BUILDING BRAINS WITH ARM PROCESSORS AND FPGAS BASED ON HIGH PERFORMANCE ARCHITECTURES CONVOLUTIONAL NEURAL NETWORKS

LAI JIAN CHANG

UNIVERSITI TEKNIKAL MALAYSIA MELAKA

BUILDING BRAINS WITH ARM PROCESSORS AND FPGAS BASED ON HIGH PERFORMANCE ARCHITECTURES CONVOLUTIONAL NEURAL NETWORKS

LAI JIAN CHANG

This report is submitted in partial fulfilment of the requirements for the degree of Bachelor of Computer Engineering with Honours

Faculty of Electronic and Computer Technology and Engineering
Universiti Teknikal Malaysia Melaka

2024

UNIVERSITI TEKNIKAL MALAYSIA MELAKA
FAKULTI TEKNOLOGI DAN KEJURUTERAAN ELEKTRONIK DAN KOMPUTER

BORANG PENGESAHAN STATUS LAPORAN
PROJEK SARJANA MUDA II

Tajuk Projek	:	Building Brains with ARM processors and FPGAs based on high performance architectures Convolutional Neural Networks
Sesi Pengajian	:	2023/2024

Saya LAI JIAN CHANG mengaku membenarkan laporan Projek Sarjana Muda ini disimpan di Perpustakaan dengan syarat-syarat kegunaan seperti berikut:

1. Laporan adalah hakmilik Universiti Teknikal Malaysia Melaka.
2. Perpustakaan dibenarkan membuat salinan untuk tujuan pengajian sahaja.
3. Perpustakaan dibenarkan membuat salinan laporan ini sebagai bahan pertukaran antara institusi pengajian tinggi.
4. Sila tandakan (✓):

☐	**SULIT***	(Mengandungi maklumat yang berdarjah keselamatan atau kepentingan Malaysia seperti yang termaktub di dalam AKTA RAHSIA RASMI 1972)
☐	**TERHAD***	(Mengandungi maklumat terhad yang telah ditentukan oleh organisasi/badan di mana penyelidikan dijalankan.
✓	**TIDAK TERHAD**	

Disahkan oleh:

[illegible]
(TANDATANGAN PENULIS)

[illegible]
(COP DAN TANDATANGAN PENYELIA)

PROF. MADYA DR. WONG YAN CHIEW
Profesor Madya
Fakulti Kejuruteraan Elektronik & Kejuruteraan Komputer
Universiti Teknikal Malaysia Melaka (UTeM)
Hang Tuah Jaya,
76100 Durian Tunggal, Melaka

Alamat Tetap: 2901 JLN RJ 5/2 TAMAN RASAH JAYA 70300 SEREMBAN NSDK

Tarikh : 10 JANUARI 2024

Tarikh : 23 JANUARI 2024

*CATATAN: Jika laporan ini SULIT atau TERHAD, sila lampirkan surat daripada pihak berkuasa/organisasi berkenaan dengan menyatakan sekali tempoh laporan ini perlu dikelaskan sebagai SULIT atau TERHAD.

DECLARATION

I declare that this report entitled “BUILDING BRAINS WITH ARM PROCESSORS AND FPGAS BASED ON HIGH PERFORMANCE ARCHITECTURES CONVOLUTIONAL NEURAL NETWORKS” is the result of my own work except for quotes as cited in the references.

Signature	:	[illegible]
Author	:	LAI JIAN CHANG
Date	:	10 JANUARY 2024

APPROVAL

I hereby declare that I have read this thesis and in my opinion this thesis is sufficient in terms of scope and quality for the award of Bachelor of Computer Engineering with Honours.

Signature	:	[illegible]
Supervisor Name	:	PM Dr. Wong Yan Chiew
Date	:	23 Jan 2024

DEDICATION

To my beloved parents, who have been my unwavering source of inspiration, motivation, and encouragement throughout my academic journey. Without your love, sacrifices, and unwavering support, I would not be where I am today. This thesis is a testament to your unrelenting faith in me, and I dedicate it to you with all my heart. To my academic supervisor, Dr. Wong Yan Chiew, whose expertise, guidance, and mentorship have shaped my research and challenged me to push beyond my limits. Your patience, constructive feedback, and valuable insights have been instrumental in helping me grow as a researcher and a scholar. I am grateful for the opportunities you have given me to learn, collaborate, and contribute to the academic community. To Universiti Teknikal Malaysia Melaka, the institution that has provided me with a rich and vibrant academic environment, cutting-edge resources, and a platform to pursue my intellectual passions. My experience at this university has been transformative, empowering me to think critically, to explore new ideas, and to engage with diverse perspectives. I am honoured to be a part of this academic community, and I dedicate this thesis to the university that has given me so much. Thank you.

ABSTRACT

The advent of Convolutional Neural Networks (CNNs) has transformed the landscape of artificial intelligence, particularly in visual information processing. This thesis embarks on a comprehensive exploration of advanced architectures for CNNs, focusing on the strategic integration of ARM processors and Field-Programmable Gate Arrays (FPGAs). The overarching goal is to harness the synergies between these heterogeneous computing platforms, capitalizing on their respective strengths to engineer high-performance systems capable of intricate visual interpretation. The research unfolds through an in-depth investigation into both hardware and software aspects, aiming to optimize the design, deployment, and performance of CNNs. Special attention is given to the development of tailored algorithms that align with the unique features of ARM processors and FPGAs. This includes the implementation of efficient memory utilization strategies and parallelization techniques to fully exploit the parallel processing capabilities inherent in these architectures. A critical facet of the study involves addressing challenges related to power consumption, thermal considerations, and resource utilization. By exploring novel approaches to mitigate these challenges, the thesis seeks to establish a foundation for creating intelligent systems with brain-like processing capabilities while maintaining energy efficiency.

The research methodology employs rigorous experimentation and performance evaluations, with a keen focus on determining the trade-offs between computational efficiency and model accuracy. Insights derived from this exploration contribute to the overarching goal of advancing the field, offering a nuanced understanding of how the integration of ARM processors and FPGAs can yield optimized architectures for CNNs. The findings of this thesis not only extend the current understanding of high-performance architectures for CNNs but also lay the groundwork for future developments at the intersection of hardware design, neural networks, and artificial intelligence. The implications of this research resonate across various applications, from enhancing computer vision capabilities to empowering autonomous systems with sophisticated visual perception and interpretation.

ABSTRAK

Keberadaan Rangkaian Neural Konvolusional (CNNs) telah mengubah landskap kecerdasan buatan, khususnya dalam pemrosesan informasi visual. Tesis ini memulai eksplorasi komprehensif terhadap arsitektur canggih untuk CNNs, dengan fokus pada integrasi strategis prosesor ARM dan Field-Programmable Gate Arrays (FPGAs). Tujuan utamanya adalah memanfaatkan sinergi antara platform komputasi heterogen ini, memanfaatkan keunggulan masing-masing untuk merancang sistem berkinerja tinggi yang mampu melakukan interpretasi visual yang rumit.Penelitian ini dilaksanakan melalui penyelidikan mendalam terhadap aspek perangkat keras dan perangkat lunak, dengan tujuan mengoptimalkan desain, implementasi, dan kinerja CNNs. Perhatian khusus diberikan pada pengembangan algoritma yang disesuaikan dengan fitur unik dari prosesor ARM dan FPGAs. Ini termasuk penerapan strategi penggunaan memori yang efisien dan teknik paralelisasi untuk sepenuhnya memanfaatkan kemampuan pemrosesan paralel yang melekat dalam arsitektur ini.Fase penting dari penelitian ini melibatkan penanganan tantangan terkait konsumsi daya, pertimbangan termal, dan pemanfaatan sumber daya. Dengan mengeksplorasi pendekatan baru untuk mengatasi tantangan ini, tesis ini berusaha membentuk dasar bagi pembuatan sistem cerdas dengan kemampuan pemrosesan *mirip otak sambil tetap menjaga efisiensi energi.Metodologi penelitian menggunakan eksperimen yang ketat dan evaluasi kinerja, dengan fokus pada penentuan kompromi antara efisiensi komputasional dan akurasi model. Wawasan yang diperoleh dari eksplorasi ini memberikan kontribusi pada tujuan umum untuk memajukan bidang ini, menawarkan pemahaman yang mendalam tentang bagaimana integrasi prosesor ARM dan FPGAs dapat menghasilkan arsitektur yang dioptimalkan untuk CNNs. Temuan dari tesis ini tidak hanya memperluas pemahaman saat ini tentang arsitektur berkinerja tinggi untuk CNNs tetapi juga meletakkan dasar untuk pengembangan di masa depan di persimpangan desain perangkat keras, jaringan saraf, dan kecerdasan buatan. Implikasi dari penelitian ini mencakup berbagai aplikasi, mulai dari meningkatkan kemampuan visi komputer hingga memberdayakan sistem otonom dengan persepsi visual dan interpretasi yang canggih.*

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and gratitude to the individuals who have been instrumental in the completion of this thesis. First and foremost, I extend my sincere thanks to my advisor, Profesor Madya Dr. Wong Yan Chiew for their unwavering guidance, invaluable insights, and continuous support throughout the entire research process. Their expertise and encouragement were instrumental in shaping the direction of this work. I am indebted to the members of my thesis committee for their constructive feedback, scholarly input, and commitment to ensuring the quality of this research. Their collective expertise significantly enriched the content and methodology of this thesis. Special thanks are due to Eng Yong Ming for their collaborative spirit, insightful discussions, and shared commitment to advancing our understanding in this field. The exchange of ideas and collective effort greatly contributed to the success of this research. I express my gratitude to Universiti Teknikal Malaysia Melaka for providing the necessary resources and research facilities that facilitated the execution of this project. Heartfelt appreciation goes to my family for their unwavering support, understanding, and encouragement throughout this academic journey. Their love and encouragement were my pillars of strength. Lastly, I extend my thanks to all those friends and mentors who, in various

ways, contributed to the completion of this thesis. Your support has been truly invaluable. This thesis is a testament to the collaborative spirit and collective efforts of those who have touched my academic journey. Thank you all for your support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

CNN	: Convolutional Neural Network

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Introduction

In this chapter, there will be 4 sections. The background study of the project will be discussed in Section 1.2 followed by the problem statement in Section 1.3. The objectives of the project are discussed in Section 1.4 and the scope of work is discussed in Section 1.5.

1.2 Background Study

Deep learning has become a game-changing technology with a wide range of applications in the quickly developing field of artificial intelligence. Convolutional Neural Networks (CNNs) are the most well-known deep learning architectures because of their outstanding results in image and video processing applications. But as of right now, we know that deploying CNNs on conventional computing platforms like CPUs and GPUs frequently presents difficulties including excessive power consumption, expense, and thermal problems, which restricts their use in real-time settings.

1.3 Objectives

- To design and implement a CNN on an FPGA.
- To explore optimization techniques for improving the performance of the CNN on the FPGA.
- To analyze the performance of the FPGA-based CNN in terms of resource utilization, power consumption, and accuracy.

The first objective of the project is to design and implement a CNN on an Field Programmable Gate Arrays (FPGA) with validation on board to make sure the CNN is actually working on board. With the resource constraint on FPGA, optimization technique needs to be done on the CNN would need to be explore in order to make sure the implementation works on real life FPGA board. Finally, analyzing the performance for the FPGA based CNN is also essential to gain an insight on the project.

1.4 Problem Statement

CNN has become a popular method in image and video processing. The trend of implementing the CNN are usually done on CPU and GPU since they have a better speed and resource constraint. However, the power consumption, expenses, and resource utilization of the traditional computing platform is elevating for the push of accuracy and speed. This project aims to tackle the issue discussed above by investigating the use of FPGA on implementing the CNN. This is because FPGA has become a possible alternative for implementing CNN since it uses lesser power consumption, lesser resource and more cheaper compared to traditional computing platforms. FPGA is well known for its portability, reconfigurable and power consumption level.

The basic concept of solving the issue with the traditional computing platform is to reduce its high computational power and high-power consumption. The plan is to utilize the parallel processing and the reconfigurable ability of the FPGA to provide a more economic and effective platform for implementing the CNN. However, the challenges of doing so are also a foreseen bump. With the resource and power constraint and architecture of the FPGA, a series of work needs to be done to make the implementation of CNN successful in the FPGA. Analysis needs to be done as well to validate the problems of traditional computing platforms and the solution to use FPGA based CNN.

The high cost of off-chip communication and the need for energy-efficient hardware acceleration of CNNs on FPGAs also needed to be emphasized. [1] The challenges of implementing CNNs on platforms with limited resources, such as FPGAs should also be highlighted. [2] This project aims to tackle this issue as well.

1.5 Scope of work

The scope of work for this project is to design and implement the CNN on the PYNQ Z1 board with optimization technique used and to also perform the performance analysis of the implemented CNN on the board. The dataset used are MNIST and Street View House Number (SVHN) datasets. Software used are the Docker, Jupyterlab, Vivado and Python.

1.6 Thesis Organization

The 1st chapter of this thesis will be focusing on the introduction of the whole project with objectives and problem statements included. In the 2nd chapter, the background study of the project will be included as well as the literature review for a better understanding on the state-of-art for this project. The 3rd chapter will focus on the methodology of the project to achieve the objectives and solving the problems stated. The result and comparison of analysis will be in the 4th chapter of the thesis followed by the chapter 5 with conclusions and future works related to the project.

CHAPTER 2

BACKGROUND STUDY

2.1 Introduction

This chapter will be focusing on the background study of the project and the literature review for investigating the state-of-art of the project. Background study will be focusing on what is Artificial Intelligence and its branch, Field Programmable Gate Array (FPGA) and High-Level Synthesis.

2.2 Background

2.2.1 Artificial Intelligence

The field of artificial intelligence (AI) has evolved from humble beginnings to a field with global impact. The definition of AI and of what should and should not be included has changed over time. Experts in the field joke that AI is everything that computers cannot currently do. Although facetious on the surface, there is a sense that developing intelligent computers and robots means creating something that does not exist today. Artificial intelligence is a moving target.[3]

The paper “What is AI?” has provided three definitions for AI. The most basic element common to all of them is that AI involves the study, design and building of intelligent agents that can achieve goals. The choice of an AI makes should be appropriate to its perceptual and cognitive limitations. If an AI is flexible and can learn from experience. sense, plan, and act on the basis of its initial configuration, it might be said to be more intelligent than an AI that just has a set of rules that guides a fixed set of actions. However, there are some contexts in which you might not want the AI to learn new rules and behaviors. Perception of AI differs for humans from different field. For example, developers of expert systems see AI as a repository of expert knowledge that humans can consult, whereas developers of machine learning systems see AI as something that might discover new knowledge. As we shall see, each approach has strengths and weaknesses. [3]

2.2.2 Deep Learning

Deep Learning (DL) is a subset of machine learning (ML). DL is to use the AI to imitate the human brains by using neurons and connect them creating a neural network to perform computational calculations for machines to learn from a large amount of data. DL is very commonly used for image classification tasks.

DL uses a trainable features extractor as compared to ML uses a hand-crafted feature extractor. The representation of features is hierarchical and trained which usually are low, mid, and high-level features. Low level features are features that is the basic attribute of the image or video that can be extracted easily such as contours and edges. The mid-level feature acts as the bridge for connecting the low- and high-level features. The high-level feature are conceptual and more significant, they are formed by pairing the low-level feature and mid-level feature, they are often the characteristics or components of an object,

However, DL requires a large number of labeled data or huge, labeled datasets to achieve a high accuracy not to mention that the configuration of the model is challenging to prevent overfitting due to large number of datasets.

2.2.3 Convolutional Neural Network

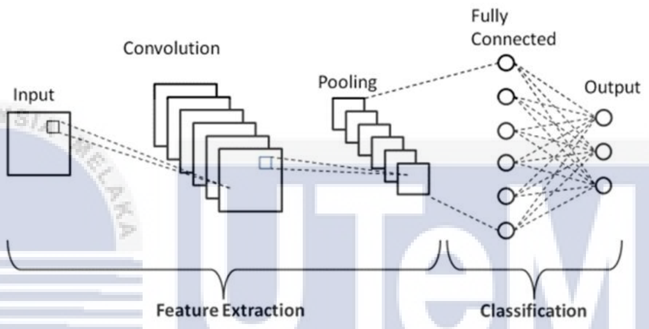


Figure 2.1: Basic Architecture of CNN

CNN: CNNs are specialized neural networks designed for processing grid-like data, such as images and videos. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply convolution filters to detect local patterns, while pooling layers downsample the data to reduce the size of data and its dimensionality while preserving key information. Activation functions introduce non-linearity into the network, allowing it to learn more complex patterns since CNN is normally dealing with classification tasks not trend or linear prediction. Commonly used activation functions are ReLU and sigmoid activation function. The fully connected layers will do the classification task based on the features extracted. CNN is normally well-suited for tasks like image classification, object detection, and image segmentation.

CNN also presents some challenges for implementation. Training and running CNNs is computationally expensive since it is requiring powerful hardware resources to make sure the performance meets the requirement. CNNs can easily achieving overfit, especially when dealing with small datasets where the accuracy of the testing does not provide a significant insight since the CNN is predicting the data using memorization. Despite the good performance, a deep CNN has been considered a black-box model with weak feature interpretability for decades. Boosting the feature interpretability of a deep model gradually attracts increasing attention recently, but it presents significant challenges for state-of-the-art algorithms.[4] Understanding how CNNs make decisions can be difficult due to their complex internal representations.

Convolutional Neural Networks is a great tool to tackle with problems such as image classification. Its ability to learn and extract relevant features from images makes it a powerful tool for various applications. While challenges mentioned above exist, research and analysis of the CNN on various platforms is required for a better understanding and application using it to improve the level of lifestyle of human beings.

2.2.4 Field Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) is a kind of integrated circuits that offers a combination of reconfiguration and performance. Unlike traditional processors such as ASICS, FPGA is reprogrammable which allows users to configure them based on the needs and use. The flexibility of FPGA makes them suitable for many applications and researcher for the reuse of it based on each project needs.

Field Programmable Gate Arrays are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing.[5] The ability to be reprogrammed of FPGAs makes them different compared to Application-Specific Integrated Circuits (ASICs) and general-purpose processors. The adaptive power of FPGA makes them a choice for a variety of applications. FPGAs are made from multiple logic blocks, which are the building blocks that can be configured to perform a specific logic functions based on the usage of the algorithm. The blocks that are normally found in a FPGA are Look-Up Tables (LUTs), flip-flops, multiplexers, BRAM and DSP. The LUT is basically a table for producing output whenever an input is given. Flip flops or FF is the components that keeps the state of the chip, where it can store a single bit of information. Multiplexer is a selector which selects a single input from a variety of inputs. The BRAM or the memory is used to stored a lot of data which flows in the FPGA. The DSP in the FPGA acts as the computation unit for processing complex mathematical tasks.

FPGAs are programmed by using Hardware Description Languages (HDLs) such as Verilog or VHDL. These languages allow designers to describe the desired functionality of the digital circuit, specifying how logic blocks should be interconnected and how data should flow through the system. FPGA has the ability to utilize parallelism during processing which can leads to a good performance for certain applications, offering advantages in terms of speed and efficiency compared to traditional processors such as CPU and GPU. While FPGAs offer parallelism and efficiency, their programming and optimization can be complex without decent understanding. The architecture, resource utilization, and timing constraints needs to be configured and considered in order to get a good performance.

FPGA allows users to reprogrammed and change the use of the FPGA based on their own needs which is a benefit for them as compared to the traditional computing platform.

2.2.5 High Level Synthesis (HLS)

High Level Synthesis (HLS) is an automated process which is also known as C synthesis. HLS takes an abstract behavioral specification of a digital system and finds a register-transfer level structure that realizes the given behavior. HLS allows programmers or design to write algorithms in a high level programming language such as C, C++ and Python and through the HLS process, the algorithm converts into the HDL code which is then can be use in the FPGA. This is crucial since designing a complex algorithm in the HDL is a very complex task and allowing the synthesis to happen automatically from high level to low level language allows developers to implement complex model to the FPGA easier.

Despite its advantages, HLS usage requires significant knowledge on the hardware architecture of the hardware that we want to implement. Flow, memory and the constraints are parts where the designer need to be careful on making sure the performance is significantly comparable to the traditional computing platform. Many FPGA vendors provide HLS tools integrated into their development environments. For example, Xilinx comes out with Vivado and Vitis as their HLS tools for the Xilinx FPGA board.

As a conclusion, High Level Synthesis is a tools for designer to implement their high level language algorithms onto a hardware such as the FPGA that needs to be code in low level language such as the HDL by synthesizing the high level language to the low level language automatically.

2.3 Literature review

2.3.1 An FPGA-Based Convolutional Neural Network Coprocessor

In this study by the researchers in this paper, a convolutional neural network (CNN) coprocessor based on field-programmable gate arrays (FPGA) is introduced. In their research, the coprocessor features a row stationary (RS) streaming mode 1D convolutional computation unit PE and a pulsating array structure with a 3D convolutional computation unit PE chain. The flexibility of the coprocessor proposed lies in its ability to control the number of PE array openings based on the output channels of the convolutional layer. The paper also outlines the design of a storage system with multilevel cache, utilizing multiple broadcasts to distribute data to local caches. An image segmentation method compatible with the hardware architecture is proposed. [6]

The presented coprocessor successfully implements the convolutional and pooling layers of the VGG16 neural network model. Quantization of activation values, weight values, and bias values is performed using 16-bit fixed-point quantization. The coprocessor achieves a peak computational performance of 316.0 GOP/s and an average computational performance of 62.54 GOP/s at a clock frequency of 200MHz, with a power consumption of approximately 9.25 W. These findings contribute to the literature on FPGA-based CNN coprocessors and their application in neural network models.[6]

Table 1: Summary of An FPGA-Based Convolutional Neural Network Coprocessor

Index	Model	Target	Description
[6]	CNN	ZC706 evaluation board	Platform: ZynqXC7Z045 Frequency (MHz): 200 Quantification: 16 bits Power: 9.3W

			Performance (GOP/s): 62.54

2.3.2 MRI-based brain tumor segmentation using FPGA-accelerated neural network

In contrast to conventional computing platforms, the FPGA accelerator introduced in this study demonstrates significant enhancements in both speed and power efficiency. Utilizing the BraTS19 and BraTS20 datasets, our FPGA-based brain tumor segmentation accelerator exhibits performance gains of 5.21 and 44.47 times compared to the TITAN V GPU and the Xeon CPU, respectively. Moreover, in terms of energy efficiency, our design outperforms the GPU and CPU, achieving 11.22 and 82.33 times higher energy efficiency, respectively. These results shows a significant computational advantages and energy efficiency offered by their FPGA-based accelerator in the context of brain tumor segmentation.[7]

Table 2: Summary of MRI-based brain tumor segmentation using FPGA-accelerated neural network

Index	Model	Target	Description
[7]	CNN	Xilinx's Alveo U280 accelerator card	FPGA accelerator demonstrates significant improvements in speed and power efficiency for brain tumor segmentation. BraTS19 and BraTS20 datasets Performance gains of 5.21 and 44.47 times compared to TITAN V GPU and Xeon CPU, respectively. Energy efficiency outperforms GPU and CPU, achieving 11.22 and 82.33 times higher efficiency, respectively.

2.3.3 FPGA-Based CNN for Real-Time UAV Tracking and Detection

Neural networks (NNs) play a crucial role in modern artificial intelligence applications, particularly in tasks such as image classification and real-time object tracking. This research paper introduces an innovative approach to address the challenge of real-time monitoring and detection of unmanned aerial vehicles (UAVs) using a convolutional neural network (CNN) implemented on a Zynq UltraScale FPGA. The primary obstacle faced in implementing real-time algorithms on FPGA platforms is the constraint of DSP hardware resources. The proposed design in this journal successfully tackles the challenge in the context of autonomous real-time UAV detection and tracking, which leverage the capabilities of Xilinx's Zynq UltraScale XCZU9EG system on a chip (SoC).[8]

The solution presented in this study consists of two interconnected modules: a UAV tracking module and a neural network-based UAV detection module. The tracking module incorporates a novel background-differencing algorithm, while the UAV detection module utilizes a modified CNN algorithm optimized for maximum field-programmable gate array (FPGA) performance. These modules are synergistically designed to enhance real-time UAV detection in any given video input. The proposed system has undergone rigorous testing with actual flying UAVs, demonstrating an accuracy of 82%. It operates at the full frame rate of the input camera for both tracking and neural network detection, achieving performance comparable to an equivalent deep learning processor unit (DPU) with UltraScale FPGA-based HD video and tracking implementation. Notably, our approach exhibits lower resource utilization, as evidenced by the results obtained in this study.[8]

2.3.4 Briefly Analysis about CNN Accelerator based on FPGA.

Given the extensive and time-intensive nature of convolutional computations in deep learning, researchers frequently resort to leveraging GPU or FPGA acceleration to expedite these processes. This paper elucidates the merits of employing FPGA accelerators for convolutional computations. Specifically, it presents research findings on convolutional computation utilizing FPGA and elucidates the prevalent approach to FPGA accelerator design, employing high-level synthesis and Vitis AI.[9]

The paper further demonstrates the practical application of these concepts by deploying and executing the YOLOv4 model on the ZCU102 evaluation board using Vitis AI. The experimentation involves object detection with a tableware dataset, resulting in a recognition accuracy of 96.2%. Notably, the FPGA-accelerated implementation showcases a performance enhancement of 72.5 times compared to a CPU-based approach. These findings underscore the efficacy of FPGA accelerators in optimizing convolutional computations for deep learning applications.[9]

Table 3: Summary of Briefly Analysis about CNN Accelerator based on FPGA

Index	Model	Target	Description
[9]	CNN	FPGA (ZCU102)	The specific configuration of the CPU is: memory of 32GB, processor of Intel Core i7-8700 CPU @ 3.20GHz × 12, operating system of ubuntu 16.04 LTS (64 bit). After testing, the time required for YOLOv4 inference process on CPU is 1164.21 seconds, while it only takes 16.04 seconds on ZCU102 evaluation board, which is about 72.5 times higher than the performance of CPU. It can be seen that, compared to traditional CPU, FPGA can bring a lot of performance improvements for CNN inference process. The mean accuracy precision is up to 96.2%

2.3.5 A Review of the Optimal Design of Neural Networks Based on FPGA

The widespread adoption of deep learning, rooted in neural networks, has led to remarkable advancements in image recognition, speech recognition, natural language processing, automatic driving, and various other domains. FPGA emerges as a standout technology in the accelerated deep learning landscape due to its flexible architecture, versatile logic units, high energy efficiency, robust compatibility, and minimal latency. To stay current with the latest advancements in neural network optimization on FPGA, this review delves into related technologies and research themes.[10]

The paper begins by outlining the developmental trajectory and application domains of key neural networks, underscoring the significance of studying deep learning technology. It highlights the reasons and advantages of leveraging FPGA for accelerating deep learning tasks. Several prevalent neural network models are introduced, followed by an extensive review of contemporary FPGA-based neural network acceleration technologies, methods, accelerators, and framework designs. The paper also provides insights into the current challenges faced by FPGA-based neural network applications and proposes corresponding solutions. Lastly, it anticipates future research directions in this domain, aiming to offer valuable research perspectives for individuals involved in the field of neural network acceleration using FPGA.[10]

2.3.6 FPGA implementation for CNN-based optical remote sensing object detection

In recent years, optical remote sensing object detection has witnessed widespread application of convolutional neural network (CNN)-based methods, showcasing impressive performance. Aerospace systems, including satellites and aircraft, often employ these methods to observe ground objects. However, due to constrained logical resources and power budgets in these systems, the adoption of embedded devices becomes an attractive option for implementing CNN-based methods. Striking a balance between performance and power consumption remains a challenge.[11]

This paper presents an efficient hardware-implementation approach for optical remote sensing object detection. Initially, we optimize the CNN-based model for hardware implementation, laying the groundwork for effectively mapping the network onto a field-programmable gate array (FPGA). Additionally, we introduce a hardware architecture tailored for CNN-based remote sensing object detection. This architecture incorporates a general processing engine (PE) designed to execute various convolutions in the network using a uniform module. A streamlined data storage and access scheme is proposed, achieving low-latency calculations and a high memory bandwidth utilization rate.[11]

To validate their approach, they deploy the enhanced YOLOv2 network on a Xilinx ZYNQ xc7z035 FPGA. Experimental results reveal that the FPGA implementation achieves a performance only 0.18% lower than that on a graphics processing unit (GPU) in mean average precision (mAP). Operating at a 200 MHz frequency, our design attains a throughput of 111.5 giga-operations per second (GOP/s) with a 5.96 W on-chip power consumption. [11]

2.3.7 FPGA-Based accelerators of deep learning networks for learning and classification: A review

With recent advancements in digital technologies and the availability of reliable data, the field of artificial intelligence has witnessed the emergence of deep learning, showcasing its effectiveness in addressing complex learning challenges previously deemed insurmountable. Convolutional neural networks (CNNs), in particular, have proven highly effective in applications such as image detection and recognition.

However, the intensive CPU operations and memory bandwidth requirements pose challenges for general-purpose CPUs to achieve desired performance levels.[12]

As a remedy, hardware accelerators utilizing application-specific integrated circuits, field-programmable gate arrays (FPGAs), and graphic processing units have been increasingly employed to enhance CNN throughput. Notably, FPGAs have gained traction for accelerating deep learning network implementations due to their capacity to maximize parallelism and energy efficiency. This paper undertakes a comprehensive review of recent techniques for accelerating deep learning networks on FPGAs. Emphasis is placed on the key features employed by these techniques to enhance acceleration performance. Additionally, the paper provides recommendations for optimizing FPGA utilization in CNN acceleration.[12]

The techniques examined in this paper capture the latest trends in FPGA-based accelerators for deep learning networks, serving as a valuable reference for future advancements in efficient hardware accelerators. This work is anticipated to guide future research endeavors and prove beneficial for researchers delving into the realm of deep learning.[12]

2.3.8 Hardware implementation of neural network-based engine model using FPGA

This paper implements an artificial neural network (ANN)-based engine model using the Field Programmable Gate Array (FPGA). The developed (ANN)-based engine model will be used to estimate the engine gas emissions to mitigate the harmful effects of these emissions on human health. Getting reliable and robust FPGA-based ANNs implementations depends on the optimal choice of activation function that will provide minimal area occupation on FPGA. This study introduces, implements, and investigates FPGA-based ANN-based engine models using five different activation functions. These implemented engine models were described using MATLAB/Simulink and hardware description language coder and carried out by Spartan -3E-500.CP132 FPGA platform from Xilinx. The performance of the implemented engine models was investigated in terms of area-efficient implementation and the regression values (R) to build a robust ANN-based engine model.[13]

2.3.9 Summary of other Literature Reviews

Table 4: Literature Review

Index	Model	Target	Accuracy	LUT FF	Power (W)	Memory (RAM)	DSP
[14]	CNN	FPGA (Xilinx Zedboard)	Dataset: MNIST Accuracy : 97%	25436 -	-	257	188
[15]	CNN	FPGA (xc7vx980t of the Virtex-7 family)	-	57438 79327	3.25	-	937
[16]	CNN	FPGA (Xilinx ZYNQ 7100)	-	142291 187146	4.083	708	1926
[17]	CNN MobilеNet	FPGA (XCZU7EV)	87% top-5	118233 128614	7.35	532	340
[18]	NN	FPGA (Xilinx Artix 7 xc7a35t)	97.25%	10678 16568	-	9	6
[19]	NN	FPGA (VIRTEX-7 FPGA)	-	26499 -	-	12	126

[20]	NN	FPGA	96.0%	(ReLU) 7086 6653 (Sigmoid) 6256 5297	-	(ReLU) 15 (Sigmoid) 35	(ReLU) 160 (Sigmoid) 160
[21]	CNN	FPGA (Virtex-6 of part number XC6VLX130T-2 FPGA)	-	1 PE 339 Register	-	-	1 PE 16

2.4 Research gap

Review of FPGA-Based Accelerators of Deep Convolutional Neural Networks, by Philip and Sivamangai states 4 aspect that we can focus on to push the state of art. Optimize the remaining computation process where only a few experts are currently working on the activation part of the matrix operation because most of the research focuses on the loop. Next, is to access optimization. There needs to be more research on other data access optimization methods. FPGA integration. As illustrated in the paper, managing scheduling and allocation issues properly can lead to better performance on a multi-FPGA cluster. Moreover, there is not much research in this area at the moment. Hence, this direction deserves further exploration. Finally the automatic configuration. If it was possible to deploy applications on FPGAs more easily, such as NVIDIA’s CUDA (Compute Unified Device Architecture), complex programming could be eliminated.[22]

2.5 Summary

The literature review delves into existing research on integrating ARM processors and FPGAs for high-performance Convolutional Neural Networks (CNNs). It covers foundational CNN concepts, surveys relevant literature on FPGA-based neural networks, and identifies gaps for the current study. The background study contextualizes the research within the broader hardware acceleration landscape, emphasizing the rationale for choosing ARM processors and FPGAs. Overall, this chapter provides a foundation for the thesis objectives, including FPGA-based CNN design, optimization exploration, and comprehensive performance analysis.

CHAPTER 3

METHODOLOGY

3.1 Introduction

In pursuit of the aforementioned research goal, the methodology employed in this study encompasses the comprehensive development and implementation of a Convolutional Neural Network (CNN) designed for image classification. Emphasizing the convergence of machine learning and hardware acceleration, this methodology seeks to optimize the deployment of the trained CNN model on an FPGA, leveraging the capabilities of the PYNQ Z1 board. The subsequent sections delineate the step-by-step processes involved in data collection, model training, High-Level Synthesis (HLS), FPGA implementation, validation, and performance evaluation.

The methodology will be covering on how the project is being conducted. The first step is to select suitable data and then preprocessing of the data. The next step would be creating a CNN model and a quantized CNN model. Next, Prune and train both of the model before testing them to get the performance of the models. After that, the models is converted to HDL using HLS to prepare the models to be implemented on the FPGA. Testing of the HLS models is then conducted before the bitstream generation of the models. Then, the bitfiles of both model is transferred to the FPGA and perform validation. A thorough analysis will be conducted to provide insight on the FPGA based CNN.

3.2 Approach to the project

3.2.1 Dataset selection

The Street View House Numbers (SVHN) dataset[23] is a good datasets for performing image recognition and classification tasks using the CNN. The reason for SVHN dataset to be good for the tasks is because it is real-world images of house numbers from Google Street View and this makes the dataset relevant to practical applications, such as optical character recognition in street-level images. The data quality of the SVHN datasets is a key for its advantage because it contains noise such as light diversity and noise on the image collected. The quality of data which is closest to the real-life environment makes sure that the model is able to adapt to images captured in real time. MNIST datasets is also 1 of the famous datasets to be used for the same reasons, but SVHN dataset is selected for this project to provide a more realistic approach to the image recognition and classification tasks since SVHN is a dataset collected in real life. The MNIST and SVHN is one of the benchmark datasets for models on digit recognition and classification. If the model successfully implemented on the SVHN, the capabilities of the model is shown.

SVHN dataset contains 72k images for training and 20k images for testing. The sufficient amount of data in a dataset is a key factor for a good model designation. It contains image with single digit and multiple digits with different colours, fonts and background. The variety of features in the datasets prevents the model from overfitting and achieve a higher precision with a good configuration of model designation.

The SVHN dataset is preferred for CNN training due to its real-world relevance, large and diverse nature, varied digit appearances, provision of bounding box information, preprocessing challenges, benchmarking value, accessibility, and educational benefits. Figure 3.1 shows the data in the SVHN datasets with its labels.

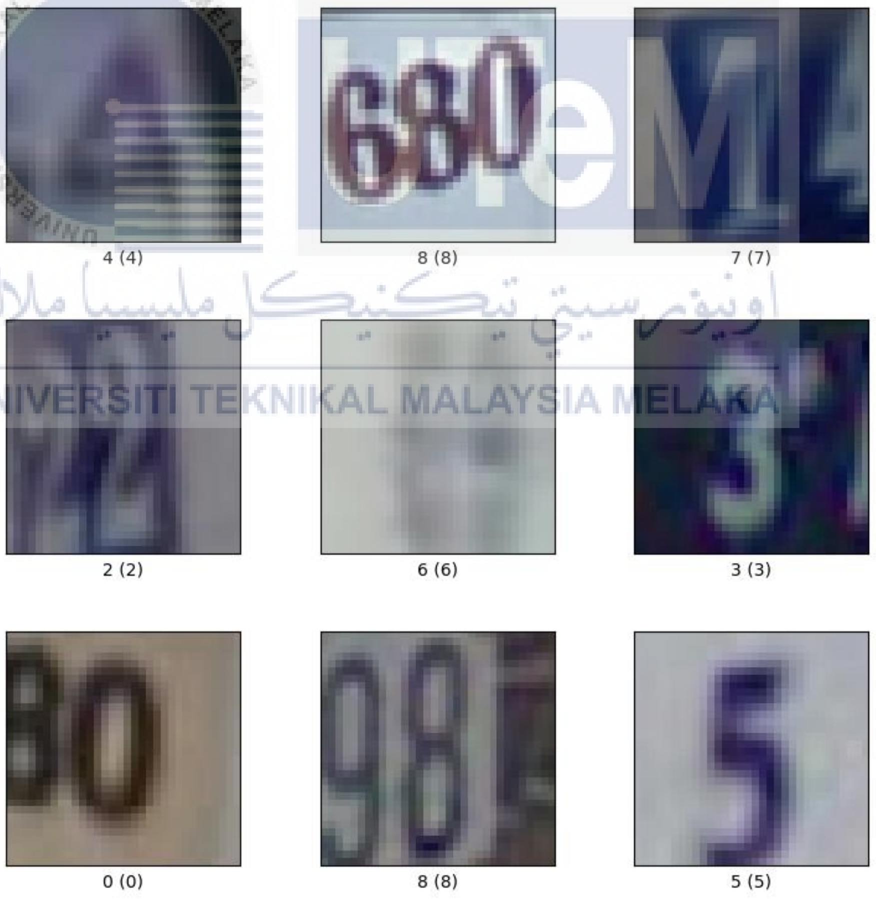


Figure 3.1: SVHN dataset

3.2.2 HLS4ML

hls4ml is a Python package for machine learning inference in FPGAs. The library create firmware implementations of machine learning algorithms using high level synthesis language (HLS). It can be used translate traditional open-source machine learning package models into HLS that can be configured for needed case based on users needs. [24] Workflow of the HLS4ML is shown in Figure 3.2.

With the help of open-source tools like PyTorch and Keras, machine learning models can be quickly and efficiently translated into high level synthesis (HLS) code, which can be transpiled and executed on an FPGA. This is the aim of hls4ml. Subsequently, the HLS project can be utilised to generate an IP that can be integrated into intricate designs or employed to develop a kernel for co-processing CPUs. Many of the parameters of the algorithm can be freely defined by the user to best suit their needs.[24] The hls4ml package makes it possible to quickly prototype the implementation of a machine learning algorithm in FPGAs, which significantly shortens the time it takes to get results. It also provides users with guidance on how to best design a machine learning algorithm for their application while balancing latency, resource consumption, and performance requirements. [24]

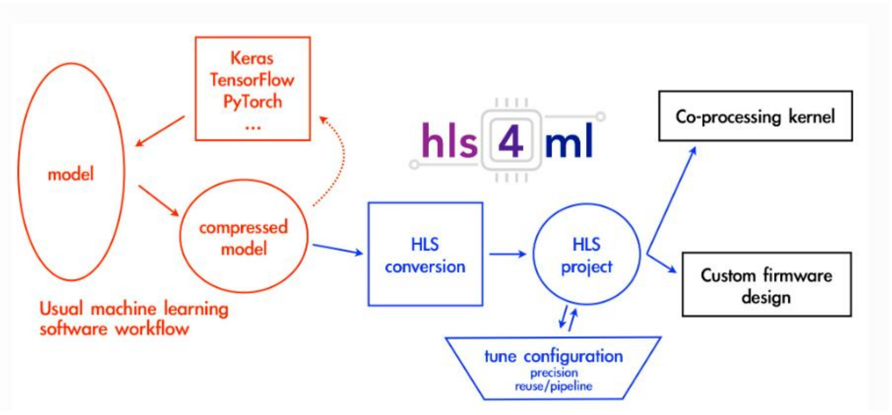


Figure 3.2: HLS4ML workflow

An FPGA can be specifically programmed to do a certain task, in this case evaluate neural networks given a set of inputs, and as such can be highly optimized for the task, with tricks like pipelining and parallel evaluation. However, this means dynamic remapping while running isn't really a possibility. FPGAs also often come at a comparatively low power cost with respect to CPUs and GPUs. This allows hls4ml to build HLS code from compressed neural networks that results in predictions on the microsecond scale for latency. The hls4ml tool saves the time investment needed to convert a neural network to a hardware design language or even HLS code, thus allowing for rapid prototyping. [24]

In summary, HLS4ML serves as a bridge between high-level machine learning model development and efficient FPGA implementations. By automating the translation process and dealing with FPGA-specific optimizations, HLS4ML eases the deployment of machine learning models onto hardware, providing a valuable tool for developers seeking to leverage FPGA acceleration.

3.2.3 PYNQ Z1 board implementation

With the help of PYNQ, a brand-new open-source framework, embedded programmers can take advantage of the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to create programmable logic circuits by using the PYNQ-Z1 board. Rather, Python is used to programme the APSoC, and the PYNQ-Z1 is used to test and develop the code. The process of importing and programming programmable logic circuits is much the same as that of software libraries; they are imported as hardware libraries and programmed via their respective APIs.

PYNQ provides a high-level programming abstraction, allowing developers to program the FPGA using Python. It provides the Jupyter Notebook in the PYNQ board image. This allows the designers to write code and algorithms in Python directly in the board which solves the challenge of needing to write HDL code on a FPGA as they need to in the normal FPGA. PYNQ board uses the Zynq System on Chip which combines multiple features in a single chip but still able to perform the same desired function with multiple chips on board. ARM processor is also in the Zynq SoC. This enables hardware acceleration with the CPU, DSP and other components all on the same chip or board. The flexibility of the PYNQ board is also its advantage for reprogramming the SoC following the needs.

To conclude, the PYNQ z1 board is chosen for its flexibility, reprogrammable ability and lower challenge compare to the other FPGA. The Python interface with jupyter Notebook built in the image of PYNQ and its Zynq SoC provides a huge advantage to implement CNN on the board for this project. The physical board is shown in Figure 3.3 and its architecture is shown in Figure 3.4.

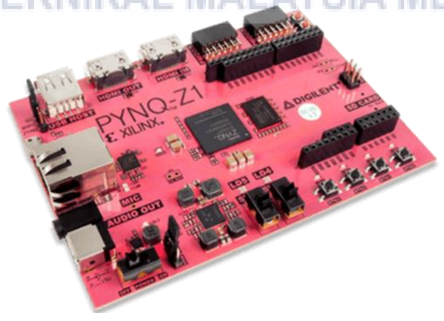


Figure 3.3: PYNQ Z1 board

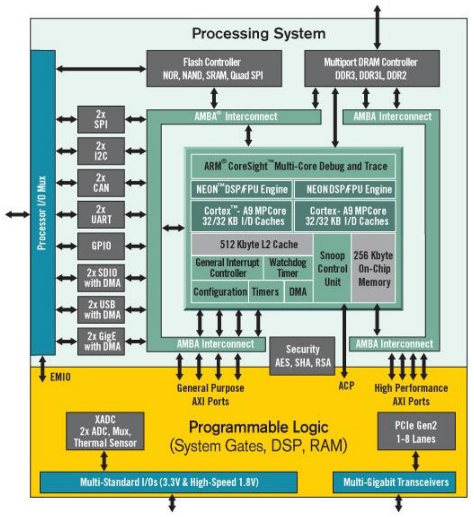

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA Interconnect
AMBA Interconnect
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
ARM CoreSight Multi-Core Debug and Trace
NEON DSP/FPU Engine
NEON DSP/FPU Engine
Cortex- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 3.4: Zynq APSoC architecture

3.3 Proposed Method Flowchart

The flowchart of this project is shown in Figure 3.5. The process starts with the research on the background and a thorough literature review on the project. After that, the dataset selection and the preprocessing of the data is conducted. After that, the CNN model is build on a Docker container. The Docker container is installed in the Ubuntu system on host machine, the reason to use the docker container because it is a isolated, standalone package to run all the applications. This saves the time and challenges to install software such as Vivado into the host machine, where installation of the software can be done on a single script with the command to do it all.

The second objectives is to optimize the model created in the steps before. Quantization of the CNN model is created and the pruning with the training process is conducted. The model is then trained and tested using SVHN datasets. The configuration of the model is tuned until the accuracy reaches above 80% for the training and testing. Testing process makes sure the model is generalized for any new data can be classified correctly by the model. After the accuracy reaches the desired level which is 80%, the HLS conversion of both of the models is conducted using the hls4ml library.

The HLS conversion and configuration step optimizes the model for hardware implementation. This is followed by simulation and synthesis to test the model's functionality before actual deployment. Bitstream generation is conducted after the synthesis process runs without any critical error. The model is then deployed on board by transferring the bitfile, hardware configuration file and the AXI stream driver to the PYNQ z1 board and validated. If the accuracy remains 80% or above, the process proceeds. If not, adjustments are made in the previous steps and the model is retested.

The report of the implementation and the accuracy of all the processes is recorded and analyzed. These steps provide an insight on the implementation of CNN on the PYNQ Z1 board.

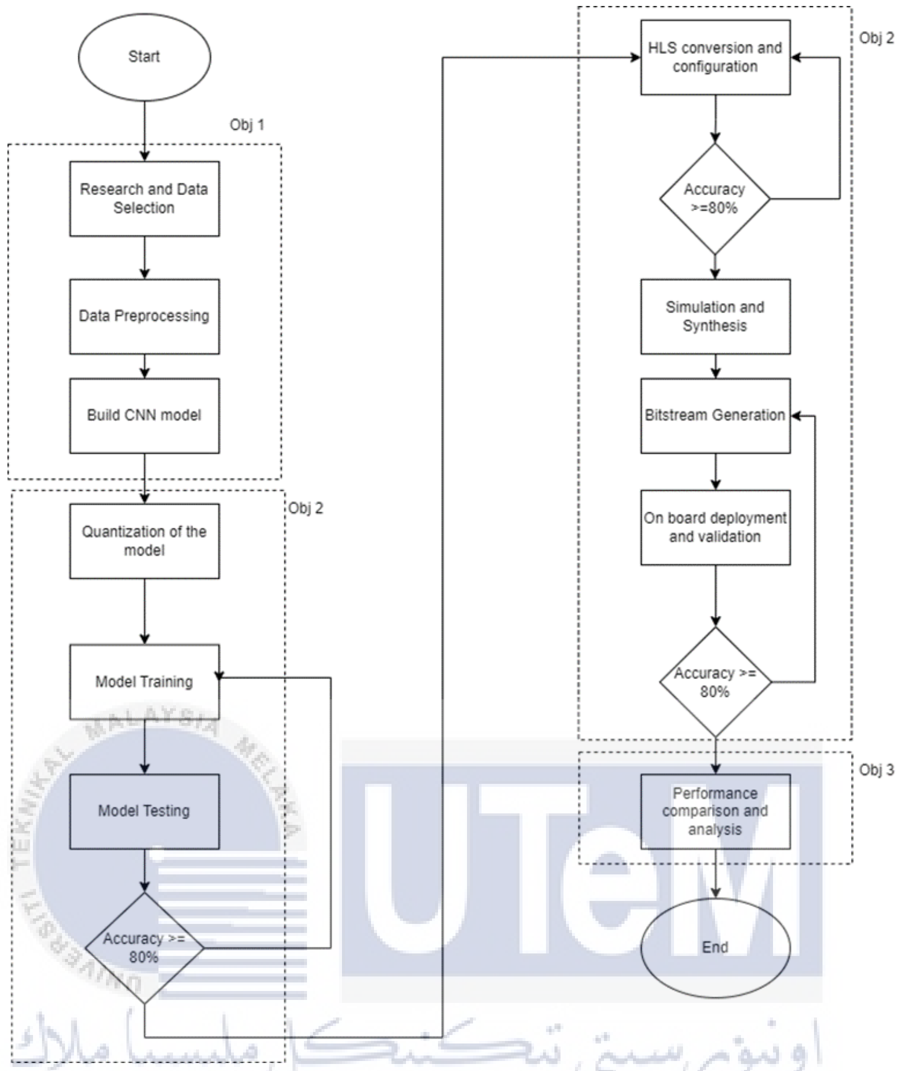


Figure 3.5: Flowchart

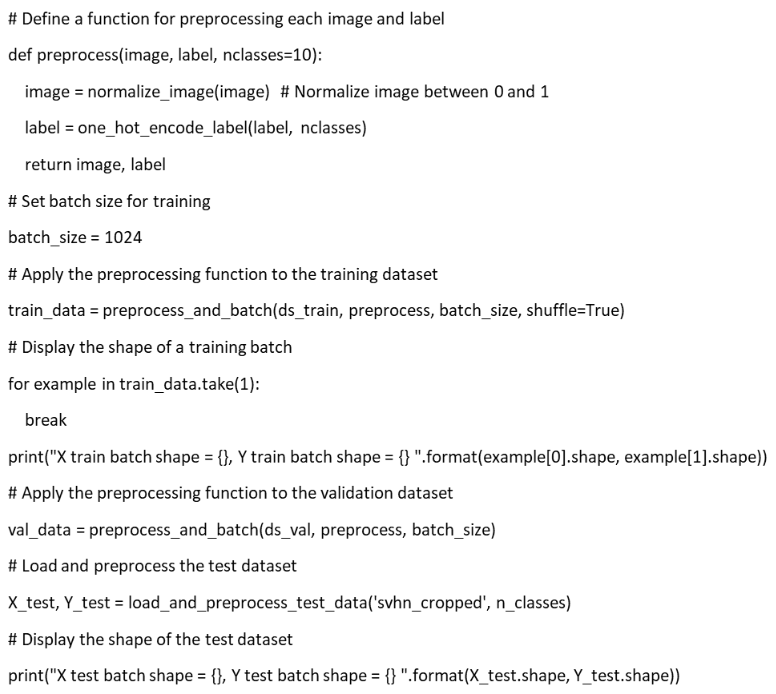
3.3.1 Data Preprocessing

The pseudocode of the data preprocessing is shown in Figure 3.6. The SVHN dataset is provided by the TensorFlow Dataset. After loading the dataset, the preprocessing step is conducted. The dataset is first split into 90/10 where 90% of the data will be used in the training and the remaining will be the validation datasets.

A preprocess function is build for doing the preprocessing of datasets. The function starts by normalizing the pixel values which are normally in 8 bit image[0 255] into a standardized floating points ranging from [0 1]. Normalization needs to be done to make sure that training process of the training process runs steadily to achieve good convergence and performance of the model. The next step is to encode the class label convert the integer-encoded labels into a one-hot encoded representation. Any unnecessary dimensions in removed from the label tensor. The encoding process generates binary vector representation of the data. For example for image with image 3 in the SVHN datasets which has 10 classes (0-9), the representation would be [0 0 0 1 0 0 0 0 0 0]. This format enables a better interpretation of the image or data for the CNN model.

Next, we let the training dataset to undergo the preprocess stage to achieve the format that is reliable to be used in the CNN model. The preprocessed data is then shuffled, batched and prefetch to increase the performance of the training process. Shuffling of the data avoids bias and introduce randomness to the model. The bath size is set to 1024 for training purposes. The validation dataset also undergoes the preprocessed stage for the same purpose. The testing dataset is fetch and undergo the preprocessing stage as well to ensure the consistency of the data fed into the model.

In summary, the datasets undergo preprocessing for a better training and testing process so that the CNN model is able to do the digit recognition task from the image in the dataset. It meticulously handles preprocessing intricacies, batch organization, and prefetching, thereby laying a robust foundation for subsequent model training and evaluation endeavors



```
# Define a function for preprocessing each image and label
def preprocess(image, label, nclasses=10):
    image = normalize_image(image)  # Normalize image between 0 and 1
    label = one_hot_encode_label(label,  nclasses)
    return image, label
# Set batch size for training
batch_size = 1024
# Apply the preprocessing function to the training dataset
train_data = preprocess_and_batch(ds_train, preprocess, batch_size, shuffle=True)
# Display the shape of a training batch
for example in train_data.take(1):
    break
print("X train batch shape = {}, Y train batch shape = {} ".format(example[0].shape, example[1].shape))
# Apply the preprocessing function to the validation dataset
val_data = preprocess_and_batch(ds_val, preprocess, batch_size)
# Load and preprocess the test dataset
X_test, Y_test = load_and_preprocess_test_data('svhn_cropped', n_classes)
# Display the shape of the test dataset
print("X test batch shape = {}, Y test batch shape = {} ".format(X_test.shape, Y_test.shape))
```

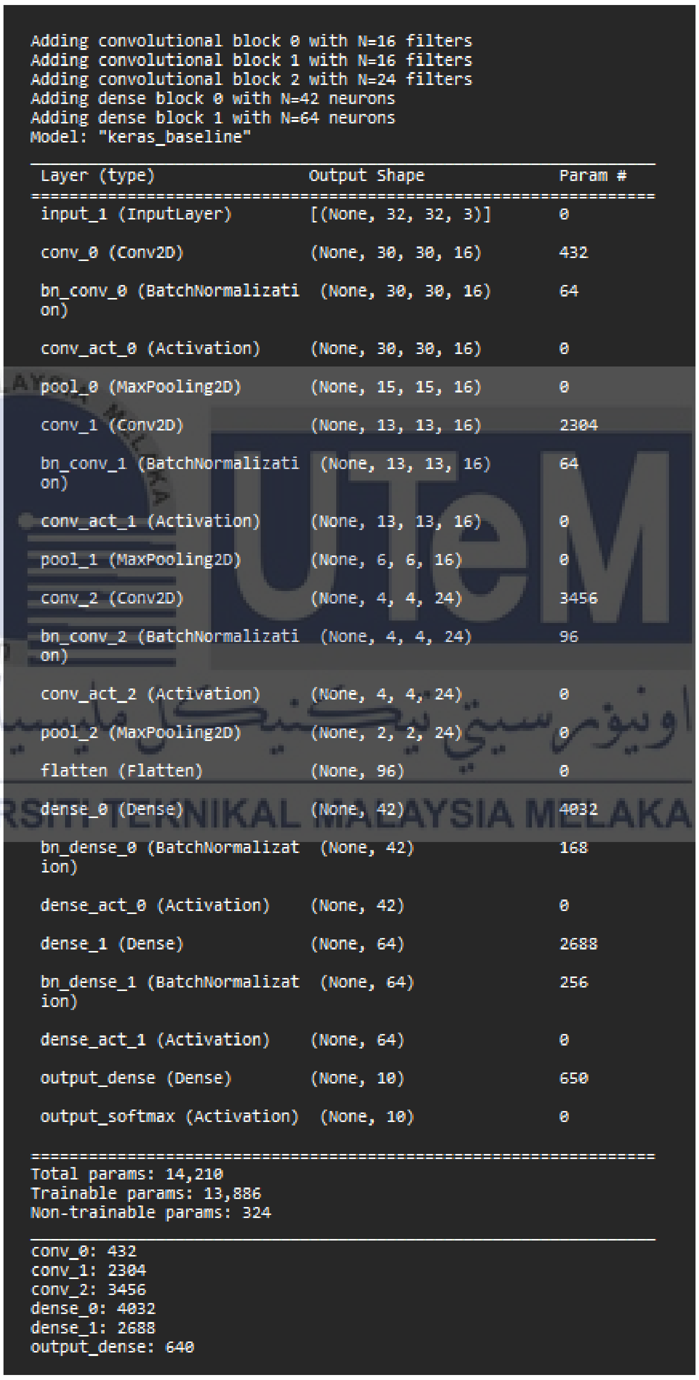
Figure 3.6: Pseudocode of the Data Preprocessing

3.3.2 Model Development

A convolutional neural network (CNN) architecture is meticulously crafted using the Keras API in conjunction with the TensorFlow backend.[25] The architecture unfolds through a series of convolutional blocks, where each block encapsulates a 3x3 convolutional layer, batch normalization for stability, rectified linear unit (ReLU) activation for introducing non-linearity, and 2x2 max pooling to downsample spatial dimensions is shown in Figure 3.7. The configuration of each convolutional block, specifically the number of filters, is specified by the filters_per_conv_layer list.

Following the convolutional blocks, the flattened output is directed through dense blocks, signifying fully connected layers. Each dense block is composed of a dense layer, batch normalization to mitigate internal covariate shift, and ReLU activation for non-linearity. The number of neurons in each dense block is determined by the neurons_per_dense_layer list.

The final dense layer outputs logits corresponding to the number of classes (n_classes), and a softmax activation function is applied to obtain class probabilities. The model is compiled for training using categorical cross-entropy as the loss function and the Adam optimizer.



```
Adding convolutional block 0 with N=16 filters
Adding convolutional block 1 with N=16 filters
Adding convolutional block 2 with N=24 filters
Adding dense block 0 with N=42 neurons
Adding dense block 1 with N=64 neurons
Model: "keras_baseline"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv_0 (Conv2D)             (None, 30, 30, 16)        432

 bn_conv_0 (BatchNormalizati  (None, 30, 30, 16)       64
 on)

 conv_act_0 (Activation)     (None, 30, 30, 16)        0

 pool_0 (MaxPooling2D)       (None, 15, 15, 16)        0

 conv_1 (Conv2D)             (None, 13, 13, 16)        2304

 bn_conv_1 (BatchNormalizati  (None, 13, 13, 16)       64
 on)

 conv_act_1 (Activation)     (None, 13, 13, 16)        0

 pool_1 (MaxPooling2D)       (None, 6, 6, 16)          0

 conv_2 (Conv2D)             (None, 4, 4, 24)          3456

 bn_conv_2 (BatchNormalizati  (None, 4, 4, 24)         96
 on)

 conv_act_2 (Activation)     (None, 4, 4, 24)          0

 pool_2 (MaxPooling2D)       (None, 2, 2, 24)          0

 flatten (Flatten)           (None, 96)                0

 dense_0 (Dense)             (None, 42)                4032

 bn_dense_0 (BatchNormalizat  (None, 42)               168
 ion)

 dense_act_0 (Activation)    (None, 42)                0

 dense_1 (Dense)             (None, 64)                2688

 bn_dense_1 (BatchNormalizat  (None, 64)               256
 ion)

 dense_act_1 (Activation)    (None, 64)                0

 output_dense (Dense)        (None, 10)                650

 output_softmax (Activation)  (None, 10)               0

=================================================================
Total params: 14,210
Trainable params: 13,886
Non-trainable params: 324
_________________________________________________________________
conv_0: 432
conv_1: 2304
conv_2: 3456
dense_0: 4032
dense_1: 2688
output_dense: 640
```

Figure 3.7: Architecture of CNN model

3.3.3 Pruning

Pruning is a technique employed to reduce the size of neural networks by eliminating certain weights, thereby promoting sparsity. This is done with the goal of achieving a more efficient model in terms of memory usage, inference speed, and potentially improving generalization performance.

The pruning process is done with a custom designed function, with applying weight pruning to the layers in the CNN. The pruning process is determined by the PolynomialDecay function provided by TensorFlow which will increase the sparsity from 0 until 50% sparsity is reached for the models. The function will increase the sparsity every 2 epoch during the training process.

The pruneFunction is a custom function designed to apply weight pruning to specific layers of the neural network. The pruning schedule is determined by the PolynomialDecay function from TF-MOT, which gradually increases sparsity from an initial value of 0.0 to a final sparsity level of 50%. This increase occurs every 2 epochs, starting at the specified begin_step and ending at the end_step, with a specified frequency determined by frequency. The layers that will be pruned is the convolutional layers and the dense layers. These layers are pruned since they have the most complexity and pruning them can eventually lead to a significant reduction in model size without affecting the performance of the model. Pruning is done by setting a portion of weight to 0 gradually during the training process so that the model can adapt to the sparsity constraint.

Finally the pruned model is created after the training process of the model. This is to create a model that is smaller so that it is more resource efficient. This optimization step is crucial since we are to deploy the model in the PYNQ Z1 board which is a

FPGA that has tighter resource constraint and limited computational power. With a pruned model, the model could have a faster model inference and development. The pruning process is shown in Figure 3.8.

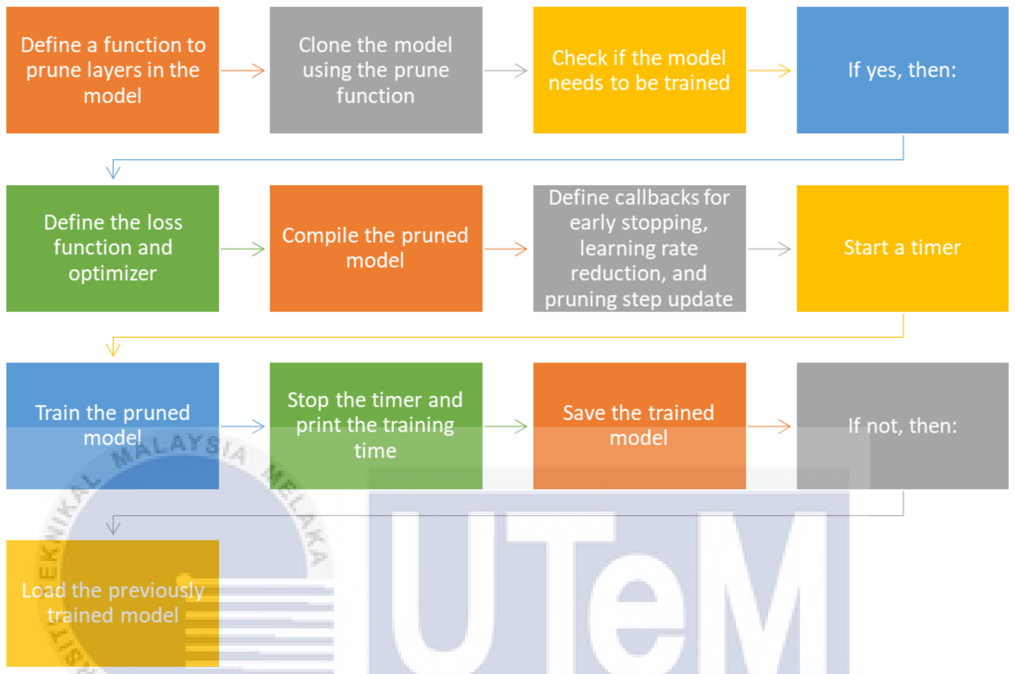

Define a function to prune layers in the model
Clone the model using the prune function
Check if the model needs to be trained
If yes, then:
Define the loss function and optimizer
Compile the pruned model
Define callbacks for early stopping, learning rate reduction, and pruning step update
Start a timer
Train the pruned model
Stop the timer and print the training time
Save the trained model
If not, then:
Load the previously trained model

Figure 3.8: Pruning Process

3.3.4 Model Training

The training process is conducted following a logic statement, where the training will only conducted if the statement is True whereas when it is false, the training function will only load the pre-saved model in the previous session.

The training is done with the following steps, specifying the loss function which is the categorical cross-entropy in my project and the Adam optimizer with the learning rate, beta values, epsilon and the ANSGrad. The model undergoes 30 epoch or iterations of training with callbacks such as early stopping and learning rate reduction for training efficiency. After the training has been done, the pruned and trained model is then saved as a h5 file.

3.3.5 Model Testing

The testing process is conducted after the training process to see if the model is well trained. The accuracy desired is above 80%. The testing input data is defined as X_test and the ground truth label is defined as Y_test. The data is then put in the trained model for prediction of the class probabilities and performance of the model is recorded.

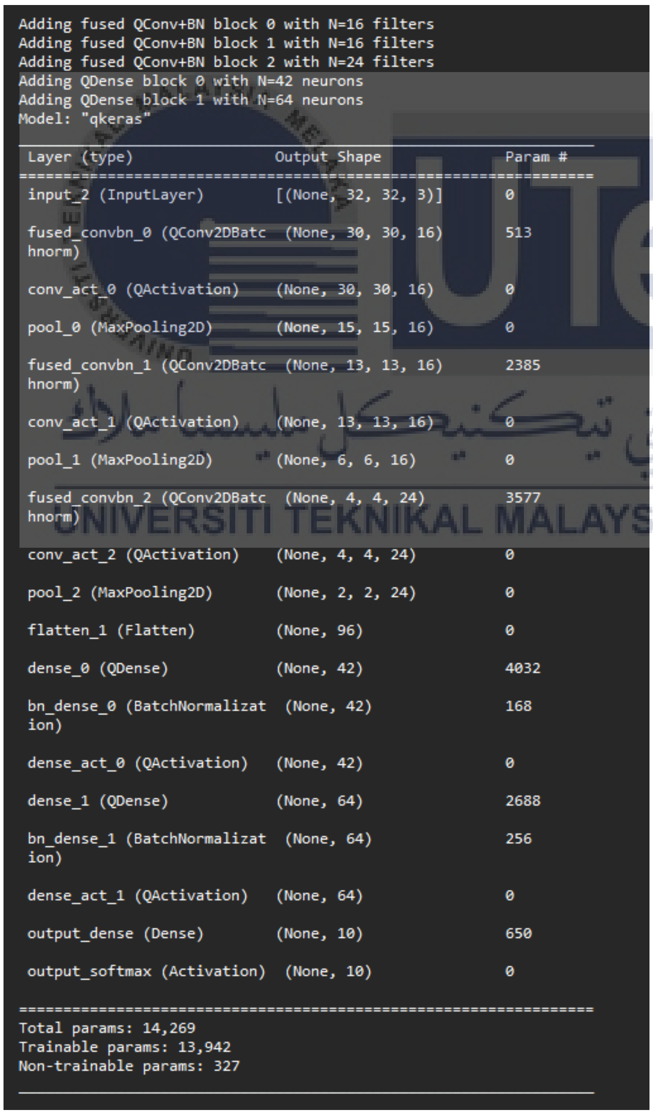
3.3.6 Quantization

Quantization is an optimization technique which reduces the precision of weights and activations in the neural network. Quantization can improve the model efficiency and reduce the resource utilization which may leads to hardware inference and implementation which is the aim of this project.

The quantization of the CNN model is done by using the QKeras library which provides the simplicity to use the quantized layer in model developing. The development process of the CNN model is the same as the non-quantized model with just using the function QConv and QDense while developing the CNN model. By using this function, the quantized convolutional and dense layers of CNN is created. The kernel and bias for the convolutional layer is set to 6 bits quantization with 0 fractional bits. The activation used in the quantized model is also a 6 bits ReLU activation function. The configuration of the Dense layer is also set to 6 bits quantization and 0 fractional bit. With these configuration, a qkeras which is a quantized CNN model is created.

The model is then gone through the same process as the not quantized model which is training, pruning and testing. The architecture of the quantized model is shown in the Figure 3.9.

Quantization is performed to leverage the benefits of reduced precision arithmetic, leading to more memory-efficient models and potentially faster inference on hardware with support for quantized operations. While quantization introduces a trade-off between model accuracy and computational efficiency, it is particularly useful in scenarios where computational resources are constrained, such as edge devices and embedded systems. The specific choice of quantization parameters, such as the number of bits and alpha values, allows for a balance between model efficiency and performance.



```
Adding fused QConv+BN block 0 with N=16 filters
Adding fused QConv+BN block 1 with N=16 filters
Adding fused QConv+BN block 2 with N=24 filters
Adding QDense block 0 with N=42 neurons
Adding QDense block 1 with N=64 neurons
Model: "qkeras"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

 fused_convbn_0 (QConv2DBatc  (None, 30, 30, 16)       513
 hnorm)

 conv_act_0 (QActivation)    (None, 30, 30, 16)        0

 pool_0 (MaxPooling2D)       (None, 15, 15, 16)        0

 fused_convbn_1 (QConv2DBatc  (None, 13, 13, 16)       2385
 hnorm)

 conv_act_1 (QActivation)    (None, 13, 13, 16)        0

 pool_1 (MaxPooling2D)       (None, 6, 6, 16)          0

 fused_convbn_2 (QConv2DBatc  (None, 4, 4, 24)         3577
 hnorm)

 conv_act_2 (QActivation)    (None, 4, 4, 24)          0

 pool_2 (MaxPooling2D)       (None, 2, 2, 24)          0

 flatten_1 (Flatten)         (None, 96)                0

 dense_0 (QDense)            (None, 42)                4032

 bn_dense_0 (BatchNormalizat  (None, 42)               168
 ion)

 dense_act_0 (QActivation)   (None, 42)                0

 dense_1 (QDense)            (None, 64)                2688

 bn_dense_1 (BatchNormalizat  (None, 64)               256
 ion)

 dense_act_1 (QActivation)   (None, 64)                0

 output_dense (Dense)        (None, 10)                650

 output_softmax (Activation)  (None, 10)               0

=================================================================
Total params: 14,269
Trainable params: 13,942
Non-trainable params: 327
_________________________________________________________________
```

Figure 3.9: Quantized CNN model Architecture

3.3.7 Visualization

A Receiver Operating Characteristic (ROC) curve is created for a multi-class classification task, comparing the performance of a baseline Keras model with a quantized Keras (QKeras) model. The ROC curve is a graphical representation of the trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity) at various classification thresholds. This comparison is performed for each class in the classification task. For each class label, the true positive rate (tpr) and false positive rate (fpr) are calculated and the area under the ROC curve (AUC) is also computed.

This visualization serves to compare the performance of the baseline Keras model with the quantized Keras model. A higher AUC value generally indicates better model performance in distinguishing between positive and negative instances. The comparison also extends to accuracy metrics, providing a good evaluation of the models.

3.4 Implementation

3.4.1 HLS conversion

The pruning layer is removed to make sure that the appropriate custom layer is passed. The hls4ml library is utilized to convert and compile a pruned convolutional neural network (CNN) model, which was originally defined in Keras, into a hardware description language (HDL) representation suitable for FPGA (Field-Programmable Gate Array) deployment.

Configuring hls4ml using the `hls4ml.utils.config_from_keras_model` function, extracting configuration parameters from the Keras model is needed to be done as the first step. The granularity is set to 'name,' meaning that each layer's configuration is specified by its name. The precision for the entire model is set to 'ap_fixed<16,6>,' indicating a fixed-point representation with 16 bits (total width) and 6 bits for the fractional part. The reuse factor is set to 1, meaning that the hardware design aims to fully parallelize the computation for each layer. For each layer, the strategy is set to 'Latency,' suggesting that the design aims to minimize the latency (execution time) for that layer. Additionally, the strategy for the output layer ('output_softmax') is set to 'Stable' to ensure better numerical stability, especially for high-accuracy models. A configuration dictionary (`cfg`) is created for hls4ml conversion, specifying parameters such as the backend ('Vivado'), IO type ('io_stream'), the HLS configuration (`hls_config`), the Keras model (`model`), the output directory ('pruned_cnn/'), and the target Xilinx FPGA part. The `hls4ml.converters.keras_to_hls` function is called to convert the Keras model into an HLS representation using the provided configuration (`cfg`).The resulting HLS model is then compiled using the `compile` method.

The goal is to prepare a pruned CNN model for deployment on an FPGA by converting it into an HLS representation. The configuration settings, such as precision, reuse factor, and layer-wise strategies, are crucial for optimizing the hardware design based on the characteristics of the model and the target FPGA device. The resulting HLS model can be further synthesized and implemented on an FPGA for efficient and high-performance inference. After that, we check the accuracy of the hls4ml model and plot the ROC for it. The architecture of the HLS models is shown in Figure 3.10 and Figure 3.11

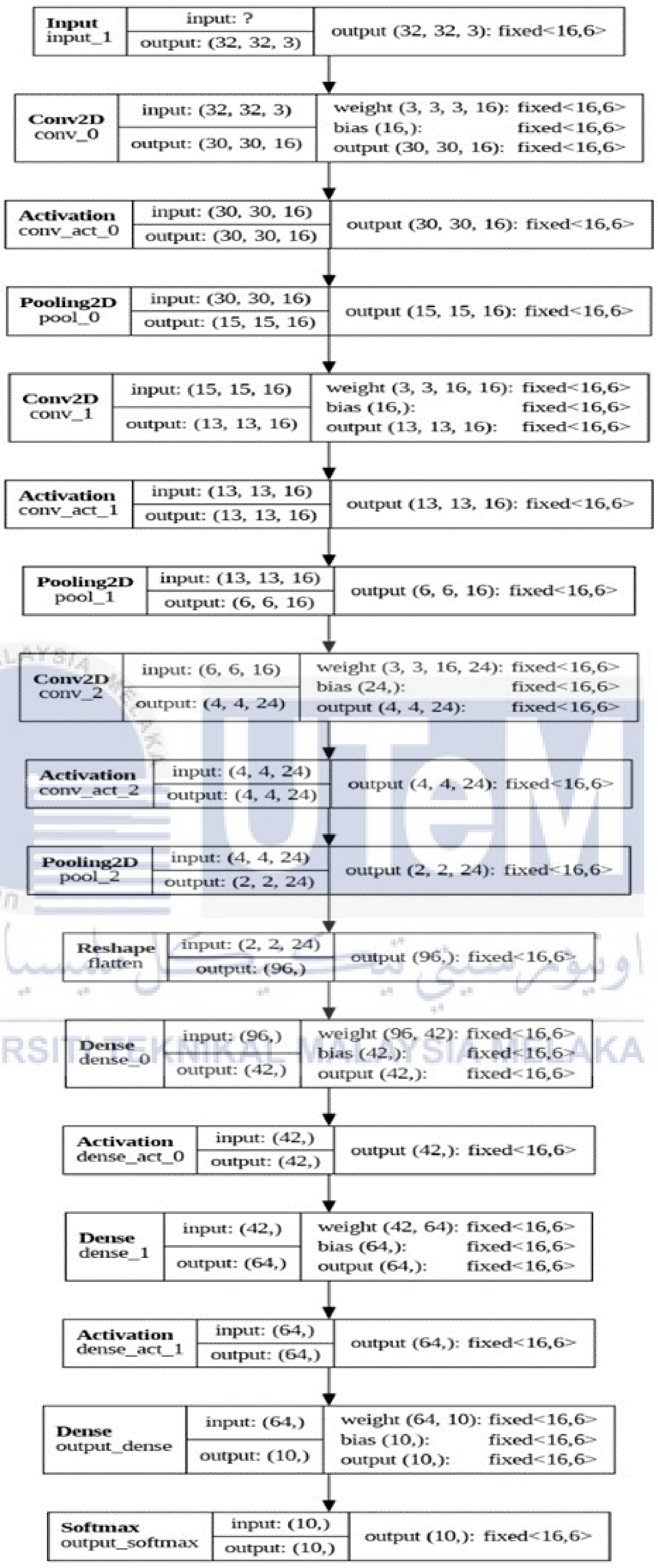


Figure 3.10: HLS Pruned CNN model architecture

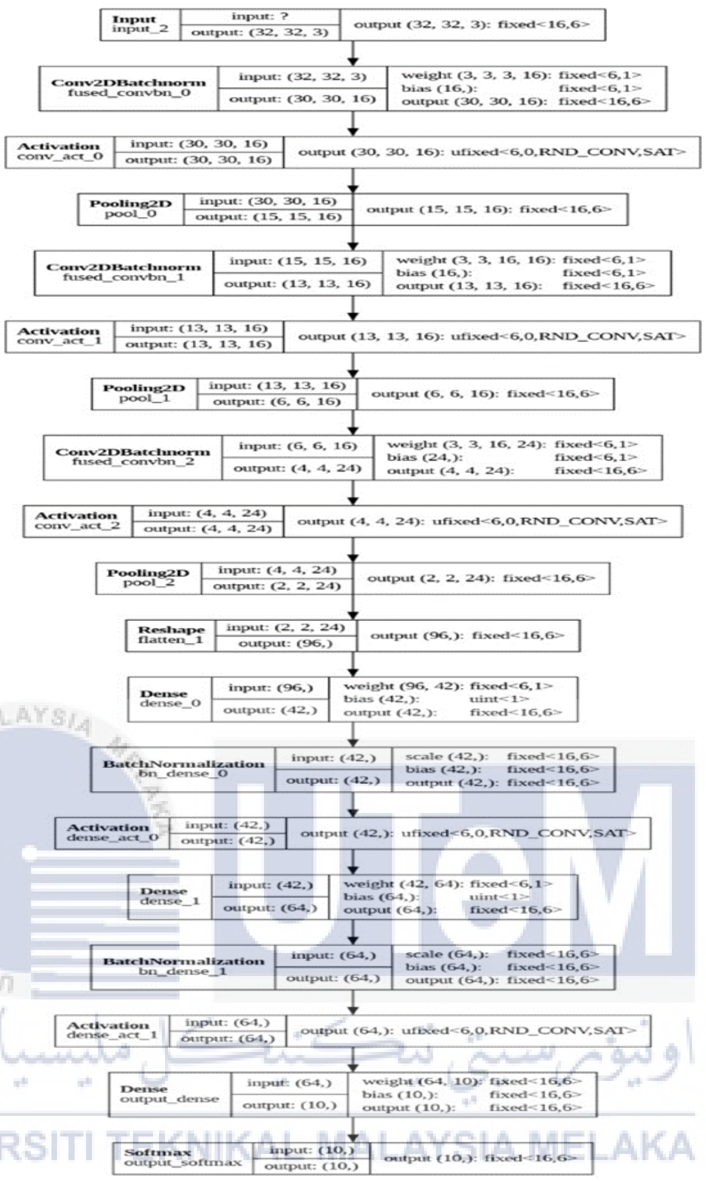

Input input_2 | input: ? | output: (32, 32, 3) | output (32, 32, 3): fixed<16,6>
Conv2DBatchnorm fused_convbn_0 | input: (32, 32, 3) | output: (30, 30, 16) | weight (3, 3, 3, 16): fixed<6,1> bias (16,): fixed<6,1> output (30, 30, 16): fixed<16,6>
Activation conv_act_0 | input: (30, 30, 16) | output: (30, 30, 16) | output (30, 30, 16): ufixed<6,0,RND_CONV,SAT>
Pooling2D pool_0 | input: (30, 30, 16) | output: (15, 15, 16) | output (15, 15, 16): fixed<16,6>
Conv2DBatchnorm fused_convbn_1 | input: (15, 15, 16) | output: (13, 13, 16) | weight (3, 3, 16, 16): fixed<6,1> bias (16,): fixed<6,1> output (13, 13, 16): fixed<16,6>
Activation conv_act_1 | input: (13, 13, 16) | output: (13, 13, 16) | output (13, 13, 16): ufixed<6,0,RND_CONV,SAT>
Pooling2D pool_1 | input: (13, 13, 16) | output: (6, 6, 16) | output (6, 6, 16): fixed<16,6>
Conv2DBatchnorm fused_convbn_2 | input: (6, 6, 16) | output: (4, 4, 24) | weight (3, 3, 16, 24): fixed<6,1> bias (24,): fixed<6,1> output (4, 4, 24): fixed<16,6>
Activation conv_act_2 | input: (4, 4, 24) | output: (4, 4, 24) | output (4, 4, 24): ufixed<6,0,RND_CONV,SAT>
Pooling2D pool_2 | input: (4, 4, 24) | output: (2, 2, 24) | output (2, 2, 24): fixed<16,6>
Reshape flatten_1 | input: (2, 2, 24) | output: (96,) | output (96,): fixed<16,6>
Dense dense_0 | input: (96,) | output: (42,) | weight (96, 42): fixed<6,1> bias (42,): uint<1> output (42,): fixed<16,6>
BatchNormalization bn_dense_0 | input: (42,) | output: (42,) | scale (42,): fixed<16,6> bias (42,): fixed<16,6> output (42,): fixed<16,6>
Activation dense_act_0 | input: (42,) | output: (42,) | output (42,): ufixed<6,0,RND_CONV,SAT>
Dense dense_1 | input: (42,) | output: (64,) | weight (42, 64): fixed<6,1> bias (64,): uint<1> output (64,): fixed<16,6>
BatchNormalization bn_dense_1 | input: (64,) | output: (64,) | scale (64,): fixed<16,6> bias (64,): fixed<16,6> output (64,): fixed<16,6>
Activation dense_act_1 | input: (64,) | output: (64,) | output (64,): ufixed<6,0,RND_CONV,SAT>
Dense output_dense | input: (64,) | output: (10,) | weight (64, 10): fixed<16,6> bias (10,): fixed<16,6> output (10,): fixed<16,6>
Softmax output_softmax | input: (10,) | output: (10,) | output (10,): fixed<16,6>

Figure 3.11: HLS Quantized Pruned CNN model architecture

3.4.2 Simulation, Synthesis and Bitstream Generation

Simulation and synthesis is done using the command build from the hls4ml library. The estimated resource utilization report is generated for analysis purposes. The configuration of hls4ml is slightly different compared to the synthesis part where the backend used is the Vivado accelerator where the synthesis process uses the Vivado as backend. The bitfile, hardware handsoff, driver and testing data is archived into a tar.gz file and copied to the PYNQ board.

3.4.3 Deployment and Validation

The image of the board is downloaded and loaded into a SD card for the pynq board to work. After that, the NN inference can be done with the driver AXI and the output is saved for analysis. The accuracy on board is being compared with the host machine’s accuracy.

3.5 Analyzation performance

3.5.1 Inference Time:

In the context of FPGA deployment, the inference time is a critical metric reflecting the speed at which the model processes input data and produces classification results. The FPGA's parallel processing capabilities are harnessed to optimize inference time, and measurements are taken to quantify the reduction achieved compared to a purely software-based implementation.

3.5.2 Resource Utilization:

Resource Utilization is the amount of resources used by an FPGA for the particular design, in my project, it is the HLS pruned CNN model and HLS quantized pruned CNN model. The aspects that is often consider is Lookup table, Digital Signal Processing (DSP), Flip Flops (FF), Block RAM and I/O block. They are affected by the architecture of the model and the designation of the model and also the HLS conversion technique.

3.5.3 Power Consumption:

Power Consumption is the power needed for the FPGA to run the model. It is usually affected by the function and the frequency of the FPGA running. The design complexity and operating condition can also changes the power consumption of FPGA.

3.5.4 Comparison with Traditional Platform Implementation:

In order to visualize the advantages and limitations of FPGA comparing to the CPU and GPU, the comparison between them is needed to be done and analyze for gaining insight on both of the platforms implementing the same model performing the same classification task.

3.5.5 Trade-offs and Optimization Strategies:

With optimization technique done on the models for resource and power efficiency, there will also be trade-offs in the accuracy of the model. Performance analysis is done to inspect the significance of the optimization technique. Balancing the efficiency and accuracy is one of the main key aspect.

3.5.6 Validation of FPGA-Deployed Model:

The performance metrics derived from FPGA implementation are cross-validated against the metrics obtained during the training and validation phases in Python. This step is to perform an analysis on the performance of the FPGA. This makes sure the implementation of FPGA based CNN is significant.

CHAPTER 4

RESULT AND DISCUSSION

4.1 Introduction

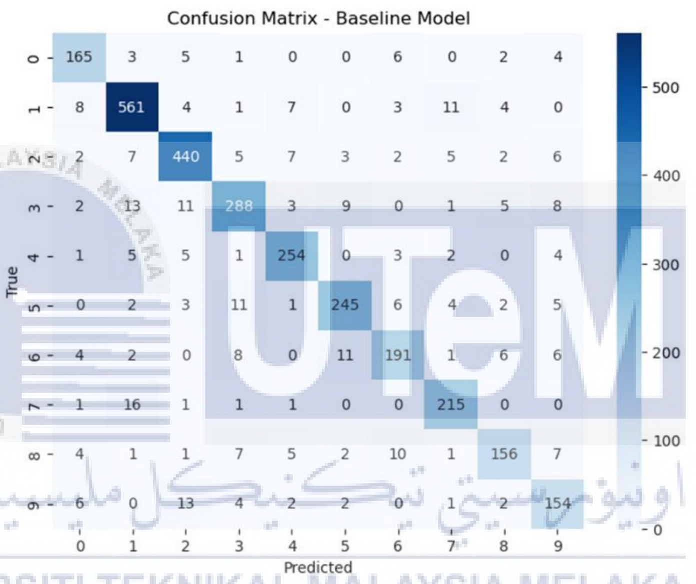
In this section, the accuracy, resource utilization and power of the pruned CNN model and the quantized pruned CNN model will be discussed. The validation and accuracy of both models in the PYNQ Z1 will also be displayed as the result.

4.2 Result of Pruned CNN model

4.2.1 Accuracy

The accuracy of the pruned CNN model is 88.97% and the pruned CNN HLS model has the accuracy of 88.73%. The accuracy stated is the accuracy running on the CPU i5 7th generation. There is a slight decrease in the accuracy for the HLS model. This is a normal phenomenon since there are certain optimizations done during the HLS process to make the model more suitable for hardware implementation.

The confusion matrix in the Figure 4.1 is a useful tool for understanding the performance of a machine learning model. It shows the number of times each class was predicted for every true class. Darker shades in the matrix indicate higher numbers, providing insights into which classes the model is accurately predicting and which ones it’s struggling with. In the context of the statement, the confusion matrix could be used to analyze the performance of the pruned CNN model both before and after the HLS conversion.



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	165	3	5	1	0	0	6	0	2	4
1	8	561	4	1	7	0	3	11	4	0
2	2	7	440	5	7	3	2	5	2	6
3	2	13	11	288	3	9	0	1	5	8
4	1	5	5	1	254	0	3	2	0	4
5	0	2	3	11	1	245	6	4	2	5
6	4	2	0	8	0	11	191	1	6	6
7	1	16	1	1	1	0	0	215	0	0
8	4	1	1	7	5	2	10	1	156	7
9	6	0	13	4	2	2	0	1	2	154

Figure 4.1: Confusion matrix of pruned CNN model

The graph in Figure 4.2 is the ROC of the pruned CNN model and the pruned CNN model HLS. It is the representation of the performance for both the models. They are evaluated based on their Area under Curve, AUC which is a metric to measure the performance of classification task. Higher AUC indicates the model performance is good. As we can see in the graph, the x axis is the signal efficiency and the y axis is the background efficiency, Signal Efficiency is the rate at which true positive events are correctly identified, and Background Efficiency is the rate at which false positive events are incorrectly identified as true. A model with perfect classification would have a signal efficiency of 1 and a background efficiency of 0. Both models are closing in towards 1.o which indicates both model perform well for all the classes in the classification task. In summary, the graph helps us to visualize the model’s performance.

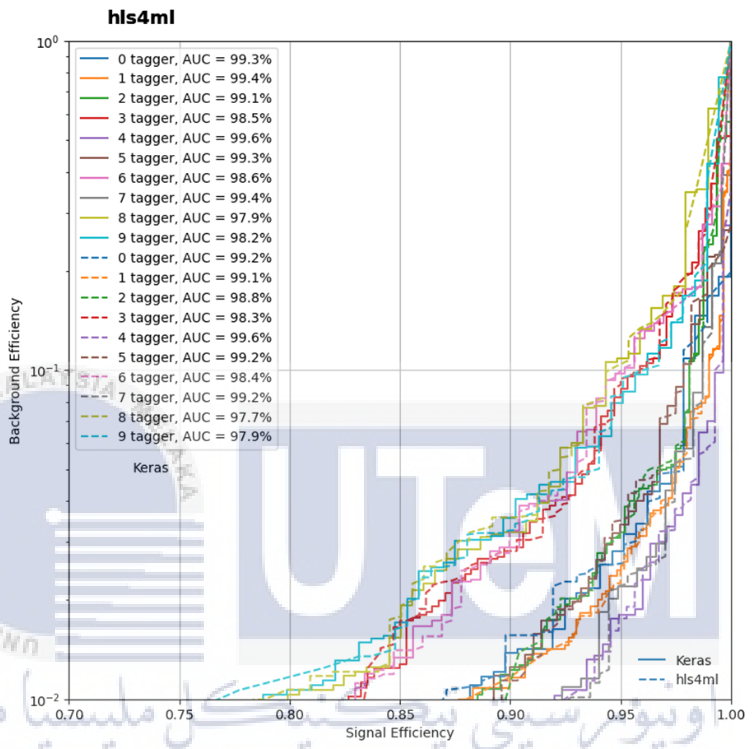


Figure 4.2: ROC of pruned CNN model

In conclusion, the goal of using high-performance architectures like ARM processors and FPGAs is to strike a balance between computational efficiency and model accuracy. A small drop in accuracy might be acceptable if the gains in efficiency are significant. This is particularly true for applications where real-time processing and low power consumption are critical. The success of deploying Convolutional Neural Networks (CNNs) on Field-Programmable Gate Arrays (FPGAs) lies not only in achieving hardware acceleration but also in preserving the model's predictive accuracy. This section presents a comprehensive analysis of the accuracy results obtained through various implementations

4.2.2 Resource Utilization

This section presents the implementation report focusing on the resource utilization report by Vivado. The resource utilization report is shown in Figure 4.3, Figure 4.4 and Figure 4.5. Slice LUT is used for implementing combinational logic. Out of the available 53,200 Slice LUTs, 36,460 are utilized, achieving a utilization rate of 68.53%. The breakdown reveals that 63.80% of these LUTs are configured for general logic purposes, while 14.48% are allocated for use as memory. A portion of the Slice LUTs is dedicated to specific functions such as Distributed RAM and Shift Register. In the report, it states that 22 LUTs serve as Distributed RAM, and 2,497 LUTs are configured as Shift Registers. The slice registers is used for storing intermediate and final results, displays a utilization of 55.61%. All 59,173 registers are employed, with the entirety configured as Flip Flops, indicating a predominant use for sequential logic. The utilization of F7 and F8 multiplexers, critical for routing signals within the FPGA, is presented. F7 Muxes demonstrate a utilization of 6.70%, with 1,781 out of 26,600 in use. Similarly, F8 Muxes exhibit a utilization of 4.31%, with 573 out of 13,300 utilized.

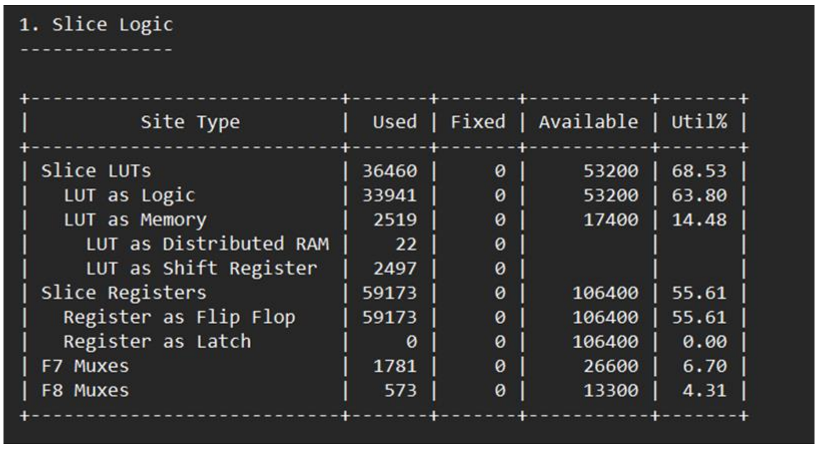

```
1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 36460 |     0 |     53200 | 68.53 |
|   LUT as Logic             | 33941 |     0 |     53200 | 63.80 |
|   LUT as Memory            |  2519 |     0 |     17400 | 14.48 |
|     LUT as Distributed RAM |    22 |     0 |           |       |
|     LUT as Shift Register  |  2497 |     0 |           |       |
| Slice Registers            | 59173 |     0 |    106400 | 55.61 |
|   Register as Flip Flop    | 59173 |     0 |    106400 | 55.61 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |  1781 |     0 |     26600 |  6.70 |
| F8 Muxes                   |   573 |     0 |     13300 |  4.31 |
+----------------------------+-------+-------+-----------+-------+
```

Figure 4.3: Slice Logic Report of Pruned CNN model

The memory report is shown in Figure 4.4. The Block RAM Tile serves as a fundamental memory component in the FPGA design, providing 71 utilized instances out of an available pool of 140 which shows the utilization rate of 50.71%. Block RAM Tiles are versatile and commonly used for storing data in FPGA designs, contributing significantly to the overall memory landscape. Specific instances of memory, namely RAMB36/FIFO and RAMB36E1, are employed in the design with 4 instances each. These instances are drawn from an available pool of 140, resulting in a modest utilization rate of 2.86% for both types. These memories are essential for applications requiring specialized memory structures, such as First-In-First-Out (FIFO) implementations. The RAMB18 module, known for its capacity and versatility, is utilized with 134 instances out of an available 280, achieving a utilization rate of 47.86%. This memory component is commonly employed for various data storage and retrieval operations within the FPGA design. Similar to RAMB18, the RAMB18E1 module exhibits a utilization rate of 47.86%, with all 134 instances actively contributing to the design. RAMB18E1, with its enhanced features, further enhances the memory capabilities of the FPGA.

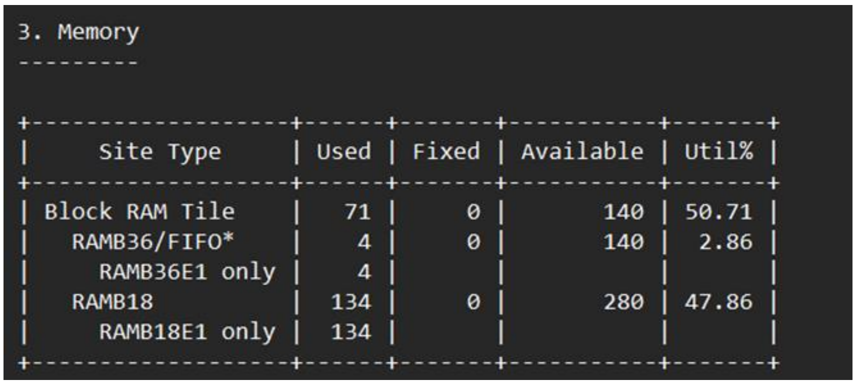
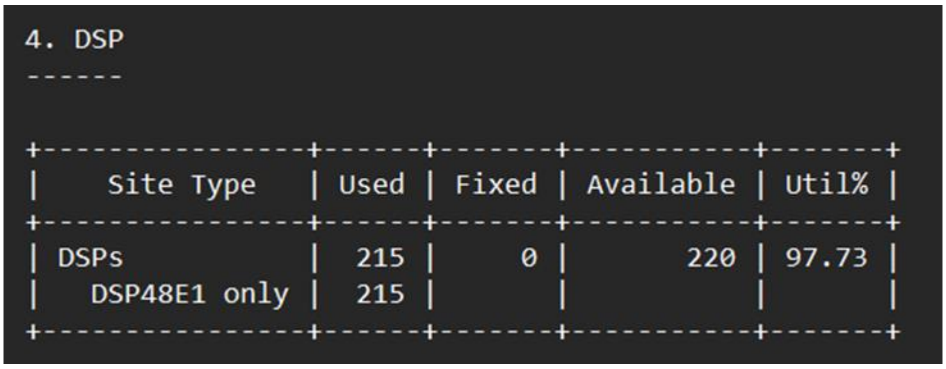

```
3. Memory
---------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |   71 |     0 |       140 | 50.71 |
|   RAMB36/FIFO*    |    4 |     0 |       140 |  2.86 |
|     RAMB36E1 only |    4 |       |           |       |
|   RAMB18          |  134 |     0 |       280 | 47.86 |
|     RAMB18E1 only |  134 |       |           |       |
+-------------------+------+-------+-----------+-------+
```

Figure 4.4: Memory usage report of Pruned CNN model

The DSP resource utilization shown in Figure 4.5 is 215 instances actively in use out of an available pool of 220, resulting in a utilization rate of 97.73%. DSP modules are crucial for accelerating complex mathematical computations and signal processing tasks within FPGA designs.



```
4. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |  215 |     0 |       220 | 97.73 |
|   DSP48E1 only |  215 |       |           |       |
+----------------+------+-------+-----------+-------+
```

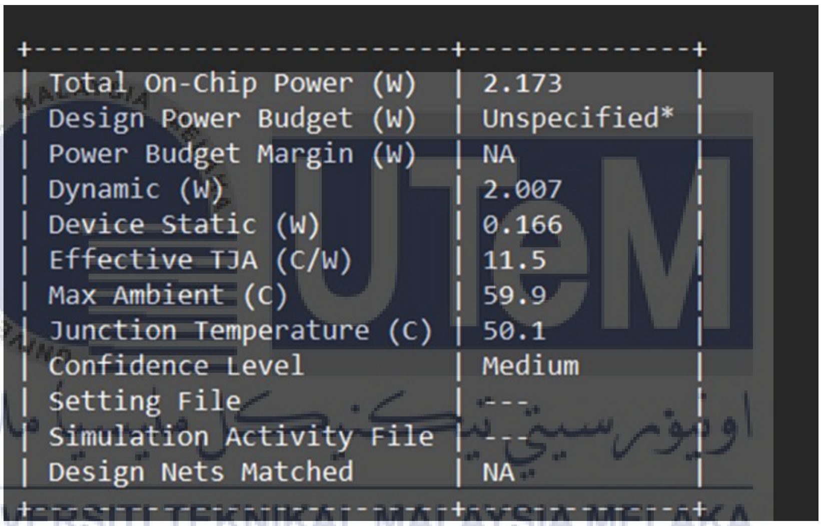
Figure 4.5: DSP usage report of Pruned CNN model

The integrated analysis of Slice Logic, DSP, and Memory utilization showcases a holistic approach to resource management. The near-maximal use of DSP resources suggests an efficient alignment of computational tasks with specialized hardware capabilities. The balance between Slice Logic and Memory utilization indicates a harmonized allocation of resources for both computational and data storage requirements.

4.2.3 Power

The power report is appended in Figure 4.6. Total On-Chip Power (W) is the total power consumption of the chip in Watts. The value 2.173W means that the chip is consuming approximately 2.173W of power. Dynamic (W) is the dynamic power consumption of the chip, which is the power consumed when the chip is active or in operation. The chip is consuming approximately 2.007 Watts of dynamic power. Device Static (W) is the static power consumption of the chip, which is the power consumed when the chip is idle or not in operation. The chip is consuming approximately 0.166 Watts of static power. Effective TJA (C/W) stands for Thermal

Resistance Junction to Ambient. It’s a measure of how effectively the chip can transfer heat from the junction (the part of the chip that gets hot) to the ambient environment. The lower the TJA, the better the chip is at cooling itself. Max Ambient is the maximum ambient temperature, in Celsius, at which the chip can operate. Junction Temperature is the temperature, in Celsius, at the junction of the chip. This is typically the hottest point on the chip. The total on-chip power is also lower compared to the pruned CNN model which indicates that the quantization process leads to a lesser power usage model.



```
+--------------------------+--------------+
| Total On-Chip Power (W)  | 2.173        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 2.007        |
| Device Static (W)        | 0.166        |
| Effective TJA (C/W)      | 11.5         |
| Max Ambient (C)          | 59.9         |
| Junction Temperature (C) | 50.1         |
| Confidence Level         | Medium       |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```

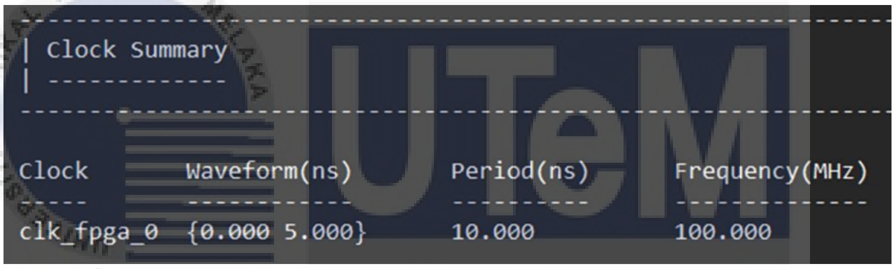
Figure 4.6: Power report

4.2.4 Clock Constraint and Frequency

As shown in Figure 4.7 and 4.8, the clock frequency used is 100 MHz and there is no violation of clock for the setup, hold and PW which is a good design features. In Vivado, the "worst slack" in the Clock Report refers to the timing slack of the critical path with the least amount of margin in terms of meeting the specified timing constraints. The timing constraints define the desired performance goals for the model’s design, such as maximum clock frequency, setup time, and hold time requirements.

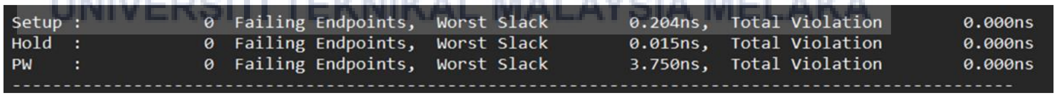
The timing slack represents the amount of time by which a signal can be delayed without violating the specified timing constraints. A positive slack value indicates that the design meets the timing requirements, whereas a negative slack value indicates a timing violation. The "worst slack" is the smallest (most negative) slack value among all the critical paths in your design.

If the worst slack is negative, it means that the design is failing to meet timing at that particular critical path. This could be due to various reasons, such as congested routing, inefficient placement of logic elements, or inadequate clock-to-q delays in the sequential elements along the critical path.



```
------------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------------

Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
clk_fpga_0   {0.000 5.000}      10.000          100.000
```

Figure 4.7: Clock summary



```
Setup :           0  Failing Endpoints,  Worst Slack        0.204ns,  Total Violation        0.000ns
Hold  :           0  Failing Endpoints,  Worst Slack        0.015ns,  Total Violation        0.000ns
PW    :           0  Failing Endpoints,  Worst Slack        3.750ns,  Total Violation        0.000ns
------------------------------------------------------------------------------------------------
```

Figure 4.8: Clock constraint

4.3 Quantized Pruned CNN model

4.3.1 Accuracy

In the assessment of Convolutional Neural Networks (CNNs) within the scope of our research on 'Building Brains with ARM Processors and FPGAs Based on High-Performance Architectures,' we evaluated the model's performance using two distinct frameworks: qkeras and qkeras_hls4ml. The accuracy metric, a key indicator of the model's classification performance, was determined for each framework. The achieved accuracy for the qkeras implementation was 87.5%, while the accuracy for the qkeras_hls4ml implementation was slightly lower at 87.46666666666667 %. These values reflect the models' abilities to correctly classify input data, providing insights into the effectiveness of each framework in implementing high-performance CNNs on the selected hardware architectures.

The confusion matrix in Figure 4.9 allows us to identify the performance of all the classes for classification. We can specifically look at each class to determine which class has the highest performance and which has the lowest. We are also able to look at which class has the highest number of data in the test dataset we used.

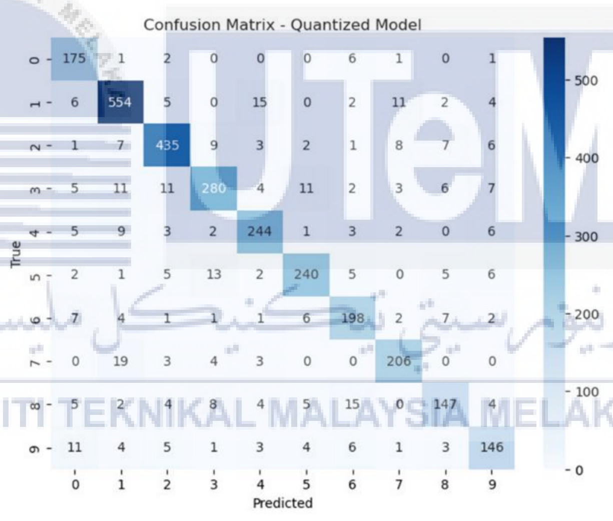


Figure 4.9: Confusion matrix of Quantized Model

The performance is evaluated based on the Area Under Curve (AUC) values, which is a common metric used in machine learning to measure the quality of binary classification problems.

The graph in Figure 4.10 plots Signal Efficiency against Background Efficiency, which can be interpreted as the model’s ability to correctly classify signal (true positives) and background (true negatives) instances. This is crucial in the context of

CNNs as it directly relates to the model's accuracy and precision. The different colored lines represent different taggers, each with a specific AUC value. These taggers is the classes of prediction which is 0 to 9. "Signal Efficiency" is the rate at which true positive events are correctly identified, and "Background Efficiency" is the rate at which false positive events are incorrectly identified as true. A model with perfect classification would have a signal efficiency of 1 and a background efficiency of 0. As we can see in the figure, there is less performance drop after the HLS conversion which is a good phenomenon for our aim.

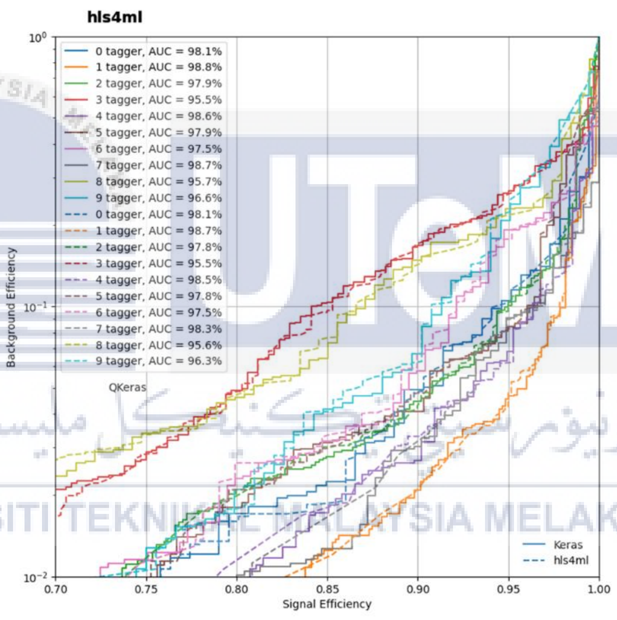
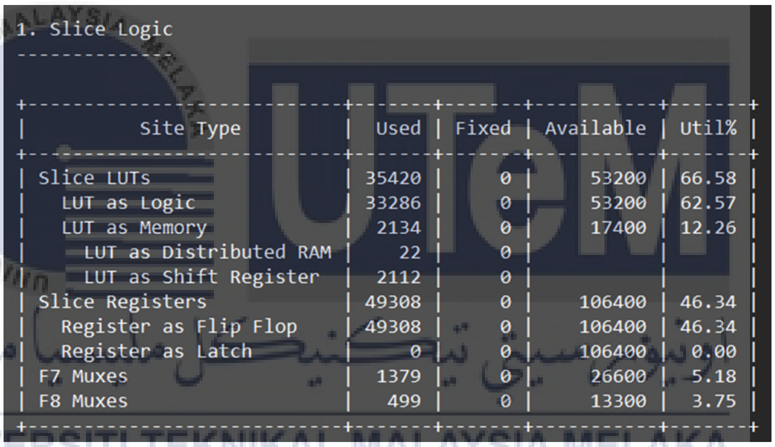


Figure 4.10: ROC of Quantized Pruned CNN model

4.3.2 Resource Utilization

The resource utilization report is presented in Figure 4.11, 4.12 and 4.13. Slice Look-Up Tables (LUTs) are fundamental building blocks in an FPGA. They implement arbitrary Boolean logic functions. The utilization of 66.58% indicates that a significant portion of the logic in the design is implemented using these LUTs. A utilization of 62.57% suggests that a majority of the LUTs are used for this purpose.

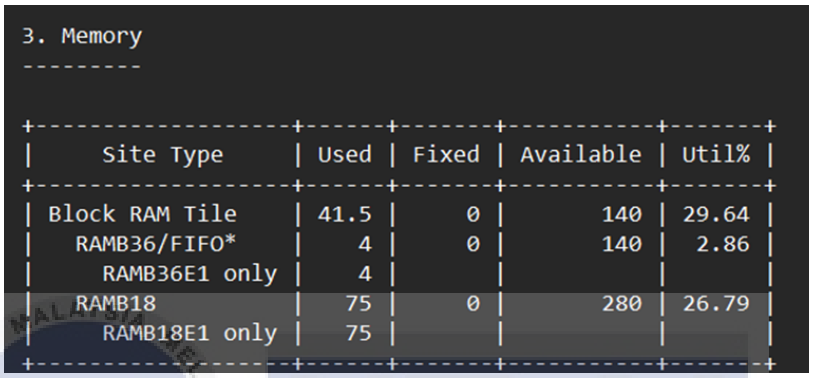
LUTs can also be used as small memories or shift registers. The utilization of 12.26% indicates that a small portion of the LUTs are used as memory elements. Slice registers are used to store data or state information in sequential logic. The utilization of 46.34% suggests that about half of the available slice registers are used in the design. Register as flip flops refers to registers that are used as flip-flops, a basic unit of memory in digital circuits. The utilization matches that of the slice registers, suggesting that all used registers are configured as flip-flops. All the utilization percentages drop compared to the pruned CNN model. This indicates that the quantized pruned CNN model uses lesser resource compared to the pruned CNN model.



```
1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 35420 |     0 |     53200 | 66.58 |
|   LUT as Logic             | 33286 |     0 |     53200 | 62.57 |
|   LUT as Memory            |  2134 |     0 |     17400 | 12.26 |
|     LUT as Distributed RAM |    22 |     0 |           |       |
|     LUT as Shift Register  |  2112 |     0 |           |       |
| Slice Registers            | 49308 |     0 |    106400 | 46.34 |
|   Register as Flip Flop    | 49308 |     0 |    106400 | 46.34 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |  1379 |     0 |     26600 |  5.18 |
| F8 Muxes                   |   499 |     0 |     13300 |  3.75 |
+----------------------------+-------+-------+-----------+-------+
```

Figure 4.11: Slice Logic report of Quantized Pruned CNN model

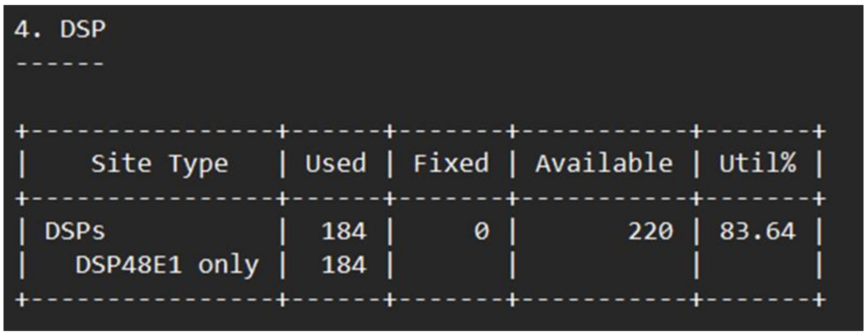
Block RAMs are used to implement larger memory arrays. They are used in designs that require storage of large amounts of data or coefficients. The utilization of 29.64% indicates that less than a third of the available Block RAM Tiles are used in the design. There is a significant drop in the memory utilization of the quantized pruned CNN model compared to the pruned CNN model. Quantization reduces the precision of the weights in the neural network. For example, weights that were originally 32-bit floating point numbers might be reduced to 8-bit integers. This can significantly reduce the memory requirements of the model, as well as the computational requirements of the forward pass. In conclusion, the low memory utilization in the report can be attributed to the use of quantization and pruning techniques. These techniques have reduced the memory footprint of the CNN model, allowing it to fit on the FPGA with plenty of resources to spare. This is particularly beneficial in embedded systems where memory and computational resources are limited.



```
3. Memory
---------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    | 41.5 |     0 |       140 | 29.64 |
|   RAMB36/FIFO*    |    4 |     0 |       140 |  2.86 |
|     RAMB36E1 only |    4 |       |           |       |
|   RAMB18          |   75 |     0 |       280 | 26.79 |
|     RAMB18E1 only |   75 |       |           |       |
+-------------------+------+-------+-----------+-------+
```

Figure 4.12: Memory report of Quantized Pruned CNN model

DSPs are used to perform arithmetic functions, such as multiply-accumulate, in a single operation. They are crucial for implementing the convolutional layers in a CNN. The utilization of 83.64% indicates that a significant portion of the DSPs are used in the design. Quantization can significantly reduce the computational requirements of the forward pass, which in turn can lead to a reduction in DSP usage.



```
4. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |  184 |     0 |       220 | 83.64 |
|   DSP48E1 only |  184 |       |           |       |
+----------------+------+-------+-----------+-------+
```

Figure 4.13: DSP usage report of Quantized Pruned CNN model

4.3.3 Power

The power report is appended in Figure 4.14. Total On-Chip Power (W) is the total power consumption of the chip in Watts. The value 1.953 suggests that the chip is consuming approximately 1.953 Watts of power. Dynamic (W) is the dynamic power consumption of the chip, which is the power consumed when the chip is active or in operation. The chip is consuming approximately 1.798 Watts of dynamic power. Device Static (W) is the static power consumption of the chip, which is the power consumed when the chip is idle or not in operation. The chip is consuming approximately 0.154 Watts of static power. Effective TJA (C/W) stands for Thermal Resistance Junction to Ambient. It’s a measure of how effectively the chip can transfer heat from the junction (the part of the chip that gets hot) to the ambient environment. The lower the TJA, the better the chip is at cooling itself. Max Ambient is the maximum ambient temperature, in Celsius, at which the chip can operate. Junction Temperature is the temperature, in Celsius, at the junction of the chip. This is typically the hottest point on the chip. The total on-chip power is also lower compared to the pruned CNN model which indicates that the quantization process leads to a lesser power usage model.

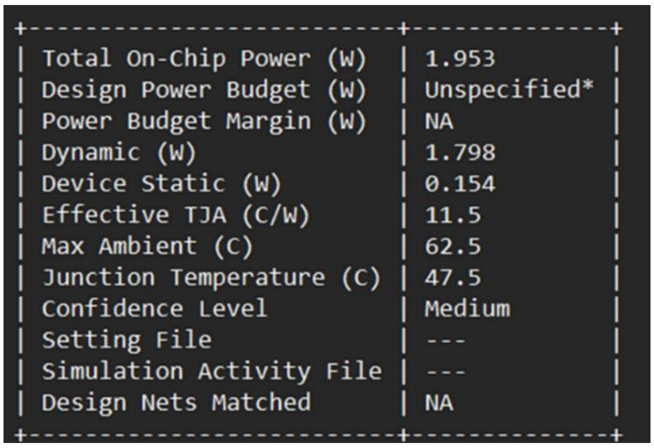
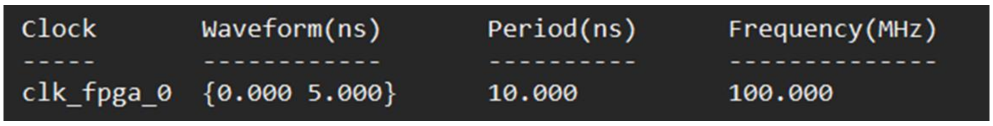

```
+--------------------------+--------------+
| Total On-Chip Power (W)  | 1.953        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 1.798        |
| Device Static (W)        | 0.154        |
| Effective TJA (C/W)      | 11.5         |
| Max Ambient (C)          | 62.5         |
| Junction Temperature (C) | 47.5         |
| Confidence Level         | Medium       |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```

Figure 4.14: Power report of Quantized Pruned CNN model

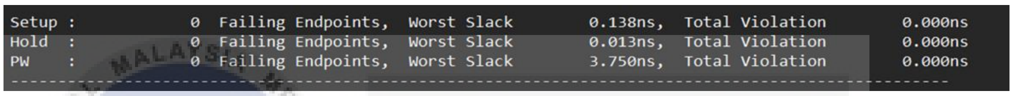
4.3.4 Clock Constraint and Frequency

As in the pruned CNN model clock summary report, the clock frequency used in the quantized pruned CNN model is also 100MHz and there is also no clock violation found after the implementation process which we are able to notice in Figure 4.15 and 4.16.



```
Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
clk_fpga_0   {0.000 5.000}      10.000          100.000
```

Figure 4.15: Clock summary of Quantized Pruned CNN model



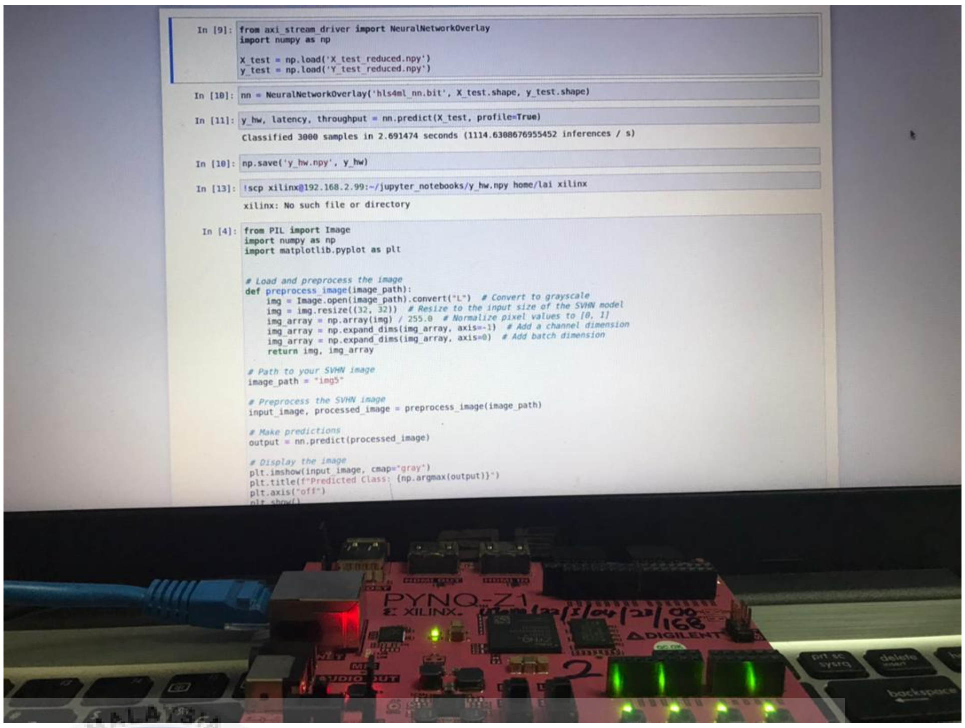
```
Setup :           0  Failing Endpoints,  Worst Slack       0.138ns,  Total Violation       0.000ns
Hold  :           0  Failing Endpoints,  Worst Slack       0.013ns,  Total Violation       0.000ns
PW    :           0  Failing Endpoints,  Worst Slack       3.750ns,  Total Violation       0.000ns
---------------------------------------------------------------------------------------------------
```

Figure 4.16: Clock Constraint of Quantized Pruned CNN model

4.4 PYNQ Z1 board

4.4.1 Validation

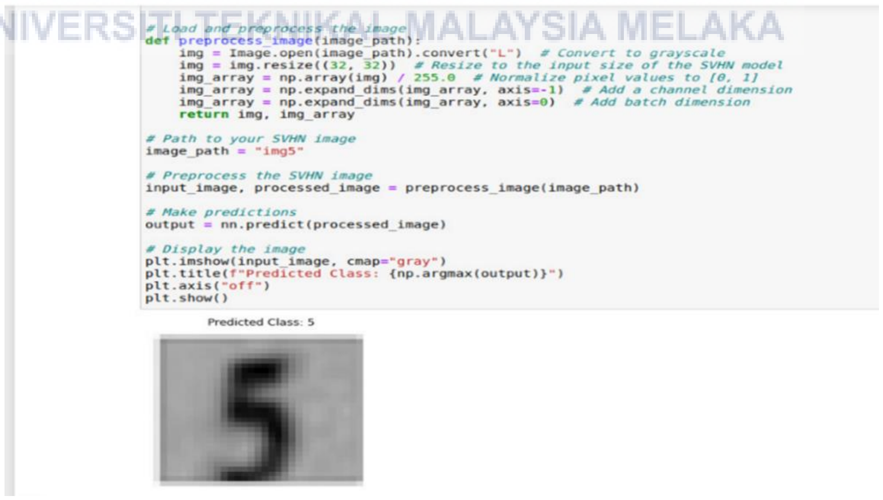
The validation process is done which is shown in Figure 4.17 and to make sure the practical implementation of the pruned CNN model and the quantized pruned CNN model works on the PYNQ Z1 board. The code shows the inference process is done using the axi driver which is the communication protocol, the bitfile and configuration file.



```
In [9]:  from axi_stream_driver import NeuralNetworkOverlay
         import numpy as np

         X_test = np.load('X_test_reduced.npy')
         y_test = np.load('Y_test_reduced.npy')

In [10]: nn = NeuralNetworkOverlay('hls4ml_nn.bit', X_test.shape, y_test.shape)

In [11]: y_hw, latency, throughput = nn.predict(X_test, profile=True)

         Classified 3000 samples in 2.691474 seconds (1114.6308676955452 inferences / s)

In [10]: np.save('y_hw.npy', y_hw)

In [13]: !scp xilinx@192.168.2.99:~/jupyter_notebooks/y_hw.npy home/lai xilinx

         xilinx: No such file or directory

In [4]:  from PIL import Image
         import numpy as np
         import matplotlib.pyplot as plt


         # Load and preprocess the image
         def preprocess_image(image_path):
             img = Image.open(image_path).convert("L")  # Convert to grayscale
             img = img.resize((32, 32))  # Resize to the input size of the SVHN model
             img_array = np.array(img) / 255.0  # Normalize pixel values to [0, 1]
             img_array = np.expand_dims(img_array, axis=-1)  # Add a channel dimension
             img_array = np.expand_dims(img_array, axis=0)  # Add batch dimension
             return img, img_array

         # Path to your SVHN image
         image_path = "img5"

         # Preprocess the SVHN image
         input_image, processed_image = preprocess_image(image_path)

         # Make predictions
         output = nn.predict(processed_image)

         # Display the image
         plt.imshow(input_image, cmap="gray")
         plt.title(f"Predicted Class: {np.argmax(output)}")
         plt.axis("off")
         plt.show()
```

Figure 4.17: PYNQ Z1 board inference

Another way shown in Figure 4.18 that is tested for validation is by using an image in the testing dataset and using it to the model on the board and check the predicted class. This is a qualitative analysis for the model on the board PYNQ Z1.



```
# Load and preprocess the image
def preprocess_image(image_path):
    img = Image.open(image_path).convert("L")  # Convert to grayscale
    img = img.resize((32, 32))  # Resize to the input size of the SVHN model
    img_array = np.array(img) / 255.0  # Normalize pixel values to [0, 1]
    img_array = np.expand_dims(img_array, axis=-1)  # Add a channel dimension
    img_array = np.expand_dims(img_array, axis=0)  # Add batch dimension
    return img, img_array

# Path to your SVHN image
image_path = "img5"

# Preprocess the SVHN image
input_image, processed_image = preprocess_image(image_path)

# Make predictions
output = nn.predict(processed_image)

# Display the image
plt.imshow(input_image, cmap="gray")
plt.title(f"Predicted Class: {np.argmax(output)}")
plt.axis("off")
plt.show()
```

Predicted Class: 5

Figure 4.18: Validation of 1 image

4.4.2 Accuracy

The accuracy onboard(Keras) is 88.73333333333333%, whereas the accuracy onboard(QKeras) is 87.46666666666667%. The accuracy is same for both model respectively to the accuracy after the HLS conversion. This indicates that there is no accuracy loss during the inference on board.

4.5 Model in Google Colab

Both model are also built in the Google Colab using the Colab's CPU and GPU workspace to perform a comparison analysis.

4.5.1 Accuracy

The accuracy with the training and testing using CPU and GPU on the Google Colab of the Keras model is 88.1% and the accuracy for quantized keras model is 84.9% which is slightly lower than the only CPU using intel i5 7th generation's accuracy stated in the section above. This can be due to the GPU limitations which the use of floating-point precision on GPUs compared to CPUs can indeed be a factor that affects the accuracy of computations and, consequently, the overall accuracy of a machine learning model. GPUs often use lower-precision floating-point formats (such as half-precision or mixed-precision) to accelerate computations, which may result in some loss of precision compared to the higher precision typically used on CPUs (single or double precision). Visualization of the graph can be seen in Figure 4.19.

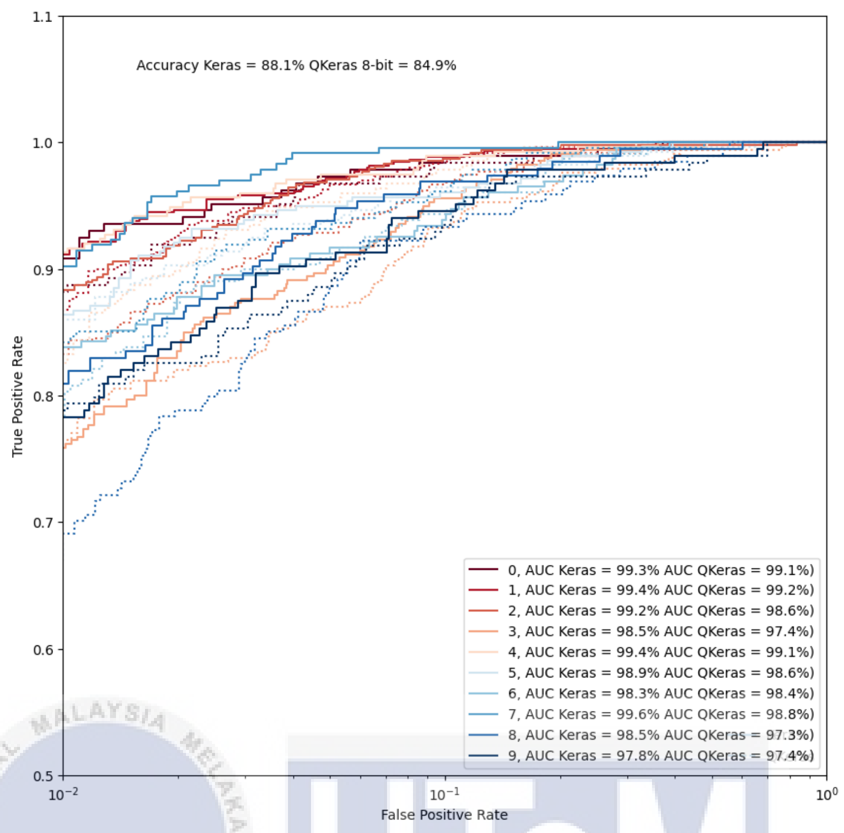


Figure 4.19: Visualize of accuracy on Colab

4.5.2 Resource Utilization

The resource Utilization graph of the Google Colab can be seen in Figure 4.20. The CPU usage of the training and testing is very high where it fluctuated from 0 to 100%, the memory usage for the models are at 30% average and the GPU usage fluctuated at 0 to 60%.

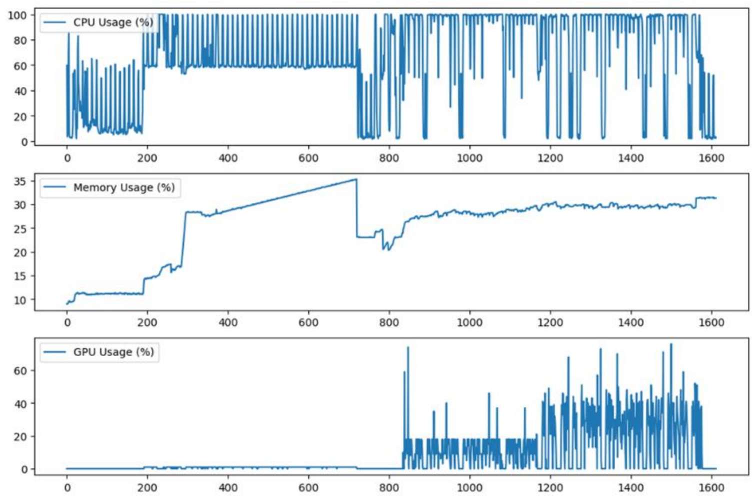


Figure 4.20: Resource graph for Colab

4.6 Comparison

Table 5 shows the comparison of both the pruned models and quantized models on different platforms in terms of accuracy, power, and resource utilization. The CPU and GPU approach have a slightly lower accuracy, but the training process is a lot quicker compared to the CPU only approach.

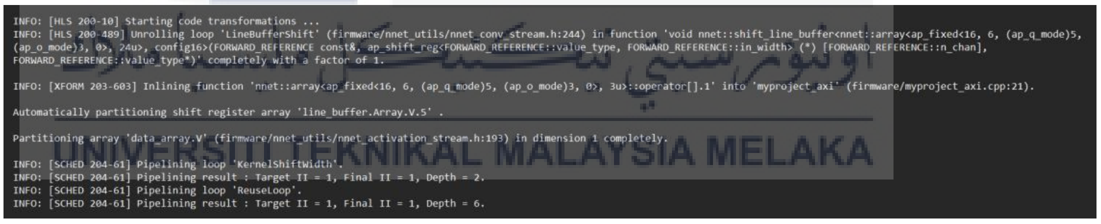
Table 5: Comparison Analysis

Device	**Model**	**Power**	**Resource Utilization**	**Testing Accuracy**
CPU & GPU (Colab) Faster Training	Pruned CNN	61W & 300W (on server)	100% CPU, 60% GPU, 35% Memory	88.1%
CPU & GPU (Colab)	Quantized Pruned CNN	61W & 300W (on server)	100% CPU, 60% GPU, 35% Memory	84.9%
CPU (Ubuntu Intel i5 7th gen) Slower Training	Pruned CNN	65W	-	88.9%
CPU (Ubuntu Intel i5 7th gen)	Quantized Pruned CNN	65W	-	87.5%

CPU (Ubuntu Intel i5 7th gen)	Pruned CNN HLS	65W	-	88.7%
CPU (Ubuntu Intel i5 7th gen)	Quantized Pruned CNN HLS	65W	-	87.4%
PYNQ Z1	Pruned CNN HLS	2.173W	68.53% LUT, 55.61% Registers, 97.73% DSP, 50.71% Memory	88.7%
PYNQ Z1	Quantized Pruned CNN HLS	1.953W	66.58% LUT 46.34% Registers 83.64% DSP 29.64% Memory	87.4%

4.7 Discussion

The PYNQ Z1 utilizes parallelism techniques such as unrolling, pipelining, inlining and partitioning array. Loop unrolling is a compiler optimization technique employed to enhance parallelism in software implementations, particularly within loops. By replicating the loop body multiple times, loop unrolling reduces loop overhead and exposes more opportunities for instruction-level parallelism. This approach allows multiple iterations of the loop to execute concurrently, facilitating improved throughput and computational efficiency. Inlining, another optimization strategy, involves incorporating the body of a function directly into the calling code, eliminating the overhead associated with function calls. In the context of parallelism, inlined code offers better optimization opportunities as the compiler can more easily identify and exploit parallelism within the broader context of the calling code. Furthermore, partitioning arrays is a technique where large arrays are broken down into smaller, manageable chunks, enabling independent or parallel processing of segments. This approach distributes the workload and enhances overall throughput by creating opportunities for parallel execution. Lastly, pipelining is a hardware design strategy that breaks down a computation into stages, allowing each stage to operate concurrently. This technique enhances concurrency by overlapping the execution of different stages, where each stage processes a different set of data. The result is efficient parallel processing of multiple data elements at various stages simultaneously, further optimizing the overall performance of the computation. The parallelism of FPGA allows it to increase the performance in terms of resource utilization, power and thermal. Some of the examples are shown in Figure 4.22 for the parallelization technique. The technique used is automated by the HLS4Ml library. However, with the resource constraint of the RAM on board, the training process is not able to run on the board, which means that the training must be done on host machine and then convert the trained model into the bitfile and hardware configuration file then only the inference can be done the PYNQ Z1 board. The example of the parallelism done is shown in Figure 4.21.



```
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-489] Unrolling loop 'LineBufferShift' (firmware/nnet_utils/nnet_conv_stream.h:244) in function 'void nnet::shift_line_buffer<nnet::array<ap_fixed<16, 6, (ap_q_mode)5,
(ap_o_mode)3, 0>, 24u>, config16>(FORWARD_REFERENCE const&, ap_shift_reg<FORWARD_REFERENCE::value_type, FORWARD_REFERENCE::in_width> (*) [FORWARD_REFERENCE::n_chan],
FORWARD_REFERENCE::value_type*)' completely with a factor of 1.

INFO: [XFORM 203-603] Inlining function 'nnet::array<ap_fixed<16, 6, (ap_q_mode)5, (ap_o_mode)3, 0>, 3u>::operator[].1' into 'myproject_axi' (firmware/myproject_axi.cpp:21).

Automatically partitioning shift register array 'line_buffer.Array.V.5' .

Partitioning array 'data_array.V' (firmware/nnet_utils/nnet_activation_stream.h:193) in dimension 1 completely.

INFO: [SCHED 204-61] Pipelining loop 'KernelShiftWidth'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 2.
INFO: [SCHED 204-61] Pipelining loop 'ReuseLoop'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 6.
```

Figure 4.21: Parallelization during Synthesize and Bitstream Generation

4.8 Environment and Sustainability

4.8.1 Needs and Importance for Sustainable Development

9 INDUSTRY, INNOVATION AND INFRASTRUCTURE
12 RESPONSIBLE CONSUMPTION AND PRODUCTION

Figure 4.22: SDG

The first aspect is developing an advanced Computing for AI Applications. The demand nowadays for AI application in the healthcare, finance and automation sector is increasing since AI eases the work needed to be done by humans not to mention excels in some of the job. Advanced computing solutions such as ARM processor and FPGA has become a necessity for answer those demands. The project is also aiming for energy efficiency. Energy efficiency is able to reduce environmental impact done by technology in order to improve the lifestyle of humans. ARM processor and FPGA are well known for their power efficiency, which can be a solution for energy-efficiency. ARM processors and FPGA can provide the scalability and flexibility in developing neural networks. This ensure the adaptability of the board for the evolution of neural networks which happens in the blink of an eye. ARM processors and FPGA has the ability to optimize resource utilization and ensuring sustainability.

4.8.2 Impact of the Engineering Solution on Society

Application of CNN on FPGA allows the medical advancement in imaging, diagnostic which can lead to better healthcare outcomes. Nowadays, autonomous systems is a trend for things such as vehicles and drones. This project provide an insight on the real time image based recognition system to enhance safety. Precision agriculture is also an aspect that can potentially benefit from this project. Wastage of resource can be avoid when precision agriculture can be implemented using FPGA based CNN.

4.8.3 Impact on the environment

The impact of this project on the environment can be reducing carbon footprint, e-waste management and natural resource conservation. Energy efficiency of the ARM processor and FPGA can lead to reduce in overall carbon footprint. The flexibility and reprogrammability of FPGA allows lesser of e-waste since it can be reuse and reprogram each time according to the needs of user. Energy efficient of the ARM processor and FPGA also allows this project to contributes in natural resource conservation.

4.9 Summary

Based on the accuracy, power, resource consumption comparison analysis, the results of my project on "Building Brains with ARM processors and FPGAs based on high-performance architectures for Convolutional Neural Networks (CNNs)" suggest that there are advantages and disadvantages for advanced computing for AI applications. The advantages are optimized resource utilization and power consumption. The drawback would be the resource constraint causing the training process to not be able to be done on the PYNQ z1 board. The results highlight the potential of ARM processors and FPGAs in addressing the computational needs of high-performance CNNs, offering a pathway for the advancement of AI applications in various sectors.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Introduction

The chapter will cover the conclusion of the project in Section 5.2 and the future work of this project in Section 5.3.

5.2 Conclusion

This thesis focuses on the development of optimized CNN models based on PYNQ z1 FPGA. All the objectives of the project has been achieved with the model on the FPGA validation successfully done, optimization technique used on the model and analysis done for the performance of the pruned and pruned quantized CNN models in the CPU, GPU and FPGA.

The first objective aimed to design and implement a CNN on an FPGA. The objective is achieved with the validation on board is completed and the result such as implementation report and accuracy are recorded. The second objective is to implement optimization technique to the CNN model, which is also achieved with pruning and quantization done to the CNN model. Parallelization technique is also utilized while converting the model into HLS model where the pipelining, inlining and partitioning array is done to utilize the parallelization abilities of the FPGA. The third objective is to perform a performance analysis to the result in order to gain insight on the comparison of traditional computing platform with the FPGA. This objective is achieved when we can conclude the advantages and the drawback of the FPGA in comparison to the traditional computing platform based on the accuracy and implementation report generated.

In conclusion, the successful achievement of these objectives underscores the significance of integrating ARM processors and FPGAs for CNN applications. The findings presented in this thesis not only advance the current understanding of hardware-accelerated neural networks but also provide a solid foundation for future research and development in the pursuit of optimized, high-performance computing solutions in the realm of artificial intelligence.

5.3 Future Work

The limitations found on this project can be improved in the future in exploring the limits of FPGA. By fine tuning the configuration of each layers in the CNN, the method of improving the accuracy, resource utilization and power consumption can be expected. Investigating on other optimization technique could also be one of the direction to work on in order to improve the performance of neural network based FPGA. Other than CNN, other NN model such as Recurrent Neural Network, Spiking Neural Network and et cetera, can also be explore to implement on the FPGA to test the limits of FPGA with various applications. Real time inference using a camara on FPGA can also be the path to explore the FPGA, since real time data from camera can be a challenging topic to work on. Cross-platform compatibility is a key consideration for broadening the implementation's reach across various FPGA architectures. Evaluating power consumption and efficiency aspects, implementing power-aware design techniques, and exploring dynamic reconfiguration for on-the-fly adjustments to FPGA configurations can contribute to energy-efficient CNN models. Furthermore, integration with edge computing platforms and thorough benchmarking against alternative FPGA-based CNN implementations will provide insights into the strengths and weaknesses of the proposed approach in comparison to existing solutions. In essence, these future research directions aim to push the boundaries of FPGA-accelerated CNN deployment, advancing the fields of hardware-accelerated deep learning and edge computing.

REFERENCES

[1] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," *IEEE Trans Very Large Scale Integr VLSI Syst*, vol. 26, no. 7, 2018, doi: 10.1109/TVLSI.2018.2815603.

[2] D. Parra, D. Escobar Sanabria, and C. Camargo, "A Methodology and Open-Source Tools to Implement Convolutional Neural Networks Quantized with TensorFlow Lite on FPGAs," *Electronics (Switzerland)*, vol. 12, no. 20, Oct. 2023, doi: 10.3390/electronics12204367.

[3] C. Bartneck, C. Lütge, A. Wagner, and S. Welsh, "What Is AI?," in *SpringerBriefs in Ethics*, 2021. doi: 10.1007/978-3-030-51110-4_2.

[4] Q. Zhang, X. Wang, Y. N. Wu, H. Zhou, and S. C. Zhu, "Interpretable CNNs for Object Classification," *IEEE Trans Pattern Anal Mach Intell*, vol. 43, no. 10, 2021, doi: 10.1109/TPAMI.2020.2982882.

[5] Xilinx Inc., "Field Programmable Gate Array (FPGA): What is an FPGA?," Xilinx.

[6] C. Qiu, X. Wang, T. Zhao, Q. Li, B. Wang, and H. Wang, "An FPGA-Based Convolutional Neural Network Coprocessor," *Wirel Commun Mob Comput*, vol. 2021, 2021, doi: 10.1155/2021/3768724.

[7] S. Xiong *et al.*, "MRI-based brain tumor segmentation using FPGA-accelerated neural network," *BMC Bioinformatics*, vol. 22, no. 1, 2021, doi: 10.1186/s12859-021-04347-6.

[8] P. Hobden, S. Srivastava, and E. Nurellari, "FPGA-Based CNN for Real-Time UAV Tracking and Detection," *Frontiers in Space Technologies*, vol. 3, 2022, doi: 10.3389/frspt.2022.878010.

[9] Z. Wang, H. Li, X. Yue, and L. Meng, "Briefly Analysis about CNN Accelerator based on FPGA," in *Procedia Computer Science*, 2022. doi: 10.1016/j.procs.2022.04.036.

[10] C. Wang and Z. Luo, "A Review of the Optimal Design of Neural Networks Based on FPGA," *Applied Sciences (Switzerland)*, vol. 12, no. 21. 2022. doi: 10.3390/app122110771.

[11] N. Zhang, X. Wei, H. Chen, and W. Liu, "FPGA implementation for CNN-based optical remote sensing object detection," *Electronics (Switzerland)*, vol. 10, no. 3, 2021, doi: 10.3390/electronics10030282.

[12] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-Based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7. 2019. doi: 10.1109/ACCESS.2018.2890150.

[13] M. Magdy Saady and M. Hassan Essai, “Hardware implementation of neural network-based engine model using FPGA,” *Alexandria Engineering Journal*, vol. 61, no. 12, 2022, doi: 10.1016/j.aej.2022.05.035.

[14] S. Zhai, C. Qiu, Y. Yang, J. Li, and Y. Cui, “Design of Convolutional Neural Network Based on FPGA,” in *Journal of Physics: Conference Series*, 2019. doi: 10.1088/1742-6596/1168/6/062016.

[15] S. Bouguezzi, H. Ben Fredj, T. Belabed, C. Valderrama, H. Faiedh, and C. Souani, “An efficient fpga-based convolutional neural network for classification: Ad-mobilenet,” *Electronics (Switzerland)*, vol. 10, no. 18, 2021, doi: 10.3390/electronics10182272.

[16] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, “An FPGA-based CNN accelerator integrating depthwise separable convolution,” *Electronics (Switzerland)*, vol. 8, no. 3, 2019, doi: 10.3390/electronics8030281.

[17] I. Pérez and M. Figueroa, “A heterogeneous hardware accelerator for image classification in embedded systems,” *Sensors*, vol. 21, no. 8, 2021, doi: 10.3390/s21082637.

[18] T. Addabbo *et al.*, “A Low-Complexity FPGA-Based Neural Network for Hand-Arm Vibrations Classification,” in *2023 IEEE International Workshop on Metrology for Industry 4.0 and IoT, MetroInd4.0 and IoT 2023 - Proceedings*, 2023. doi: 10.1109/MetroInd4.0IoT57462.2023.10180160.

[19] M. T. Ahmed and S. Sinha, “Design and Development of Efficient Face Recognition Architecture Using Neural Network on FPGA,” in *Proceedings of the 2nd International Conference on Intelligent Computing and Control Systems, ICICCS 2018*, 2018. doi: 10.1109/ICCONS.2018.8663098.

[20] S. S. Lingala, S. Bedekar, P. Tyagi, P. Saha, and P. Shahane, “FPGA Based Implementation of Neural Network,” in *Proceedings - IEEE International Conference on Advances in Computing, Communication and Applied Informatics, ACCAI 2022*, 2022. doi: 10.1109/ACCAI53970.2022.9752656.

[21] F. U. D. Farrukh, T. Xie, C. Zhang, and Z. Wang, “Optimization for Efficient Hardware Implementation of CNN on FPGA,” in *Proceedings of 2018 IEEE International Conference on Integrated Circuits, Technologies and Applications, ICTA 2018*, 2018. doi: 10.1109/CICTA.2018.8706067.

[22] N. M. Philip and N. M. Sivamangai, “Review of FPGA-Based Accelerators of Deep Convolutional Neural Networks,” in *ICDCS 2022 - 2022 6th International Conference on Devices, Circuits and Systems*, 2022. doi: 10.1109/ICDCS54290.2022.9780689.

[23] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “The Street View House Numbers (SVHN) Dataset,” NIPS Workshop.

[24] J. Duarte *et al.*, “Fast inference of deep neural networks in FPGAs for particle physics,” *Journal of Instrumentation*, vol. 13, no. 7, 2018, doi: 10.1088/1748-0221/13/07/P07027.

[25] T. Aarrestad *et al.*, “Fast convolutional neural networks on FPGAs with hls4ml,” *Mach Learn Sci Technol*, vol. 2, no. 4, 2021, doi: 10.1088/2632-2153/ac0ea1.