. This complexity can limit accessibility and hinder the practical implementation of FPGA-based solutions. There is a need for more user-friendly tools and methodologies to simplify FPGA programming and design. Research should focus on developing accessible solutions to broaden the use of FPGAs and facilitate their application in a wider range of scenarios.

Neuromorphic Computing for Emotion Recognition in Healthcare

Neuromorphic computing presents a promising approach for real-time emotion recognition, especially in healthcare settings. The emphasis on energy-efficient hardware and real-time processing is a key strength of this technology, with potential applications in areas such as post-stroke emotional rehabilitation (A. B. S., 2022).

Despite its promise, neuromorphic computing is still in the early stages of practical application. The deployment of these technologies in real-world healthcare scenarios remains limited, which constrains their impact. Further research is needed to explore the scalability of neuromorphic solutions and their integration with existing healthcare systems. Understanding how these technologies can be seamlessly integrated into real-world applications is crucial for their broader adoption.

Safety and Efficiency in Transportation

The application of FPGA-based solutions for real-time emotion recognition in transportation systems demonstrates practical utility in enhancing passenger safety and optimizing transportation systems (A. B. S., 2022). The focus on real-time performance and safety is a notable strength.

A key challenge remains in balancing high computational performance with energy efficiency. Achieving this balance is critical but difficult, and current solutions often struggle to meet these dual demands effectively. There is a need for more comprehensive models that integrate real-time emotion recognition with other safety features in transportation systems. Future research should aim to develop these models to enhance overall system performance and safety.

Edge Computing in Autonomous Vehicles

The literature underscores the vital role of edge computing in autonomous vehicles, emphasizing the need for a balance between computational power, energy efficiency, and security (S. Liu, 2019). This focus aligns with the growing demands of autonomous driving systems.

Practical implementation of edge computing solutions in autonomous vehicles involves complex challenges related to real-time processing, data security, and system integration. Addressing these challenges is crucial for the successful deployment of edge computing in this context. Further research is needed to explore innovative solutions that address these challenges. The development of novel hardware accelerators and optimization techniques will be essential for advancing edge computing in autonomous vehicles.

Optimizing Deep Neural Networks

The literature highlights ongoing efforts to optimize deep neural networks (DNNs) for improved energy efficiency and performance (Xue Lv, 2021). Techniques such as cross-layer optimizations contribute to enhancing DNN efficiency, particularly in edge-computing contexts.

While progress has been made, there is a lack of holistic approaches that integrate multiple optimization strategies into practical applications. Current research often focuses on individual techniques without addressing the broader challenges of DNN optimization. Research should focus on developing unified frameworks that incorporate various optimization strategies. Such frameworks would facilitate the creation of more effective and practical solutions for DNN optimization.

Face Recognition in Embedded Systems

The literature provides valuable insights into the advancements in face recognition algorithms and hardware accelerators for embedded systems (A. Baobaid, 2022). Improvements in accuracy and computational speed are significant achievements.

Achieving real-time performance with minimal power consumption in embedded systems remains a challenge. The trade-offs between performance and power efficiency need to be addressed more thoroughly. Further research is needed to explore the scalability and adaptability of face recognition systems across different embedded platforms. Developing solutions that work effectively across various devices and applications is essential.

Face Detection in Deep Learning

The literature review effectively captures the evolution from traditional face detection methods to advanced deep learning-based approaches. The comparison of different techniques provides a comprehensive understanding of their effectiveness.

Despite improvements, challenges related to computational efficiency persist. Metrics such as FLOPs and latency are important, but practical implementations often face limitations that are not fully addressed. Future research should focus on practical implementations of deep learning-based face detection methods. Understanding how these techniques perform in real-world applications will be crucial for their broader adoption.

This review of the literature underscores significant advancements and ongoing challenges in the fields of deep learning, edge computing, and related technologies. While considerable progress has been made, addressing the identified limitations and gaps through innovative solutions and comprehensive frameworks will be essential for advancing these fields and ensuring their practical applicability.

Chapter 3: Methodology

The primary objective of this research is to design and implement an efficient hardware accelerator that can perform real-time facial computing tasks on resource-constrained edge devices. Achieving this objective necessitates a comprehensive methodology that integrates both hardware and software components to optimize performance, reduce latency, and maintain energy efficiency.

This section outlines the methodological approach adopted to realize the proposed solution. The methodology is structured around three key components: hardware design, software driver development, and performance evaluation. The hardware design focuses on developing a custom IP for Convolutional and Max-pooling operations, which will be implemented on the PYNQ-Z2 FPGA board. The software development component involves creating the necessary interfaces and drivers to integrate the hardware accelerator with the processing system. Finally, performance evaluation is conducted to benchmark the proposed solution against traditional software-based implementations, focusing on key metrics such as processing time, energy consumption, and overall system latency.

The methodology is iterative, allowing for continuous refinement of the hardware design and software integration based on performance evaluations. This iterative approach ensures that the final implementation is optimized for the specific constraints of the edge computing system. The following sections provide detailed descriptions of each methodological component, including the tools and techniques used in the development and evaluation process.

3.1 System Overview

This project is directed to the design and implementation of a hardware-accelerated solution for real-time facial computing on resource-constrained edge devices. The entire system is implemented and evaluated on a PYNQ-Z2 development board, which includes both a Processing System (PS) and Programmable Logic (PL) to enhance the performance and efficiency of the system. The PL side central design is known as net-engine IP, which is used to perform Convolution and Max-pooling operations with configurable registers. The data and control flows are optimized through the use of Direct Memory Access (DMA) and interrupt mechanisms, ensuring minimal latency and high throughput keeping the accuracy of the system.

The proposed system architecture is designed to achieve a balance between high performance, low latency, and energy efficiency, making it good enough for a diverse range of edge AI applications. The following diagram provides a comprehensive high-level overview of the system's architecture, detailing the key components and their interactions.

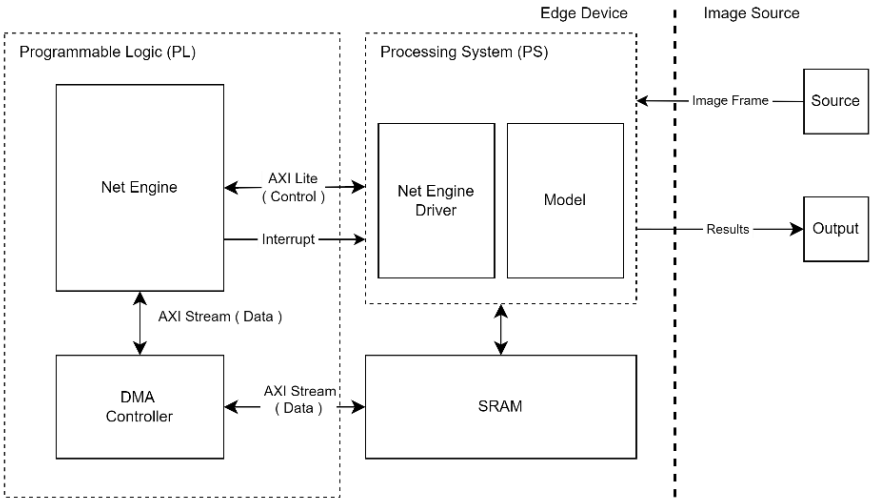


Figure 1: High-Level System Design

As depicted in the Figure 1 diagram, The system design integrates several critical components to facilitate efficient data processing and communication between the PL and PS sides. The net engine IP is responsible for executing the Convolution and Max-pooling tasks, meanwhile, the PS side is responsible for configuring, and reading the status of the net engine IP, and managing DMA transfers. Therefore, the net-engine driver has been introduced to the PS side. Additionally, the use of SRAM for temporary data storage further contributes to the system's decreasing memory access times. This architecture not only supports real-time facial computing but is also designed to be adaptable for other computing applications that utilize Convolution and Max-pooling operations to predict the output of the system, offering a versatile and scalable solution for embedded systems.

3.2 Hardware Setup

The implementation and the validation of the entire system are done by using the PYNQ-Z2 development board, which provides a robust platform for prototyping and testing the hardware design.

Typically, the PYNQ-Z2 is used with its official image that supports the Jupyter Notebook. However, for the research requirements, the PYNQ official image was not utilized. Instead of that, Vivado 2023.2 and Vitis 2023.2 were employed as primary software tools for design and implementation.

The development environment includes the following tools:

- **Vivado 2023.2**: This Xilinx software suite is utilized for the design, implementation, and debugging of the net-engine IP, Vivado provides the necessary tools for synthesizing, placing, and routing the hardware components required.
- **Vitis 2023.2**: This software is employed for developing and implementing the net-engine driver and the software components that are needed to evaluate the system's accuracy and timing performance.

The PYNQ-Z2 development board features a dual-core processing system (PS) side, but, for validation, only core 0 has been utilized. This choice simplifies the setup and focuses the evaluation on a single processing core to ensure stability and reliable operation of the system. The specifications of the development board have been mentioned below.

Category	Feature
Processor and Logic	
Chip	ZYNQ XC7Z020-1CLG400C
Processor	ARM Cortex-A9 dual-core, 650 MHz
Logic Slices	13,300 slices, each with 4x6-input LUTs and 8 flip-flops
Block RAM	630 KB
DSP Slices	220
Analog-to-Digital Converter (XADC)	On-chip Xilinx XADC
Memory and Storage	
DDR3 Memory	512 MB, 16-bit bus @ 1050 Mbps
Quad-SPI Flash	16 MB, with factory-programmed 48-bit EUI-48/64 identifier
MicroSD Slot	Yes
Power	

Power Supply	USB or external 7V-15V regulator
USB and Ethernet	
Gigabit Ethernet PHY	Yes
Micro USB-JTAG	For programming and debugging
Micro USB-UART	For serial communication
USB 2.0 OTG PHY	Supports host mode only
Audio and Video	
HDMI Ports	2 ports (input and output)
Audio	24-bit I2S DAC with 3.5mm TRRS jack; Line-in with 3.5mm jack

Table 1: The table of the specification of PYNQ-Z2 development board

3.3 Programmable Logic (PL) Implementation

The implementation of a hardware accelerator on the PYNQ-Z2 development board's Programming Logic side is the main step in achieving the desired performance improvements in real-time facial computing tasks. This section outlines the design and configuration of the net-engine IP core, which offloads Convolution and Max-pooling operations from the CPU to the Net Engine. The focus is on enhancing processing efficiency and achieving real-time performance by harnessing the parallel processing capabilities inherent in the FPGA architecture.

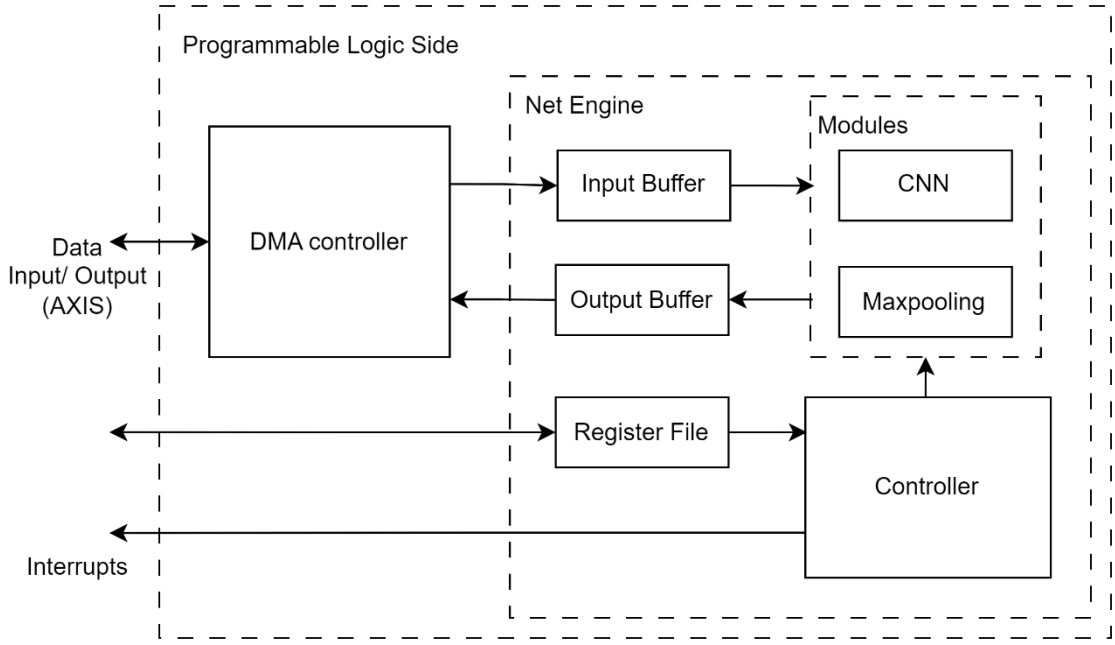


Figure 2: Programmable Logic (PL) side architecture

Figure 2 illustrates the architecture of the net engine IP. This architecture includes Direct

Memory Access (DMA) controller and net engine IP with CNN and Max-pooling modules. The register file is exiting to configure and read the status of the net engine IP. An input buffer is used to hold up to four data rows, while an output buffer stores the processed data before it is transferred back to the Processing System (PS) side. Both the width of the row and the data bit width can be customized during the synthesis stage of the net engine IP to suit specific requirements.

The Direct Memory Access (DMA) controller is a key component responsible for facilitating high-speed data transfer between the memory and the hardware IP. It ensures that input data is efficiently moved to the input buffer and that the processed output is transferred back to memory without CPU intervention.

3.4 Net-Engine IP

The net engine is the main part of the hardware design, which can perform Convolution and Max-pooling operations. This hardware component is integral to enhancing the performance of the real-time facial computing task that uses a Convolution Neural Network (CNN).

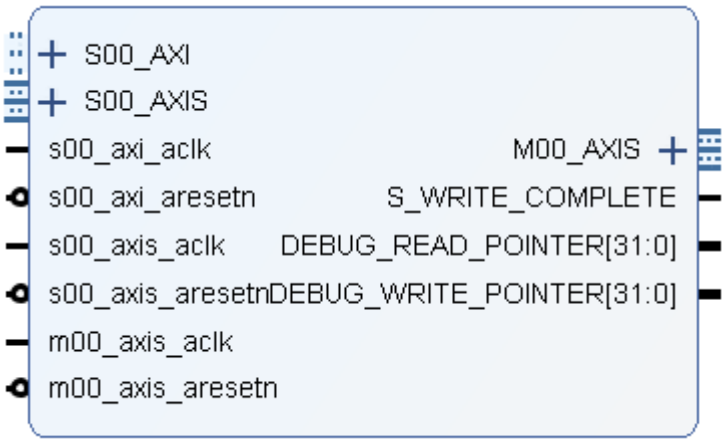


Figure 3: The Block Structure of the Net Engine IP

Figure 3 illustrates the block structure of the Net Engine IP with its output ports. The **input buffer** of the Net Engine is a crucial temporary storage area that holds incoming data before it is processed by the Convolution module. By providing a continuous data stream to the Convolution module, the input buffer ensures that data processing occurs seamlessly and without interruptions, thereby optimizing processing efficiency. The S00_AXIS and M00_AXIS communication ports are utilized by the Direct Memory Access (DMA) controller to transfer data from memory to the Net Engine and the Net Engine to Memory.

Once data has been processed, it is temporarily stored in the **output buffer**. This buffer holds the results from both the Convolution and Max-pooling modules with the operation selection register. The processed data is then transferred back to memory via the Direct Memory Access (DMA)

controller. The output buffer plays a significant role in decoupling the processing stages from memory transfer operations, which enhances the overall throughput and performance of the system. There is only a 5-clock cycle output latency concerning the input.

The **register file** is another critical component of the net engine. It contains essential status, configuration, and data registers that are necessary for the hardware IP's operation. These registers store various parameters such as the width of the image data, kernel data, bias values, and other operational controls that govern the functionality of the net engine.

The **Convolution** and **Max-Pooling** modules are the core processing units within the net engine. The convolution module performs convolution operations on the input data, while the Max-pooling module reduces the dimensionality of the data by selecting the maximum values within specific regions. Both modules are optimized for high-speed execution and can handle multiple convolutional filters and large input dimensions efficiently. They incorporate a 5-stage pipelining technique, which facilitates efficient processing at the hardware level.

Finally, the controller oversees the operation of the entire net engine IP. It manages the sequencing and synchronization of data flow between the different components, including the DMA controller, input/output buffers, Convolution module, and Max-pooling module. The controller ensures that all parts of the net engine work in unison, achieving optimal performance and seamless operation.

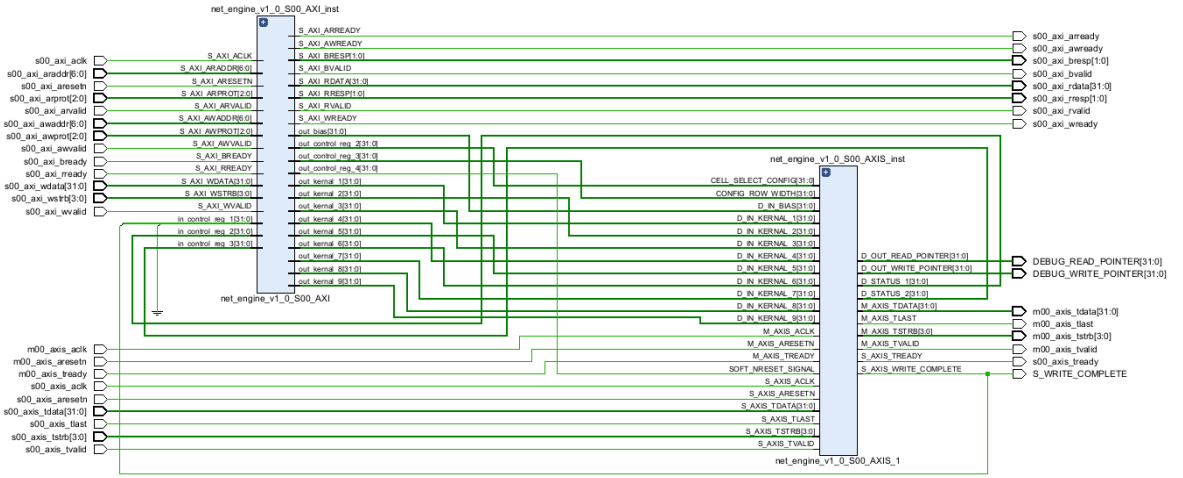


Figure 4: The schematic of the net engine IP

Figure 4 shows the schematic of the top-level net engine hardware schematic. It has 2 main modules called net_engine_v1_0_S00_AXI which is AXI lite interfacing module, that is used to configure and read a register in the net engine IP, and net_engine_v1_0_S00_AXIS which is AXI stream interface module is usually connecting to the Net Engine IP via the DMA controller. Therefore, the memory data transferring is happening efficiently.

3.4.1 Input/ Output Ports and Interrupt

The net-engine IP core interfaces with the rest of the system through several critical ports, designed to facilitate high-speed data transfer and efficient processing:

- **AXI Lite Interface**: This interface allows for the configuration of the net-engine IP by writing configuration parameters and control signals. It is directly connected to the AXI Interconnection IP, enabling seamless communication between the IP core and other system components.
- **AXI Stream Slave Port**: This port is used to stream input data, such as image pixels, into the net-engine IP. It supports high-speed data transfers from memory to the processing units, ensuring that the input data is processed without unnecessary delays.
- **AXI Stream Master Port**: The output of the IP is returned to memory via the AXI Stream Master Port. This port works in conjunction with the DMA controller, allowing for high-bandwidth data transfers that are crucial for maintaining system performance.
- **Write Complete Interrupt**: This interrupt is triggered when the processing of a single row is completed. It signals the Processing System (PS) to transfer the next data row to the Net Engine.
- **Receive Complete Interrupt:** This interrupt is generated by the Direct Memory Access (DMA) controller once all processed data has been received by the Processing System. It indicates that the image frame processing is complete and can be used to determine the completion status of the processing.

3.4.2 Net-Engine Register File

The net-engine IP core is configured and controlled through a series of 20 registers, which are categorized as follows:

- **Configuration Registers (5 Registers)**: These registers are used to configure key operational parameters of the net-engine IP, such as activation functions, kernel sizes, strides, and padding. By setting these parameters, the IP core can be tailored to handle various convolution tasks effectively.
- **Kernel and Bias Registers (10 Registers)**: These registers store the kernel weights and bias values required for the convolution operations. They facilitate the precise loading of convolutional parameters into the processing units, ensuring accurate and efficient computation.

- **Status Registers (5 Registers)**: These registers provide real-time status updates, including processing progress and completion flags, which are essential for monitoring the IP core's operations and ensuring that tasks are completed within the required timeframes

Below Table 2 provides a detailed map of the net engine's registers, organized into three main categories, there are Status, Configuration, and kernel and bias Registers.

Register Name	Description	Offset
Status Registers		
NET_ENGINE_STATUS_REG_1	Status information	0x00
NET_ENGINE_STATUS_REG_2	Status information	0x04
NET_ENGINE_STATUS_REG_3	Status information	0x08
NET_ENGINE_STATUS_REG_4	Status information	0x0C
NET_ENGINE_STATUS_REG_5	Status information	0x10
NET_ENGINE_STATUS_REG_6	Status information	0x14
Configuration Registers		
NET_ENGINE_CONFIG_REG_1	Select convolution / max-pooling Operation	0x18
NET_ENGINE_CONFIG_REG_2	Input Row Length	0x1C
NET_ENGINE_CONFIG_REG_3	Net Engine Enable/Disable	0x20
NET_ENGINE_CONFIG_REG_4	Reserved	0x24
Kernel and Bias Registers		
NET_ENGINE_BIAS_REG	Bias values for convolution operations	0x28
NET_ENGINE_KERNEL_REG_1	Kernel weights for convolution operations	0x2C
NET_ENGINE_KERNEL_REG_2	Kernel weights for convolution operations	0x30
NET_ENGINE_KERNEL_REG_3	Kernel weights for convolution operations	0x34
NET_ENGINE_KERNEL_REG_4	Kernel weights for convolution operations	0x38
NET_ENGINE_KERNEL_REG_5	Kernel weights for convolution operations	0x3C
NET_ENGINE_KERNEL_REG_6	Kernel weights for convolution operations	0x40
NET_ENGINE_KERNEL_REG_7	Kernel weights for convolution operations	0x44
NET_ENGINE_KERNEL_REG_8	Kernel weights for convolution operations	0x48
NET_ENGINE_KERNEL_REG_9	Kernel weights for convolution operations	0x4C

Table 2: The Register File of the Net Engine IP

3.4.3 Floating-Point Operations

The net-engine IP core is equipped with 32-bit floating-point units for addition and multiplication, which are vital for performing the precise computations required by convolution and max-pooling tasks. The inclusion of these floating-point operations ensures that the system can handle the complex mathematical operations involved in convolution and max-pooling with the accuracy needed for reliable performance.

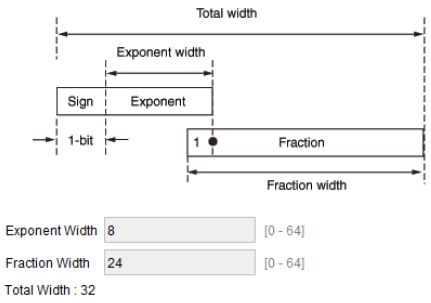


Figure 5: The structure of the 32-float representation

The above Figure 5 shows the components of the 32-bit floating point value. It includes the 8 bits of exponent width and 24 bits of fraction width to represent a float value.

3.4.4 Convolution module Implementation

The convolutional module implemented in this project is implemented to perform the convolution operation commonly used in the Convolutional Neural Network (CNN). The module is parameterized by the data width and the kernel size, allowing it to be adapted for various precision and filter sizes. In this particular implementation, the module is configured with a 32-bit data width and a 3x3 kernel size, making it suitable for high-precision computations typically required in deep learning applications.

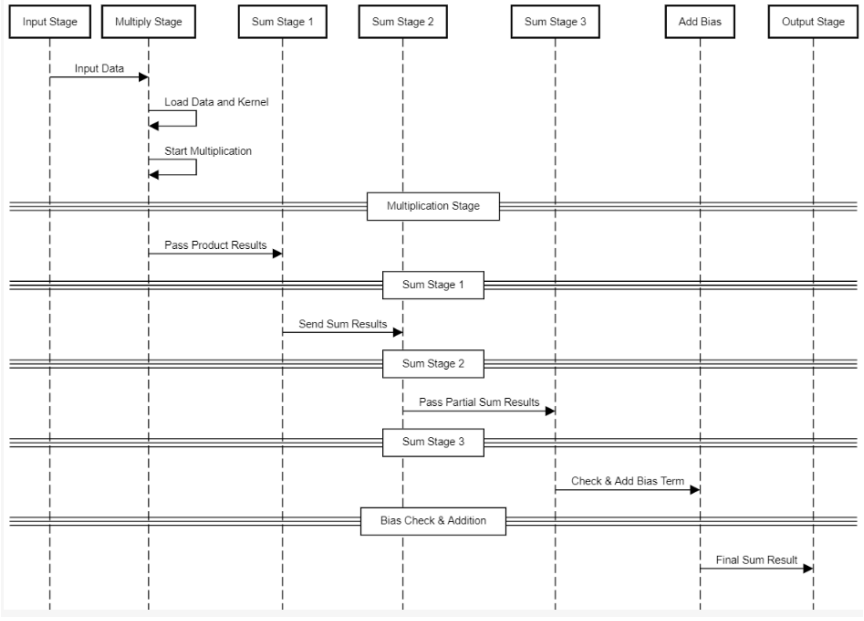


Figure 6: Convolutional Pipeline Sequence Diagram

The Convolution module implemented in the described sequence diagram performs convolution operations on input data using a fixed-size kernel. The module processes the input data through five stages, including multiplication, summation, and bias addition, with specific handling for cases where the bias is zero.

The sequence diagram for the convolution cell describes the operation of its unit implemented in a pipelined architecture. Here's a breakdown of each stage:

1. **Net Engine Configuration for Convolution**:
    - **Data Inputs**: The convolution module accepts 9 input data values, each representing a pixel or feature from the input channel.
    - **Kernel Inputs**: It also receives 9 kernel weights, which are used to perform the convolution operation. The kernel is a 3x3 matrix in this implementation.
    - **Bias Input**: The bias input is used at the last computing stage.
2. **Multiplication Stage**:
    - **Component**: float32_multiply modules.
    - **Function**: Each data input is multiplied by its corresponding kernel weight. This is done through 9 separate multiplication operations, one for each pair of data and kernel values.
    - **Outputs**: The results of these multiplications are stored in internal registers.
3. **Summation Stage**:
    - **Component**: float32_add modules.
    - **Function**: The results of the multiplications are summed to produce intermediate values. The summation is carried out in multiple stages:
        - **Stage 1**: The products from the first 8 multiplications are grouped and summed in pairs, resulting in 5 intermediate sums (including a zero addition for the last odd element if required).
        - **Stage 2**: The results from Stage 1 are further summed in pairs to produce 3 intermediate sums (again, including a zero addition if needed).
        - **Stage 3**: The results from Stage 2 are summed to produce a single value.
4. **Bias Addition**:
    - **Component**: float32_add module for bias.
    - **Function**: After obtaining the final sum from Stage 3, the bias value is added. If the bias value is zero, it is skipped in the addition process.
    - **Handling**: The module includes logic to conditionally add the bias only if it is non-zero, ensuring efficient computation.

5. **Output Stage**:
    - **Final Output**: The result after bias addition is assigned to the output register.
    - **Output Valid Signal**: The validity of the output data is controlled by a flag, which indicates when the output data is ready to be used.
6. **Control Signals**:
    - **Data Valid Flags**: Various control signals are used to manage the validity of data at different stages of processing, ensuring that operations are carried out sequentially and correctly.

3.4.5 Max-pooling Implementation

The max pooling cell performs a series of comparisons to determine the maximum value from a 3x3 grid of data inputs. The operation is divided into three stages, each progressively reducing the number of values compared until a single maximum value is obtained. This final maximum value is then outputted, representing the result of the max pooling operation.

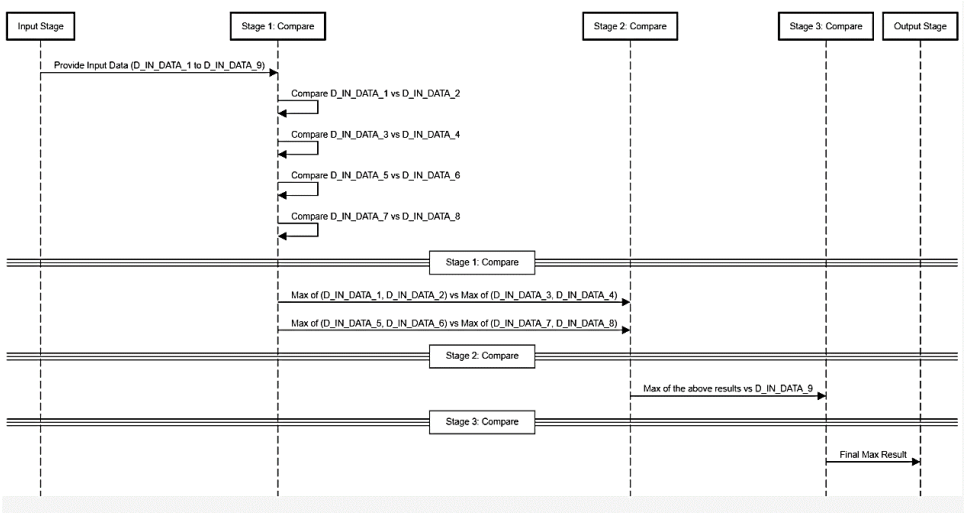


Figure 7: Max-pooling Cell Pipeline Sequence Diagram

The sequence diagram for the max pooling cell describes the operation of a max pooling unit implemented in a pipelined architecture. Here's a breakdown of each stage:

1. **Input Stage**:
    - **Provide Input Data**: The initial stage involves feeding the input data into the pipeline. This data consists of 9 values (D_IN_DATA_1 to D_IN_DATA_9) that need to be processed.
2. **Stage 1: Compare**:
    - **Compare D_IN_DATA_1 vs D_IN_DATA_2**: The first comparison occurs between D_IN_DATA_1 and D_IN_DATA_2, where the larger of the two values is selected.
    - **Compare D_IN_DATA_3 vs D_IN_DATA_4**: Second, the comparison is made between D_IN_DATA_3 and D_IN_DATA_4, selecting the larger value.
    - **Compare D_IN_DATA_5 vs D_IN_DATA_6**: Third, the comparison is made between D_IN_DATA_5 and D_IN_DATA_6, selecting the larger value.
    - **Compare D_IN_DATA_7 vs D_IN_DATA_8**: Finally, the comparison is made between D_IN_DATA_7 and D_IN_DATA_8, selecting the larger value.

    These comparisons result in four intermediate maximum values:

    - max_1_2 (Max of D_IN_DATA_1 and D_IN_DATA_2)
    - max_3_4 (Max of D_IN_DATA_3 and D_IN_DATA_4)
    - max_5_6 (Max of D_IN_DATA_5 and D_IN_DATA_6)
    - max_7_8 (Max of D_IN_DATA_7 and D_IN_DATA_8)
3. **Stage 2: Compare**:
    - **Max of (D_IN_DATA_1, D_IN_DATA_2) vs Max of (D_IN_DATA_3, D_IN_DATA_4)**: The maximum value obtained from the comparison of D_IN_DATA_1 and D_IN_DATA_2 is compared with the maximum value obtained from D_IN_DATA_3 and D_IN_DATA_4.
    - **Max of (D_IN_DATA_5, D_IN_DATA_6) vs Max of (D_IN_DATA_7, D_IN_DATA_8)**: Similarly, the maximum values from the comparisons of D_IN_DATA_5 and D_IN_DATA_6 are compared with those of D_IN_DATA_7 and D_IN_DATA_8.

    This stage reduces the four intermediate maximum values into two new maximum values:

    - max_1_2_3_4 (Max of max_1_2 and max_3_4)
    - max_5_6_7_8 (Max of max_5_6 and max_7_8)

4. **Stage 3: Compare**:
    - **Max of the above results vs D_IN_DATA_9**: The maximum values from Stage 2 (max_1_2_3_4 and max_5_6_7_8) are compared with the remaining data value, D_IN_DATA_9.

    This stage produces the final maximum value by comparing the results from Stage 2 with D_IN_DATA_9:

    - max_pool (Final maximum value after comparing max_1_2_3_4, max_5_6_7_8, and D_IN_DATA_9)
5. **Output Stage**:
    - **Final Max Result**: The output stage provides the final result of the max pooling operation. The maximum value obtained from all comparisons is sent to the output as C_OUT_DATA.

3.5 Processing System (PS) Implementation

The programmable Logic design cannot function without the support of the processing system, Therefore, The net engine driver is crucial to facilitate the manipulation of incoming data and achieve the desired output using the hardware. The processing system (PS) includes the Net Engine driver implementation, convolution neural network implementation, and proposed network (Pnet Model in Multi-task Cascaded Convolutional Network) implementation. each of these software components plays a crucial role in interacting with the hardware IP or other software to enhance overall performance and achieve optimal results. By working in conjunction with the hardware, these components ensure efficient data processing and system performance.

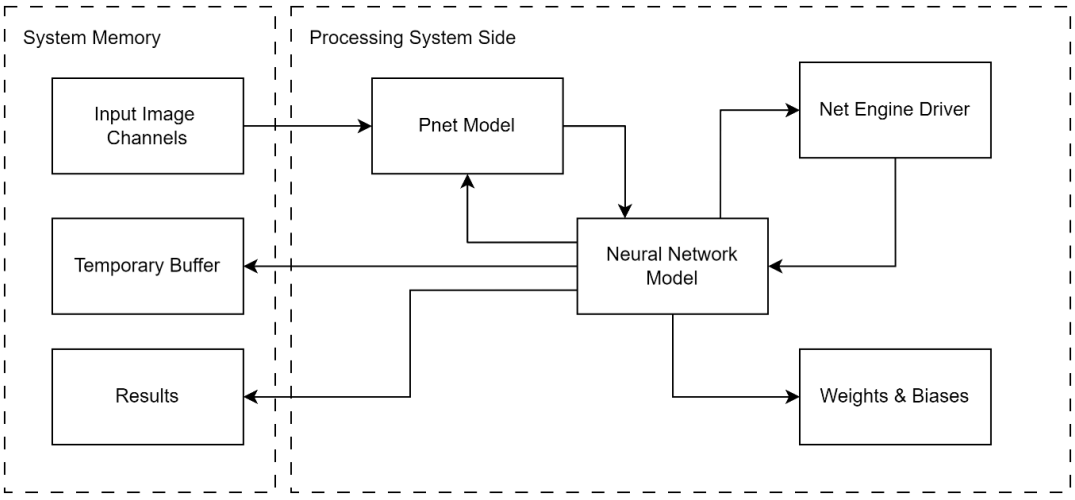


Figure 8: The High-Level design of the Processing System Implementation

Figure 8 illustrates the high-level design of the processing system. In the system's memory, there is a static buffer allocated for three input channels called red, green, and blue channels. Additionally, a temporary memory pool is used to hold processed data for reprocessing by subsequent layers. The neural network model leverages the Net Engine driver to execute convolution and max-pooling operations. The static memory stores weights and bias data required for these operations. When processing begins with the PNet model, the neural network component retrieves the weights and biases from static memory and transfers them to the Net Engine to configure the net engine. After the configuration, the channel data from the neural network is sent to the Net Engine IP. After all layers have been processed, the output results are stored for further use or analysis.

3.5.1 Net Engine Driver Implementation

The Net Engine driver is crucial for facilitating communication between the processing system and the Net Engine hardware implementation. It handles the configuration of the Net Engine and manages data transfer between memory and the IP using the Direct Memory Access (DMA) controller. Additionally, the driver is responsible for receiving processed data from the Net Engine IP and storing it at the specified memory location.

Two interrupts are triggered from the Programmable Logic (PL) side: the Write Complete Interrupt and the Receive Complete Interrupt. Before utilizing the Net Engine IP, it must be properly enabled and configured. The width of the image is specified as the row length in the Net Engine, which requires setting the **NET_ENGINE_CONFIG_REG_2** register accordingly. Both interrupts and the DMA controller must be enabled for proper operation.

For configuration, set the **NET_ENGINE_CONFIG_REG_1** to 0x00000001 to initiate a convolution operation, or to 0x00000000 for max-pooling. Note that max-pooling operations are restricted to a three-by-three matrix. For convolution operations, kernel values in 32-bit float format must be configured in the **NET_ENGINE_KERNEL_REG_1** through **NET_ENGINE_KERNEL_REG_9** registers. If bias values are used, they should be set in the **NET_ENGINE_BIAS_REG** register. This setup ensures that the Net Engine operates correctly for both convolution and max-pooling tasks. Before sending the data to the net engine, the **NET_ENGINE_CONFIG_REG_3** register must be enabled.

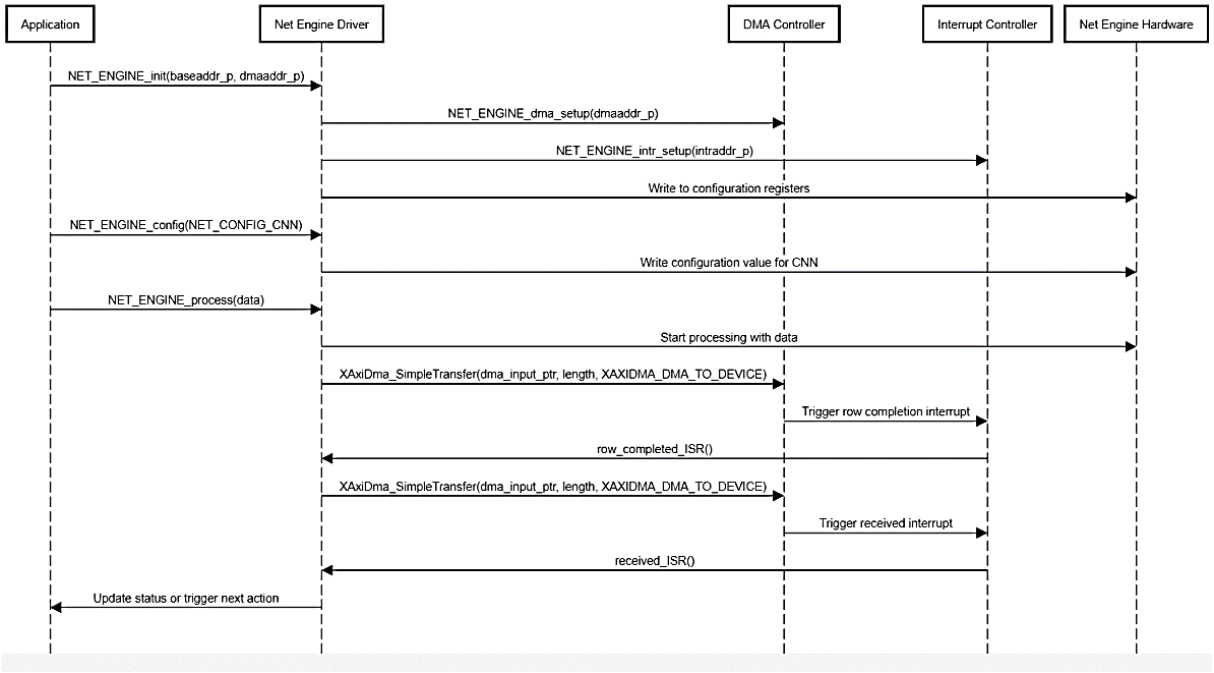


Figure 9: The sequence diagram of the Net Engine Driver Implementation

The sequence diagram for the Net Engine driver details the comprehensive process involved in the initialization and utilization of the Net Engine hardware for convolution-based image processing and convolution neural network applications. Initially, the application invokes the **NET_ENGINE_init** function within the Net Engine Driver to establish the base address and net engine's DMA base address required for hardware operation. Subsequently, the Net Engine Driver configures the DMA Controller by executing the **NET_ENGINE_dma_setup** function, which facilitates the setup of Direct Memory Access (DMA) for efficient data transfer. Following this, the driver configures the Interrupt Controller using the **NET_ENGINE_intr_setup** function, ensuring proper interrupt handling for the Net Engine. The final step in initialization involves the driver writing to the configuration registers of the Net Engine Hardware to complete the setup process.

After completing the initialization process, the Net Engine is configured for Convolutional Neural Network (CNN) processing. This is achieved when the application sends a configuration command to the Net Engine Driver via the **NET_ENGINE_config** function. The driver subsequently writes the appropriate configuration values to the Net Engine Hardware to prepare it for process CNN-based processing.

For image data processing, the application calls the **NET_ENGINE_process** function on the Net Engine Driver, transferring the image data for processing. The Net Engine Driver initiates the processing sequence by directing the Net Engine Hardware to commence operations. Concurrently, the driver triggers a DMA transfer on the Controller, which facilitates the transfer of image data to the Net Engine for processing.

As the processing progresses, the DMA Controller generates a write completion interrupt upon finishing the transfer of a single data row, which is conveyed to the Interrupt Controller. The Interrupt Controller then notifies the Net Engine Driver through the **row_completed_ISR()**, prompting the driver to issue another DMA transfer command to the DMA Controller for the subsequent image row. Additionally, upon completion of the entire processing, the DMA Controller triggers a received interrupt. This interrupt is communicated to the Net Engine Driver via the **received_ISR()** function, enabling the driver to update the application on the processing status or initiate any required follow-up actions.

This sequence diagram provides a detailed representation of the interactions and procedures involved in the Net Engine driver's operation, illustrating the systematic approach to managing image data processing through initialization, configuration, data handling, and interrupt management.

3.5.2 Neural Network Implementation

The neural network module consists of several software components, such as layers, channels, and kernels. The weights and bias of each layer should be implemented on the PS as in side the static memory location before initiating the processing.

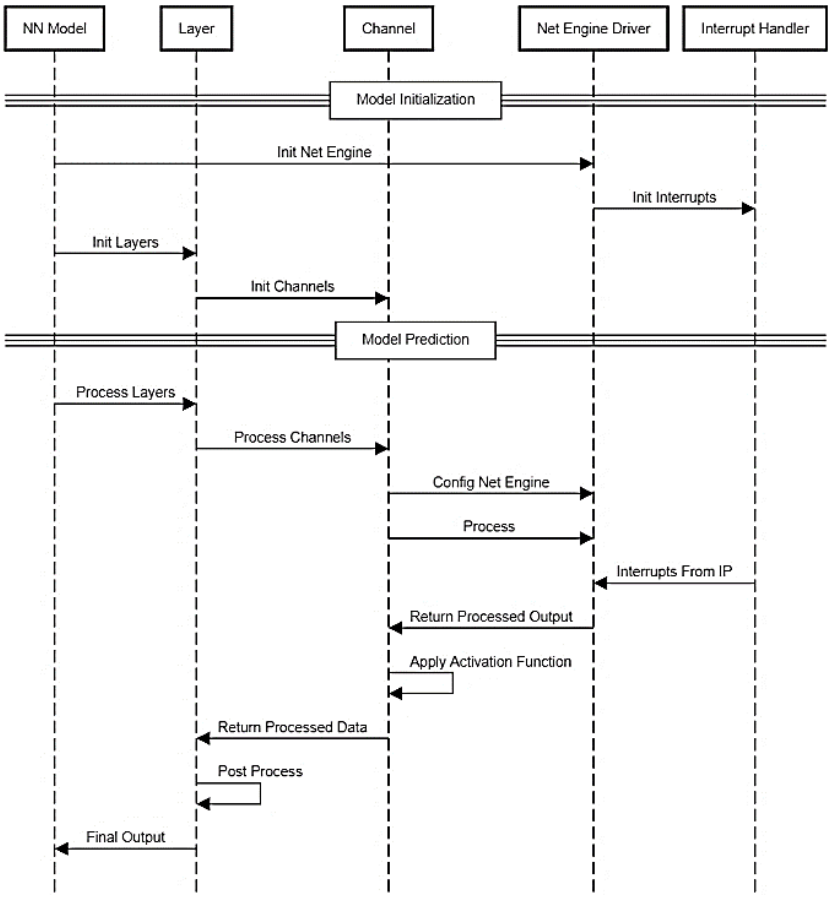


Figure 10: The sequence diagram of the Neural Network Component

In the neural network processing workflow with interrupt handling, the sequence diagram outlines the interactions between various components during model initialization and prediction phases. Initially, the Neural Network (NN) Model interacts with the Net Engine Driver to initialize the engine with their hardware addresses, followed by setting up interrupts with the Interrupt Handler. The NN Model then proceeds to initialize the layers, which in turn initialize their respective channels.

During model prediction, the NN Model triggers the Layer processing, which manipulates the processing of channels. Channels are responsible for configuring and processing data with the Net Engine Driver. An important aspect of this process is the handling of interrupts: the Interrupt Handler communicates with the Net Engine Driver to manage interruptions from the processing unit. This ensures that data processing is appropriately paused and resumed as needed.

After processing, the Net Engine Driver returns the processed output to the channels, where activation functions are applied. The processed data is then passed back to the Layer, which performs any necessary post-processing before delivering the final output to the NN Model. This sequence ensures efficient handling of neural network operations, with robust mechanisms for managing interrupts and integrating various processing stages.

3.6 Memory Map

The memory map for the neural network (NN) processing system is defined through a series of memory regions in PS7 DDR memory, each allocated for specific tasks such as input channels, receive buffers, and memory pools. The NN_INPUT_SIZE is set to 0xA000, which corresponds to the size of each input channel. The memory addresses for the red, green, and blue input channels are defined sequentially, starting from MEM_BASE_ADDR + 0x00300000 for the red channel. The green and blue channels follow each offset by the size of the input (0xA000), ensuring that the input data for each color channel is stored in contiguous but separate memory regions.

Memory Region	Size (bytes)
NN_INPUT_RED_CHANNEL	0xA000
NN_INPUT_GREEN_CHANNEL	0xA000
NN_INPUT_BLUE_CHANNEL	0xA000
NN_RECEIVE_MEM_BASE	0xA000
NN_MEM_POOL_1_BASE	0x5DCC8
NN_MEM_POOL_2_BASE	0x5DCC8
NN_MEM_POOL_3_BASE	0x5DCC8

Table 3: The Memory Map of the Convolution Neural Network

The receive memory region is defined starting from MEM_BASE_ADDR + 0x00400000, with a

length of 0xA000, allowing space for receiving processed data. The end of this region is marked by NN_RECEIVE_MEM_HIGH, which is the sum of the base address and the length.

Three memory pools are allocated for different stages or components of the NN processing. The first pool begins at MEM_BASE_ADDR + 0x00500000 and has a length of 0x5DCC8. The second and third pools are allocated immediately after the preceding pool, maintaining the same length (0x0005DCC8) for consistency. This sequential allocation ensures that each memory pool is distinct yet contiguous, allowing efficient memory management during NN operations. These defined memory regions facilitate organized and predictable memory access, which is crucial for performance.

3.7 Proposed Network (Pnet in MTCNN)

In this project, I have used the proposed network (Pnet) model in the Multi-Task Cascaded Convolutional Neural Network (MTCNN) to validate the results of the implemented system. This proposed network model can be utilized to detect a face in the image.

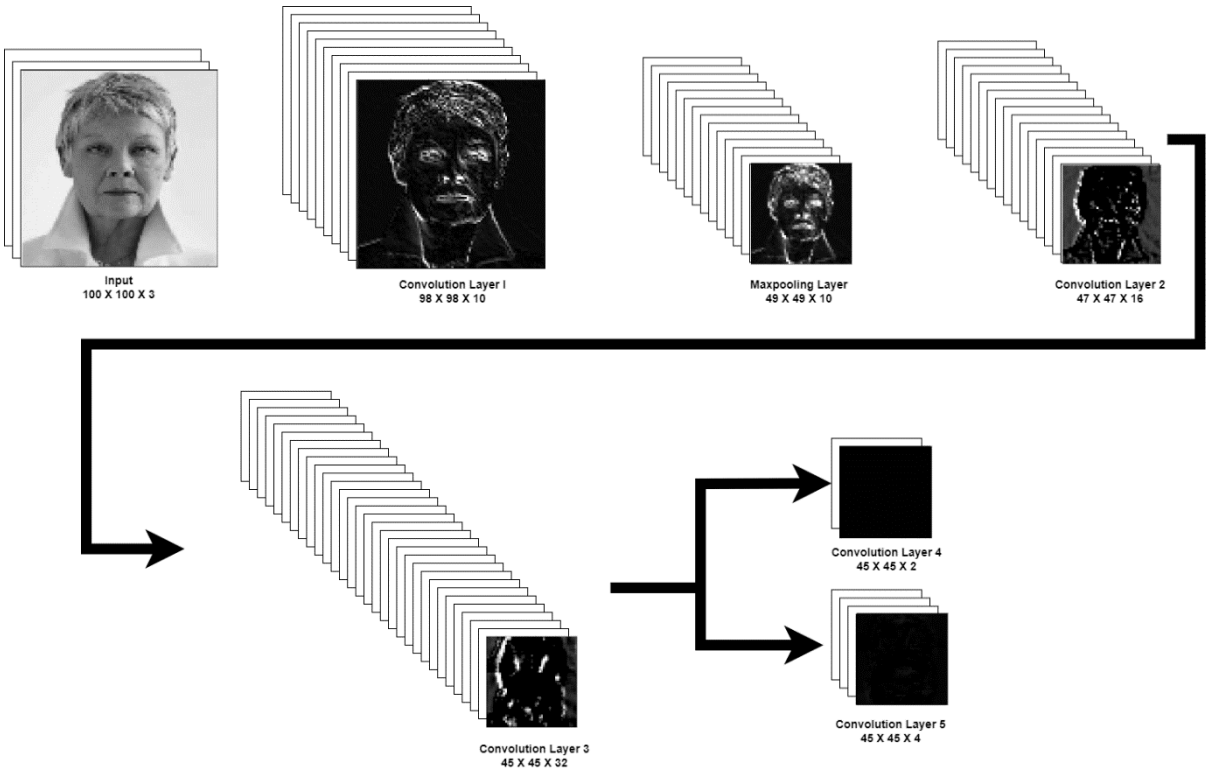


Figure 11: The Pnet Model Architecture

The Proposal Network (P-Net) is the initial stage in the Multi-task Cascaded Convolutional Networks (MTCNN) architecture, a prominent framework used for face detection. The primary role of P-Net is to generate candidate facial bounding from the input image. This network operates as a convolutional neural network (CNN) designed to rapidly identify potential faces by proposing regions in an image that are likely to contain faces.

P-Net is structured to achieve two main tasks: bounding box regression and facial landmark localization. It begins by processing the input image through a series of convolutional layers, which extract and enhance features relevant to face detection. The network employs a sliding window approach, where different portions of the image are analyzed to predict face candidates. Each window is classified as either containing a face or not, while simultaneously refining the bounding boxes to better fit the detected faces. The network architecture of P-Net is designed to be computationally efficient, making it suitable for handling the high volume of candidate regions generated during the face detection process. It uses a combination of convolutional and pooling layers to progressively reduce the spatial dimensions of the image while increasing the depth of feature maps. This hierarchical processing allows the network to capture both global and local features of potential faces.

In terms of implementation, P-Net employs a small-scale CNN architecture with multiple convolutional layers followed by fully connected layers. It is trained on a large dataset of labeled face images, using a combination of classification and regression loss functions. The training process involves optimizing these loss functions to improve the accuracy of face classification and bounding box prediction.

The outputs of the P-Net are then passed to the next stages in the MTCNN framework R-Net (Refine Network) and O-Net (Output Network)—which further refine the face candidates and perform final face detection and alignment. This cascading approach allows MTCNN to achieve high detection accuracy and robustness across varying face sizes and orientations.

Overall, P-Net plays a crucial role in the MTCNN framework by providing the initial face proposals and landmark predictions, setting the stage for subsequent processing by the more advanced networks in the cascade. Its efficiency and effectiveness in generating face candidates are essential for the overall performance of the MTCNN face detection system.

3.8 Experimental Method

The objective of this experiment is to evaluate and compare the performance of hardware-based and software-based implementations of convolution and max-pooling algorithms. The evaluation focuses on accuracy and processing time to determine the effectiveness of each approach.

The experimental results will be analyzed to determine the accuracy and efficiency of the hardware-based and software-based solutions. The comparison focused on identifying the trade-offs between accuracy and processing time, as well as assessing the advantages of hardware acceleration in real-time image processing applications.

The following sections provide the software implementations for the convolution and max-pooling algorithms:

3.8.1 Pseudocode for Convolution Operation

1. **Loop Through Image:**
    - For each row index i from 0 to channel->height - 3:
        - For each column index j from 0 to channel->width - 3:
            - Initialize sum to 0.0f.
2. **Apply Convolution Kernel:**
    - For each row offset m from 0 to 2:
        - For each column offset n from 0 to 2:
            - Calculate the position:
                - x ← j + n
                - y ← i + m
            - Check if x and y are within the bounds of the image:
                - If true:
                    - Update the sum by adding the product of:
                        - channel->input_ptr[y * channel->width + x]
                        - placeholder for the kernel value
3. **Store Result:**
    - Set output to sum.

3.8.2 Pseudocode for Max-pooling Operation

1. Set Dimensions and Parameters:
    - Set in_height, in_width, out_height, out_width, stride, and size from the channel data.
2. Process Each Channel:
    - For each position in the output feature map:
        - Set max_value to -FLT_MAX.
        - Calculate start_y and start_x for the pooling window.
        - For each position within the pooling window:
            - Check if the position is within bounds.
            - Update max_value if the current value is greater.
        - Store max_value in the output feature map.
    - Move to the next input_channel and output_channel.
3. Finish:
    - Return 0.

3.8.3 Time Measuring Method

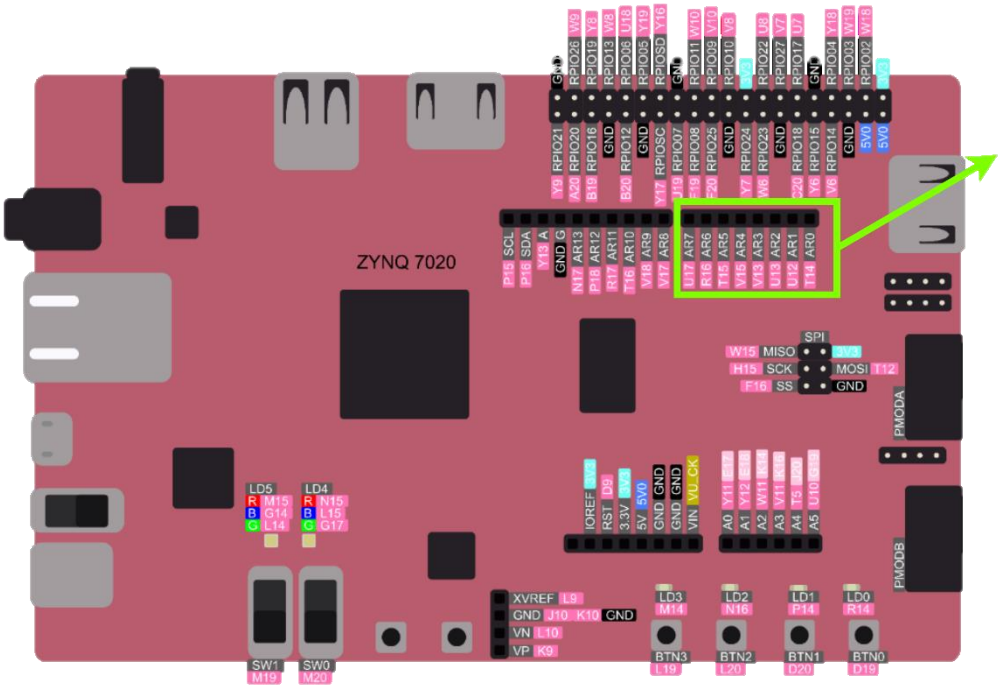


Figure 12: The Timing Measurement Setup

The experimental setup is designed to assess the time utilization of a system implemented on the PYNQ-Z2 development board. On the Programmable Logic (PL) side, the system comprises a net engine, DMA controller, and PL-to-PS interrupts. Meanwhile, the Processing System (PS) side hosts the Net Engine driver and the P-Net model implementation, running on the 0th core of the dual Cortex-A9 processors available on the PYNQ-Z2 board.

To measure time intervals, the AR0 to AR6 GPIO ports are utilized to provide external outputs. These outputs are captured using a Logic Analyzer, specifically the Saleae Logic 8 Device, which operates with a 100 MS/s sampling rate. This setup ensures accurate measurement of time differences between various system components.

For the measurement process, the AXI GPIO IP in Vivado is configured to enable 'Time Measuring GPIOs' on the PL side. Additionally, the 'time_measure' software component is implemented on the Processing System side to trigger these time-measuring GPIOs, facilitating precise time tracking and analysis of the system's performance.

Chapter 4: Results Evaluation

This chapter presents the findings of the research, including numerical and visual evaluations. The results are categorized into three main sub-topics: (1) Comparison of Net Engine vs. Software Output, (2) Accuracy of the Pnet Model with Software and Net Engine Implementations, and (3) Processing Time Differences Between Software-Based and Hardware-Supported Implementations.

4.1 First CNN Layer Output Comparison

The performance comparison between the Net Engine IP and the software-based implementation was rigorously assessed through both statistical and visual methods. This section provides a comprehensive overview of the numerical comparisons, including correlation coefficients, Chi-Square statistics, and histogram intersections, along with visual representations of the results.

4.1.1 Numerical Comparison:

The numerical evaluation of the output from the Net Engine versus the software-based implementation is summarized in Table 4. This comparison is based on three statistical measures: Correlation Coefficient, Chi-Square Statistic, and Histogram Intersection. These metrics provide insights into the similarity between the outputs generated by the two methods.

Channel	Correlation Coefficient	Chi-Square Statistic	Histogram Intersection
Channel 1	1.000	0.099	1.000
Channel 2	1.000	0.032	0.999
Channel 3	1.000	0.012	1.000
Channel 4	1.000	0.011	1.000
Channel 5	1.000	0.004	1.000
Channel 6	1.000	0.078	0.999
Channel 7	1.000	0.074	0.999
Channel 8	1.000	0.029	0.999
Channel 9	1.000	0.039	1.000
Channel 10	1.000	0.039	0.999

Table 4: The Statistical Comparison of Net Engine and Software-Based Output

4.1.2 Visual Comparison:

Both the images processed by the net engine IP and the software-based implementation were visually inspected. As observed, the images are nearly identical, with no discernible differences to the naked eye, indicating that the net engine IP is functioning as intended. A detailed histogram analysis was conducted for each output, providing a quantitative comparison of pixel intensity distributions. The histograms reveal that the distribution of pixel intensities in the outputs from the net engine IP

closely matches those from the software implementation. The congruence in the histograms further validates the accuracy and reliability of the net engine IP.

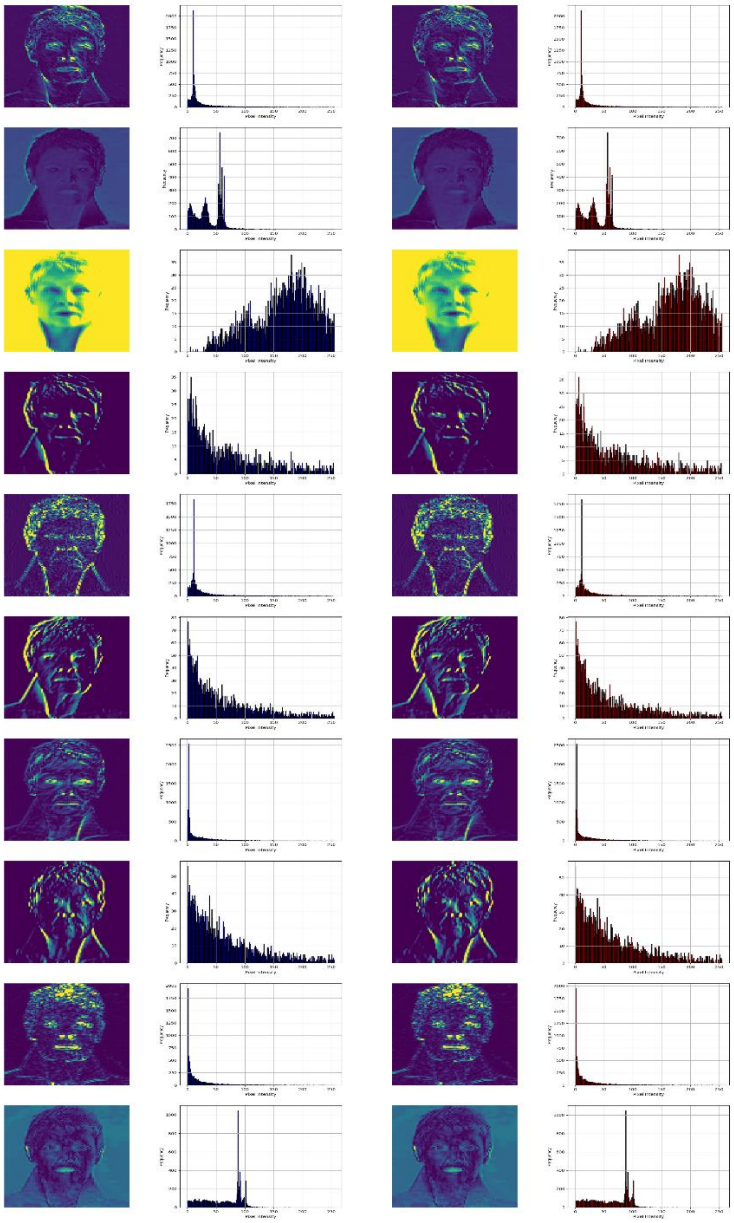


Figure 13: The Comparison of the First CNN Layer Output Channel and their histogram

Based on the evidence presented in the images and histograms, the net engine IP has been demonstrated to produce outputs that are not only visually indistinguishable from the software-based outputs but also quantitatively comparable. This confirms that the net engine IP achieves accurate processing and can be reliably used in place of the software-based implementation.

4.2 Final Layer Output Comparison

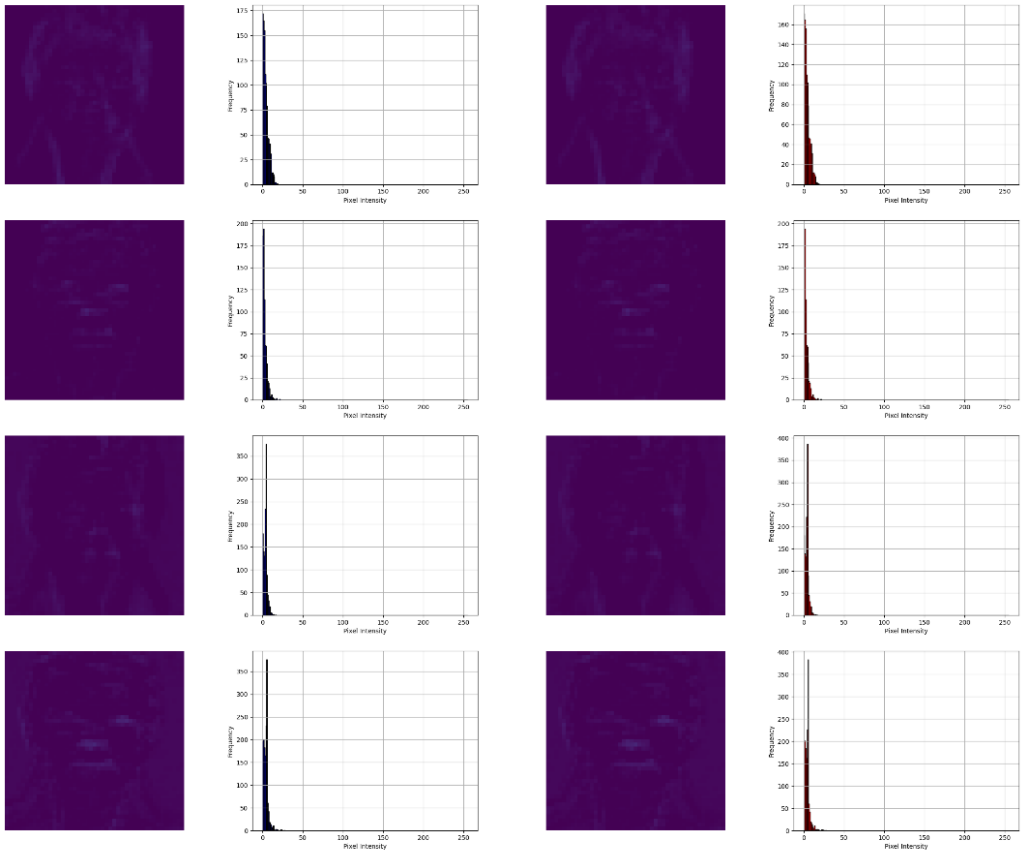


Figure 14; The Comparison of Final Layer Output Channel and their Histogram

The results indicate a remarkable level of similarity between the datasets compared. The Correlation Coefficient consistently shows a perfect positive linear relationship between the datasets, with values of 1.0 across most channels, and values very close to 1.0 (0.9999999999999998) for a few channels. This signifies that the datasets move together perfectly, demonstrating an exact alignment in their trends and patterns.

The Histogram Intersection value of 1.0 across all channels confirms a perfect overlap between the histograms of the datasets. This result highlights that the frequency distributions of pixel intensities (or other measured values) are identical across the two datasets. Additionally, the Additional Correlation values, which are all 1.0, corroborate the perfect positive correlation observed with the primary Correlation Coefficient. This consistency reinforces the high similarity between the datasets.

In summary, the results collectively validate that the net engine's output is highly accurate and consistent with the software-based output, reflecting a high degree of agreement between the two methods.

4.3 Model Accuracy Comparison

we assess the accuracy of the implemented model across different layers and channels compared to the software-based output. The accuracy scores are tabulated and analyzed for each layer and channel to evaluate how closely the Net Engine IP implementation aligns with the software-based implementation.

The calculation of accuracy score is calculated by using the *accuracy_score* function in the sklearn library. The accuracy score calculation has been done by a net engine used process concerning the software-based implementation. And this model consists of 6 layers as mentioned in the previous section.

Accuracy Score of the Hardware Implementation Respect to the Software						
	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5 (Out 1)	Layer 6 (Out 2)
Channel 1	0.9898	1.0000	0.9574	1.0000	1.0000	0.9778
Channel 2	0.9898	1.0000	1.0000	1.0000	0.9333	1.0000
Channel 3	0.9898	0.9796	1.0000	0.9556	-	1.0000
Channel 4	0.9592	0.9796	1.0000	0.9778	-	0.9778
Channel 5	0.9898	1.0000	0.9787	1.0000	-	-
Channel 6	0.9490	0.9592	1.0000	1.0000	-	-
Channel 7	1.0000	1.0000	0.9787	1.0000	-	-
Channel 8	0.9694	1.0000	0.9787	1.0000	-	-
Channel 9	1.0000	1.0000	1.0000	1.0000	-	-
Channel 10	0.9796	0.9796	1.0000	1.0000	-	-
Channel 11	-	-	0.9787	1.0000	-	-
Channel 12	-	-	1.0000	1.0000	-	-
Channel 13	-	-	0.9787	1.0000	-	-
Channel 14	-	-	1.0000	1.0000	-	-
Channel 15	-	-	1.0000	0.9778	-	-
Channel 16	-	-	1.0000	1.0000	-	-
Channel 17	-	-	-	0.9778	-	-
Channel 18	-	-	-	1.0000	-	-
Channel 19	-	-	-	1.0000	-	-
Channel 20	-	-	-	0.9778	-	-
Channel 21	-	-	-	1.0000	-	-
Channel 22	-	-	-	1.0000	-	-
Channel 23	-	-	-	1.0000	-	-
Channel 24	-	-	-	1.0000	-	-
Channel 25	-	-	-	0.9556	-	-
Channel 26	-	-	-	1.0000	-	-
Channel 27	-	-	-	1.0000	-	-
Channel 28	-	-	-	1.0000	-	-

Channel 29	-	-	-	1.0000	-	-
Channel 30	-	-	-	0.9778	-	-
Channel 31	-	-	-	1.0000	-	-
Channel 32	-	-	-	1.0000	-	-

Table 5: The Model Output Accuracy Score concerning the Software-based Output

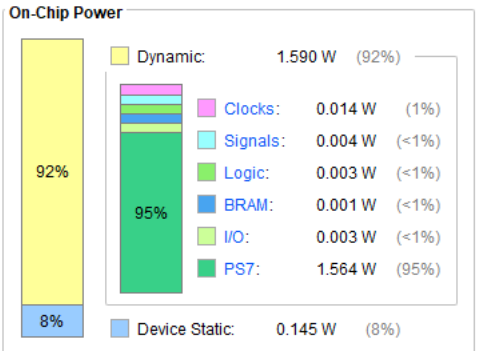
The analysis of the accuracy scores reveals that the Net Engine IP implementation demonstrates a high degree of alignment with the software-based implementation across various layers and channels. Most channels achieve near-perfect accuracy, with many reaching a score of 1.000 across multiple layers, indicating that the Net Engine IP replicates the software output with exceptional precision. Channels such as Channel 1 and Channel 2 consistently show high accuracy in nearly all layers, suggesting a robust performance across different processing stages. Although some channels exhibit variability in accuracy, with a few not having scores for certain layers, the overall trend highlights a strong correlation between the hardware and software implementations. The observed accuracy underscores the reliability and effectiveness of the Net Engine IP in delivering results comparable to the software-based approach. This consistency in high accuracy across various layers and channels affirms the success of the Net Engine IP in accurately replicating the computational results of the software implementation.

4.4 The estimated Power utilization of PL

The estimated power consumption of the programmable side is mentioned below. The estimated power utilization of the PL side has been taken using vivado software.

Power Consumption

Dynamic Power	Clocks	0.014w
	Signals	0.004w
	Logic	0.003w
	BRAM	0.001w
	I/O	0.003w
	PS7	1.564w
Device Static Power		0.145w
Total Power		1.735w



The power consumption analysis of the Net Engine IP implementation reveals that the majority of the energy is consumed by the Processing System 7 (PS7) on the Zynq 7000 chip, which alone accounts for 1.564 watts. This significant figure underscores the central role of the PS7 in the system's overall power usage. In comparison, the dynamic power consumption from other components is relatively minimal. Specifically, the power used by clocks is 0.014 watts, signals consume 0.004 watts,

and logic elements account for 0.003 watts. The Block RAM (BRAM) and I/O operations together contribute a small fraction, with the BRAM consuming 0.001 watts and I/O operations using 0.003 watts. Additionally, the static power consumption of the device stands at 0.145 watts, representing the baseline power required when the system is idle. Summing both dynamic and static power, the total power consumption is 1.735 watts. This comprehensive breakdown highlights that while the PS7 is the primary contributor to power usage, the total power consumption reflects a balance between active processing and idle power requirements, providing a clear picture of the system's energy demands and efficiency.

4.5 Timing Analysis

The timing analysis compares the performance of the system with software-only implementation and software with Net Engine to evaluate its impact on processing efficiency. The comparative timing data illustrates that the implementation with the Net Engine achieves substantially faster execution, demonstrating its effectiveness in accelerating computations and improving overall performance. The analysis of processing times across four critical stages of the model demonstrates substantial improvements with the integration of the Net Engine. This section presents the findings in terms of processing time reduction at different scale factors. The scale factor is the factor that is used to scale the input channels.

4.5.1 Model Processing Time

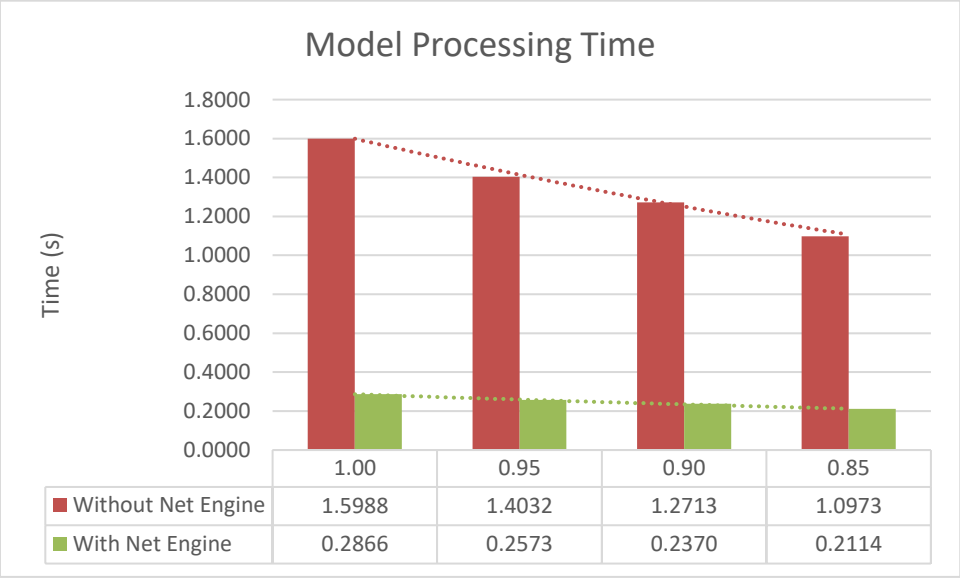


Figure 15: The chart of Modet Processing Time

At Scale Factor 1.00, the processing time decreased from 1.5988 seconds without the Net Engine to 0.2866 seconds with it, resulting in a reduction of 1.3122 seconds. This trend is consistent across all scale factors, with reductions of 1.1459 seconds, 1.0343 seconds, and 0.8860 seconds at Scale

Factors 0.95, 0.90, and 0.85, respectively.

4.5.2 CNN Layer Processing Time

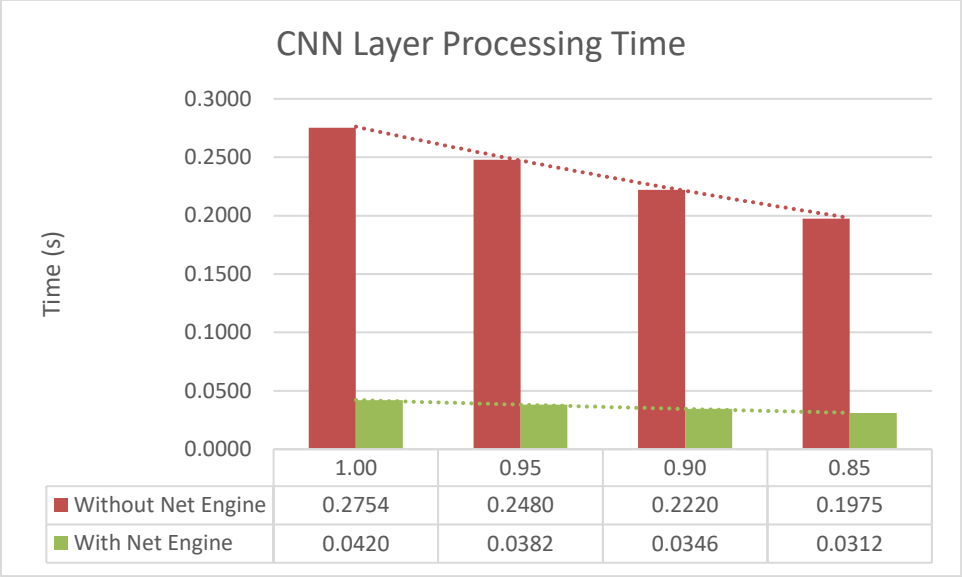


Figure 16: The chart of CNN Layer Processing Time

The integration of the Net Engine significantly reduced processing times here as well. For Scale Factor 1.00, the processing time fell from 0.2754 seconds to 0.0420 seconds, a difference of 0.2333 seconds. This reduction ranges from 0.2097 seconds at a Scale Factor of 0.95 to 0.1663 seconds at a Scale Factor of 0.85.

4.5.3 Channel Processing Time

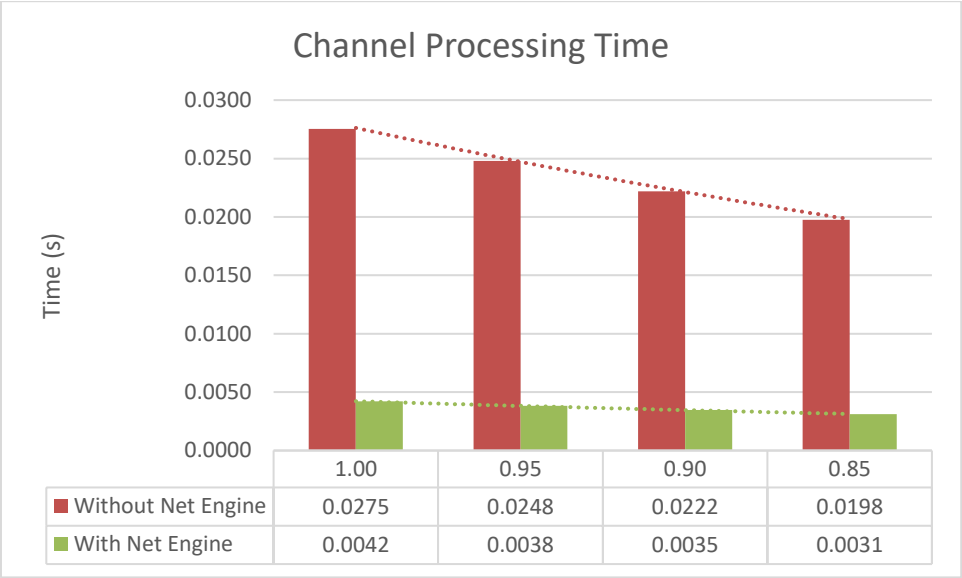
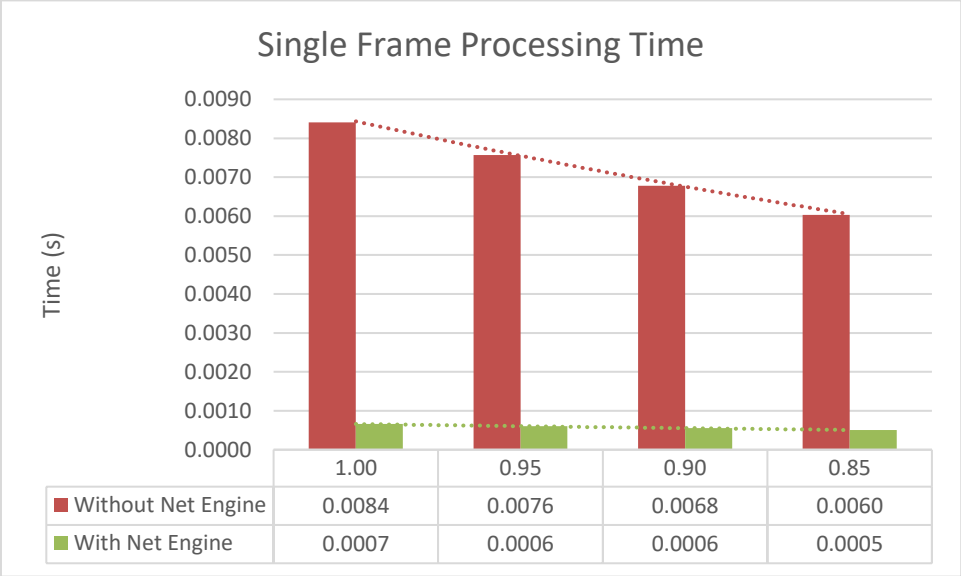


Figure 17: The Chart of Channel Processing Time

The Net Engine also led to notable reductions in processing times for the first layer's channels. At Scale Factor 1.00, the time dropped from 0.0275 seconds to 0.0042 seconds, representing a

reduction of 0.0233 seconds. The reductions range from 0.0210 seconds at a Scale Factor of 0.95 to 0.0166 seconds at a Scale Factor of 0.85.

4.5.4 Single Frame Processing Time



	1.00	0.95	0.90	0.85
Without Net Engine	0.0084	0.0076	0.0068	0.0060
With Net Engine	0.0007	0.0006	0.0006	0.0005

Figure 18: The chart of Single Frame Processing Time

Finally, for the processing time of the first layer's frames, the reduction is evident. The time decreased from 0.0084 seconds to 0.0007 seconds at Scale Factor 1.00, a difference of 0.0077 seconds. The reduction in time ranges from 0.0070 seconds at a Scale Factor of 0.95 to 0.0055 seconds at a Scale Factor of 0.85.

The data illustrates the substantial performance improvements afforded by the Net Engine across various processing stages and scale factors. The consistent reductions in processing times across the model, CNN layers, channels, and frames underscore the Net Engine's effectiveness in optimizing computational efficiency and enhancing overall model performance. This improvement is critical for applications requiring high-speed processing and real-time analysis, confirming the Net Engine's significant contribution to performance enhancement.

The logic analyzer results show in Figure 19 and Figure 20 that the image processing time with the net engine is 0.2866 seconds, while without the net engine, it is 1.5988 seconds.

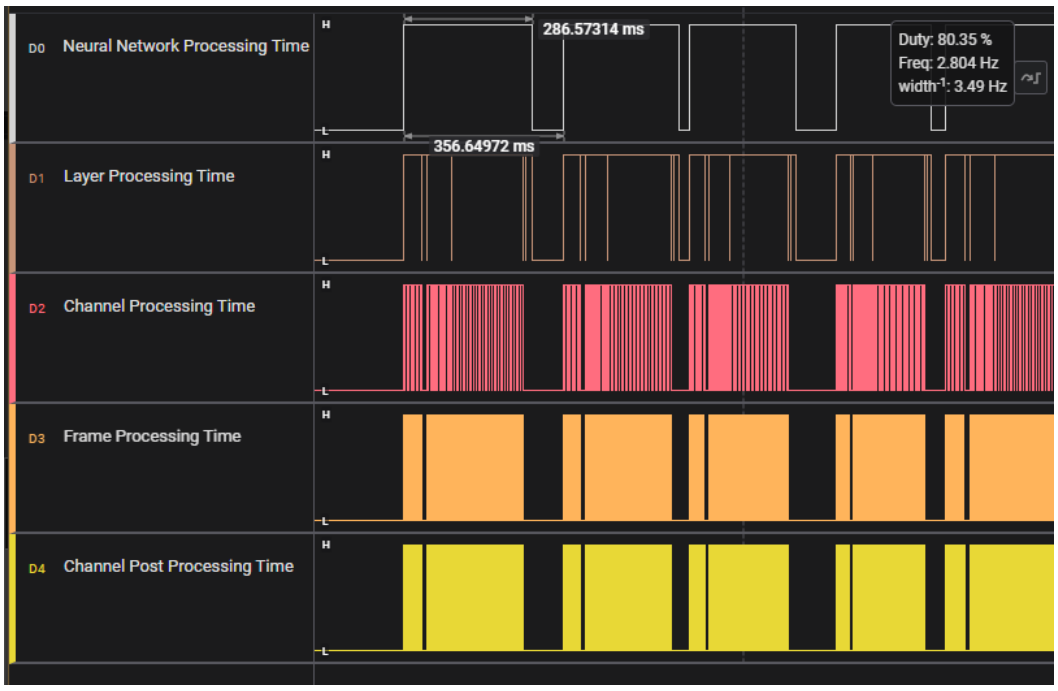


Figure 19: The time measurement diagram of software and hardware implementation

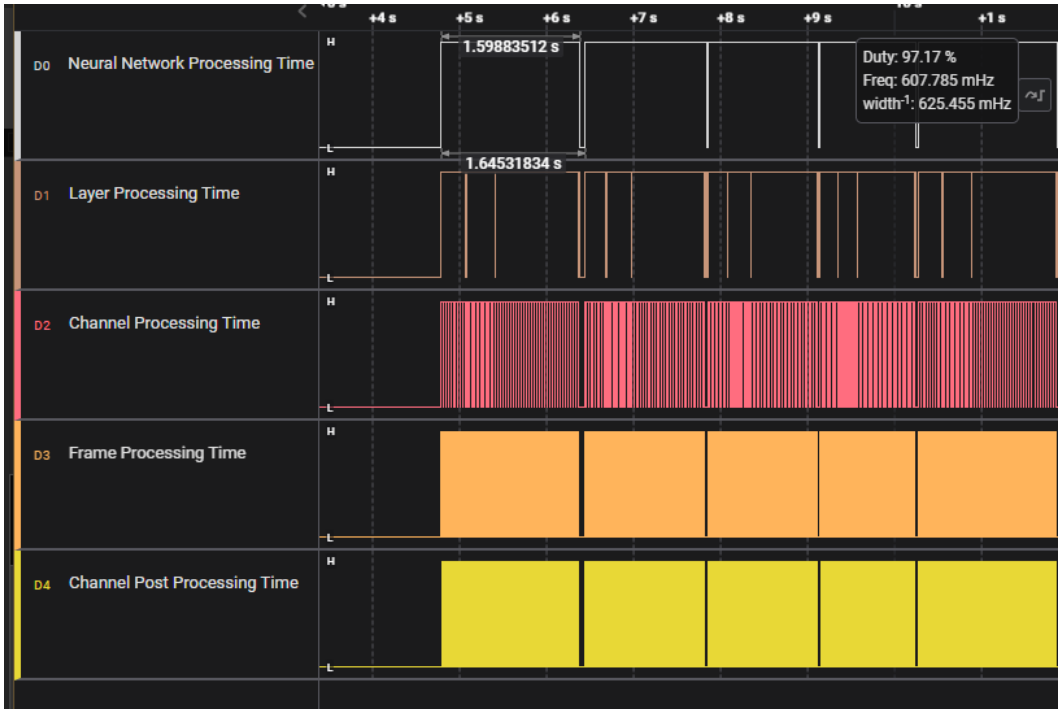


Figure 20: The time measurement diagram of only software implementation

Chapter 5: Discussion

In this research, the performance of hardware-based and software-based implementations of convolution and max-pooling algorithms using a PYNQ-Z2 development board has been evaluated. The primary focus was on analyzing the accuracy and processing time of each approach to determine the effectiveness of the implementation of the convolution and max-pooling for convolution neural networks or other image processing devices. This hardware design and driver software can be implemented to the edge device to perform a complex operation at the edge while keeping high accuracy on the output results.

5.1 Performance Analysis

The experimental results demonstrated that the hardware-based implementation was significantly higher performing rather than the software-based solution in terms of processing time. However, the output results of both methods are approximately equal. The custom hardware IP designed for convolution and max-pooling operations leveraged the pipelined processing capabilities of the FPGA, resulting in a substantial reduction in computation time compared to the software-based implementation running on the ARM Cortex-A9 processor in the PYNQ-Z2 development board.

The processing time reduction was particularly evident in scenarios involving larger input images and more complex convolution kernels. The hardware acceleration allowed for real-time processing, which is critical for applications like facial recognition and object detection, where speed is paramount.

However, the analysis also revealed trade-offs. The hardware implementation required considerable development time and resources, including the design of the custom IP and the integration with the system. In contrast, the software-based approach, while slower, offered greater flexibility and ease of implementation, as it did not require specialized hardware knowledge or tools.

5.2 Accuracy Considerations

Both the hardware and software implementations achieved similar accuracy levels, as expected. Since the core algorithms were identical, the accuracy of the output was consistent across both hardware and software implementations. This consistency suggests that hardware acceleration does not compromise the accuracy of convolution and max-pooling operations, making it a viable option for applications requiring both high speed and precision.

5.3 Practical Implications

The selection between hardware-based and software-based implementations depends on the

specific requirements of the device application. For real-time systems where processing speed is critical, such as live video streaming or high-frequency image processing, the hardware-based approach offers clear advantages. But still, the hardware design needs to be improved to do operation faster for streaming data at high frequency. The ability to reduce latency provided by FPGA acceleration makes it the preferred choice for these use cases. However, there may still be some time lags between consecutive image frames.

On the other hand, software-based implementations are more suitable for applications where flexibility and ease of development are more important than processing speed. Examples include environments where the hardware configuration may frequently change, or where development time is limited. Additionally, software implementations are easier to update and modify, allowing for quicker iterations and adaptations to new requirements.

5.4 Limitations and Future Work

Despite the benefits of hardware acceleration, this study highlighted several limitations. The development of custom hardware IP is resource-intensive, requiring specialized skills and tools. Additionally, the fixed nature of hardware designs can limit flexibility, making it challenging to adapt to new algorithms or changes in system requirements without significant redesign efforts.

Future work could explore the integration of more advanced neural network operations into the hardware design, potentially including layers beyond convolution and max-pooling, such as fully connected layers or batch normalization. Instead of a single net engine, the count of the net-engine modules can be increased to archive two frames processing at the same time. However, for the additional net-engine module, the processing system side needs to allocate another memory pool. Additionally, investigating the use of high-level synthesis (HLS) tools could reduce the complexity and development time associated with hardware IP design, making hardware acceleration more accessible to a broader range of developers.

Further optimization of the hardware design could focus on minimizing resource usage and power consumption, which are critical factors in embedded systems. The exploration of hybrid approaches, combining hardware acceleration for computationally intensive tasks with software-based implementations for more flexible operations, could also be a promising direction for future research.

Chapter 6: Conclusion

This study has addressed the challenge of achieving real-time facial computing on resource-constrained edge devices through the design and implementation of a hardware accelerator. The primary focus was on developing an FPGA-based accelerator capable of delivering high-performance facial computing by optimizing both hardware and software components. The findings from the evaluation of the Net Engine IP contribute significantly to the advancement in process timing and accuracy.

The major findings of the research have been summarized below. These results provide a comprehensive summary of the key outcomes.

- **Comparison of Net Engine Output with Software Implementation**: The empirical analysis of the Net Engine IP against software-based solutions revealed a high degree of accuracy and consistency. The comparative metrics, including correlation coefficients, Chi-Square statistics, and histogram intersections, demonstrate that the hardware accelerator replicates software outputs with notable precision. These results validate the hardware's capability to perform facial computing tasks comparably to traditional software implementations.
- **Accuracy of the Proposed Network Model**: The Net Engine IP exhibited exceptional accuracy across various layers and channels of the proposed network model, with accuracy scores consistently approaching 1.000. This indicates that the hardware accelerator effectively mirrors the performance of the software-based approach, confirming its robustness and reliability in processing convolution neural networks.
- **Processing Time Efficiency**: The integration of the Net Engine IP resulted in significant reductions in processing times for various stages of the convolution neural network. The hardware accelerator markedly enhanced the efficiency of model processing, convolutional neural network (CNN) layers, individual channels, and frame processing even with the programmable logic (PL) side operating at 50Mhz clock frequency. These improvements underscore the Net Engine's capability to deliver real-time performance, thereby addressing the computational demands of applications.
- **Power Consumption Analysis**: The power utilization analysis revealed that the Processing System 7 (PS7) constitutes the predominant portion of the total system power consumption, measured at 1.735 watts. This analysis provides insights into the energy efficiency of the Net Engine IP and highlights the balance between active processing and idle power requirements.
- **Contributions to the Field**: This research makes substantial contributions to the field of edge computing and real-time computing by presenting a viable solution for deploying net engines

on resource-limited devices to perform convolution neural networks (CNN). The successful implementation of the FPGA-based hardware accelerator provides a practical framework for future developments in edge AI applications, potentially enhancing user experiences and operational efficiency.

The research underscores the effectiveness of the Net Engine IP as a robust solution for real-time computing on resource-constrained edge devices. By demonstrating high accuracy, significant reductions in processing time, and efficient power utilization, the study highlights the Net Engine's capability to perform comparably to traditional software implementations while addressing the computational challenges. The contributions made by this dissertation are the way for future advancements in edge AI applications, offering practical insights and frameworks for enhancing operational efficiency and user experiences in real-time computing scenarios.

References

A. B S, S. R. (2022). "NC-Emotions: Neuromorphic hardware accelerator design for facial emotion recognition,". *29th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 1-4. doi:Glasgow, United Kingdom

A. Baobaid, M. M. (2022). "Hardware Accelerators for Real-Time Face Recognition: A Survey,". *IEEE Access, 10*, 83723-83739. doi:10.1109/ACCESS.2022.3194915

al., A. M. (2019). "Deep Learning for Edge Computing: Current Trends, Cross-Layer Optimizations, and Open Research Challenges,". *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 553-559. doi:10.1109/ISVLSI.2019.00105

al., X. L. (2022, Feb). "Collaborative Edge Computing With FPGA-Based CNN Accelerators for Energy-Efficient and Time-Aware Face Tracking System,". *IEEE Transactions on Computational Social Systems, 9*, 252-266. doi:10.1109/TCSS.2021.3059318

Rao, B. S. (2021). "Binary neural network based real time emotion detection on an edge computing device to detect passenger anomaly,". *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, 175-180. doi:10.1109/VLSID51830.2021.00035

S. Liu, L. L. (2019, Aug). "Edge Computing for Autonomous Driving: Opportunities and Challenges,". *Proceedings of the IEEE, 107*, 1697-1716. doi:10.1109/JPROC.2019.2915983

Talib, M. M. (2021, Feb). A systematic literature review on hardware implementation of artificial intelligence algorithms. *J Supercomput 77*, 1897–1938. doi:10.1007/s11227-020-03325-8

Xue Lv, M. S. (2021). Application of Face Recognition Method Under Deep Learning Algorithm in Embedded Systems. *Microprocessors and Microsystems*. doi:10.1016/j.micpro.2021.104034

Y. S. Mounika, B. B. (2022). "FPGA BASED HARDWARE ACCELERATOR FOR CONVOLUTION NEURAL NETWORK,". *2022 International Conference on Recent Trends in Microelectronics, Automation, Computing and Communications Systems (ICMACC)*, 260-264. doi:10.1109/ICMACC54824.2022.10093648.

Yu, C. F. (2019). "FPGA-based Power Efficient Face Detection for Mobile Robots,". *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 467-473. doi:10.1109/ROBIO49542.2019.8961745

Appendices

8.1 Added main.py python script that used to process the results

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.stats import chisquare, pearsonr
from scipy.ndimage import rotate
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

def load_hwframe(filename=None):
    """Load hardware frame data from a .npy file."""
    channels = []
    out = np.load(filename).astype(np.float32)
    for i in range(out.shape[-1]):
        channels.append(out[0, :, :, i])
    return channels

def load_swframe(filename=None):
    """Load software frame data from a .npy file."""
    channels = []
    out = np.load(filename)
    for i in range(out.shape[-1]):
        channels.append(out[0, 0, :, :, i])
    return channels

def plot_histograms(hist1, hist2, title1='Image 1 Histogram',
    title2='Image 2 Histogram'):
    """Plot histograms of two images."""
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.bar(range(256), hist1, width=1.0, color='blue')
    plt.title(title1)
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')

    plt.subplot(1, 2, 2)
    plt.bar(range(256), hist2, width=1.0, color='red')
    plt.title(title2)
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')

    plt.tight_layout()
    plt.show()
```

```
def numerical_comparison(hist1, hist2, channel_index):
    """Compute and print numerical comparisons between two histograms."""
    hist1, _ = np.histogram(hist1, bins=256, range=(0, 255))
    hist2, _ = np.histogram(hist2, bins=256, range=(0, 255))

    corrcoef = np.corrcoef(hist1, hist2)[0, 1]
    chi2_stat, p_value = chisquare(hist1, hist2)
    corr, _ = pearsonr(hist1, hist2)
    intersection = np.sum(np.minimum(hist1, hist2)) / np.sum(hist2)

    print(f"Channel {channel_index}:")
    print(f"Correlation Coefficient: {corrcoef:.4f}")
    print(f"Chi-Square Stat: {chi2_stat:.4f}, p-value: {p_value:.4f}")
    print(f"Intersection: {intersection:.4f}")
    print(f"Pearson Correlation: {corr:.4f}")

def plot_image_comparisons(sw_channels, hw_channels, layer):
    """Plot and compare the first 5 images and their histograms."""
    num_images = min(len(sw_channels), len(hw_channels), 5)
    plt.figure(figsize=(30, num_images * 6))

    for i in range(num_images):
        rot_sw = rotate(sw_channels[i], -90, reshape=True)
        rot_hw = rotate(hw_channels[i], -90, reshape=True)

        numerical_comparison(rot_sw.ravel(), rot_hw.ravel(), i)

        plt.subplot(num_images, 4, 4 * i + 1)
        plt.imshow(rot_sw, cmap='gray', vmin=0, vmax=255)
        plt.axis('off')
        plt.title(f'SW Image {i + 1}')

        plt.subplot(num_images, 4, 4 * i + 2)
        plt.hist(rot_sw.ravel(), bins=256, range=(0, 255), color='blue',
        edgecolor='black')
        plt.xlabel('Pixel Intensity')
        plt.ylabel('Frequency')
        plt.grid(True)

        plt.subplot(num_images, 4, 4 * i + 3)
        plt.imshow(rot_hw, cmap='gray', vmin=0, vmax=255)
        plt.axis('off')
        plt.title(f'HW Image {i + 1}')

        plt.subplot(num_images, 4, 4 * i + 4)
        plt.hist(rot_hw.ravel(), bins=256, range=(0, 255), color='red',
        edgecolor='black')
```

```
        plt.xlabel('Pixel Intensity')
        plt.ylabel('Frequency')
        plt.grid(True)

    plt.suptitle(f'Layer {layer} Image Comparisons', fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.savefig(f"Layer-{layer}-image-comparison.png")
    plt.show()

def main():
    """Main function to load data, compare images"""
    sw_files = ["sw_layer_1_CNN_1_RELU_98X98.npy",
                "sw_layer_2_MAXPOOL_49X49.npy",
                "sw_layer_3_CNN_2_RELU_47X47.npy",
                "sw_layer_4_CNN_3_RELU_45X45.npy",
                "sw_layer_5_CNN_4_Softmax_45X45.npy",
                "sw_layer_6_CNN_5_45X45.npy"]
    hw_files = ["hw_layer_1_output.npy",
                "hw_layer_2_output.npy",
                "hw_layer_3_output.npy",
                "hw_layer_4_output.npy",
                "hw_layer_5_output.npy",
                "hw_layer_6_output.npy"]

    # For full comparison across all layers
    for i in range(len(sw_files)):
        print(f"Layer {i}")
        sw_channels = load_hwframe(filename=sw_files[i])
        hw_channels = load_swframe(filename=hw_files[i])
        plot_image_comparisons(sw_channels, hw_channels, i)

if __name__ == "__main__":
    main()
```

--END--