ACCELERATING THE CONVOLUTIONAL NEURAL NETWORKS(CNN) USING FPGA

THAM WEI JIAN

UNIVERSITI TEKNIKAL MALAYSIA MELAKA

ACCELERATING THE CONVOLUTIONAL NEURAL NETWORKS(CNN) USING FPGA

THAM WEI JIAN

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

Faculty of Electronic and Computer Engineering
Universiti Teknikal Malaysia Melaka

2020

DECLARATION

I declare that this report entitled “Accelerating the Convolutional Neural Networks (CNN) using FPGA” is the result of my own work except for quotes as cited in the references.

Signature	:	…………………………………
Author	:	Tham Wei Jian…………………
Date	:	24 June 2020……………………

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 Electronic Engineering with Honours.

Signature	:	…………………………………
Supervisor Name	:	PM Dr. Wong Yan Chiew………
Date	:	……30/06/2020………………

DEDICATION

I would like to dedicate my project report to my beloved family.

ABSTRACT

The convolutional neural network (CNN) is inspired by the behavior of optic nerves in the living creatures and also has a huge application in video surveillance, mobile robot vision, image search engine in database, etc. Besides, the rapid growth of CNN has shown that the performance of CNN now surpasses of the other type of visual recognition algorithms, and even beyond the human accuracy on certain conditions. In this work, the FPGA platform is used to implement the CNNs of different application. This is because FPGA has good performance, high energy efficiency, fast development round, and capability of reconfiguration. We also set up two different platforms which are CPU-only (Intel i5-4200M) and GPU-only (NVIDIA GTX-750Ti) to run the YOLOv2 so that the data can be obtained and compared with the FPGAs. The results show that the YOLOv2 and ResNet50 on FPGA have achieved low power consumption and high-power efficiency in this project. Besides, the accuracy of CNN on ZedBoard for digits recognition is satisfactory and the power consumption is very low. For the BNN, the time taken for hardware implementation to classify an image is faster than the software implementation when using the PYNQ-Z2.

ABSTRAK

Rangkaian saraf konvolusional (Convolutional Neural Network) inspirasi daripada saraf optik manusia dan juga memiliki aplikasi besar dalam pengawasan video, penglihatan robot bergerak, mesin pencari gambar dalam pangkalan data dan lain-lain. Selain itu, pertumbuhan CNN yang rapat telah menunjukkan bahawa prestasi CNN kini melebihi jenis algoritma pengenalan visual lain, dan juga melebihi ketepatan manusia pada keadaan tertentu. Dalam projek ini, platform FPGA digunakan untuk melaksanakan CNN aplikasi yang berbeza. Hal ini demikian kerana FPGA mempunyai prestasi yang baik, kecekapan tenaga yang tinggi, putaran pengembangan yang cepat, dan kemampuan konfigurasi ulang. Kami juga menyediakan dua platform berbeza yang hanya CPU (Intel i5-4200M) dan hanya GPU (NVIDIA GTX-750Ti) untuk menjalankan YOLOv2 sehingga data dapat diperolehi dan dibandingkan dengan FPGA. Hasil kajian menunjukkan bahawa YOLOv2 dan ResNet50 pada FPGA telah mencapai penggunaan kuasa rendah dan kecekapan kuasa tinggi dalam projek ini. Selain itu, ketepatan CNN pada ZedBoard untuk pengecaman digit adalah memuaskan dan penggunaan kuasa sangat rendah. Bagi BNN, masa yang diperlukan untuk pelaksanaan perkakasan untuk mengklasifikasikan gambar lebih cepat daripada pelaksanaan perisian apabila kami menggunakan PYNQ-Z2.

ACKNOWLEDGEMENTS

I want to express my sincerest gratitude to my thesis supervisor, PM Dr. Wong Yan Chiew, for allowing me to work on the topic and providing me with the knowledge and information. PM Dr. Wong' s guidance allows me to understand the project better so that I can advance the knowledge in this field to help my future work.

Moreover, I also want to thank the lab assistant of Universiti Teknikal Malaysia Melaka (UTeM) who had lent me the FPGA board to complete my final year project.

Furthermore, I would like to appreciate to my thesis panels, Ir. Dr. Ranjit Singh and Dr. Mai Mariam for their valuable comments to improve my thesis.

Finally, I would like to express my earnest gratitude to my parents, my family and friends who directly and indirectly lend their hand for assisting me to accomplish this project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

For example:

CNN	:	Convolutional Neural Network
BNN	:	Binarized Neural Network
YOLO	:	You Only Look Once
FPGA	:	Field-Programmable Gate Array
CPU	:	Central Processing Units
GPU	:	Graphic Processing Units
DPU	:	Deep Processing Unit
PS	:	Processing System
PL	:	Programmable Logic

LIST OF APPENDICES

CHPATER 1

INTRODUCTION

1.1 Introduction

This chapter will describe the details of the background and purpose of the project. It also includes the problem statement and the scope of the project will be covered.

1.2 Project Background

An artificial neural network develops the deep learning architecture that we called convolutional neural network (CNN). The convolutional neural network (CNN) has a huge application in video surveillance, mobile robot vision, image search engine in database, etc. The data processing of CNN with multiple layers of neural connections is inspired by the behavior of optic nerves in the living creatures to achieve high accuracy in image recognition. Moreover, the development of deep learning algorithms has enhanced the research on deep convolutional neural network [13].

Besides, the CNN can use to recognize the human gestures in video. This is because CNN was efficiently captured both relevant shape information and detailed spatiotemporal variation of the gestures and solved the environmental issues such as complex background, occlusion, lighting conditions and so on [1]. Moreover, the CNN also being applied for detecting and recognizing East Asian characters and this project has achieved average end-to-end accuracies of 98.2% and 98.3% on 40 videos in Simplified Chinese and 40 videos in Traditional Chinese respectively. This breakthrough results have narrowed the gap between human cognitive ability and state-of-the-art algorithms used for a similar project [4]. Furthermore, the CNN also implemented to reduce the road traffic problems in the city. The researchers mentioned that they adopted the CNN method on CCTV camera to detect the congestion of traffic. The research result shows that CNN has an average classification accuracy of 89.50% on their project [1]. Due to the rapid growth of CNN have shown that the performance of CNN now surpasses of the other type of visual recognition algorithms, and even beyond the human accuracy on certain conditions.

In recent years, many researchers have designed to implement GPU-based, ASIC and FPGA-based accelerators to implement CNN algorithms to improve the performance of CNN designs. This is because the general-purpose processor cannot meet the specific calculation mode of CNN, which makes the performance of CNN difficult to meet the design requirements. In contrast， field-programmable grid arrays (FPGAs) have great potential for applying the CNN. FPGAs have the advantages of good performance, high energy efficiency, fast development cycles, and reconfigurability, which have attracted researchers to focus on this area. In addition, the combination of low-precision arithmetic and small memory footprint is also an advantage of FPGAs.

In deep learning, the GPU has the best performance in computational efficiency. Although the GPU has excellent performance in terms of computational efficiency in deep learning, it is expensive and consumes a lot of power have prompting researchers to change their choices and focus on FPGAs. Moreover, GPU is very expensive and high-power consumption compared to the FPGA and even GPU is not very stable in large-scale deployment and operation platforms. The advantage of using the FPGA is because it is reprogrammable and has a short design cycle [3].

1.3 Problem Statement

The CNN based on the GPU and ASIC are high power consumption and cost expensive compared to FPGA during the same performance. Besides, CNN is computationally intensive and implementation of high performance depends on the computing platform. So, the traditional CPUs is inappropriate in accelerating CNN. Therefore, FPGA is the better choice used as an accelerator of CNN because the

FPGA has good performance, high energy efficiency, fast development round, and capability of reconfiguration.

1.4 Objectives

The objectives of this project are:

i. To investigate the method used to accelerate CNN on FPGA development board.
ii. To optimize and analyze an accelerated CNN on FPGA development board.
iii. To evaluate the performance of the developed system on FPGA board.

1.5 Scope of Project

The scope of the project will cover how to apply CNN to FPGAs without involving an ASIC. The FPGA board model used is the Zedboard Zynq-7000 development board and PYNQ-Z2. The software covered is Jupyter Notebook, Vivado and Petalinux. The programming types used in this project are python, C/C++ and Verilog. The approaches CNN apply in this project are ResNet50, YOLOv2, BNN and CNN for digits recognition. The application of CNNs implement in this project are image classification, object detection, and digit classification. This project will use the IP integrator to accelerate the hardware implementation of CNNs for the Zedboard and PYNQ-Z2 and the Overlay method to accelerate the BNN on PYNQ-Z2.

1.6 Expected Outcome

The expected outcome of the project is an accelerated CNN developed on the FPGA board with different approaches such as ResNet50, YOLOv2, BNN and CNN for digits recognition. The CNNs application such as image classification, object detection and digits recognition will be applied on ZedBoard and PYNQ-Z2. The expected time taken to implement the CNNs will be faster and the power consumption of ZedBoard and PYNQ-Z2 will be lower compare with another platform such as GPU and CPU.

1.7 Thesis Outline

This thesis is divided into five chapters. Chapter 1 introduces the project and describes the project's overview, background, objectives, problem statements, expected outcome and scope of work. Chapter 2 introduces the background study about CNN models and each of the hardware accelerator. This chapter also shows the benchmarking for each of the platform used to implement the CNNs. In Chapter 3, the flow chart is described for each board, including the data flow in FPGA to accelerate the CNNs and how quantization and compilation of pre-trained CNN data and implement it on ZedBoard. Chapter 4 is the discussion of the results of CNNs on ZedBoard and PYNQ-Z2. Finally, Chapter 5 gives the conclusions and the recommendation of the project for the evaluation of future works.

CHAPTER 2

BACKGROUND STUDY

2.1 Convolutional Neural Networks

The biological brain system inspired researchers to develop neural networks as a computational architecture. There are approximately 85 billion neurons in our human brain that are used to receive the signals and give respond. The part of the neuron as the input signal is called dendrites and the axon is the part that produces the output signal. However, there is another called synapse that connects the branches of the dendrites to connect the other neurons. Synapse is the part that affects the input and output and even suppress the transmission of signals. The connection of ten billion neurons have formed the neural network of our human brain so that humans can think, remember, analyze, understand, dream and other reaction that related to the human brain [1].

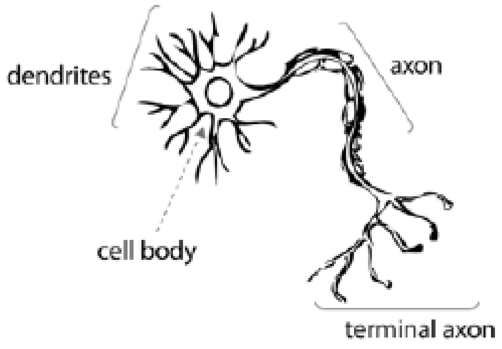


Figure 2.1: Biological Neuron. (Image adopted from [1])

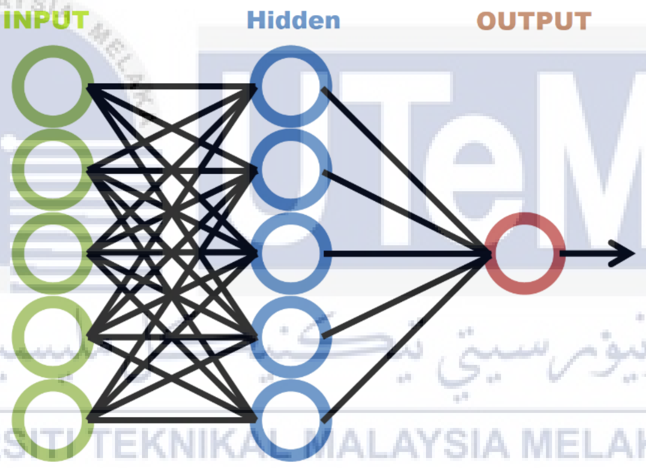


Figure 2.2: Example of Neural Network

The special of neural networks is they can improve performance through self-learning processes and it will make their performance become better. A neural network consists of several layers of nodes that are connected in layers and those layers are divided into input layer, output layer and middle layer (also known as hidden layer). The output of each node is determined by the specific values of weights and biases. So, the weight and bias values in each iteration will update for every training network [2].

Convolutional neural network (CNN) also called ConvNet popularly used for analyzing images although image analysis has been the most widespread use of CNN. They can also be used for other data analysis or classification problem as well. Most generally we can think of a CNN as an artificial neural network that has some type of specialization for being able to pick out or detect the patterns. In a typical CNN, CNN layers are mainly classified into convolution layers， pooling layers, and fully connected layers [8]. The following will explain all the layer types used in convolutional neural networks.

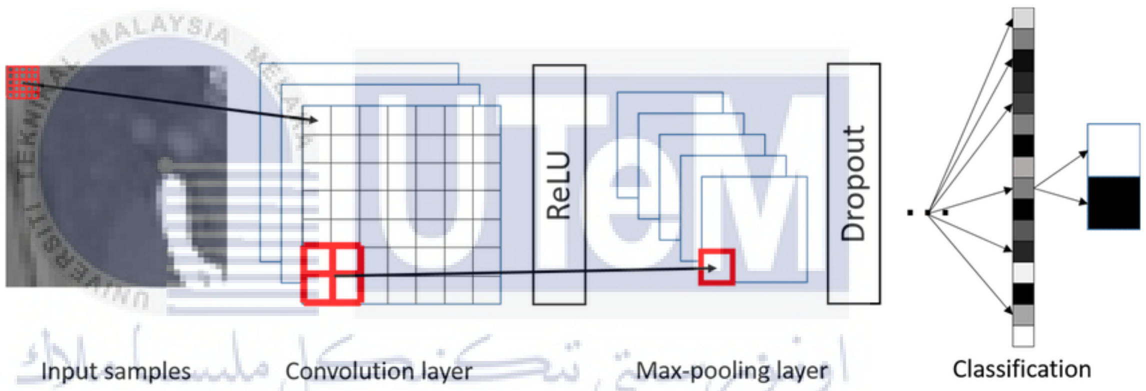


Figure 2.3: Example of standard convolutional neural network

1) Convolution Layer:

Convolution layer usually is the first layer for CNN where CNN convolves the images or data using a filter or kernel. Filters are small units that multiply across the data through the sliding window. The depth of the filter is same as the input. For example, the color image whose RGB values give the value of depth is 3, so a filter with depth 3 would be applied in the convolve. The convolution operation taking the element-wise product of filters in the image and then summing the values for every sliding action. The output of the convolution of the 3D filter with a color image is a 2D matrix.

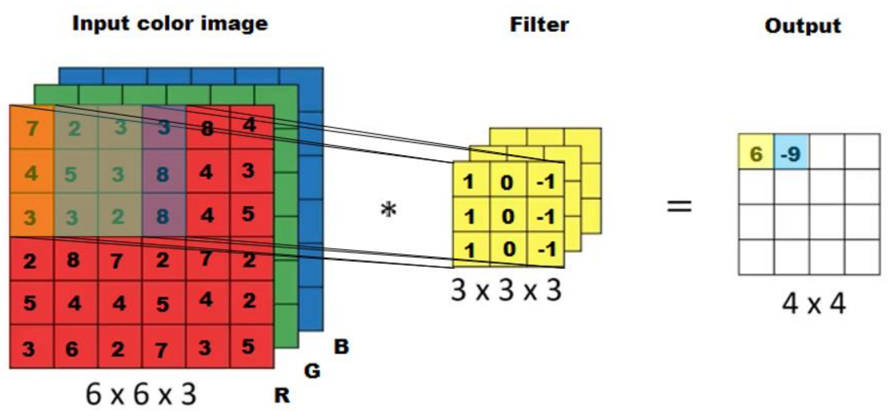


Figure 2.4: Convolution layer.

2) **Pooling Layer**

The pooling layer involved the down sampling of features by summarizing the multiple input pixels to a single pixel. There are 2 popular used of pooling layers are max pooling and average pooling [1]. Max pooling is a max filter returns the maximum value among the features in the region. For the average pooling, an average filter which returns the average values of features in the region.

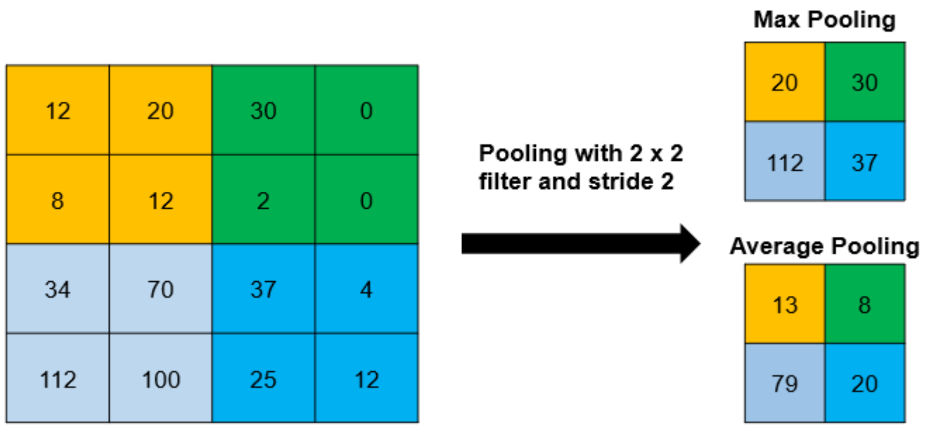


Figure 2.5: Example of max pooling and average pooling.

3) Fully Connected Layer

The fully connected layer is the last layer of the CNN. The fully connected layer usually used to calculate the category score for image classification. While the input image reaches this layer, the dimension of the image will become 1 x 1 and the fully connected layer will constitute the majority of the weights in the CNN [1].

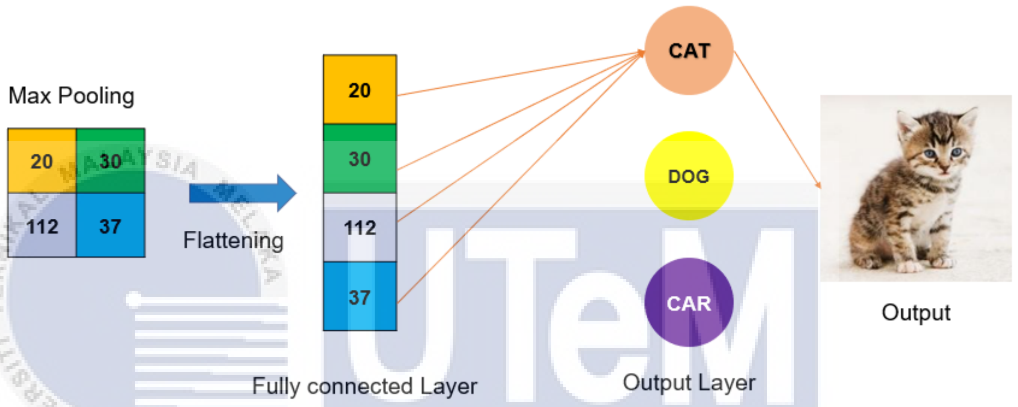


Figure 2.6: Example of Fully-connected layer.

2.2 Caffe Framework

Caffe is a deep learning framework developed by Berkeley AI Research (BAIR) and community contributors. Jia Yangqing created the project during his Ph.D. at the University of California at Berkeley. Also, Caffe's language library is written in C++ and implemented in Matlab and python languages. Caffe has multiple hidden layers that give this framework has the ability to recognize images and speech as well as natural language processing. It is often used for research and development because the code has modular capabilities and the network definition is completely separate from the actual implementation. Therefore, the optimization does not require coding to complete the setting file. Since Caffe's processing speed is extremely fast, it can effectively handle the image classification problem of millions of images [4].

According to previous research work in [6], the authors used the MNIST dataset of the Caffe framework as the neural network model for handwritten font recognition. Moreover, the research results indicate that the accuracy of the MNIST model of the Caffe framework is 99.93%. The researchers mentioned that the reason they used Caffe as a deep learning library is because its operating speed is an advantage and also Caffe supports CUDA, if necessary, the operating system can be converted from CPU to GPU, and even CPU and GPU both compatible.

2.3 Existing Approach for Convolutional Neural Network

2.3.1 ResNet50

ResNet50 means there are 50 layers Residual Network. ResNet was proposed in 2015 and won first place in the classification competition of the ImageNet test set because it is simple and practical. After that, there are many types of classification neural network approaches were built based on the concept of ResNet50 and ResNet101 such as detection, segmentation and recognition. K. He *et al.* [14] discovered that as the networks going to become deeper, the accuracy of the training set will decrease and caused the degradation problem happens. The author determined that this is not caused by the overfitting; therefore, the author proposed a new network for solved this problem, called a deep residual network (ResNet) which allows the networks to deepen as much as possible. The

ResNet provides two options, namely identity mapping and residual mapping for the problem of accuracy decreases as the network deepens. If the identity mapping has reached the optimal level, the residual mapping will be pushed as zero and left identity mapping only. This allows the network is always staying in an optimal state, and the performance of the network will not decrease as the depth is increasing.

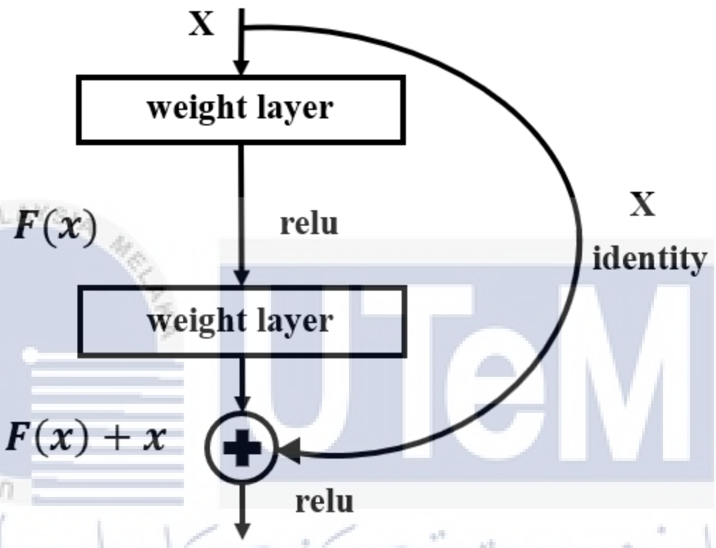


Figure 2.7: A building block of residual learning.

Y. Zhao *et al.* [15] proposed a new idea to deal with the fully-connected layers to improve the ResNet50 acceleration framework on FPGA. The experimental results showed that the FPGA has good performance and high energy efficiency ratio when compared with NVIDIA's M4 GPU because the FPGA has low latency and low energy consumption. The performance of Intel's FPGA has achieved 54 images per second and 36.7W pictures processed per second per Watt when implemented the ResNet50.

2.3.2 Binarized Neural Networks (BNNs)

The concept of BNNs is to reduce the memory size and computation requirement to improve the power efficiency by binarized both activation and weight values to single-bit values. In BNNs, the Boolean value of 0 is representing the binary values -1 in these single-bit values. The XNOR and *popcount* also used to compute the sum of multiplication between input activations and weight parameters, $\sum_i x_i w_i$[14]. The reason applied the XNOR to replace the binary multiplication between $+1$ and -1 is the outputs of binary multiplication values are the same as XNOR's truth table.

Table 2.1: Comparison between original multiplication and XNOR.

Original Multiplication		
$x_{<-1,1>}$	$w_{<-1,1>}$	$x.w_{<-1,1>}$
1	1	1
1	-1	-1
-1	1	-1
-1	-1	1

Affine transformed		
$x_{<0,1>}$	$w_{<0,1>}$	$x.w_{<0,1>}$
1	1	1
1	0	0
0	1	0
0	0	1

Y. Umuroglu *et al.* [15] proposed that the binary input activations, binary synapse weights and binary output activations are contained in binarized neural network. There are two types of binarization for neural network which are full binarization and partial binarization. Full binarization means all these three components are binary while partial binarization means either one or two components are binary. In paper Rastegari. M *et al.*

[17], author implemented the BNNs on ImageNet dataset and successfully achieved top-1 accuracy of 51.2% for full binarization and 65.5% for partial binarization.

2.3.2.1 CIFAR-10 Dataset

The CIFAR-10 database is composed of 60,000 32 x 32 colored pictures into 10 different classes of objects. The types of objects include airplane, automobile, bird, cat, deer, frog, dog, horse, ship and truck. Regarding the Y. Umuroglu *et al.* [15], one of the BNN topology called CNV has successfully achieved 80.1% accuracy for images classification by using CIFAR-10 dataset.

2.3.2.2 MNIST Dataset

MNIST is a dataset of handwritten digits which consist of 60,000 28 x 28 grayscale handwritten digits from 0 to 9. Y. Umuroglu *et al.* [15] mentioned that the BNN topology used to classify the handwritten digits is called LFC which is a three-layer fully connected network and also achieved 98.4% accuracy on FPGA.

2.3.3 You Only Look Once (YOLO)

YOLO [16] stands for You Only Look Once. YOLO algorithm uses a single CNN model to achieve end-to-end on object detection. Compared to the R-CNN algorithm, YOLO is a unified framework which is faster compared to R-CNN.

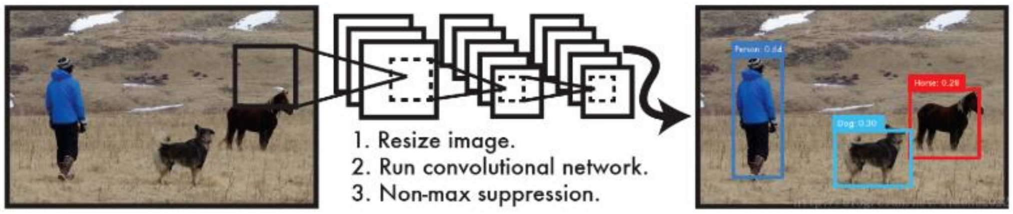


Figure 2.8: YOLO detection system (Image adopted from [22]).

Firstly, YOLO will resize the input image to 448x448, then send it to the CNN network layers to obtain the prediction results of the object detection. Secondly, YOLO would divide the input picture into N × N grids and then each of the grid cells is responsible for detecting the targets whose fall within the grid by using the cell center points. Each grid cell predicts the N bounding boxes and confidence score of these bounding boxes. As we known, the confidence scores will affect the accuracy of YOLO detection. Once the bounding box contains the target, the Pr (object)= 1 while the bounding box is the background means it does not contain any target, then Pr (object) = 0. The size and position of the bounding box can be characterized by 4 values: (x, y, w, h), where (x, y) is the center coordinate of the bounding box, and w and h are the width and height of the bounding box. Each cell needs to predict as an $(N * 5 + C)$ values. If the input image is divided into S × S grid, then the final prediction value with the size of the image is $N \times N \times (B * 5 + C)$ [22]. The following network structure described the distribution of the predicted value of each cell.

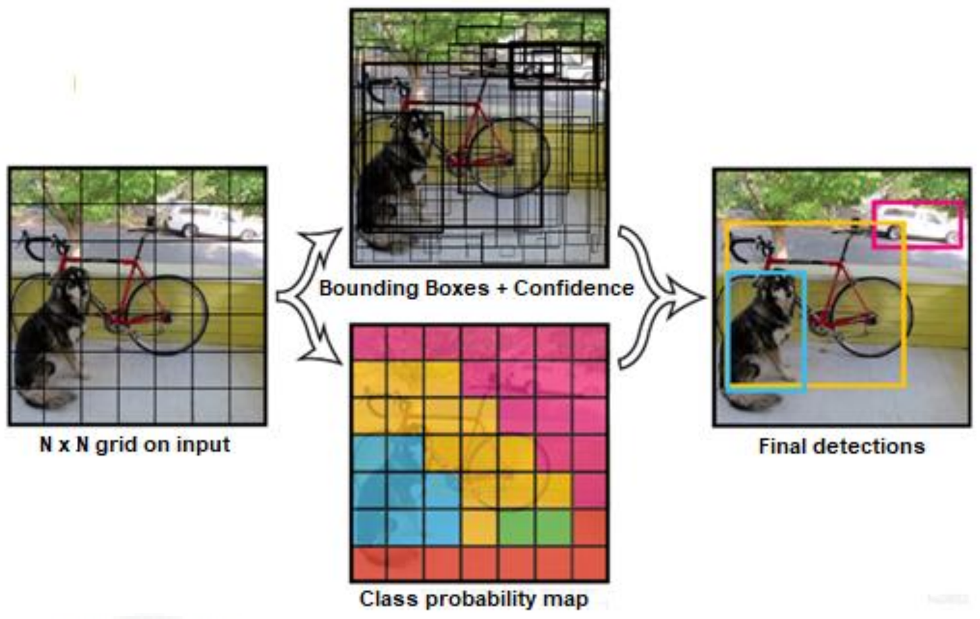


Figure 2.9: YOLO network structure (Image adopted from [22])

2.3.3.1 YOLOv2

J. Redmon *et .al* [18] applied the concept of anchor box from Faster RCNN into YOLOv2, so that the YOLOv2 can improve the positioning accuracy and also replace the YOLOv1. The YOLOv1 includes the fully connected layers, so it can directly predict the coordinates of bounding boxes in an image. However, the fully-connected layer and the last pooling layer of YOLOv2 removed from the original network, so that the last convolutional layer can have higher resolution features. The input size of YOLOv2 also replaced the original 448 * 448 with a 416 * 416. Therefore, the output of YOLOv2 can obtain a 13 * 13 feature map when the input size instead of 418. The author investigated that directly using the anchor box method caused the YOLOv2 model unstable, so the author combined the YOLOv1's algorithm into YOLOv2 to predict the coordinate position relative to the grid cell. The five variable values predict in each bounding box are tx, ty, tw, th, and to. The range of tx and ty after sigmoid function processing fall between 0 and 1. This normalization process also makes the model training more stable; cx and cy is representing the offset of a cell from the upper left corner of the image while pw and ph are representing the width and height of the bounding box respectively. Then, the bx and by are the anchors near the grid cell of cx and cy to predict the results obtained by tx and ty.

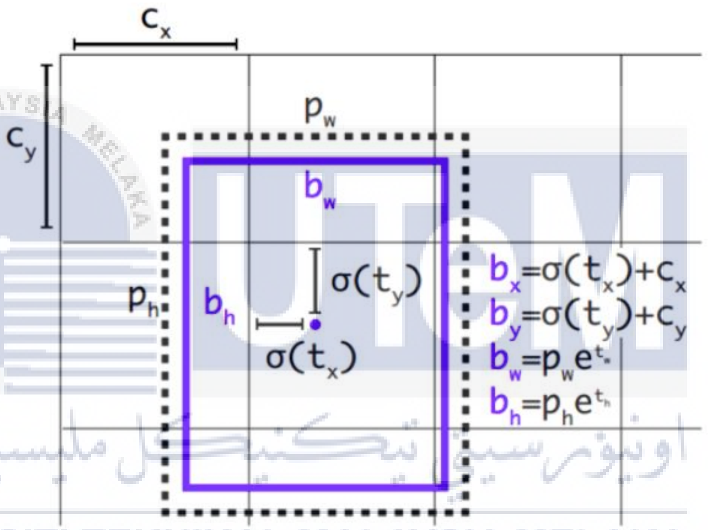


Figure 2.10: Bounding boxes with predicted dimension and its predicted location (Image adopted from [18])

2.3.4 LeNet-5

The main use of LeNet-5 is to classify the 0 to 9-digit numbers. The LeNet-5 architecture has a total of 7 layers, including three convolutional layers (C1, C3 and C5), two pooling layers or subsampling layers (S2 and S4), and the two fully-connected layers (F6 and output layer), but the input layer is not included. Y. Lecun *et al.*[36] mentioned that the input image size used by LeNet-5 is 32x32 pixels, which is larger than the input size required by other types of CNN. First, LeNet-5 will use the 5x5 sized of convolution kernel to perform the first convolution operation on the input image and form to six-channel of 28x28 feature maps to achieve the C1. Then, the layer S2 will down sample the output of C1 image become 6 channels of 14x14 feature map, and the size of the convolutional kernel used is 2x2. Then followed by another convolution layer, C3 which reduced the output of layer S2 feature map to 10x10 feature map with 16 channels. The resolution of the image gets halved again by average pooling at S4 become 16 channels of 5x5 feature map. Since the size of the layer S4 is 5x5, which is the same as the size of the convolution kernel at C5, the size of the feature map formed after the convolution is 1x1 pixel and line up become a fully-connected layer. This is followed by the layer F6 contains 84 units and it is fully-connected to C5. The Output layer is also a fully-connected layer. There are 10 units in the output layer, representing the numbers 0 to 9, and if the value of unit i is 0, the result of network identification is the number i.

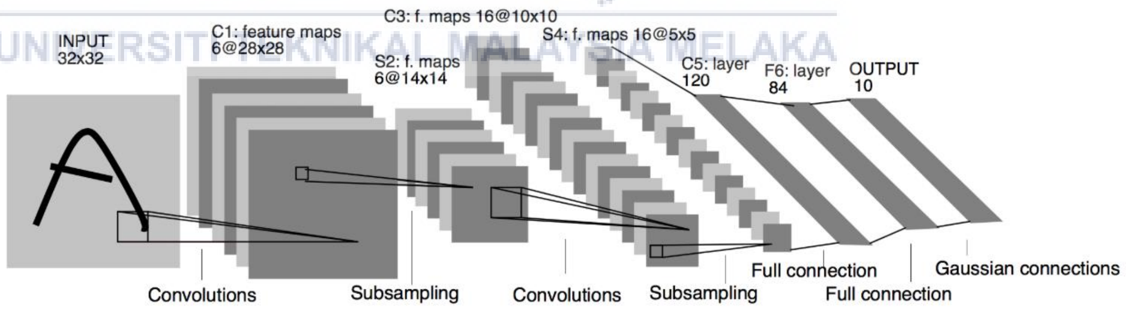


Figure 2.11: Network architecture of LeNet-5 (Image adopted from [36])

D. Rongshi *et al.* [37] purposed the researchers implemented the LeNet-5 on Zybo Z7 FPGA. This is because general processor cannot meet the performance requirement and the performance efficiency is low. Therefore, the researchers replaced the traditional processor by FPGA to optimize the convolution operation, data throughput and energy efficiency of traditional processors. They found that the data throughput of FPGA is higher than traditional processors and the power consumption of FPGA is lower than traditional processors [37].

2.4 Different types of Hardware Acceleration using for CNNs

2.4.1 Central Processing Units

The central processing unit (CPU) is the processor core in the electronic devices and it is widely used in desktops, laptops and smartphones. There are different types of processors available on the market for embedded systems, but different trade-offs are depending on speed and power requirements. D. Gschwend *et al.*[1] investigated that the CPU computes the results sequentially, so it is not suitable for convolutional neural network implementation. The advantage of the CPU is that it supports any programming framework such as C / C ++, Scala, Java, Python or any other new language so that the developers can program by using those supported languages. However, when the CPU involves machine learning training, it is only suitable for small and straight forward models with short training time. The CPU will be forbidden the total execution time of machine learning training when it is running large models and large data sets.

Figure 2.12: Intel i7 CPUs.

In research work by A. Mohanty *et al.*[16], he investigated that the combination of CPU and FPGA has achieved a speed up 30x compared to a CPU implementation for face detection. When using a CPU for a face detection algorithm, the face detection execution time is 7.284s. In contrast, the face detection execution time for combined CPU and FPGA only used 0.2378 for implements the face detection algorithm.

2.4.2 Graphics Processing Units

A graphics processing unit (GPU) is a multi-core processor dedicated to images and video processing. GPUs can host a large number of cores so that it is suitable for the applications that require to process data in parallel. Besides, the GPU supports the OpenCL and CUDA programming frameworks, which enables it to compute the algorithm of deep learning efficiently. Still, its flexibility is also limited compared to the CPU. D. Gschwend *et al.*[1] proposed two different models of GPUs which suitable for implementing the CNNs, called GeForce GTX Titan X and NVidia Tegra X1. The reason these two GPUs are ideal for deep learning is that they have high memory bandwidth, which leads to these two GPUs having a breakneck computing speed for deep learning. Therefore, the specification of memory bandwidth of GPU is the core reference require to refer while choosing a GPU as a hardware accelerator for CNN. However, M.Zhu *et al.* [29] stated that the GDDR memory of the graphics card increases the frequency to obtain a higher bandwidth, which causes excessive power consumption. Also, the more GDDR memory, the higher the price of GPU.

Figure 2.13: GeForce GTX Titan X GPU.

2.4.3 Embedded System (NVIDIA Jetson)

The embedded system often used by developers to develop artificial intelligence and deep learning is the NVIDIA Jetson series. Y. Han *et al.* [11] claimed that NVIDIA Jetson TX1 is a supercomputer module. NVIDIA's development tools provide the NVIDIA Jetson TX1 with a Linux environment as a development system for artificial intelligence and deep learning. Moreover, the specifications of NVIDIA Jetson TX1 also meet the basic requirements of standard CNNs. It has included MaxwellTM architecture, which provides for 256 NVIDIA cores and 64-bit ARM CPU cores, so it has excellent power efficiency and performance. Therefore, NVIDIA Jetson's embedded system is regarded as the ideal artificial intelligence development system. Besides, NVIDIA Jetson also includes a variety of standard hardware interfaces that provide a highly flexible and scalable platform to developers. Nevertheless, the embedded system cannot adequately perform the original code to achieve specific tasks. S. Aldegheri *et al.* [30] stated that the Jetson TX2 unable to meet real-time performance while direct using the original source code which, run on desktop computers. Real-time performance only can be achieved in embedded systems when the original code has modified.

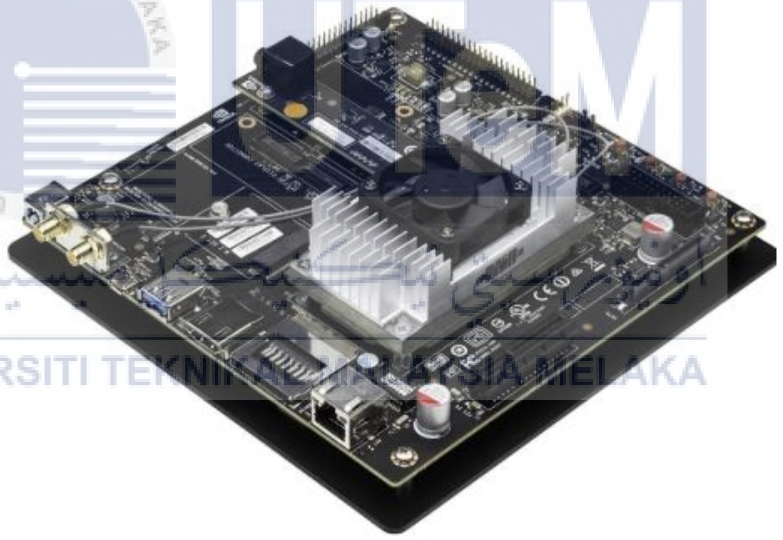


Figure 2.14: NVIDIA Jetson TX1 development kit.

2.4.4 Raspberry Pi

Raspberry Pi is a microcomputer that contains a Linux system, and it supports Scratch and Python language programming. Besides, it can do everything a desktop can do such as browsing the Internet and playing high-definition videos, word processing, and playing games. However, Raspberry Pi is not the first choice in the field of deep learning to become a hardware accelerator and even implements the CNNs. The reason is that CNN required a large number of calculations. To achieve high accuracy, CNN is too large to fit into mobile devices or small devices such as Raspberry Pi. And also, the runtime memory (RAM) required to run these CNN models is even more substantial which is challenging to find in Raspberry Pi [12].

Figure 2.15: Raspberry Pi 3B+.

To solve these problems, Z. Jiao *et al.* [12] proposed a lightweight convolutional neural network that can run fluently on Raspberry Pi 3B+. The design of lightweight CNN is based on the depthwise separable convolution and the improved Linear Bottlenecks block. The experimental results have shown that the Raspberry Pi 3B+ achieved 91% accuracy and the average speed to recognize the image is about 176ms by using the lightweight CNN. Although the lightweight CNN can implement on Raspberry Pi, unfortunately, the shallower CNNs do not achieve human-level accuracy.

2.4.5 Field-Programmable Gate Arrays

Field-programmable gate array（FPGA） is a programmable logic device that contains ten thousand to more than a million logic gates connected with programmable interconnects. The feature of the Field Programmable Gate Array (FPGA) is that FPGA provides the user with the ability to reprogrammable and available to connect the logic blocks inside the FPGA through editable connections so that the data flow can be reconfigured. The primary language used to program the FPGA and construct the FPGA model is HDL (i.e., Verilog and VHDL), which can implement the logic synthesis and the model layout. When the programming process is completed, you need to use a software compiler to verify the function and error and then flash it to the FPGA for testing. During this process, a bitstream file will be created. The bitstream file contains the wiring information of the FPGA components and users need to download the file to the FPGA board via a USB data cable [2].

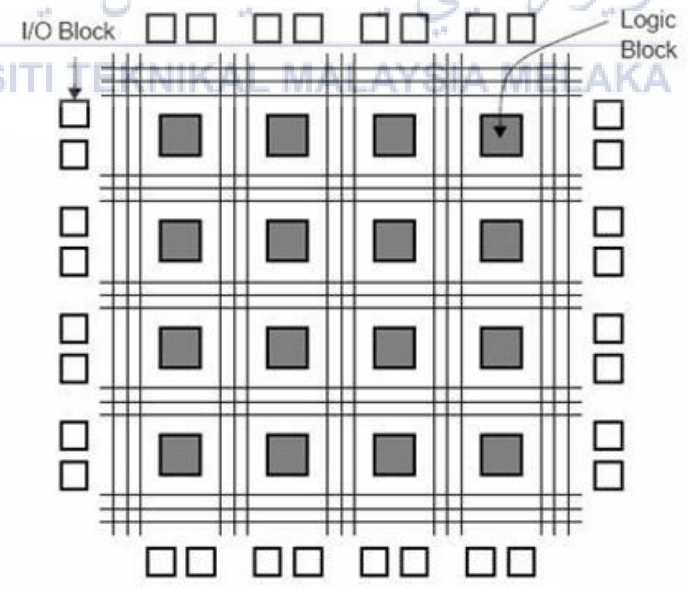


Figure 2.16: Model FPGA chip.

The reason why FPGA can attract developers' focus to research it is that FPGA only takes a short time from the design process to the functional chip, and FPGA does not involve any physical manufacturing procedure. Besides, FPGAs consume less power than ASICs, so it is very friendly to the users [35].

D. Gschwend *et al.* [1] mentioned that, the significant difference between FPGAs and General-Purpose Processors is that FPGAs can program logic blocks freely, which cannot be achieved by GPU and traditional processors. The advantages of FPGAs can be used as highly specialized accelerators to accomplish a particular task, thereby increasing the processing speed of the system and saving energy in the system. Although FPGAs have such advantages to developers, the trade-off is the designers need to consider the available hardware resources during the development process carefully. This reduces the development of agility and increased design complexity. Moreover, the difference between FPGA and Application-Specific Integrated Circuit (ASIC) is that ASIC is customized for a specific purpose, so it is not affected by any area or timing overhead of configuration logic and general-purpose interconnect. Therefore, ASIC usually has the smallest, fastest and most energy-saving effect. However, the manufacturing process of ASIC is very complicated, which leads to a long development cycle and a very high cost for manufactured. So, ASIC is generally given to mass-produced products, so that these products can cover the manufacturing costs with each other. Due to the programmability of FPGAs, it is more suitable for prototyping and shorter development cycles.

D. Danopoulos *et al.* [5] has implemented the DNN on FPGAs and used Caffe as a framework. Researchers have mentioned that hardware acceleration can help to improve the performance and efficacy of deep learning. Their research has proved that the implementation of CNN on FPGA is feasible, and they list FPGA as the preferred platform for CNN accelerator.

2.4.5.1 Zedboard

ZedBoard is a low-cost development board developed by Avnet using Xilinx Zynq-7000 All Programmable SoC. This board combines the Zynq-7000 AP SoC's ARM processor with seven series programmable logic so that the single silicon chip can be used to implement the functionality of the entire system on ZedBoard rather than several different physical chips is required. This combination has proved to be flexible and forms a compelling platform for a wide variety of applications. This SoC solution enables faster and more secure data transfer between the various system elements that have high overall system speed and low power consumption. Designers also can implement their design in different based platforms such as Windows, Linux, and Android on this development board. The target applications of ZedBoard usually for video processing, motor control, software acceleration, different OS development, embedded ARM processing and general Zynq-7000 AP SoC prototyping.

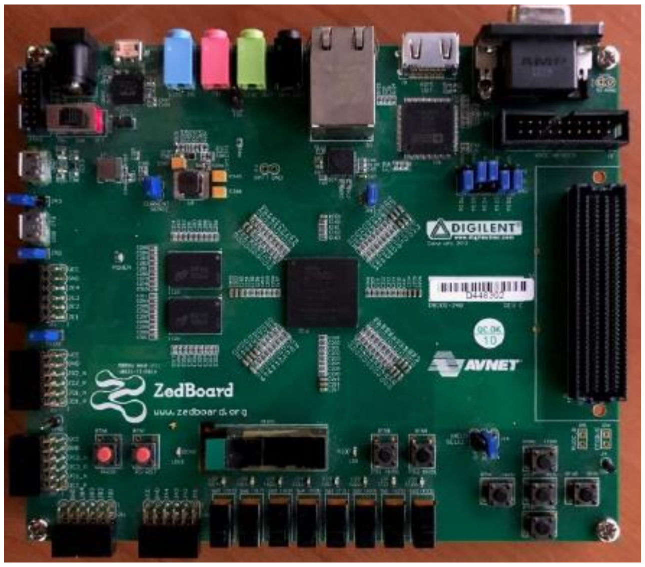


Figure 2.17: Overall view of ZedBoard.

2.4.5.1.1 Deep Processing Unit

Deep Processing Unit (DPU) is a processing engine for the convolutional neural network (CNN) running on Xilinx Zynq-based SoC. DPU IP integrated as a block in the programmable logic (PL) with a direct connection to the processing system (PS). Besides, the DPU IP is designed to be efficient, low latency and scalable for various edge AI applications. DPU IP commonly used as a hardware accelerator to takes full advantage of the Xilinx FPGA architecture to achieve the tradeoff between latency, power and cost. The DPU IP has specialized in the instruction set to work efficiency for CNNs.

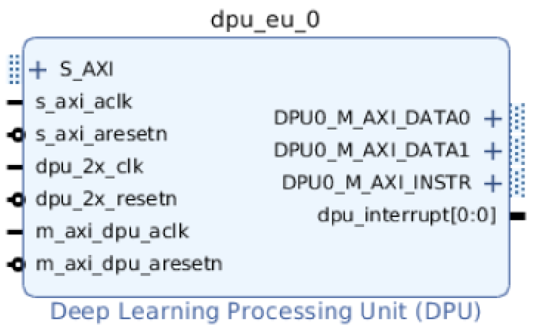


Figure 2.18: Deep Processing Unit (DPU) IP.

The DPU IP mainly contains three parts which are convolution computing module, a configuration module, and a data controller module. The convolution computing module includes a processing engine (PE) that performs all the primary convolution calculations. The configuration module provides user-configurable parameters to optimize the resources for all the support from different features. Lastly, the data controller module schedules the all flow in the DPU IP.

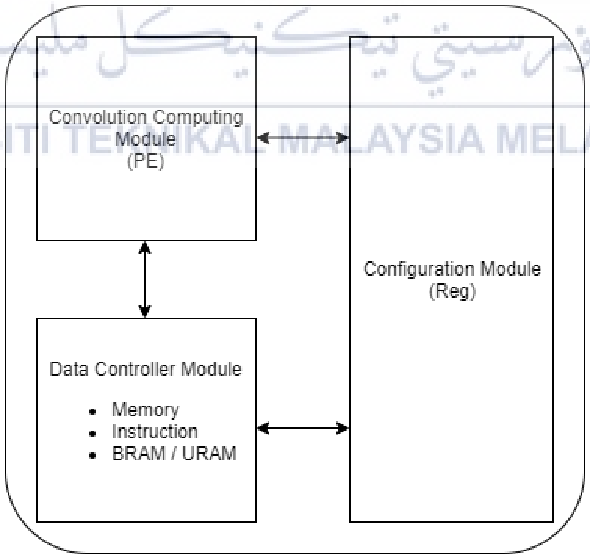


Figure 2:19: Programmable logic (PL) for DPU.

The DPU IP is scalable and configurable means that it is available in different sizes to fit into different FPGAs. For example, the DPU configuration can set as B1024 and B4096. For B1024 DPU configuration targets smaller Zynq devices, smaller configurations and lower parallelism. In contrast, B4096 DPU configuration is for more extensive configuration, higher parallelism, and larger Zynq devices.

2.4.5.2 PYNQ-Z2

PYNQ is a development board embedded with Xilinx Zynq SoC. PYNQ supports the Python language and libraries which allows the developers to implement PYNQ's programmable logic functions directly from Python applications. These libraries that can take advantage of the Zynq SoC's heterogeneous hardware architecture are also called hardware libraries or overlays. The PYNQ is more flexible than other platforms because it has a 12-pin PMOD connector, Arduino and Raspberry Pi compatible interfaces and audio/video I/O pins [13]. Unlike Zedboard, the PYNQ development board does not require any additional software for programming. Xilinx provided PYNQ a Python application called Jupyter Notebook as an online programming tool. Jupyter notebook is a web-based development environment and it supports various workflows in data science, scientific computing, and machine learning. Moreover, Jupyter notebook supports more than 40 programming languages which including the Python.

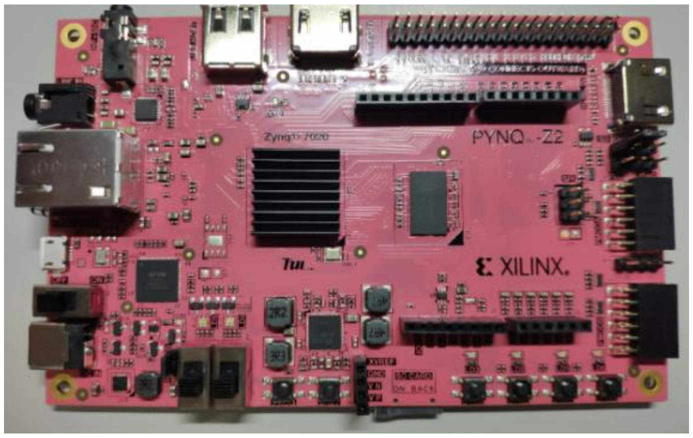


Figure 2.20: Overall view of PYNQ-Z2 board.

2.5 Benchmarking of Implementation CNNs on Different Platform

Table 2.2 shows the benchmarks of previous work during the implementation of CNN on different platforms. Power consumption is essential for systems that rely on plug-in power. If the power consumption is too high, the cost of electricity will increase, and the system will overheat when a large amount of energy is consumed. The frame per second will affect the viewing experience of a video, and the high frame rate can make the video played smoothly. The standard of frame rate for a video is 30fps, which is proposed by H. Nakahara *et al.* [20].

Table 2.2: Benchmark of the previous work.

Author	Year	Approach	Related Work	Platform	Devices	FPS	Power Consumption (W)	Application
C. Zhang *et al.* [9]	2019	VGG16	Utilizing the FPGA to implement the CNNs and show the comparative performance between CPU, GPU and FPGA to accelerate the CNNs.	CPU	E5-2609	-	150	Image Classification
				CPU +GPU	E5-2609 + K40	-	250	
				CPU +FPGA	E5-2609 + VX690t	-	26	
Y. Tu *et al.* [10]	2019	DNN	Utilizing DNNs on FPGA and embedded systems and compared the experimental results with previous works from C. Zhang *et al*. paper [9].	Embedded System +FPGA	Jetson TX2 + A7-100T	-	3.7	
H. Nakahara *et al.* [31]	2017	VGG-11 (CIFAR-10)	Utilizing the binarized VGG-11 on the CIFAR-10 dataset for images classification using the FPGA.	FPGA	ZedBoard	-	2.3	
Y. Umuroglu *et al.* [15]	2017	BNN (MNIST)	Utilizing the BNN inference accelerators on FPGA for classifying the MNIST dataset based on FINN framework.	FPGA	ZC706 Evaluation Kit	-	8.7	

K. Rungsuptaweekoon *et al.* [19]	2017	YOLO	Utilizing the object detection with YOLO on the embedded systems and compared with Tesla P40 for implemented the object detection with YOLO. Then, using the LPIRC system to obtain the energy, but the fps is computed by fps counter.	Embedded System	Jetson TX1	3	10	Object Detection
				Embedded System	Jetson TX2	5	7.5	
				GPU	Tesla P40	42	250	
H. Nakahara *et al.* [20]	2018	YOLOv2	Utilizing the FPGA (Zynq Ultrascale+ MPSoC) to implement the YOLOv2 and compared with embedded CPU and GPU.	Embedded CPU	ARM Cortex-A57	0.23	4	
				Embedded GPU	Pascal GPU	2	7	
				FPGA	Zynq Ultra MPSoC	35.71	4.5	

By referring to Table 2.2, for the application of image classification, the combination of CPU and GPU has the highest power consumption compared with FPGAs. Besides, the combination of embedded systems and FPGA achieved lower power consumption, which is closed to FPGA. For the application of object detection, the GPU platform achieved the highest frame per second (fps) during the implementation of YOLO. Although the GPU has the best performance of fps, the embedded system is the most suitable choice to implement the YOLO with high accuracy and low power consumption which is suggested by K. Rungsuptaweekoon *et al.* [19]. Moreover, Nakahara *et al.* [20] proposed the frame rate of FPGA is 35fps, which meets the standard frame rate of video and also higher than the fps of embedded CPU and embedded GPU.

2.6 Chapter Summary

This chapter summarized the characteristic, advantages and disadvantages of each platform for CNNs implementation. In addition, CNN's network structure has been discussed in detail. The ResNet50, YOLOv2, BNN and LeNet-5 are introduced and discussed the network architecture, respectively. Lastly, the benchmark of previous work for the power consumption and frames rate of each platform is evaluated and analyzed to determine the suitable platform to accelerate the CNNs in order to meet the requirements of low power consumption and high accuracy at the same time.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter will present the methodology to create the IP subsystems with the custom IP core to accelerate the YOLOv2 and ResNet50 model on FPGAs. This process performed using Vivado Development Tools. For Zedboard, the block design is validated correctly, then we move on to build the application templates for CNNs by using PetaLinux so that we can easily compile and install the applications into the root file system. Next, the hardware connection to implement the CNN on ZedBoard for digits recognition will explain in this chapter. For PYNQ-Z2, the installation of Linux OS is required in order to implement the YOLOv2 and BNN. Moreover, the environmental setup to implement the CNNs on FPGAs has provided in this chapter and the explanation operation data flow on FPGAs will be covered also.

3.2 Flow Chart

Figure 3.1 shows the flow chart of ZedBoard from the beginning to the end for the implementation of ResNet50 and YOLOv2. Firstly, the Vivado HLS software used to export the YOLOv2 IP by using the code in C language, while the DPU IP is provided inside the DNNDK [33]. Secondly, the target IP needs to import to the repository of Vivado software for further development. Then, start to connect the ResNet50 or YOLOv2 block design by referring the Figure 3.20 and Figure 3.21 respectively and valid the block diagram to ensure there is no error in the design. Once the block design is validated, generate the HDL file of the design and copy to the Petalinux's project directory. Lastly, we can configure the rootfs file and BOOT.bin file by using the PetaLinux and copy both files to the SD card.

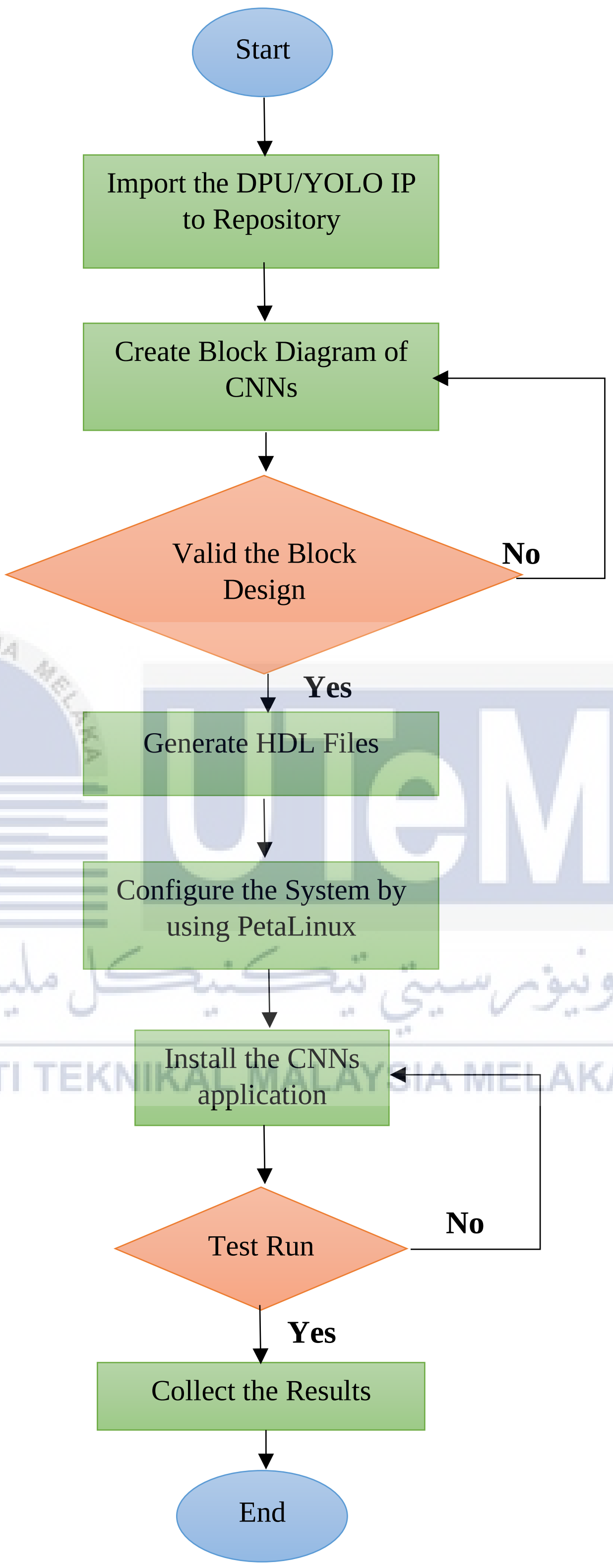

Start
Import the DPU/YOLO IP to Repository
Create Block Diagram of CNNs
Valid the Block Design
No
Yes
Generate HDL Files
Configure the System by using PetaLinux
Install the CNNs application
Test Run
No
Yes
Collect the Results
End

Figure 3.1: Flow Chart of ZedBoard for ResNet50 and YOLOv2

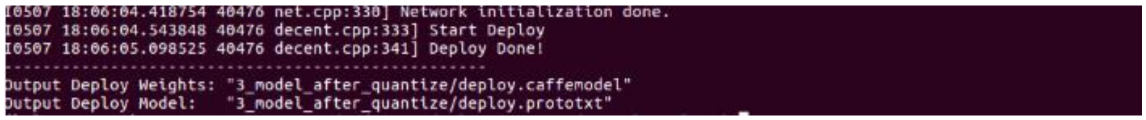
3.3 ZedBoard

3.3.1 Quantization and Compilation of pre-trained model

In this project, the pre-trained model for ResNet50 will be used to demonstrate the CNNs application. There is no allow the original pre-trained model of CNN directly implement on ZedBoard due to the large bandwidth and bits number of the weight file. To solve this problem, the quantization and compilation is required to apply for the pre-trained model of ResNet50.

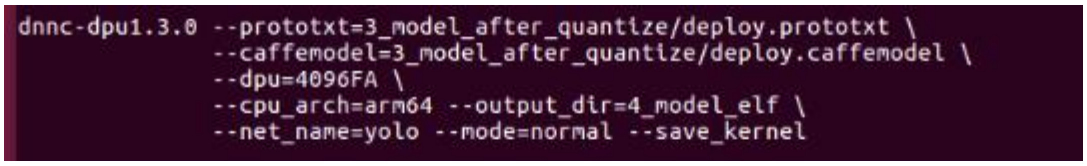
Quantization is playing an important role to reduce the bandwidth or the bits number so that the ZedBoard can meet the requirement to implement the CNNs computation. This is because the original source files of CNNs are huge and consist of 32-bit of floating-point. The quantization process is able to minimize the number of bits to 8-bit, so CNN's computation demands can be reduced and increased the power efficiency on the FPGA board. However, the quantization of CNN models will not incur a significant loss in accuracy. Xilinx has provided an AI Edge Toolkit called DNNDK to allow users to quantize and compile the models of CNN before implementing CNNs on ZedBoard.

After quantized the CNN models, the process of compilation required to take part so that the ZedBoard can run the compiled CNN model in SD card. In this process, the elf file will be generated by the DNNDK toolkit after compiled the CNN models by using the DNNDK toolkit[33]. The elf file is the same as compressed CNN’s file. While demonstrating the CNN models on ZedBoard, the elf file will temporarily export all CNN’s usage files such as libraries, frameworks, coding, etc.



```
I0507 18:06:04.418754 40476 net.cpp:330] Network initialization done.
I0507 18:06:04.543848 40476 decent.cpp:333] Start Deploy
I0507 18:06:05.098525 40476 decent.cpp:341] Deploy Done!
--------------------------------------------------
Output Deploy Weights: "3_model_after_quantize/deploy.caffemodel"
Output Deploy Model:   "3_model_after_quantize/deploy.prototxt"
```

Figure 3.2: Files generated after quantization.



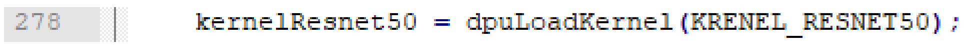
```
dnnc-dpu1.3.0 --prototxt=3_model_after_quantize/deploy.prototxt \
              --caffemodel=3_model_after_quantize/deploy.caffemodel \
              --dpu=4096FA \
              --cpu_arch=arm64 --output_dir=4_model_elf \
              --net_name=yolo --mode=normal --save_kernel
```

Figure 3.3: Script to compile prototxt and caffemodel

3.3.2 Create Inference Code and Deploy Code in C/C++

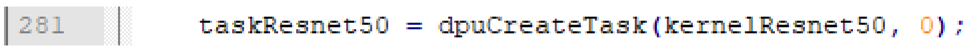
The inference codes for the ZedBoard use to run image classification in this project. The programming language required to create the inference code is C/C++ programming. For the ResNet model, the size of the input needs to set as 640 x 480 pixels and set the ethernet IP address the same as the subnet of the host computer. This is because the output will export to the host display from ZedBoard by following the IP address through the ethernet cable. Next, the directory of the input image needs to indicate at the function, so that the ResNet50 can detect the images and classify those images.

In this project, the DPU IP is an accelerator to accelerate the CNN in ZedBoard. Therefore, the DPU kernel for running ResNet50 is required to define in the inference code in C/C++. The CNN models of ResNet50 will compute through the hardware implementation, but not the software implementation. Besides, the inference code files must be renamed as main.cc(file in C/C++).



```
kernelResnet50 = dpuLoadKernel(KRENEL_RESNET50);
```

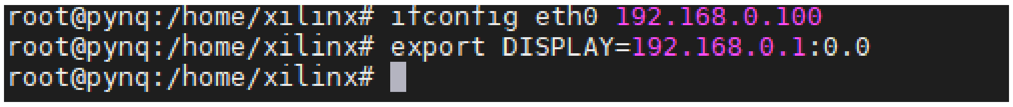
(a)



```
taskResnet50 = dpuCreateTask(kernelResnet50, 0);
```

(b)

Figure 3.4: ResNet50 coding(a)Load DPU kernel for ResNet50(b)Create DPU task for ResNet50



```
root@pynq:/home/xilinx# ifconfig eth0 192.168.0.100
root@pynq:/home/xilinx# export DISPLAY=192.168.0.1:0.0
root@pynq:/home/xilinx# 
```

Figure 3.5: Export the ZedBoard output to host display.

3.3.3 Run the Makefile and Demostarte the Models

Makefile is a file that used to install or build the defined packages in the system. In this project, makefile is used to install the created inference code in the Zedboard operating system. The name of the model and directory file needed to indicate inside the makefile. When executing the makefile on a Zedboard requires Linux instructions to make it install the CNN model. After the terminal shows the main.o file has built, the installation of the CNN model on Zedboard has completed. To demonstrate the CNN models in the SD card, the Linux command to execute the CNN model is needed to implement by using the Linux terminal regarding the path directory. After run the command, the model of ResNet50 will begin.

3.4 CNN on ZedBoard for Digits Recognition

This CNN is designed based on LeNet-5 architecture for the implementation of real-time digits recognition on ZedBoard. Also, this design is a simplified version of the LeNet-5 model. As mention in Chapter 2, the LeNet-5 architecture contains three convolutional layers and two fully-connected layers, but there are only three convolutional layers and one fully-connected layer in this CNN for digits recognition and it is only using maximum pooling layer. Besides, the steps to run this CNN on ZedBoard to achieve digit recognition is also very simple. Once connected the ZedBoard to the personal computer via the USB cable, then start using the Vivado software to deploy the structure network of the CNN with Verilog. After that, start running the synthesis and implementation design to check whether the codes have errors or not. After complete synthesis and implementation design, generate the bitstream file and upload to the PL part of ZedBoard via the data cable. Lastly, connect the ZedBoard to the monitor with VGA port and begin to implement the digit recognition. Figure 3.6 shows the flow chart of CNN on ZedBoard for digits recognition.

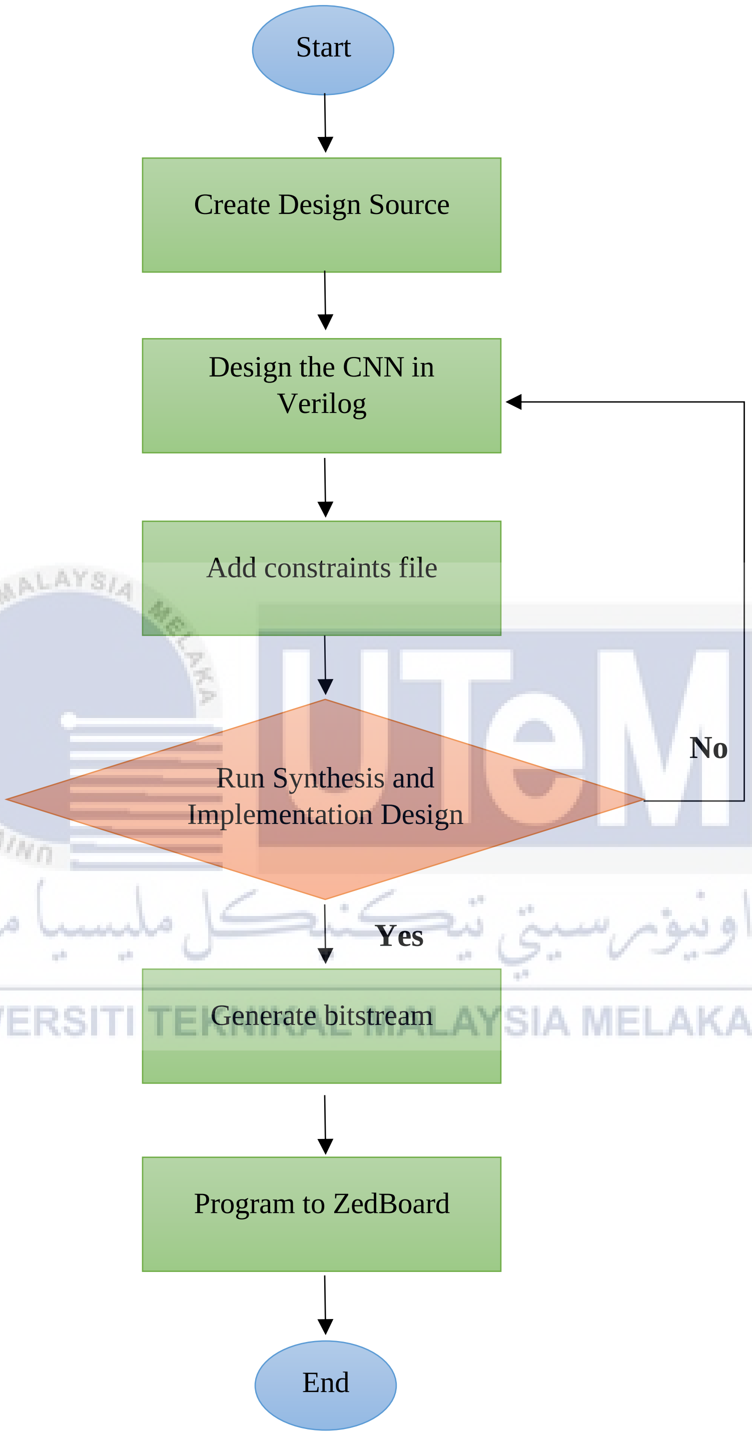


Figure 3.6: Flow Chart of CNN on ZedBoard for Digit Recognition

3.4.1 Architecture of CNN on ZedBoard for Digits Recognition

As mention in Section 3.4, this network architecture is the simplified version of LeNet-5. It only contains three convolutional layers and one fully-connected layer in this design. The input image of the CNN for digits recognition is a resized grayscale image of 32x32. The size of the convolution kernel in the first convolution layer, Conv_1 is 5x5. The maximum pooling layer S2 performs downsampling with 2x2 and stride two followed by a Sigmoid activation function for non-linear processing. Next, the size of the convolution kernel of the second convolutional layer, Conv_2 is 5x5 and the maximum pooling layer of the second convolutional layer S4 same as the S2 downsampling with 2x2 and stride two. The third convolutional layer, Conv_3 is used 5x5 convolution kernel which is same as the size of the input third layer, so the output of the third convolutional layer becomes a fully-connected layer. Figure 3.7 shows the network architecture of CNN on ZedBoard for digit recognition.

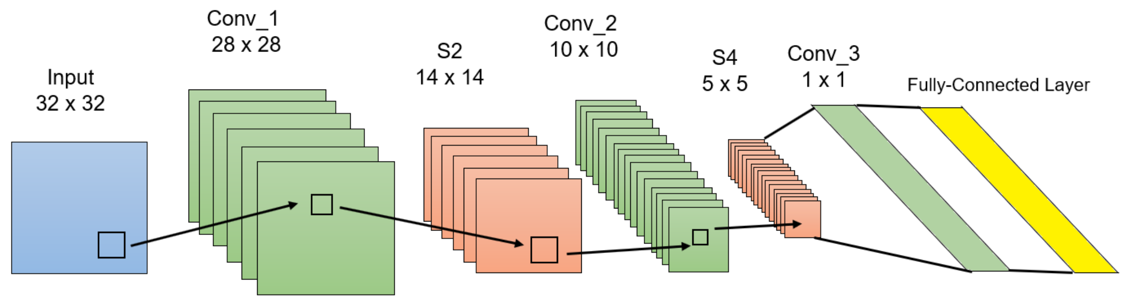
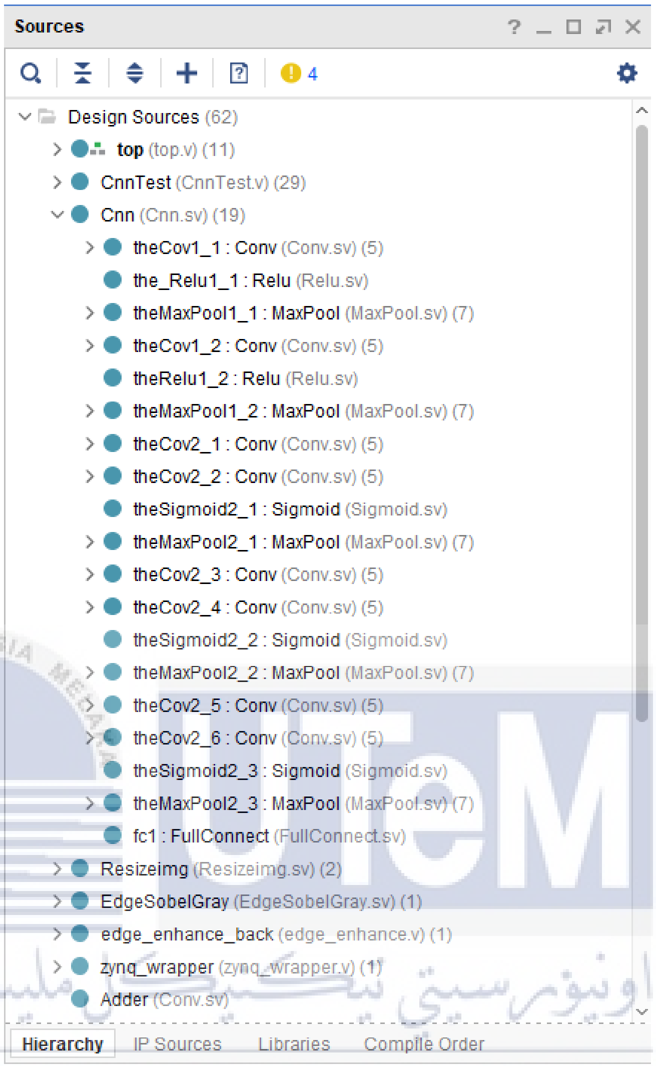


Figure 3.7: Architecture of CNN on ZedBoard for digit recognition


Sources

Design Sources (62)
- top (top.v) (11)
- CnnTest (CnnTest.v) (29)
- Cnn (Cnn.sv) (19)
  - theCov1_1 : Conv (Conv.sv) (5)
  - the_Relu1_1 : Relu (Relu.sv)
  - theMaxPool1_1 : MaxPool (MaxPool.sv) (7)
  - theCov1_2 : Conv (Conv.sv) (5)
  - theRelu1_2 : Relu (Relu.sv)
  - theMaxPool1_2 : MaxPool (MaxPool.sv) (7)
  - theCov2_1 : Conv (Conv.sv) (5)
  - theCov2_2 : Conv (Conv.sv) (5)
  - theSigmoid2_1 : Sigmoid (Sigmoid.sv)
  - theMaxPool2_1 : MaxPool (MaxPool.sv) (7)
  - theCov2_3 : Conv (Conv.sv) (5)
  - theCov2_4 : Conv (Conv.sv) (5)
  - theSigmoid2_2 : Sigmoid (Sigmoid.sv)
  - theMaxPool2_2 : MaxPool (MaxPool.sv) (7)
  - theCov2_5 : Conv (Conv.sv) (5)
  - theCov2_6 : Conv (Conv.sv) (5)
  - theSigmoid2_3 : Sigmoid (Sigmoid.sv)
  - theMaxPool2_3 : MaxPool (MaxPool.sv) (7)
  - fc1 : FullConnect (FullConnect.sv)
- Resizeimg (Resizeimg.sv) (2)
- EdgeSobelGray (EdgeSobelGray.sv) (1)
- edge_enhance_back (edge_enhance.v) (1)
- zynq_wrapper (zynq_wrapper.v) (1)
- Adder (Conv.sv)

Figure 3.8: Layer of CNN for digits recognition

3.4.2 Connection of OV7670 and Seven Segment Display LED to ZedBoard

For the implementation of this CNN, the output of digit recognition will appear on the seven segment display LED while the OV7670 camera is used to capture the patterns of a digit number as an input. Therefore, the components needed for this implementation are seven segment display LED, OV7670 camera, and some jumper wires. Figure 3.9 shows the method to connect the ZedBoard with those components, and Table 2.3 and Table 2.4 shows the net names of the FPGA pins and the connections to camera and seven segment display respectively.

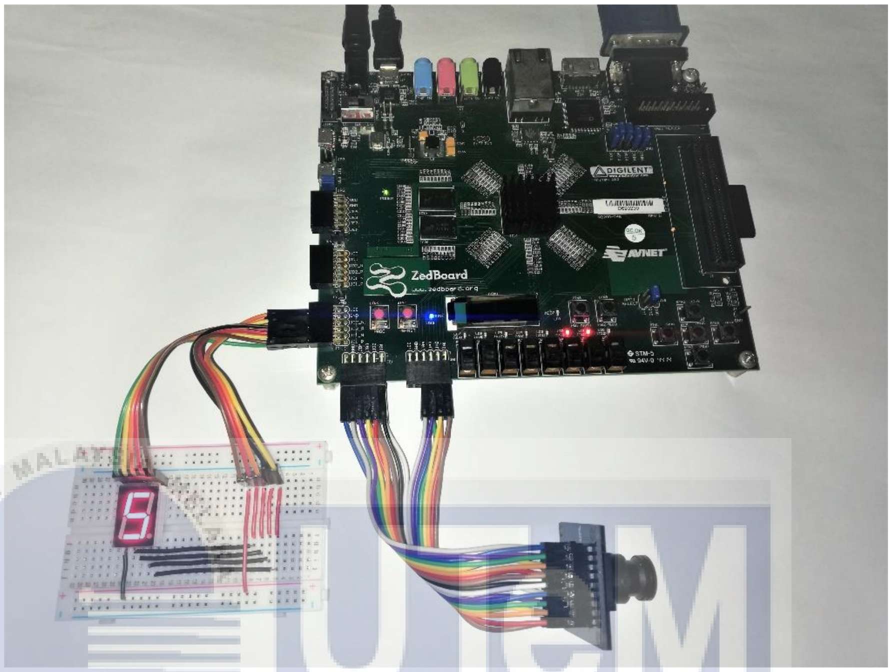


Figure 3.9: Connection of ZedBoard for digit recognition

Table 2.3: The connection of FPGA pins to OV7670 camera

Pmod	Signal Name	ZedBoard Pin	OV7670 Pin
JA1	JA1	Y11	PWDN
	JA2	AA11	D0
	JA3	Y10	D2
	JA4	AA9	D4
	JA7	AB11	RST
	JA8	AB10	D1
	JA9	AB9	D3
	JA10	AA8	D5
JB1	JB1	W12	D6
	JB2	W11	XCLK
	JB3	V10	HRE

	JB4	W8	SDA
	JB7	V12	D7
	JB8	W10	PCLK
	JB9	V9	VSY
	JB10	V8	SCL

Table 2.4:The connection of FPGA pins to seven segment display LED

Pmod	Signal Name	ZedBoard Pin	Seven Segment Diplay Led Pin
JC1 Differential	JC1_N	AB6	e
	JC1_P	AB7	d
	JC2_N	AA4	c
	JC2_P	Y4	dp
	JC3_N	T6	f
	JC3_P	R6	g
	JC4_N	U4	a
	JC4_P	T4	b

3.5 PYNQ-Z2

Figure 3.10 shows the flow chart of PYNQ-Z2 for the implementation of BNN. Compared with Figure 3.1, the flow chart of PYNQ-Z2 is simpler than ZedBoard when implementing the CNN.This is because the PYNQ-Z2 is supported by the Python libraries and framework, so that it able PYNQ-Z2 develop the CNN with Python language. Firstly, the operating system required to download from the official [32] and flash it on an SD card. The most important step is to connect the IP address of the host computer as 192.168.2.99 which is the default IP address of PYNQ-Z2. Once the PYNQ-Z2 is connected with the host computer, then go to the browser and load to the IP address of PYNQ-Z2. So, we can download the BNN framework and install the BNN package through a command on the console shown in Figure 3.13.

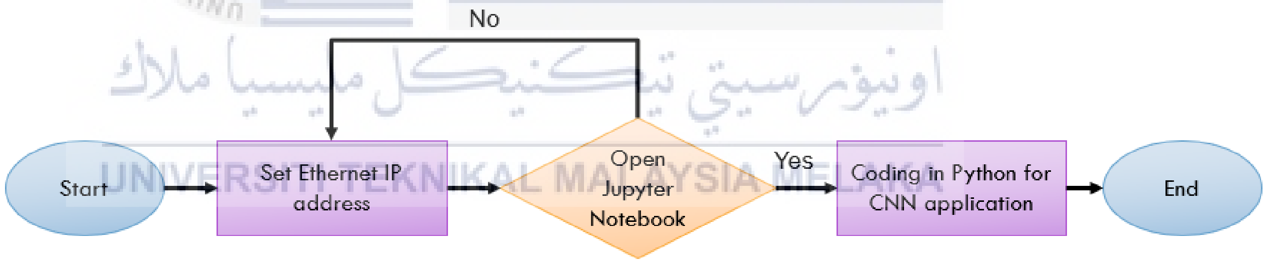


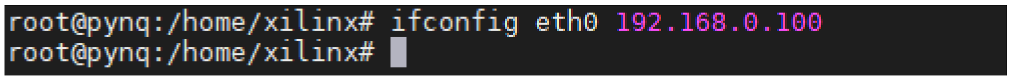
Figure 3.10: Flow chart of PYNQ-Z2.

3.5.1 Operating System Installation

To implement the Convolution Neural Networks on FPGA, there is required to burn the img file of the PYNQ-Z2[32] to SD card as the operating system for the boards. This is because the img files for the PYNQ-Z2 which contain the Linux framework and environment. The Linux operating system is an open-source operating system and suitable to implement CNNs on the FPGA board because it is supported the Python libraries so that the CNNs framework able to operate in this system.

3.5.2 Run the Jupyter Notebook

Jupyter Notebook is an online programming tool that allows users to execute the coding directly. Due to the supported Jupyter Notebook in the PYNQ-Z2, the flow chart in Figure 3.10 shows the implementation of the CNNs on PYNQ-Z2 is very efficient than ZedBoard. This means that the PYNQ-Z2 only takes a short cycle to implement CNNs applications such as image classification, and YOLOv2. The advantages of Jupyter Notebook has helped the PYNQ-Z2 does not require the extra softwares or tools to achieve the quantization and compilation of CNN models. To run the Jupyter Notebook, the IP address of PYNQ-Z2 must be the same as the host computer so that the host computer can display the files and Jupyter Notebook from PYNQ-Z2 through the ethernet cable.



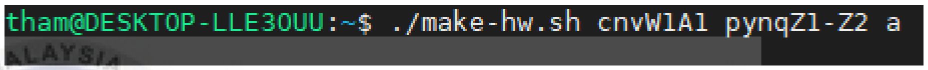
```
root@pynq:/home/xilinx# ifconfig eth0 192.168.0.100
root@pynq:/home/xilinx# 
```

Figure 3.11: Connect PYNQ-Z2 IP address to host PC.

3.5.3 Build Hardware and Software Design for BNN

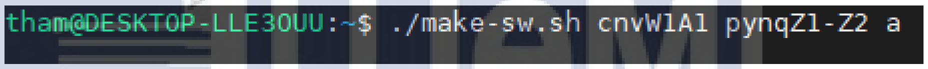
Vivado Design Suite is the main software used to build the hardware and software design of BNN. Xilinx provided the repository of BNN to allow users to design their tasks on the Linux machine. Besides, the repository included two different networks topologies which are LFC network (Three-Layer Fully

Connected network) topology and CNV network (Convolution Network) topology. In paper Y. Umuroglu *et al.* [15], the LFC network is used to classify the MNIST dataset while CNV network is used to classify CIFAR-10 dataset. In this project, the 1-bit weights and 1-bit activation (W1A1) for CNV and LFC used to build the BNN model. Launch the shell script make-hw.sh and make-sw.sh with passing parameters for the target network and the target platform with the command and the output report will be generated in a text file.



```
tham@DESKTOP-LLE3OUU:~$ ./make-hw.sh cnvW1A1 pynqZ1-Z2 a
```

(a)



```
tham@DESKTOP-LLE3OUU:~$ ./make-sw.sh cnvW1A1 pynqZ1-Z2 a
```

(b)

Figure 3.12: Command to build design of BNN (a) Hardware design (b) Software design

3.5.4 Package Installation for BNN

In order to run the BNN model in PYNQ-Z2, the BNN package is required to install through the Linux terminal on PYNQ-Z2. The purpose to the package is to build the framework of FINN for BNN. There are some source files contains in BNN package such as FINN libraries, LFC and CNV classifier, bitstreams for PYNQ-Z2 and pre-trained MNIST and Cifar10 dataset.

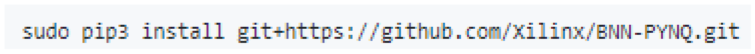
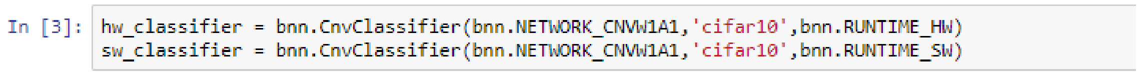
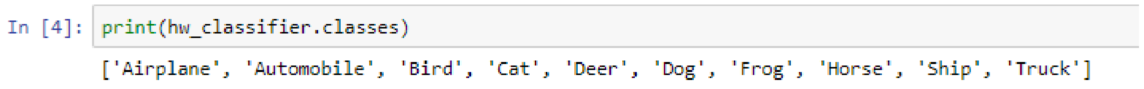

```
sudo pip3 install git+https://github.com/Xilinx/BNN-PYNQ.git
```

Figure 3.13: Command of BNN package installation.

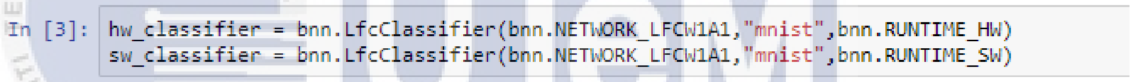

```
In [3]: hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,'cifar10',bnn.RUNTIME_HW)
        sw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,'cifar10',bnn.RUNTIME_SW)
```

(a)

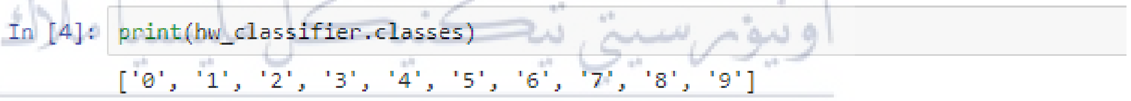

```
In [4]: print(hw_classifier.classes)

        ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']
```

(b)

Figure 3.14: Coding for Cifar10(a)Using CNV classifier(b)10 Classes classify by CIFAR-10.


```
In [3]: hw_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A1,"mnist",bnn.RUNTIME_HW)
        sw_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A1,"mnist",bnn.RUNTIME_SW)
```

(a)


```
In [4]: print(hw_classifier.classes)

        ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

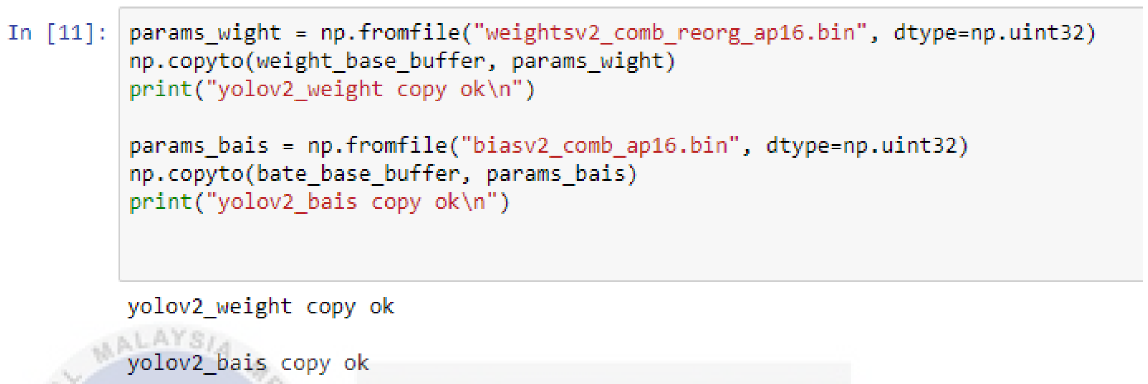
(b)

Figure 3.15:Coding for MNIST(a)Using LFC classifier(b)Digits classify by MNIST.

3.5.5 Darknet Installation for YOLOv2

In this project, the framework of YOLOv2 used for object detection called Darknet[34]. Darknet is an open-source and serves for YOLO usually Darknet uses to train the neural network for the object detection. The repository of Darknet contains some pre-trained weights (Yolov2.weight) and configuration

(Yolov2.cfg) files which trained on different architectures. The dataset used to train the YOLOv2 model is COCO dataset and there are 80 classess availabe to classify by YOLOv2.



```
In [11]: params_wight = np.fromfile("weightsv2_comb_reorg_ap16.bin", dtype=np.uint32)
         np.copyto(weight_base_buffer, params_wight)
         print("yolov2_weight copy ok\n")

         params_bais = np.fromfile("biasv2_comb_ap16.bin", dtype=np.uint32)
         np.copyto(bate_base_buffer, params_bais)
         print("yolov2_bais copy ok\n")

         yolov2_weight copy ok

         yolov2_bais copy ok
```

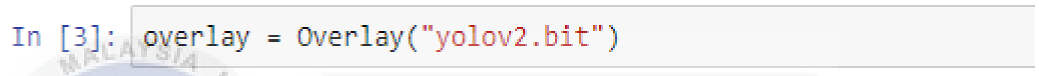
Figure 3.16: Coding applies the pre-trianed weights and bias.

3.5.6 Create and Run the inference code in Python.

Most of the machine learning and deep learning are developed based on Python. Therefore, the Jupyter Notebook is able directly to create and run the CNN models in Python. In this project, the inference codes of YOLOv2 and Cifar10 are created in Python, so that the inference codes can execute in the Jupyter Notebook.

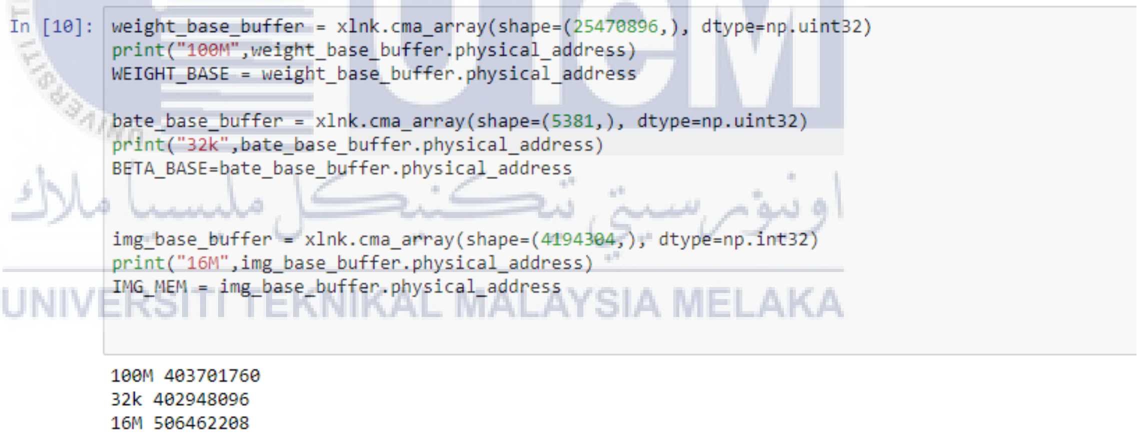
For the YOLOv2, the Vivado software required to use to generate the bin file and tcl file. The bin file contains the block design of YOLO IP core with PYNQ Processor, while the tcl file is the declaration of the I/O pins for the IP integrator. To implement the YOLOv2, these two files are crucial because the bin file is allowing the FPGA computes the YOLOv2 algorithm through the hardware implementation. The input size of images is not permitted to exceed 760 x 480 pixels. This is because the very high definition of images will cause the PYNQ-Z2 has ran out of its memory.

For the BNNs models, the bin file does not require to include inside the coding. According to the [15], the binarized CNNs have good performance and high efficiency allows the small model of devices to implement CNNs without reducing the accuracy of CNN performance. Therefore, minimize the weights and floating-points of CNNs is one of the approaches to accelerate the CNN on FPGA.



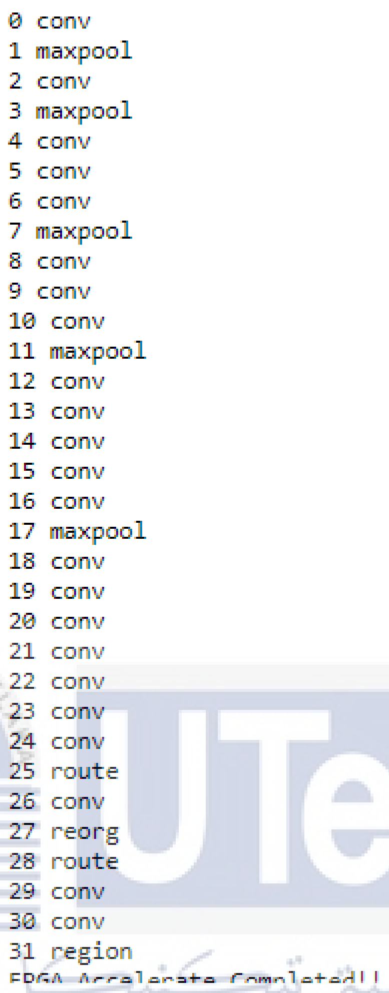
```
In [3]:  overlay = Overlay("yolov2.bit")
```

Figure 3.17: Import the YOLOv2 IP.



```
In [10]: weight_base_buffer = xlnk.cma_array(shape=(25470896,), dtype=np.uint32)
         print("100M",weight_base_buffer.physical_address)
         WEIGHT_BASE = weight_base_buffer.physical_address

         bate_base_buffer = xlnk.cma_array(shape=(5381,), dtype=np.uint32)
         print("32k",bate_base_buffer.physical_address)
         BETA_BASE=bate_base_buffer.physical_address


         img_base_buffer = xlnk.cma_array(shape=(4194304,), dtype=np.int32)
         print("16M",img_base_buffer.physical_address)
         IMG_MEM = img_base_buffer.physical_address

         100M 403701760
         32k 402948096
         16M 506462208
```

Figure 3.18: Allocated the memory of PYNQ-Z2.



```
0 conv
1 maxpool
2 conv
3 maxpool
4 conv
5 conv
6 conv
7 maxpool
8 conv
9 conv
10 conv
11 maxpool
12 conv
13 conv
14 conv
15 conv
16 conv
17 maxpool
18 conv
19 conv
20 conv
21 conv
22 conv
23 conv
24 conv
25 route
26 conv
27 reorg
28 route
29 conv
30 conv
31 region
FPGA Accelerate Completed!!
```

Figure 3.19: Network structure of YOLOv2

3.6 Designing Block Design and HDL Files Generation

After successfully importing IP into the IP repository, we are going to start creating the block design by using the Vivado Tools. Figure 3.5 and Figure 3.6 show the block design mapping of ResNet50 and YOLOv2, respectively.

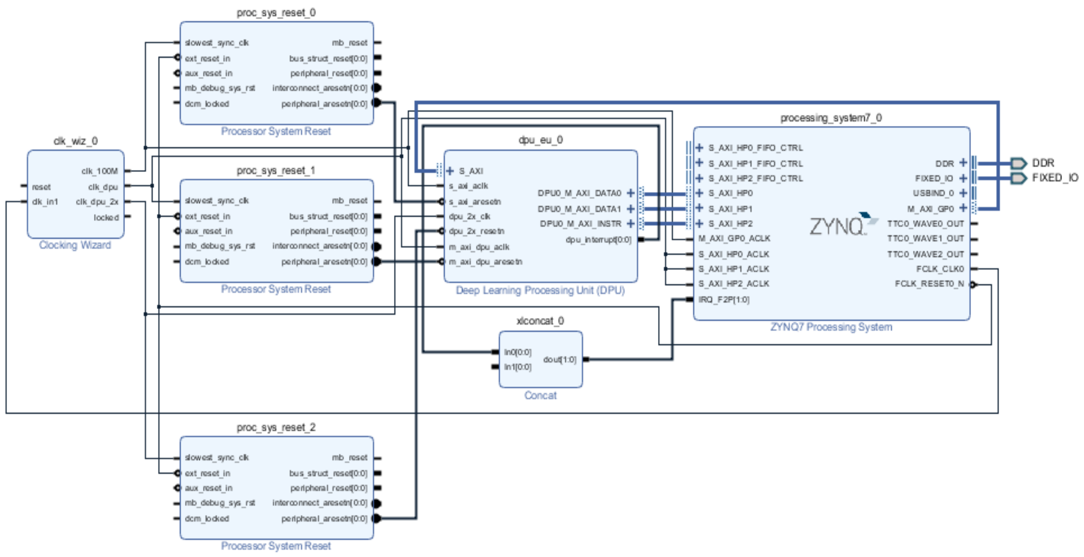


Figure 3.20: Block Design of ResNet50.

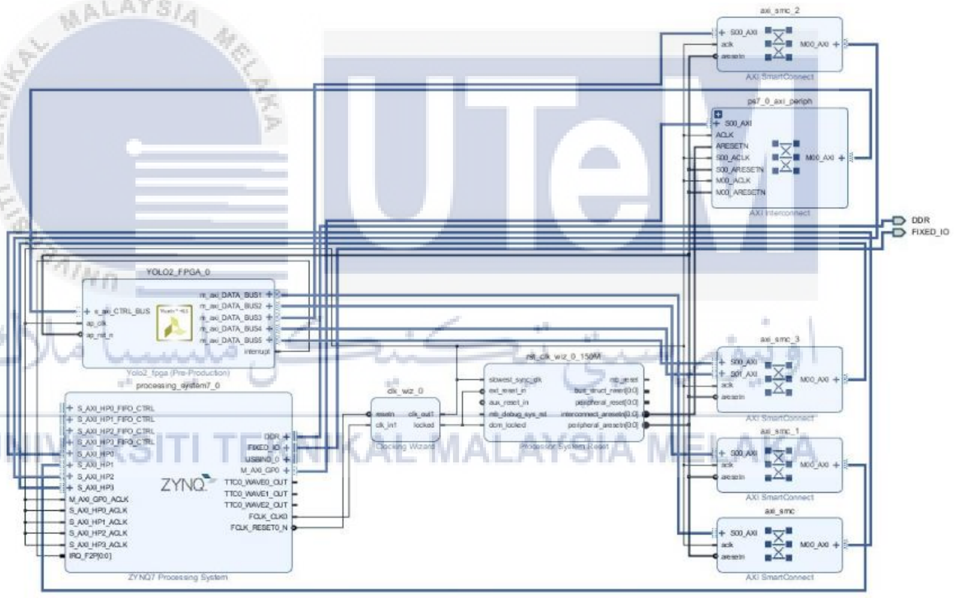


Figure 3.21: Block Design of YOLOv2.

The ARM core processing_system7_0 is the processing system related to Zynq-7000 SoC. Then, we added the clocking wizard into our module and set the output clock as 150MHz for both designs. Once the connection is complete, we required to validate the block design to check the connection error. Next, we can launch the synthesis and implementation run if there are no warning messages shown in the terminal. The results of post-synthesis and post-implementation utilization resources and power consumption will show in graphical view where we can see in Chapter 4 later. Lastly, we exported the bitstream file and HDL file to the Vivado Tools directory path and built the application template for RedsNet50 and YOLOv2.

3.7 Data Flow of CNNs on FPGAs

3.7.1 Data Flow of ResNet50 and YOLOv2 on ZedBoard

The inference operations perform using the DPU and YOLO IP have three states. The three states are image pre-processing, compute and image post-processing. Image processing the process that occurred on each image or frame before it is fed to the network. In the deep learning, each neural network is required the pre-processing. For the compute state, this is the processing in the IP integrator to accelerate the elements of the neural network graph in the Zynq device's programmable logic using the DPU IP and YOLO IP core. For the image post-processing state, this is the process that occurred following the inference.

Firstly, the ARM processor on the processing system (PS) side of the Zynq device reads the image files from the SD card or resolution video from webcam and then loads the frames into PS DDR. After this, the ARM processor scales the frame size according to the input specification of the YOLOv2 or ResNet50 and stores back to the PS DDR. For the ResNet50, the ARM processor writes the Configuration Module to set up DPU for image processing and starts execution. The IP uses the DMA from the data controller module to fetch the image or video frame from PS DDR. The IP runs the inference on the frame using the local memory such as Block RAM(BRAM) or Ultra RAM(URAM).

Lastly, the IP writes the output activations back to PS DDR and the ARM processor will read the results from PS DDR and display the results of detection or classification.

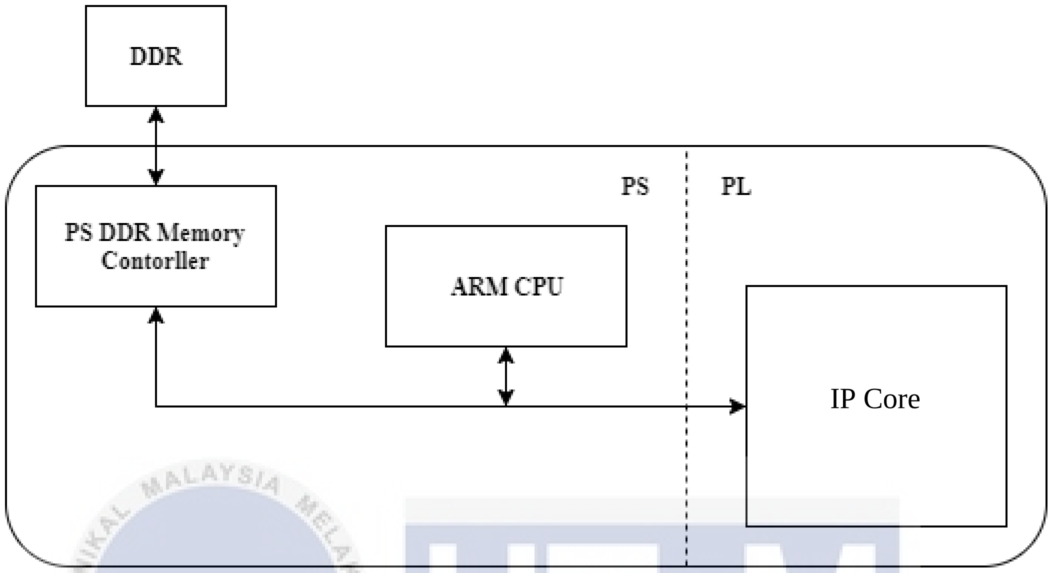


Figure 3.22: Operation and Data Flow for ZedBoard.

3.7.2 Data Flow of BNN on PYNQ-Z2

For the BNN in PYNQ-Z2, there are two different types of implementation called hardware implementation and software implementation. Although these two types of implementation are running on PYNQ-Z2, the operation flows and requirements are very different. For the hardware implementation, the computation of BNN is more rely upon PL on PYNQ-Z2. Besides, there is a function called overlays, or hardware libraries used to accelerate the software application in PYNQ-Z2. The theoretical of hardware implementation to provide acceleration is because of the image processing functions on the FPGA fabric in PL controlled from Python running in PS. Firstly, the input image in SD card passed into the PL and the image classification performed. Then, the Python application programming interface (API) will be responsible for resizing the input image to the format required by CIFAR-10 and MNIST networks and transferring the image between the hardware and software. For the software implementation, the BNN is purely using the embedded arm processor Cortex-A9 in PS to implement the image classification with Python libraries and framework. As a comparison, the same input image can be classified using a software implementation of the algorithm on PYNQ-Z2.

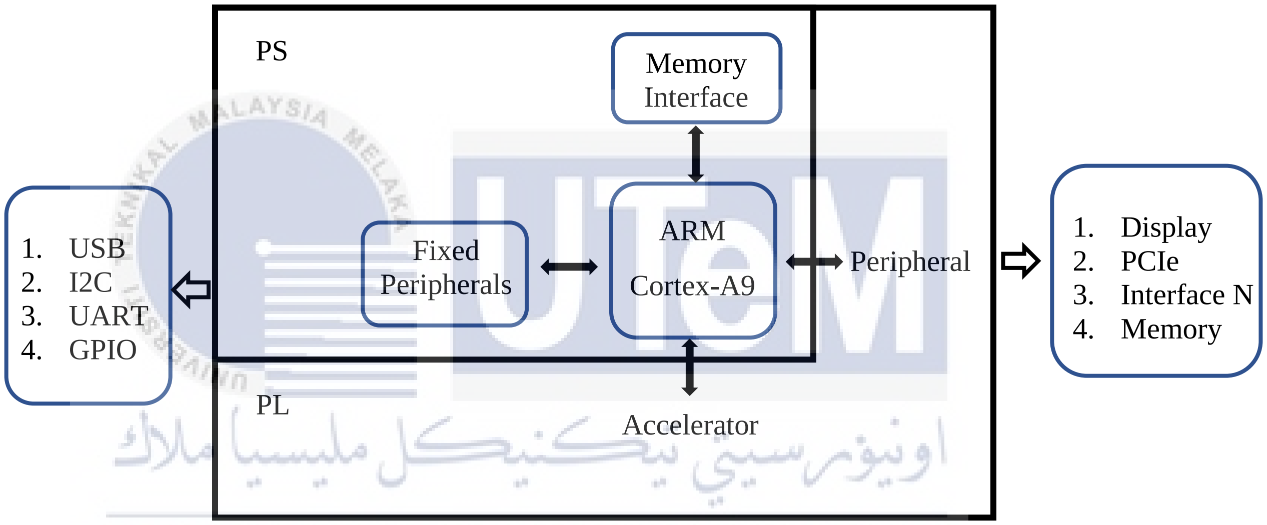


Figure 3.23: Operation and Data Flow for PYNQ-Z2.

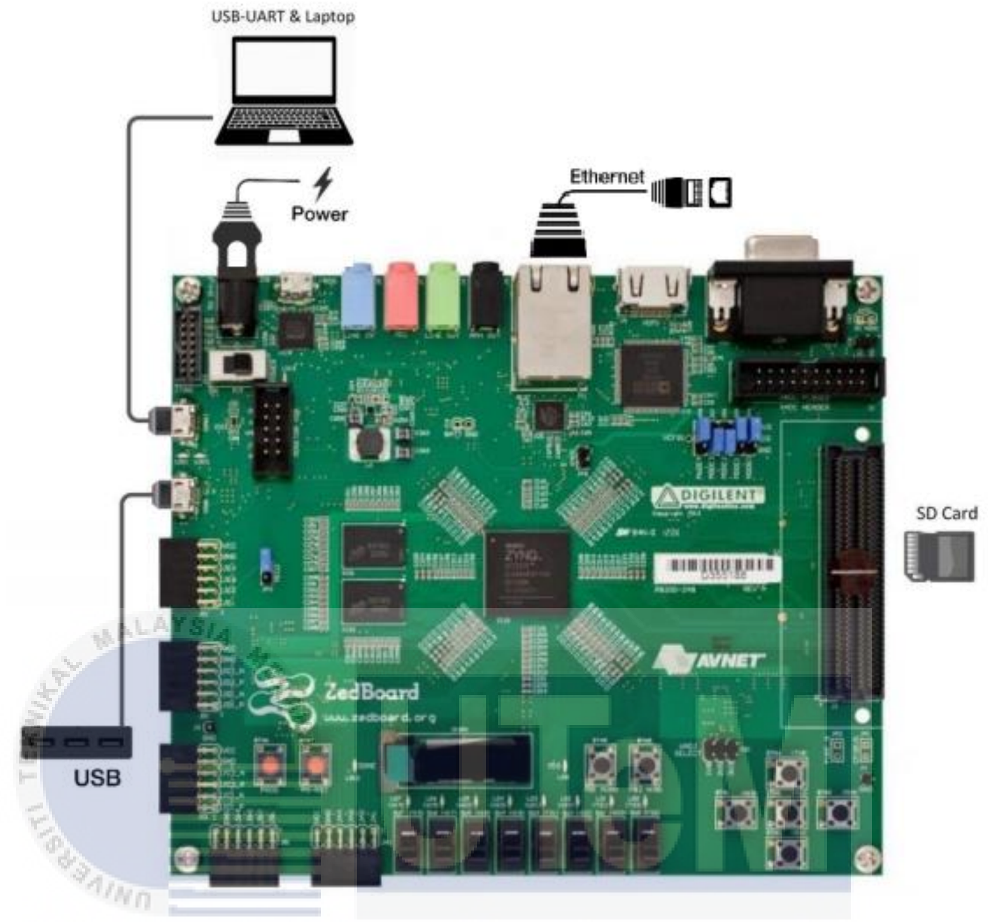


Figure 3.24: ZedBoard Setup.

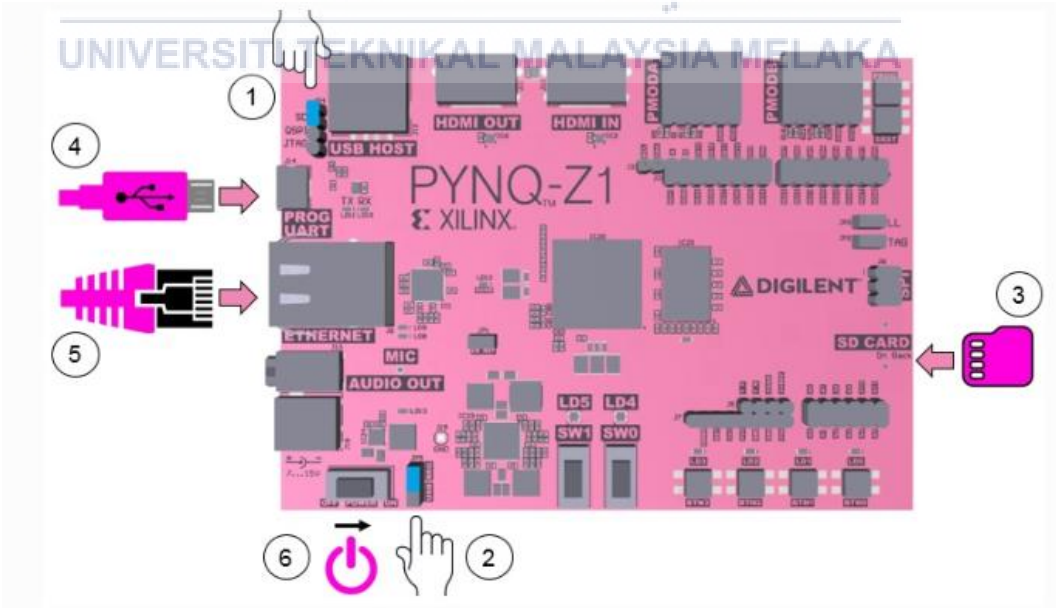


Figure 3.25: PYNQ-Z2 setup.

3.8 Chapter Summary

In this project, the ResNet50 and YOLOv2 applied on ZedBoard while the BNN and YOLOv2 applied on PYNQ-Z2. The flowchart of this project for implementation of CNNs on FPGAs and the significant steps have discussed respectively in this chapter in order to accelerate the CNNs on FPGA to allow image classification and object detection. Besides, the data flow of each CNN operated on FPGA explained in Section 3.7.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

This chapter presents the collected results from YOLOv2, ResNet50 and BNN (CIFAR-10 and MNIST) on ZedBoard and PYNQ-Z2. As stated earlier, the objectives of this work were successfully achieved by using two FPGA boards which are ZedBoard and PYNQ-Z2. The results of accelerated the CNN on FPGA boards will be shown in Section 4.2. The power consumption, execution time, performances and efficiency of each networks will be presented in Section 4.6.

4.2 Floating-point Operations Per Second and Execution Time

4.2.1 YOLOv2

For the YOLOv2 on ZedBoard, we evaluate the performance of FPGA running CNNs through the simulation test. The convolutional neural network of each layer has a FLOPS value whether it is running Resnet50 or YOLOv2. FLOPS stand for floating-point operations per second which is using to determine the FPGA performances in each convolutional layer. At the same time, the program would make the time of execution for each layer after completed its execution. Besides, the time taken to predict an image is the sum of execution time for each layer. Therefore, the performance of ZedBoard is the accumulative of FLOPS values from each convolutional layer divided by the total execution time for each layer.

$$GOP = \sum_{N=0}^{N} FLOPS$$

$$T_{Zed} = \sum_{N=0}^{N} T_E$$

$$GOP\ /s = \frac{GOP}{T_{Zed}}$$

Where:

GOP = Sum of floating-point operations

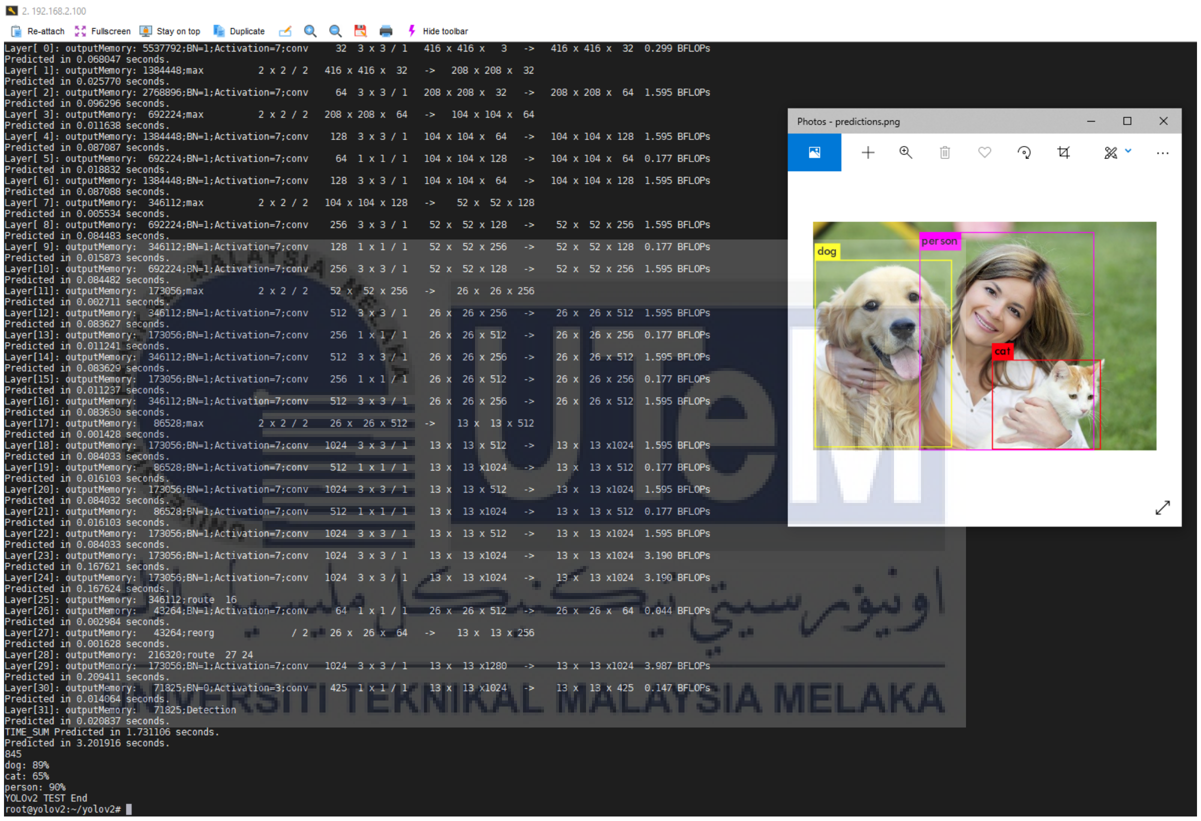
FLOPS = Floating-point operations per second for each layer

GOP /s = Performance of FPGA

T_E = Execution time for each layer

T_{Zed}= Sum of T_E on ZedBoard

N = Number of layers



```
Layer[ 0]: outputMemory: 5537792;BN=1;Activation=7;conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32  0.299 BFLOPs
Predicted in 0.068047 seconds.
Layer[ 1]: outputMemory: 1384448;max          2 x 2 / 2   416 x 416 x  32   ->   208 x 208 x  32
Predicted in 0.025770 seconds.
Layer[ 2]: outputMemory: 2768896;BN=1;Activation=7;conv     64  3 x 3 / 1   208 x 208 x  32   ->   208 x 208 x  64  1.595 BFLOPs
Predicted in 0.096296 seconds.
Layer[ 3]: outputMemory:  692224;max          2 x 2 / 2   208 x 208 x  64   ->   104 x 104 x  64
Predicted in 0.011638 seconds.
Layer[ 4]: outputMemory: 1384448;BN=1;Activation=7;conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
Predicted in 0.087087 seconds.
Layer[ 5]: outputMemory:  692224;BN=1;Activation=7;conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64  0.177 BFLOPs
Predicted in 0.018832 seconds.
Layer[ 6]: outputMemory: 1384448;BN=1;Activation=7;conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
Predicted in 0.087088 seconds.
Layer[ 7]: outputMemory:  346112;max          2 x 2 / 2   104 x 104 x 128   ->    52 x  52 x 128
Predicted in 0.005534 seconds.
Layer[ 8]: outputMemory:  692224;BN=1;Activation=7;conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
Predicted in 0.084483 seconds.
Layer[ 9]: outputMemory:  346112;BN=1;Activation=7;conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
Predicted in 0.015873 seconds.
Layer[10]: outputMemory:  692224;BN=1;Activation=7;conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
Predicted in 0.084482 seconds.
Layer[11]: outputMemory:  173056;max          2 x 2 / 2    52 x  52 x 256   ->    26 x  26 x 256
Predicted in 0.002711 seconds.
Layer[12]: outputMemory:  346112;BN=1;Activation=7;conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512  1.595 BFLOPs
Predicted in 0.083627 seconds.
Layer[13]: outputMemory:  173056;BN=1;Activation=7;conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256  0.177 BFLOPs
Predicted in 0.011241 seconds.
Layer[14]: outputMemory:  346112;BN=1;Activation=7;conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512  1.595 BFLOPs
Predicted in 0.083629 seconds.
Layer[15]: outputMemory:  173056;BN=1;Activation=7;conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256  0.177 BFLOPs
Predicted in 0.011237 seconds.
Layer[16]: outputMemory:  346112;BN=1;Activation=7;conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512  1.595 BFLOPs
Predicted in 0.083630 seconds.
Layer[17]: outputMemory:   86528;max          2 x 2 / 2    26 x  26 x 512   ->    13 x  13 x 512
Predicted in 0.001428 seconds.
Layer[18]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
Predicted in 0.084033 seconds.
Layer[19]: outputMemory:   86528;BN=1;Activation=7;conv    512  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 512  0.177 BFLOPs
Predicted in 0.016103 seconds.
Layer[20]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
Predicted in 0.084032 seconds.
Layer[21]: outputMemory:   86528;BN=1;Activation=7;conv    512  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 512  0.177 BFLOPs
Predicted in 0.016103 seconds.
Layer[22]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
Predicted in 0.084033 seconds.
Layer[23]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFLOPs
Predicted in 0.167621 seconds.
Layer[24]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFLOPs
Predicted in 0.167624 seconds.
Layer[25]: outputMemory:  346112;route  16
Layer[26]: outputMemory:   43264;BN=1;Activation=7;conv     64  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x  64  0.044 BFLOPs
Predicted in 0.002984 seconds.
Layer[27]: outputMemory:   43264;reorg              / 2    26 x  26 x  64   ->    13 x  13 x 256
Predicted in 0.001628 seconds.
Layer[28]: outputMemory:  216320;route  27 24
Layer[29]: outputMemory:  173056;BN=1;Activation=7;conv   1024  3 x 3 / 1    13 x  13 x1280   ->    13 x  13 x1024  3.987 BFLOPs
Predicted in 0.209411 seconds.
Layer[30]: outputMemory:   71825;BN=0;Activation=3;conv    425  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 425  0.147 BFLOPs
Predicted in 0.014064 seconds.
Layer[31]: outputMemory:   71825;Detection
Predicted in 0.020837 seconds.
TIME_SUM Predicted in 1.731106 seconds.
Predicted in 3.201916 seconds.
845
dog: 89%
cat: 65%
person: 90%
YOLOv2 TEST End
root@yolov2:~/yolov2#
```

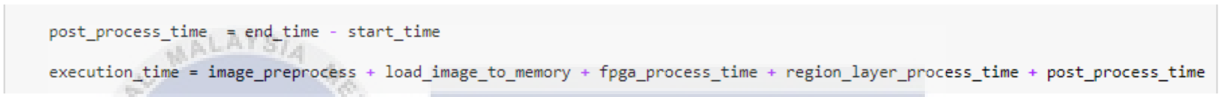
Figure 4.1: Results of YOLOv2 on ZedBoard

For YOLOv2 on PYNQ-Z2, we can obtain the FLOPS values from the Xilinx Vivado HLS simulation which is the same as ZedBoard. So, the GOP of YOLOv2 on PYNQ-Z2 is also 29.464 GOP same as ZedBoard. However, according to the source code of YOLOv2 for PYNQ-Z2, the execution time for each layer on PYNQ-Z2 is the time taken by FPGA to process the object detection. Therefore, the performance of PYNQ-Z2 is the accumulative of FLOPS values from each convolutional layer divided by the FPGA processing time the object detection.

$$GOP\ /s = \frac{GOP}{T_{PYNQ}}$$

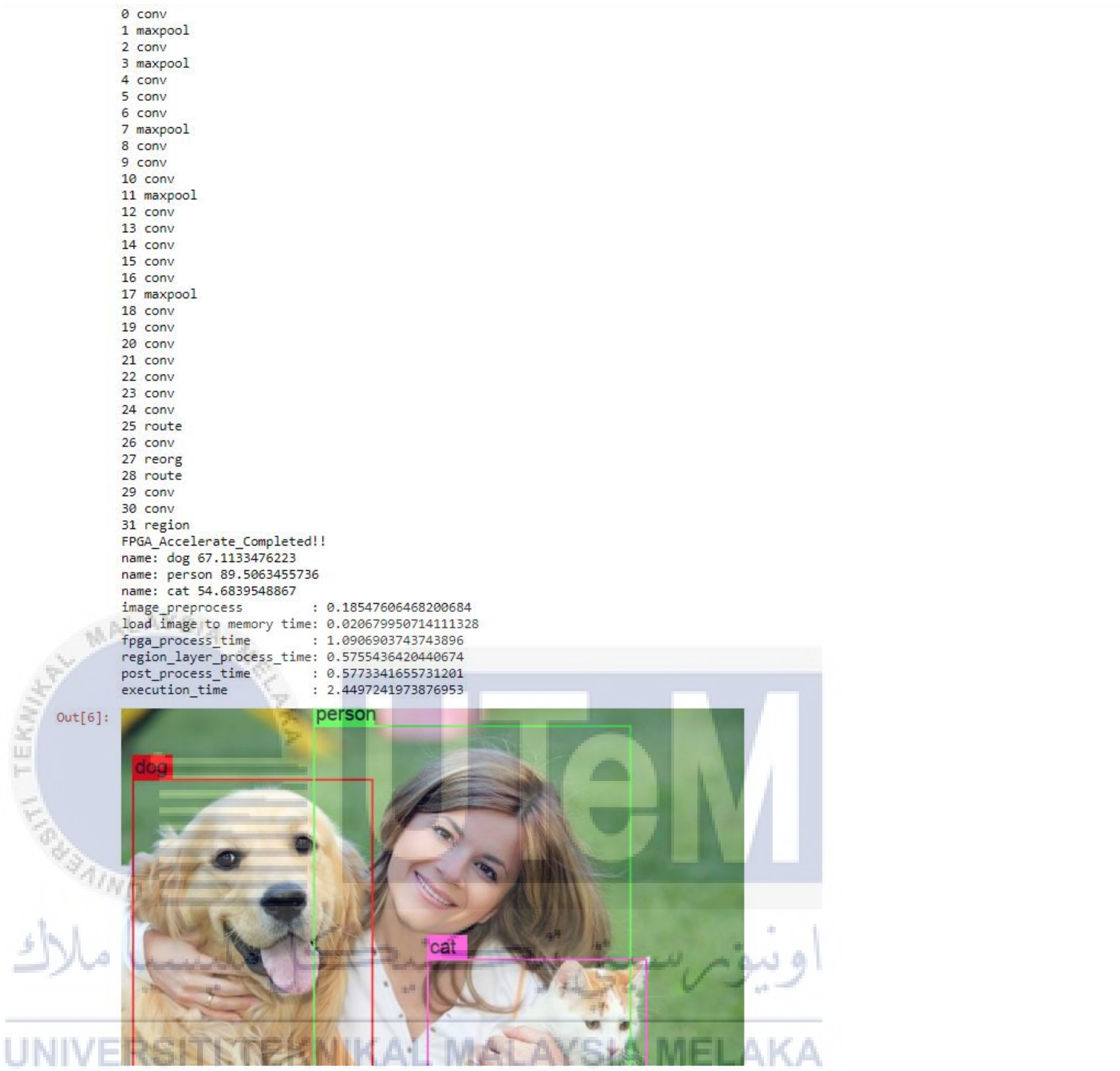
Where:

T_{PYNQ}= Time taken of FPGA to process the object detection on PYNQ-Z2



```
post_process_time  = end_time - start_time

execution_time = image_preprocess + load_image_to_memory + fpga_process_time + region_layer_process_time + post_process_time
```

Figure 4.2: Execution time for PYNQ-Z2



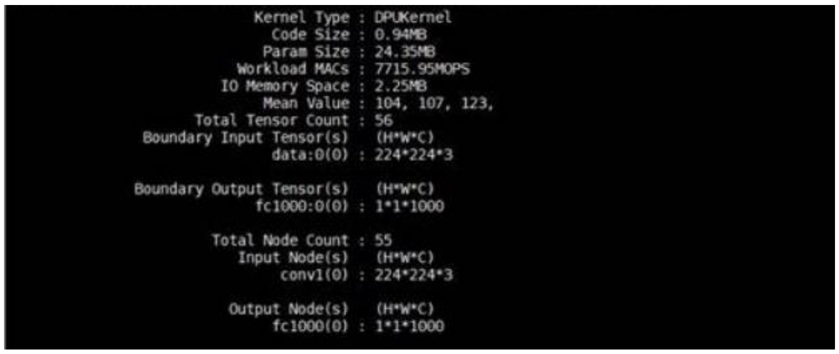
```
0 conv
1 maxpool
2 conv
3 maxpool
4 conv
5 conv
6 conv
7 maxpool
8 conv
9 conv
10 conv
11 maxpool
12 conv
13 conv
14 conv
15 conv
16 conv
17 maxpool
18 conv
19 conv
20 conv
21 conv
22 conv
23 conv
24 conv
25 route
26 conv
27 reorg
28 route
29 conv
30 conv
31 region
FPGA_Accelerate_Completed!!
name: dog 67.1133476223
name: person 89.5063455736
name: cat 54.6839548867
image_preprocess          : 0.18547606468200684
load image to memory time: 0.020679950714111328
fpga_process_time         : 1.0906903743743896
region_layer_process_time: 0.5755436420440674
post_process_time         : 0.5773341655731201
execution_time            : 2.4497241973876953
```

Out[6]:

Figure 4.3: Results of YOLOv2 on PYNQ-Z2

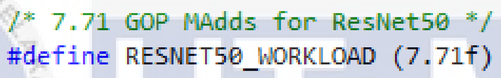
4.2.2 ResNet50

By using DNNC tools developed by Xilinx, we compile the network model of ResNet50. After the compilation is successful, the DNNC tool will generate the ELF object files and kernel information. The function of the ELF file is the files for driving the ResNet50 model on ZedBoard. Moreover, the kernel information has also included a total of FLOPS values so that we enable to calculate the performance, GOP/s of ResNet50 on ZedBoard. The ResNet50's source code has defined the values of GOP which is the sum of FLOPS for each layer as 7.71 GOP. Besides, the performance, GOP/s of ZedBoard running the Resnet50 is equal to GOP divided by the execution time.



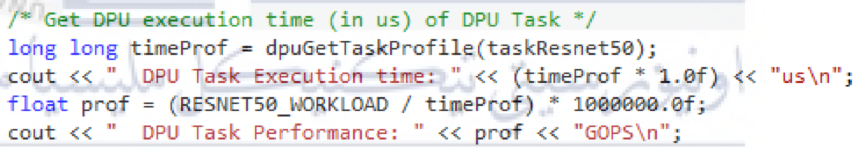
```
            Kernel Type : DPUKernel
              Code Size : 0.94MB
             Param Size : 24.35MB
          Workload MACs : 7715.95MOPS
        IO Memory Space : 2.25MB
             Mean Value : 104, 107, 123,
      Total Tensor Count : 56
Boundary Input Tensor(s)   (H*W*C)
              data:0(0) : 224*224*3

Boundary Output Tensor(s)   (H*W*C)
            fc1000:0(0) : 1*1*1000

       Total Node Count : 55
          Input Node(s)   (H*W*C)
               conv1(0) : 224*224*3

         Output Node(s)   (H*W*C)
              fc1000(0) : 1*1*1000
```

Figure 4.4: Kernel information during compilation



```
/* 7.71 GOP MAdds for ResNet50 */
#define RESNET50_WORKLOAD (7.71f)
```

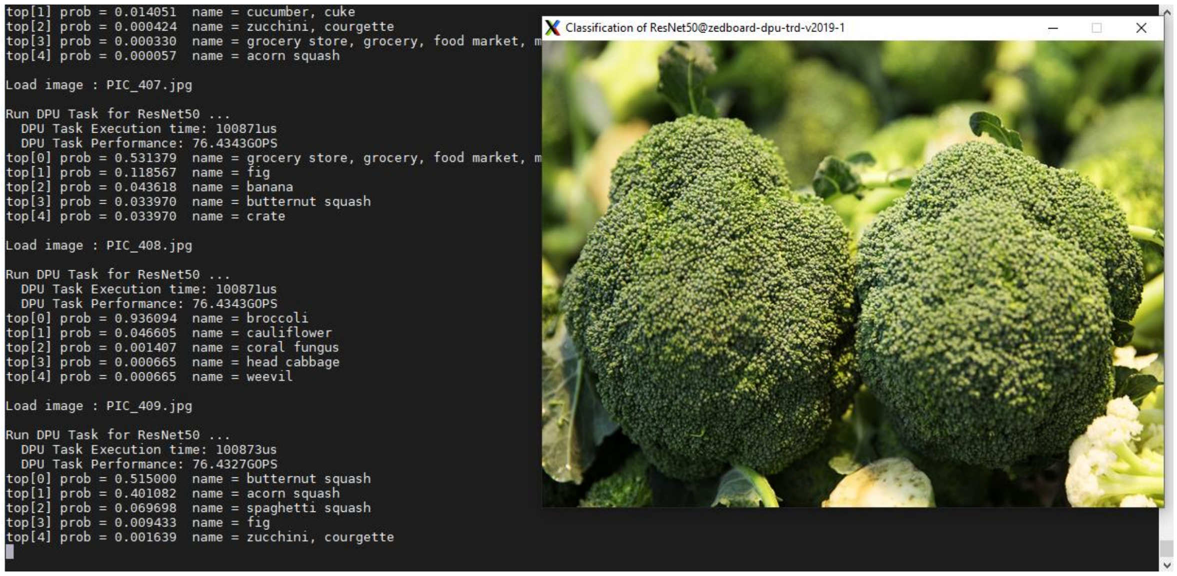
(a)



```
/* Get DPU execution time (in us) of DPU Task */
long long timeProf = dpuGetTaskProfile(taskResnet50);
cout << "  DPU Task Execution time: " << (timeProf * 1.0f) << "us\n";
float prof = (RESNET50_WORKLOAD / timeProf) * 1000000.0f;
cout << "  DPU Task Performance: " << prof << "GOPS\n";
```

(b)

Figure 4.5: Source code (a) Define the GOP value (b) Performance calculation (GOP/s)



```
top[1] prob = 0.014051  name = cucumber, cuke
top[2] prob = 0.000424  name = zucchini, courgette
top[3] prob = 0.000330  name = grocery store, grocery, food market, m
top[4] prob = 0.000057  name = acorn squash

Load image : PIC_407.jpg

Run DPU Task for ResNet50 ...
  DPU Task Execution time: 100871us
  DPU Task Performance: 76.4343GOPS
top[0] prob = 0.531379  name = grocery store, grocery, food market, m
top[1] prob = 0.118567  name = fig
top[2] prob = 0.043618  name = banana
top[3] prob = 0.033970  name = butternut squash
top[4] prob = 0.033970  name = crate

Load image : PIC_408.jpg

Run DPU Task for ResNet50 ...
  DPU Task Execution time: 100871us
  DPU Task Performance: 76.4343GOPS
top[0] prob = 0.936094  name = broccoli
top[1] prob = 0.046605  name = cauliflower
top[2] prob = 0.001407  name = coral fungus
top[3] prob = 0.000665  name = head cabbage
top[4] prob = 0.000665  name = weevil

Load image : PIC_409.jpg

Run DPU Task for ResNet50 ...
  DPU Task Execution time: 100873us
  DPU Task Performance: 76.4327GOPS
top[0] prob = 0.515000  name = butternut squash
top[1] prob = 0.401082  name = acorn squash
top[2] prob = 0.069698  name = spaghetti squash
top[3] prob = 0.009433  name = fig
top[4] prob = 0.001639  name = zucchini, courgette
```

Figure 4.6: Results of ResNet50 on ZedBoard.

4.3 Synthesized and Implemented Design

The design was synthesized and implemented by using the Xilinx Vivado development tools. This tool will be summed up all the design summary of the bit file.

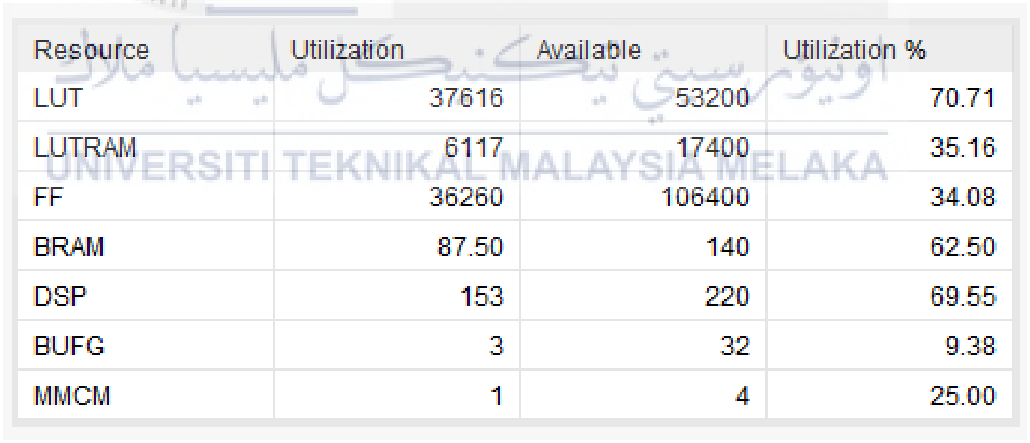
4.3.1 Resource Utilization

Figure 4.7 shows the summary of resource utilization of YOLOv2 on PYNQ-Z2 and ZedBoard respectively. The resource utilization of these two FPGA boards is the same because both are same Zynq-7000 family of SoC. The YOLOv2 is using about 71% of the FPGA Look-up Tables for PYNQ-Z2 and 70% of the FPGA Look-up Tables for ZedBoard. Besides, the PYNQ used 34% of flip-flop and ZedBoard used 33% of the flip-flops. Next, PYNQ-Z2 and ZedBoard both have used 63% of the Block RAM. Lastly, this design also used about 70% of DSPs on PYNQ-Z2 and ZedBoard.

The ResNet50 design currently used only 57% of the FPGA Look-up Tables, 55% of the flip-flops and 83% of the Block RAM. Furthermore, the design of ResNet50 has used 88% of DSPs on ZedBoard.

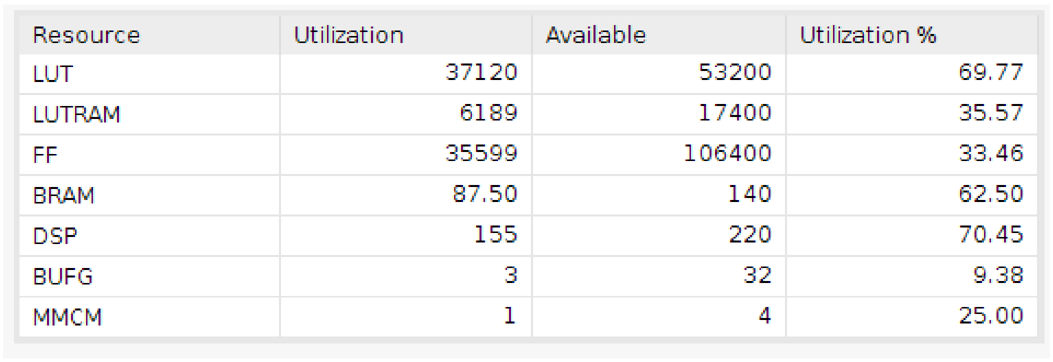
For the BNN on PYNQ-Z2, the CIFAR-10 and MNIST design have used 45.86% and 43.11% LUTs respectively. Besides, the CIFAR-10 used 36.19% of flip-flops on PYNQ-Z2 while the MNIST only used 25.47% flip-flops on PYNQ-Z2.

Next, the design of CNN on ZedBoard for digit recognition currently used only 43.73% of the FPGA Look-up Tables, 25.16% of the flip-flops and 74.29% of the Block RAM. Furthermore, this design has used 41.36% of DSPs on ZedBoard.



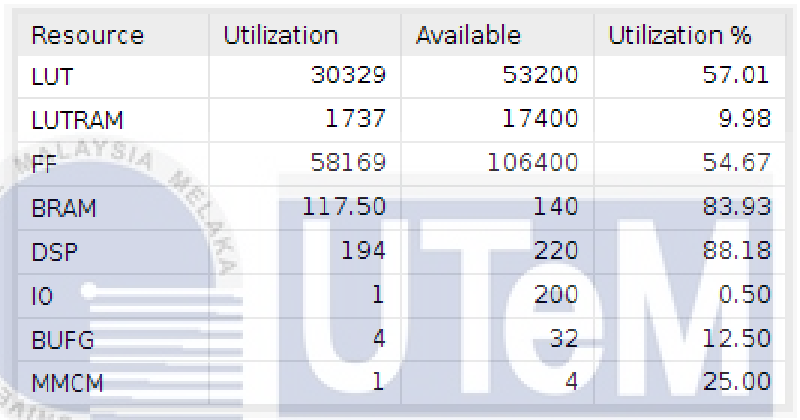
Resource	Utilization	Available	Utilization %
LUT	37616	53200	70.71
LUTRAM	6117	17400	35.16
FF	36260	106400	34.08
BRAM	87.50	140	62.50
DSP	153	220	69.55
BUFG	3	32	9.38
MMCM	1	4	25.00

(a)



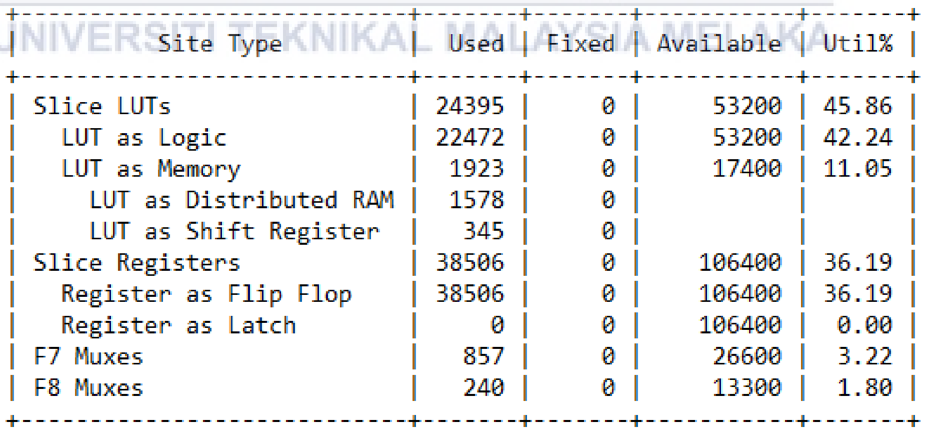
Resource	Utilization	Available	Utilization %
LUT	37120	53200	69.77
LUTRAM	6189	17400	35.57
FF	35599	106400	33.46
BRAM	87.50	140	62.50
DSP	155	220	70.45
BUFG	3	32	9.38
MMCM	1	4	25.00

(b)



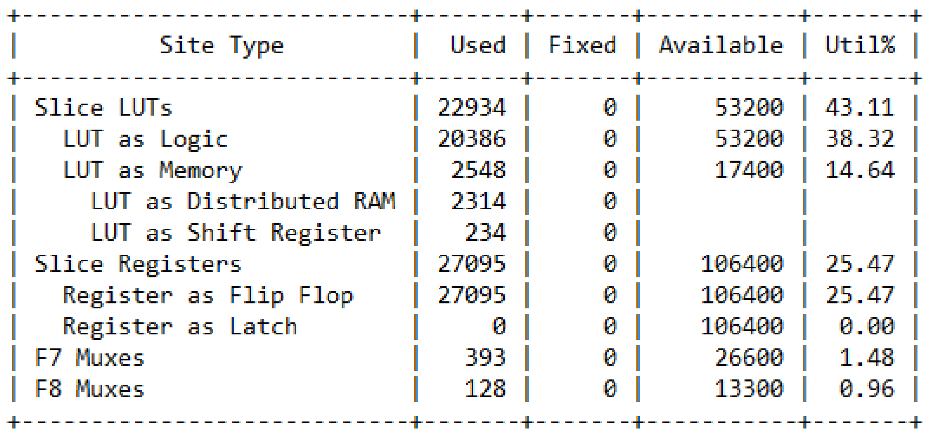
Resource	Utilization	Available	Utilization %
LUT	30329	53200	57.01
LUTRAM	1737	17400	9.98
FF	58169	106400	54.67
BRAM	117.50	140	83.93
DSP	194	220	88.18
IO	1	200	0.50
BUFG	4	32	12.50
MMCM	1	4	25.00

(c)



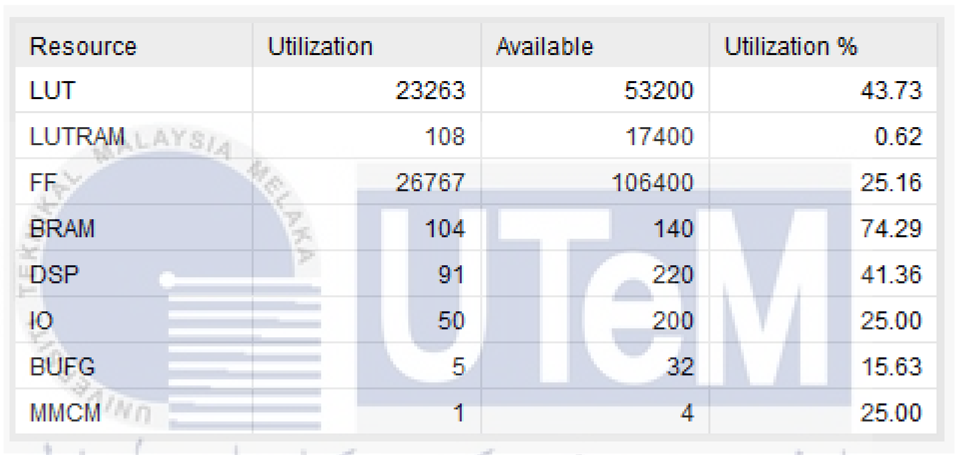
```
+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 24395 |     0 |     53200 | 45.86 |
|   LUT as Logic             | 22472 |     0 |     53200 | 42.24 |
|   LUT as Memory            |  1923 |     0 |     17400 | 11.05 |
|     LUT as Distributed RAM |  1578 |     0 |           |       |
|     LUT as Shift Register  |   345 |     0 |           |       |
| Slice Registers            | 38506 |     0 |    106400 | 36.19 |
|   Register as Flip Flop    | 38506 |     0 |    106400 | 36.19 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   857 |     0 |     26600 |  3.22 |
| F8 Muxes                   |   240 |     0 |     13300 |  1.80 |
+----------------------------+-------+-------+-----------+-------+
```

(d)



```
+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 22934 |     0 |     53200 | 43.11 |
|   LUT as Logic             | 20386 |     0 |     53200 | 38.32 |
|   LUT as Memory            |  2548 |     0 |     17400 | 14.64 |
|     LUT as Distributed RAM |  2314 |     0 |           |       |
|     LUT as Shift Register  |   234 |     0 |           |       |
| Slice Registers            | 27095 |     0 |    106400 | 25.47 |
|   Register as Flip Flop    | 27095 |     0 |    106400 | 25.47 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   393 |     0 |     26600 |  1.48 |
| F8 Muxes                   |   128 |     0 |     13300 |  0.96 |
+----------------------------+-------+-------+-----------+-------+
```

(e)



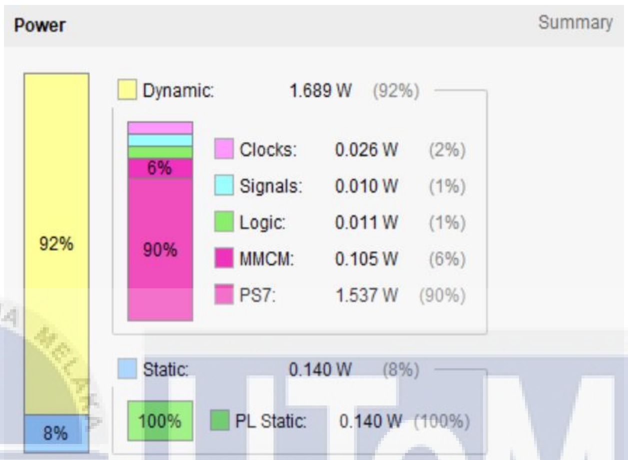
Resource	Utilization	Available	Utilization %
LUT	23263	53200	43.73
LUTRAM	108	17400	0.62
FF	26767	106400	25.16
BRAM	104	140	74.29
DSP	91	220	41.36
IO	50	200	25.00
BUFG	5	32	15.63
MMCM	1	4	25.00

(f)

Figure 4.7: Resource utilization (a) YOLOv2 for PYNQ-Z2 (b) YOLOv2 for ZedBoard (c) ResNet50 for ZedBoard (d) CIFAR-10 for PYNQ-Z2 (e) MNIST for ZedBoard (f)CNN on ZedBoard for digits recognition

4.3.2 Power Usage

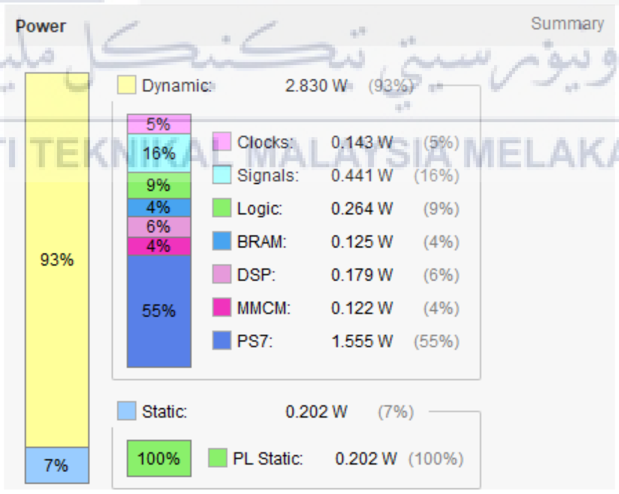
As shown in Figure 4.8, the YOLOv2 consumes 1.689W of dynamic power and 0.140W of static power on PYNQ-Z2. Therefore, the total power consumption of YOLOv2 on PYNQ-Z2 is 1.829W. Then, the total power consumption of YOLOv2 on ZedBoard is 3.032W with 2.830W of dynamic power and 0.202W of static power. Next, the ResNet50 currently consumes 3.008W of dynamic power and 0.217W of static power with a total power consumption of 3.225W on ZedBoard. Moreover, the total power consumption of CNN on ZedBoard for digits recognition is 1.911W with 1.750W dynamic power and 0.161W static power.



Power — Summary

🟨 Dynamic:	1.689 W	(92%)
🟪 Clocks:	0.026 W	(2%)
🟦 Signals:	0.010 W	(1%)
🟩 Logic:	0.011 W	(1%)
🟥 MMCM:	0.105 W	(6%)
🟥 PS7:	1.537 W	(90%)
🟦 Static:	0.140 W	(8%)
🟩 PL Static:	0.140 W	(100%)

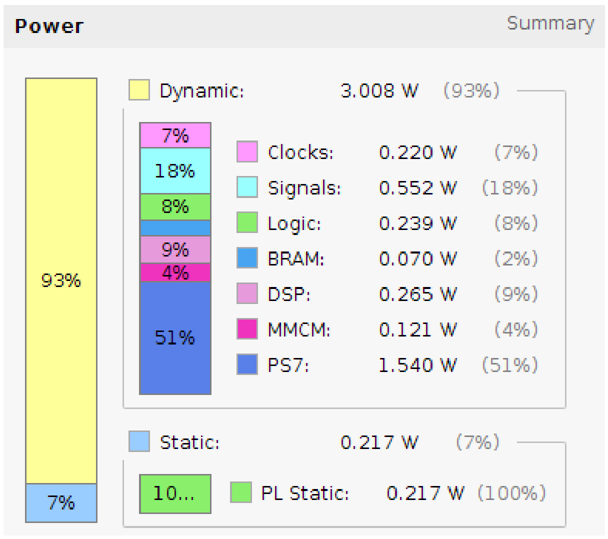
(a)



Power — Summary

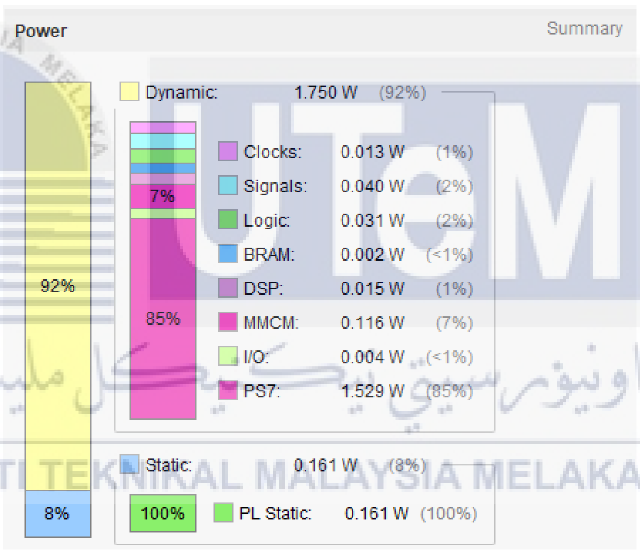
🟨 Dynamic:	2.830 W	(93%)
🟪 Clocks:	0.143 W	(5%)
🟦 Signals:	0.441 W	(16%)
🟩 Logic:	0.264 W	(9%)
🟦 BRAM:	0.125 W	(4%)
🟪 DSP:	0.179 W	(6%)
🟥 MMCM:	0.122 W	(4%)
🟦 PS7:	1.555 W	(55%)
🟦 Static:	0.202 W	(7%)
🟩 PL Static:	0.202 W	(100%)

(b)



Dynamic:	3.008 W	(93%)
Clocks:	0.220 W	(7%)
Signals:	0.552 W	(18%)
Logic:	0.239 W	(8%)
BRAM:	0.070 W	(2%)
DSP:	0.265 W	(9%)
MMCM:	0.121 W	(4%)
PS7:	1.540 W	(51%)
Static:	0.217 W	(7%)
PL Static:	0.217 W	(100%)

(c)



Dynamic:	1.750 W	(92%)
Clocks:	0.013 W	(1%)
Signals:	0.040 W	(2%)
Logic:	0.031 W	(2%)
BRAM:	0.002 W	(<1%)
DSP:	0.015 W	(1%)
MMCM:	0.116 W	(7%)
I/O:	0.004 W	(<1%)
PS7:	1.529 W	(85%)
Static:	0.161 W	(8%)
PL Static:	0.161 W	(100%)

(d)

Figure 4.8: Power Consumption (a)YOLOv2 on PYNQ-Z2 (b)YOLOv2 on ZedBoard (c)ResNet50 on ZedBoard (d)CNN on ZedBoard for digits recognition

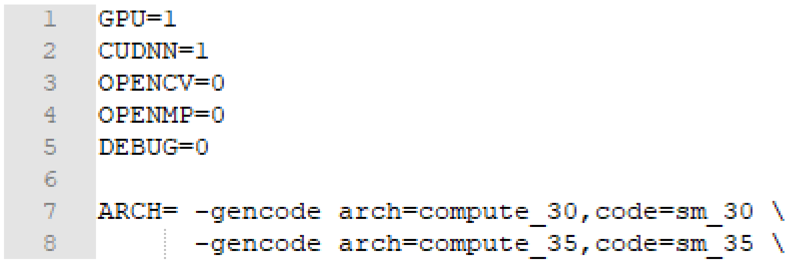
4.4 Experimental Environment

4.4.1 YOLOv2

To analyze the power efficiency and power consumption of FPGAs running CNNs. We set up two different platforms which are CPU-only and GPU-only to run the YOLOv2 so that the data can be obtained and compared with the FPGAs and the object detection tested by using the pre-trained model. Table 4.1 shows the specification of the platforms. We have installed the drivers for graphics cards (CUDA and cuDNN) to implement the object detection of YOLOv2. To ensure that the GPU-only used during the computation for object detection in this experiment, we have set the GPU and cuDNN option to ‘1’ in the script and the other set to ‘0’ before implementing the installation. Conversely， to enable CPU-only used in this experiment， we have set the OpenCV option in the script to "1" and the other option set to "0". This means that we should install the OpenCV library before the start of the installation.

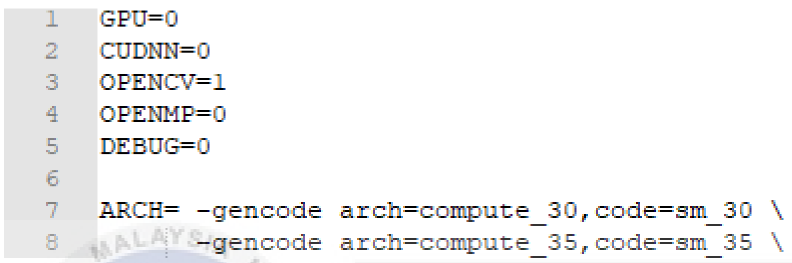
Table 4.1: Specification of experimental platforms

	Personal Computer	
OS	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS
CPU	AMD Fx-8300 @ 3.4 GHz	Intel i5-4200M @ 2.5GHz
GPU	Geforce GTX 750Ti (2GB RAM)	-
RAM	8GB	8GB
Storage	Apecer 240GB SSD	Apecer 240GB SSD



```
GPU=1
CUDNN=1
OPENCV=0
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
```

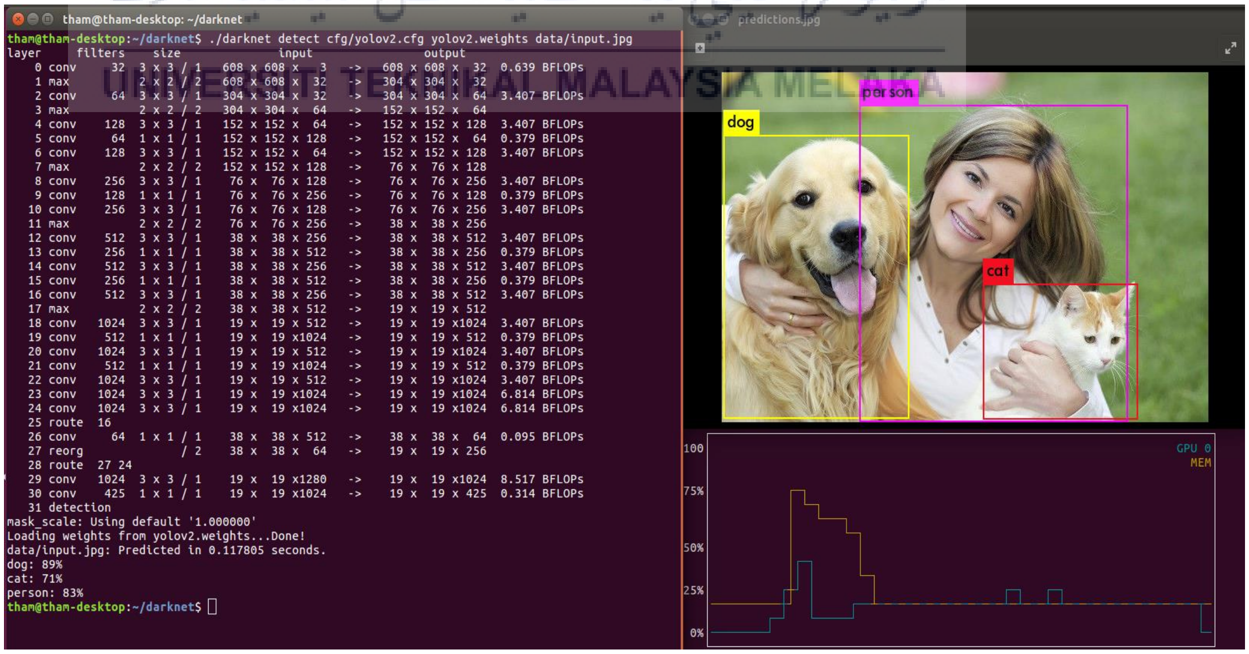
(a)



```
GPU=0
CUDNN=0
OPENCV=1
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
```

(b)

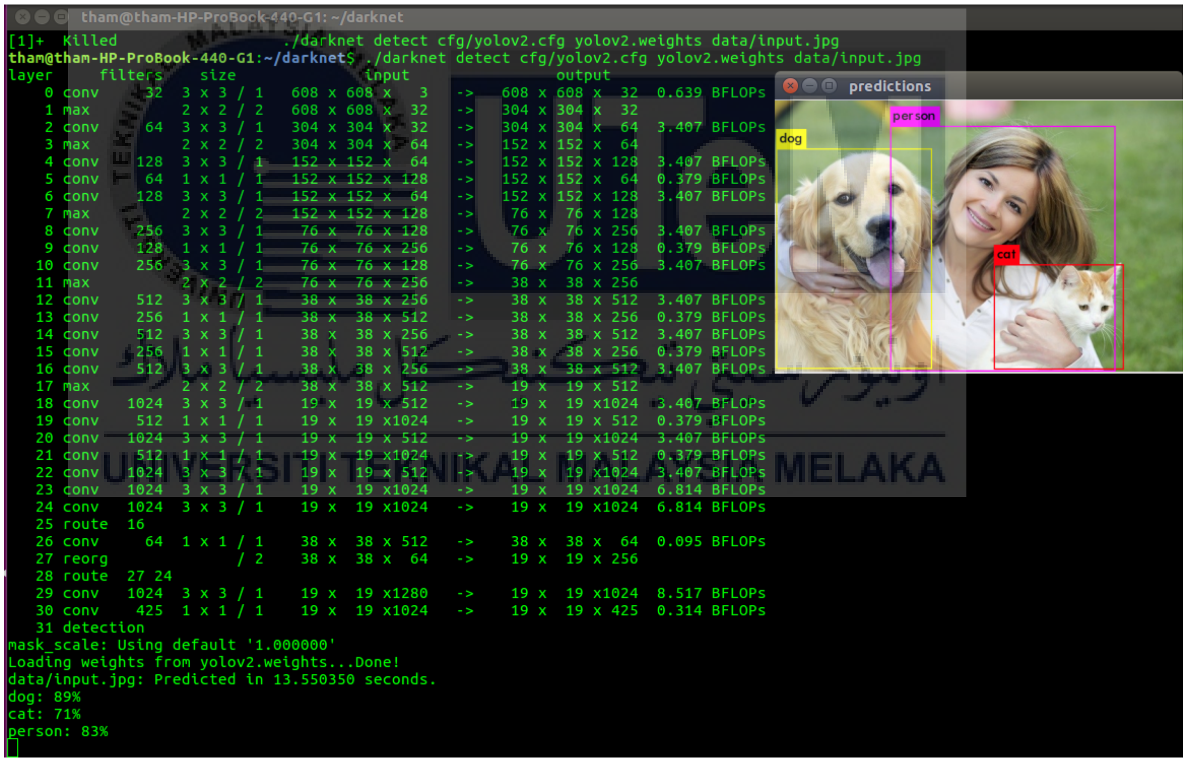
Figure 4.9: Installation script (a)Enable GPU-only computation (b) Enable CPU-only computation



```
tham@tham-desktop:~/darknet$ ./darknet detect cfg/yolov2.cfg yolov2.weights data/input.jpg
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   608 x 608 x   3   ->   608 x 608 x  32  0.639 BFLOPs
    1 max          2 x 2 / 2   608 x 608 x  32   ->   304 x 304 x  32
    2 conv     64  3 x 3 / 1   304 x 304 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    3 max          2 x 2 / 2   304 x 304 x  64   ->   152 x 152 x  64
    4 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    5 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
    6 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    7 max          2 x 2 / 2   152 x 152 x 128   ->    76 x  76 x 128
    8 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
    9 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   10 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   11 max          2 x 2 / 2    76 x  76 x 256   ->    38 x  38 x 256
   12 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   13 conv    256  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x 256  0.379 BFLOPs
   14 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   15 conv    256  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x 256  0.379 BFLOPs
   16 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   17 max          2 x 2 / 2    38 x  38 x 512   ->    19 x  19 x 512
   18 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   19 conv    512  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 512  0.379 BFLOPs
   20 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   21 conv    512  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 512  0.379 BFLOPs
   22 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   23 conv   1024  3 x 3 / 1    19 x  19 x1024   ->    19 x  19 x1024  6.814 BFLOPs
   24 conv   1024  3 x 3 / 1    19 x  19 x1024   ->    19 x  19 x1024  6.814 BFLOPs
   25 route  16
   26 conv     64  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x  64  0.095 BFLOPs
   27 reorg              / 2    38 x  38 x  64   ->    19 x  19 x 256
   28 route  27 24
   29 conv   1024  3 x 3 / 1    19 x  19 x1280   ->    19 x  19 x1024  8.517 BFLOPs
   30 conv    425  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 425  0.314 BFLOPs
   31 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2.weights...Done!
data/input.jpg: Predicted in 0.117805 seconds.
dog: 89%
cat: 71%
person: 83%
tham@tham-desktop:~/darknet$
```

Figure 4.10: Results of YOLOv2 for GPU-only

In this work, we install a GPU monitoring software called ‘nvtop’ on the personal computer to enable us to monitor the functionality of GPU during executing the object detection. Based on Figure 4.10, we clearly to see that the platform is using the graphic card of device slot 0 named GPU0 for this object detection and the used of graphic card’s memories (MEM) is achieved 75%. That means the GPU is working during implement the YOLOv2 on the personal computer.



```
[1]+  Killed                  ./darknet detect cfg/yolov2.cfg yolov2.weights data/input.jpg
tham@tham-HP-ProBook-440-G1:~/darknet$ ./darknet detect cfg/yolov2.cfg yolov2.weights data/input.jpg
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   608 x 608 x   3   ->   608 x 608 x  32  0.639 BFLOPs
    1 max          2 x 2 / 2   608 x 608 x  32   ->   304 x 304 x  32
    2 conv     64  3 x 3 / 1   304 x 304 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    3 max          2 x 2 / 2   304 x 304 x  64   ->   152 x 152 x  64
    4 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    5 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
    6 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    7 max          2 x 2 / 2   152 x 152 x 128   ->    76 x  76 x 128
    8 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
    9 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   10 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   11 max          2 x 2 / 2    76 x  76 x 256   ->    38 x  38 x 256
   12 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   13 conv    256  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x 256  0.379 BFLOPs
   14 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   15 conv    256  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x 256  0.379 BFLOPs
   16 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   17 max          2 x 2 / 2    38 x  38 x 512   ->    19 x  19 x 512
   18 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   19 conv    512  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 512  0.379 BFLOPs
   20 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   21 conv    512  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 512  0.379 BFLOPs
   22 conv   1024  3 x 3 / 1    19 x  19 x 512   ->    19 x  19 x1024  3.407 BFLOPs
   23 conv   1024  3 x 3 / 1    19 x  19 x1024   ->    19 x  19 x1024  6.814 BFLOPs
   24 conv   1024  3 x 3 / 1    19 x  19 x1024   ->    19 x  19 x1024  6.814 BFLOPs
   25 route  16
   26 conv     64  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x  64  0.095 BFLOPs
   27 reorg              / 2    38 x  38 x  64   ->    19 x  19 x 256
   28 route  27 24
   29 conv   1024  3 x 3 / 1    19 x  19 x1280   ->    19 x  19 x1024  8.517 BFLOPs
   30 conv    425  1 x 1 / 1    19 x  19 x1024   ->    19 x  19 x 425  0.314 BFLOPs
   31 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2.weights...Done!
data/input.jpg: Predicted in 13.550350 seconds.
dog: 89%
cat: 71%
person: 83%
```

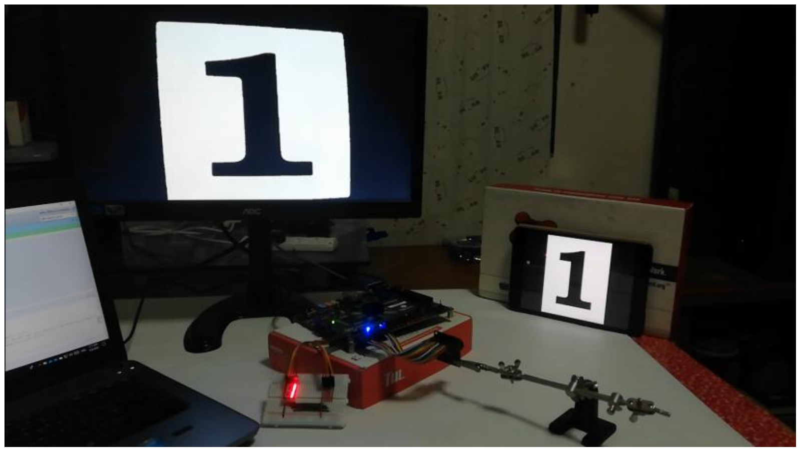
Figure 4.11: Results of YOLOv2 for CPU-only

For the ResNet model, we compare our implementations with the other platforms. Therefore, we can observe the performance and power efficiency by referring to the benchmarks of previous research to show our work has achieved high performance and efficiency.

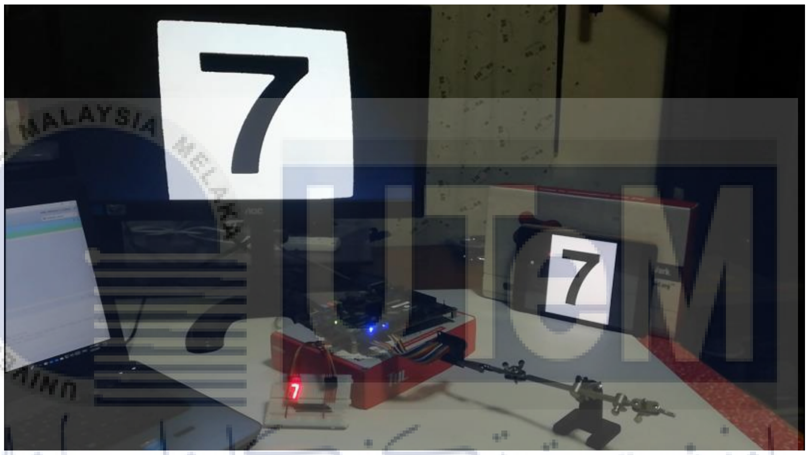
To evaluate BNN on PYNQ, we compare both CIFAR-10 and MNIST with software implementation on PYNQ-Z2 to show that the accelerate BNNs inference on FPGA. In this experiment, we will examine the inferred time and classification rate of FPGA implementation and software implementation as an accelerated evaluation.

4.4.2 CNN on ZedBoard for Digits Recognition

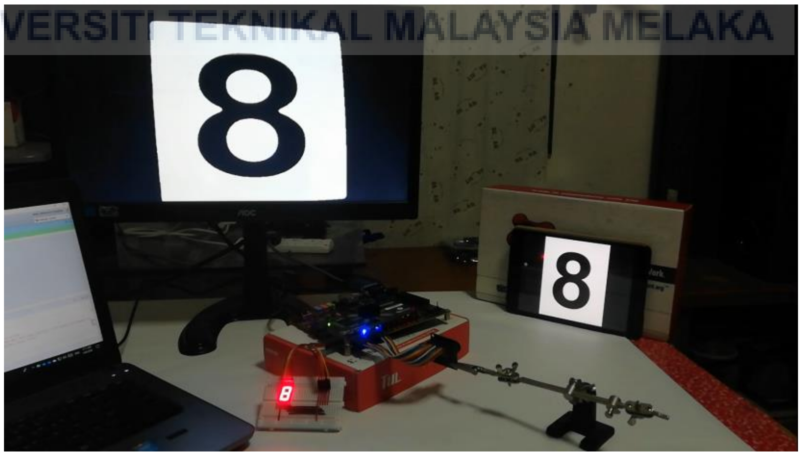
In order to test the accuracy of CNN on ZedBoard for digits recognition, the first things need to prepare is some digit images for OV7670 camera to capture as an input. When the camera detects the input images, the detected image will be exported to the monitor directly to achieve real-time detection. In this experiment, we prepared several digital images and copied to the tablet. Once the camera detects the digits, it starts the convolutional operation and finally displays the predicted digits on the seven-segment display LED. Figure 4.12 shows the results of recognition in this experiment. Obviously, the accuracy of the CNN recognition number on ZedBoard is satisfactory.



(a)



(b)

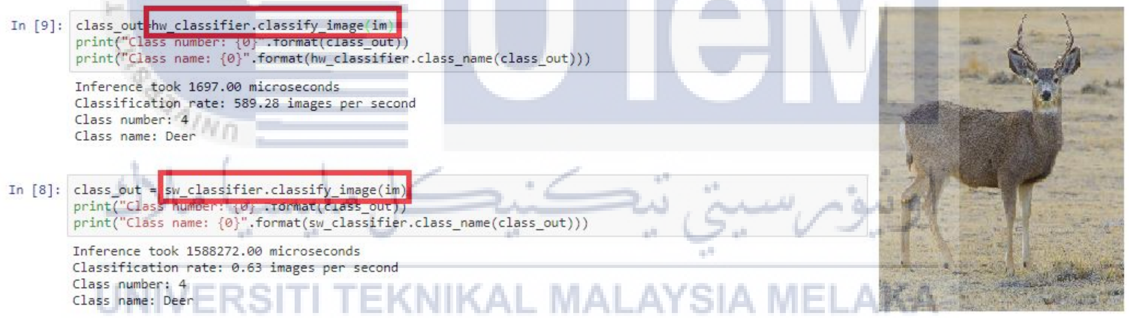


(c)

Figure 4.12: Results of recognition in this experiment (a) Digit number 1 (b) Digit number 7 (c) Digit number 8

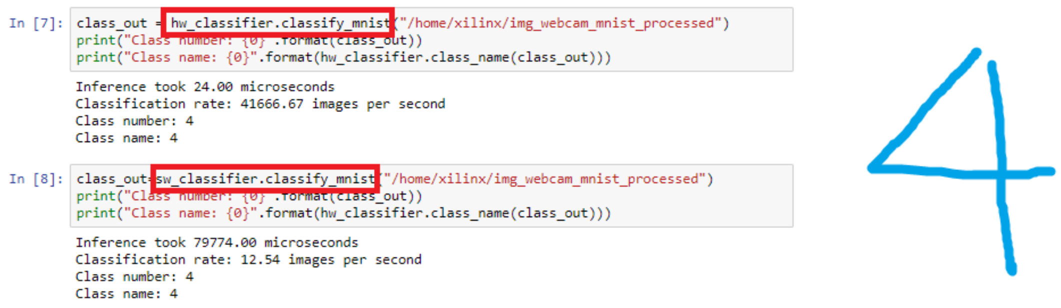
4.5 Evaluation of BNN

In this section, we test the CIFAR-10 and MNIST dataset to run image classification and handwriting digits recognition respectively on PYNQ-Z2 to achieve the hardware and software implementation of BNN. We select a deer image and two handwritten digits, which are '4' and '8' become the input of BNN in this experiment. By changing the 'hw' and 'sw' in classification command, we can target the types of implementation which we want to execute. Once the classification completed, the output of image classification will show the inference took means the time taken to predict an image in a microsecond and rate of classification means the number of images classifies per second.



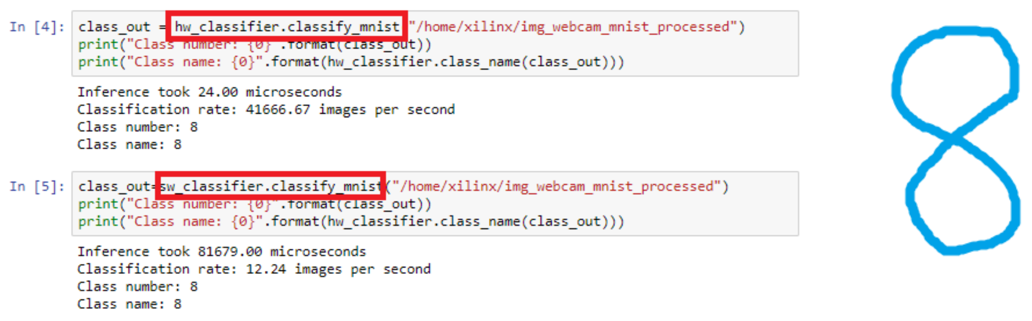
```
In [9]: class_out=hw_classifier.classify_image(im)
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(hw_classifier.class_name(class_out)))

        Inference took 1697.00 microseconds
        Classification rate: 589.28 images per second
        Class number: 4
        Class name: Deer

In [8]: class_out = sw_classifier.classify_image(im)
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(sw_classifier.class_name(class_out)))

        Inference took 1588272.00 microseconds
        Classification rate: 0.63 images per second
        Class number: 4
        Class name: Deer
```

(a)



```
In [7]: class_out = hw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(hw_classifier.class_name(class_out)))

        Inference took 24.00 microseconds
        Classification rate: 41666.67 images per second
        Class number: 4
        Class name: 4

In [8]: class_out=sw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(hw_classifier.class_name(class_out)))

        Inference took 79774.00 microseconds
        Classification rate: 12.54 images per second
        Class number: 4
        Class name: 4
```

(b)



```
In [4]: class_out = hw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(hw_classifier.class_name(class_out)))

        Inference took 24.00 microseconds
        Classification rate: 41666.67 images per second
        Class number: 8
        Class name: 8

In [5]: class_out=sw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")
        print("Class number: {0}".format(class_out))
        print("Class name: {0}".format(hw_classifier.class_name(class_out)))

        Inference took 81679.00 microseconds
        Classification rate: 12.24 images per second
        Class number: 8
        Class name: 8
```

(c)

Figure 4.13: Results of BNN in hardware and software (a) CIFAR-10 (b) MNIST for digit number 4 (c) MNIST for digit number 8

Based on the results in Table 4.2, we observed that both CIFAR-10 and MNIST of BNN running in hardware are accelerated. The parameters of evaluation in this experiment are inference took and the classification rate. For the CIFAR-10, the inference took 1.697 milliseconds and the classification rate is 589 images per second during the hardware implementation while software implementation inference took 1588.3 milliseconds and only 0.63 images per second. For the MNIST, the inference took of handwritten digit number 4 is 0.024 milliseconds and the classification rate is about 41700 images per second, which are the same as the handwritten digit 8 during the hardware implementation. As for software implementation of MNIST, the inference took to classify the handwritten digit 4 and handwritten digit 8 is 79.77 milliseconds and 81.68 milliseconds respectively. Besides, the classification rate of handwritten digit 4 is only 12.54 images per second and the classification rate of handwritten digit 8 is 12.24 images per second.

Table 4.2: Comparison of hardware and software implementation of BNN on PYNQ-Z2

Dataset	CIFAR-10		MNIST		MNIST	
Input image	Deer		Handwritten digit 4		Handwritten digit 8	
Implementation	Hardware	Software	Hardware	Software	Hardware	Software
Inference took (ms)	1.697	1588.3	0.024	79.77	0.024	81.68
Classification rate (image per second)	589.28	0.63	41666.67	12.54	41666.67	12.24

4.6 Analyzing Power Efficiency of CNNs on FPGA

In this project, the power efficiency is also one of the critical criteria through the experiments on the FPGAs. As mentioned above, the objective of these experiments is to measure and analyze the computational performance and efficiency of running CNNs on personal computer and FPGAs. Therefore, we can know whether the power consumption of the FPGAs is too large by comparing the power efficiency of the personal computer. Since we have obtained the performance and power of the FPGAs to run different CNNs, we can use the data of performance divided by the power to evaluate the power efficiency.

$$Power\ Efficiency, GOP\ /s\ /\ W = \frac{Performance, GOP\ /s}{Power, W}$$

Based on Table 4.3, we improve the power efficiency on PYNQ-Z2 and ZedBoard running YOLOv2 from 0.133GOP/s/W to 14.765GOP/s/W and 0.133GOP/s/W to 5.613GOP/s/W respectively. However，the power efficiency of ZedBoard still cannot higher than the power efficiency of the graphics card because the graphics card has the support of CUDA to improve the graphics card's ability to compute the floating-point. Hence, the graphics card can achieve object detection in just 0.12 seconds which is faster than both FPGAs. It is worth making special mention that the FPGAs only uses 3.032W and 1.829W to implement the YOLOv2. This result shows the power consumption to run YOLOv2 on FPGA is low. Besides, we compared with previous works and observed that the power efficiency of PYNQ-Z2 in this work is also improved from 0.05GOP/s /W to 14.765GOP/s /W and 8.89GOP/s /W to 14.765GOP/s /W for CPU and GPU respectively. The reason YOLOv2 has good performance on Zedboard and PYNQ-Z2 is that we use Vivado HLS to pack the RTL into the IP block. First of all, we need to import Darknet's YOLOv2 source code to Vivado HLS and perform the compilation, simulation and debugging of the corresponding C language code. After that, the Vivado HLS will integrate the C algorithm into RTL implementation and generate a comprehensive analysis report and analyze our design. Finally, we can package the RTL into the IP block and connect it with Zynq-7000 Soc IP in the Vivado. Besides, Vivado HLS can determine which steps performed in each clock cycle, and then predict the number of LUTs, registers, BRAM, and DSP48 use when running YOLOv2. In the resource utilization of YOLOv2, we know that YOLOv2 uses about 70% of the DSP, which shows that the DSP almost completely used. The DSP is mainly used for the adder and multiplier in the CNN, so Zedboard and PYNQ-Z2 have achieved good performance in calculating delay and power efficiency. Although the computing ability of Zedboard and Pynq-Z2 is not higher than the GPU platform, the high-power consumption of the graphic card causes the power efficiency of the graphics card is inefficient.

By referring to Table 4.4, we can observe that in this cross-platform experiment running YOLOv2, the mean average accuracy (mAP) value of PYNQ-Z2 is the lowest, although PYNQ-Z2 has good efficiency. This shows that running YOLOv2 on PYNQ-Z2 must trade-offs under power efficiency and accuracy. Therefore, it is better to run YOLOv2 on the ZedBoard. Although the performance of running YOLOv2 on Zedboard is not as good as the graphics card (GTX750Ti), we can observe that the mean average precision (mAP) value of Zedboard is the same as the CPU and GPU. In addition, the power consumption of Zedboard only needs 5.6W which is lower than the graphics card (GTX750Ti). Therefore, in this cross-platform experiment, we can conclude that Zedboard is the most suitable for running yolov2 to achieve object detection.

Table 4.3: Performance comparison for YOLOv2

	[18]	**[23]**	**This works**			
Devices	GTX Titan X	E5-2620v4	i5-4200M @2.5GHz	GTX750Ti	ZedBoard	PYNQ-Z2
CNN	Sim-YOLOv2	YOLOv2	YOLOv2	YOLOv2	YOLOv2	YOLOv2
Platform	GPU	CPU	CPU	GPU	FPGA	FPGA
Execution time, s	N/A	N/A	13.550	0.118	1.7311	1.091
Operations, GOP	17.18	N/A	62.944	62.944	29.464	29.464
Power, W	170	85	35	60	3.032	1.829
Performance, GOP/s	1512	4.11	4.645	533.424	17.020	27.006
Power Efficiency, GOP /s/W	8.89	0.05	0.133	8.890	5.613	14.765

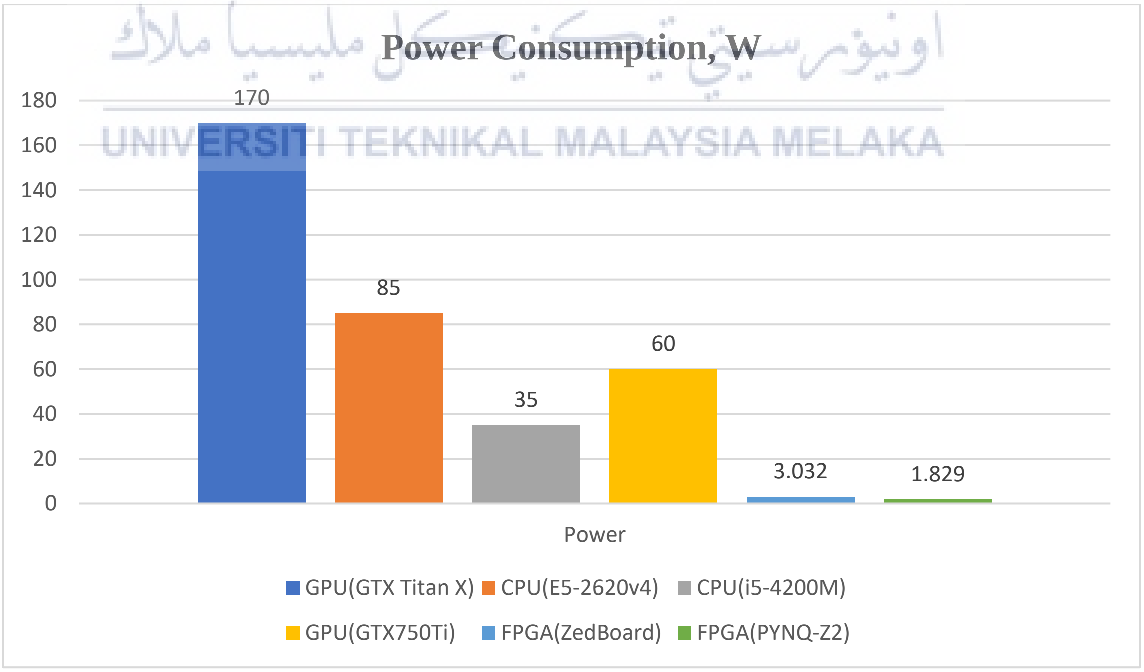


Figure 4.14: Graph of Power Consumption for YOLOv2

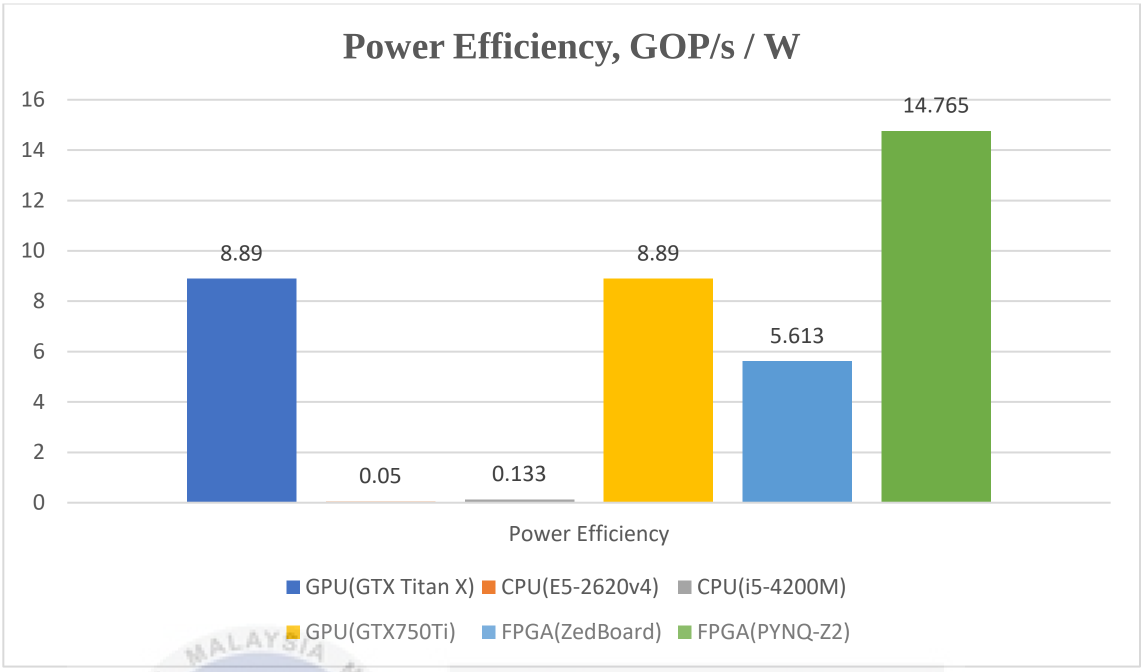


Figure 4.15: Graph of Power Efficiency for YOLOv2

Table 4.4 The cross-platform comparison data for mean average precision(mAP) in this work

Platform / Classes	CPU (i5-4200M)	GPU (GTX750Ti)	FPGA (ZedBoard)	FPGA (PYNQ-Z2)
Dog	0.89	0.89	0.89	0.67
Cat	0.71	0.71	0.65	0.55
Person	0.83	0.83	0.90	0.90
Mean Average Precision (mAP)	0.81	0.81	0.81	0.71

By referring the Table 4.5, we observed that the power efficiency of running the ResNet model on ZedBoard is 23.7GOP/s/W which is the highest efficiency on all platforms. We improve the power efficiency of ResNet from 0.63GOP/s/W to 23.7GOP/s/W. In addition, the power consumption of running ResNet on ZedBoard is only 3.225W which is the lowest on all platforms that we tested. The reason of ResNe-50 can achieve such high-power efficiency on Zedboard is that we use the DNNDK framework developed by Xilinx and this resource is public to Xilinx user. As we know that the process of CNNs requires intensive calculations and high memory bandwidth. Therefore, a tool called Decent (Deep Compression Tool) in DNNDK is used to quantify the trained data of the ResNet-50 model and weight distribution to address in order to achieve high performance and energy efficiency. After the ResNet-50 model is quantized, we use a compilation tool called DNNC (Deep Neural Network Compiler) to compile the quantized files. DNNC is a special compiler designed by Xilinx for DPU IP. It enables DPU to balance computing workload and memory access to maximize the DPU resources utilization. When the compilation is completed, an elf file will be generated and copied to the root file of the SD card. The Elf file is like a driver which is used to drive the DPU on ZedBoard to run the ResNet-50 model.

Table 4.5: Performance comparison with previous for ResNet

	[25]		[26]	[27]	[28]	This work
Devices	**Xeon E5-2650v2**	**GTX TITAN X**	**GSMD5**	**Arria 10**	**Zynq Z7045**	**ZedBoard**
Platform	CPU	GPU	FPGA	FPGA	FPGA	FPGA
Model	Res-152	Res-152	Res-50	Res-50	Res-50	Res-50
Operations, GOP	N/A	N/A	N/A	7.74	N/A	7.71
Power, W	95	250	25	45	9.61	3.225
Performance, GOP/s	119	1661	226.47	619.13	128	76.434
Power Efficiency, GOP /s/W	0.63	6.64	9.06	13.76	13.32	23.70

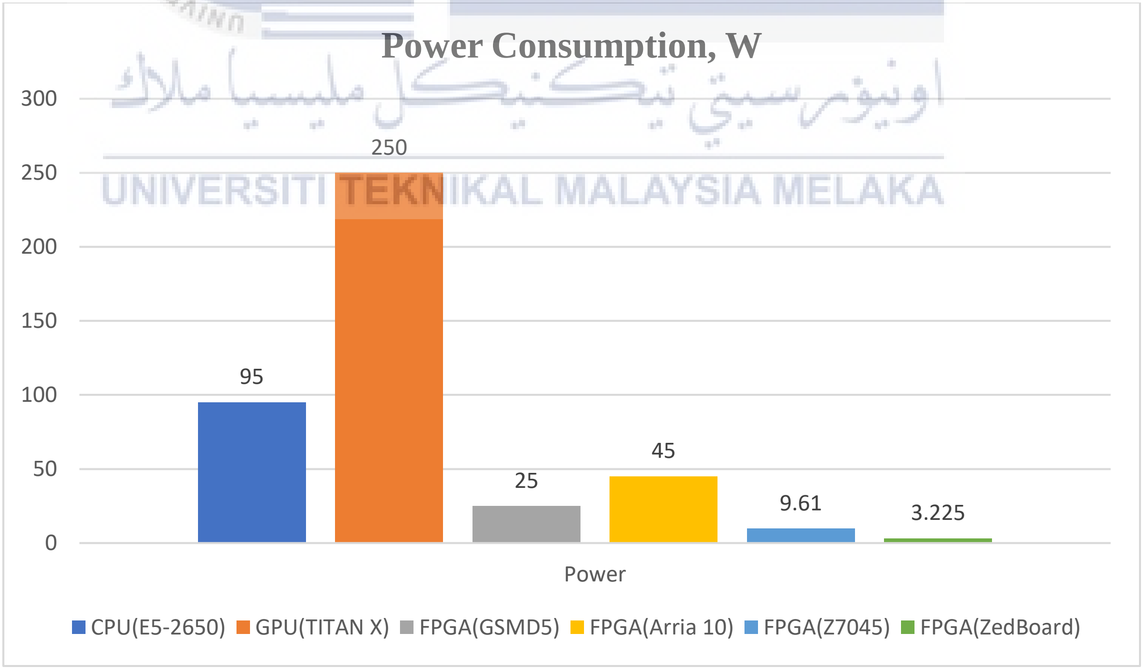


Figure 4.16: Graph of Power Consumption for ResNet

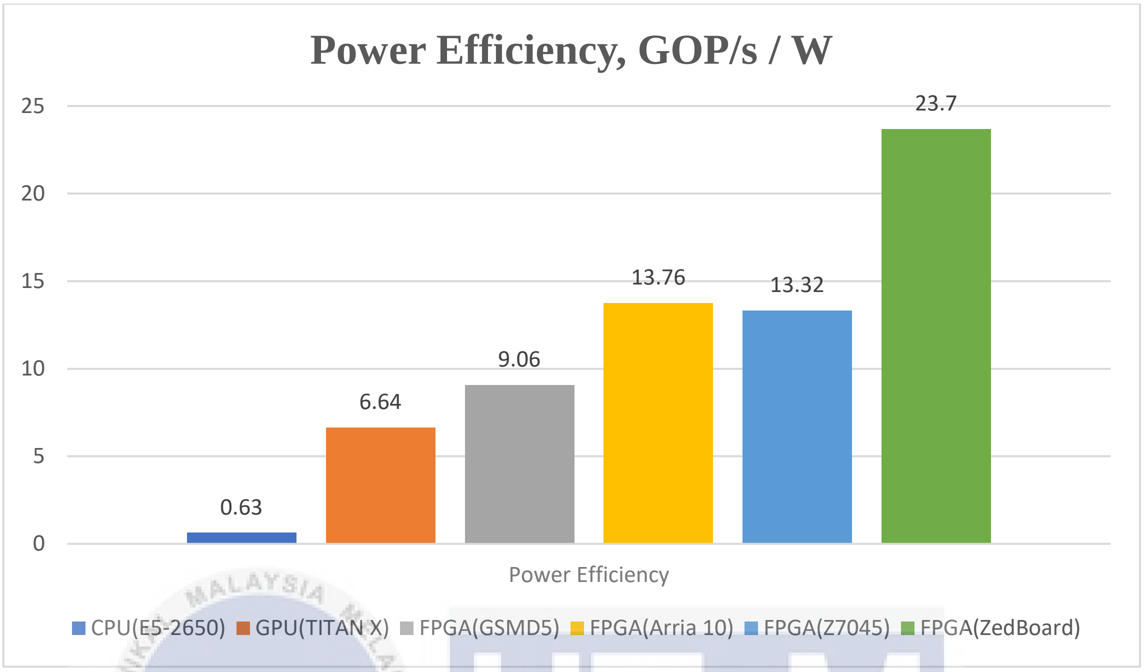


Figure 4.17: Graph of Power Efficiency for ResNet

4.7 Cost Estimation (MYR) Across Different Platforms

Another interesting section worth exploring in this project is about cost estimation. The reason is when we want to implement a detection or monitoring system, we cannot ignore the cost required to set up the system. In this section, we will compare all the platform appeared and referred in our project to accelerate the CNNs. Table 4.6 shows the price of that hardware in 2020. These estimated prices are provided by the Internet and those are converted in Malaysian ringgit units.

Table 4.6: The cost estimation across different platforms.

Hardware	**Platform**	**Cost /unit (MYR)**
NVIDIA GTX750Ti	GPU	650
NVIDIA GTX TITAN X	GPU	4,300
Intel i5-4200M	CPU	970
Intel Xeon E5-2620v2	CPU	750
Digilent ZedBoard Dev Kit	FPGA	1,900
TUL PYNQ-Z2 Dev Kit	FPGA	510
Intel Arria 10 GX 1150 Dev Kit	FPGA	20,000
NVIDIA Jetson TX2	Embedded System	3,500

In order to enable the GPU and CPU platform to be fully functional, it must be equipped with a high-configuration RAM, power supply, motherboard and hard drive to build a computer. Therefore, the total price of the computer far exceeds the price shown in Table 4.6. On the contrary, the FPGA platforms do not require any additional hardware to speed up the operation. This is because the FPGA platforms are already a complete development board, developers can directly implement CNNs on it. As we know that every extra penny deserves its value. The price of Arria 10 is higher than Zedboard and PYNQ-Z2 because of its categories as high configuration board. For that reason, Arria 10 can further design the CNNs application such as real-time object detection or face detection compared with ZedBoard and PYNQ-Z2.

4.8 Chapter Summary

In the first part of this chapter, the ResNet50 and YOLOv2 models are implemented on ZedBoard while BNN and YOLOv2 models are implemented on PYNQ-Z2. The pre-trained model of BNN is trained from the CIFAR-10 dataset for image classification and the MNIST dataset for handwriting recognition. In the second part of this chapter, the FLOPs and execution time of the CNNs are determined to evaluate the power efficiency and power consumption of the FPGAs. Moreover, the resource utilization and power usage of CNNs on FPGAs also obtained respectively through the Vivado software. The ResNet-50 and YOLOv2 results are compared with the previous research so that the accelerated performances are able to observe. Besides, the BNN model trained with CIFAR-10 and MNIST dataset are compared to the hardware and software implementation to observe CNN is accelerated on FPGA. Lastly, the cost estimation across the different platforms is also compared in MYR.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Introduction

This chapter will summarize the research results and conclude all the works of this project. Next, the recommendation for the future work and the enhancements of this project are suggested in this chapter so that the quality of the project can be improve.

5.2 Conclusion

The convolutional neural network (CNN) has huge applications in this modern world such as the video surveillance system, mobile robot vision, image search engine in the database, etc. The rapid growth of CNN has shown that the performance of CNN now surpasses the other type of visual recognition algorithms, and even beyond the human accuracy on certain conditions. However, CNN is computationally intensive and implementation of high performance depends on the computing platform. Many researchers have tried to accelerate the computation of CNN on different platforms such as CPU, GPU, ASIC, embedded system and FPGA in order to improve the performance of CNN.

Until now, GPUs still have the best performance of computation in deep learning. Due to the GPUs have high-performance computing ability, the electricity cost, power consumption, and price of GPUs are the highest among all the platforms. Therefore, most of the researchers have been attracted by the flexibility of FPGAs and turned to develop the FPGAs to achieve CNN acceleration on FPGAs. Compared with the graphics card, FPGA has a short development cycle and it can be reprogrammed, so that FPGA is easier to achieve the target of high performance with low power consumption.

Although the CPU platform supports the different programming frameworks, it is not the main choice for designers to select as the CNN accelerator. This is because CPU is only suitable for simple and small CNN models with short training time and the CPU will be forbidden the total execution time machine learning training when it is running a large model and large dataset. For embedded systems such as NVIDIA Jetson TX2, it cannot effectively perform the original code to achieve real-time performance even the embedded systems have the advantage of low power consumption like the FPGA. This means the embedded system only can execute the modified code to achieve real-time performance. Obviously, FPGA has great potential to be applied to CNN.

In Chapter 4, the power consumption and performance of ResNet50, YOLOv2 and BNN are analyzed and evaluated. By referring to the prediction results of YOLOv2 on the ZedBoard board, the accuracy of object recognition will not be affected by changing the platform, and it still achieved the similar prediction results compared with the graphics card results. Compared with other platforms of the previous research, CNN's performance on ZedBoard and PYNQ-Z2 has improved. However, the power efficiency of ZedBoard in YOLOv2 still cannot be higher than GTX TITAN X because the resource utilization in ZedBoard has not achieved maxima.

For the ResNet50, the power consumption of ZedBoard is less than both CPU (Xeon E5-2650) and GPU (GTX TITAN X) platform about x29 times and x78 times respectively. Then, the power efficiency of ZedBoard is increased about x38 times compared with CPU (Xeon E5-2650) and x4 times compared with GPU (GTX TITAN X). For the YOLOv2, the power consumption of ZedBoard is less than CPU (i5-4200M) and GPU (GTX 750Ti) about x12 times and x19 times respectively, while the power consumption of PYNQ-Z2 is less than CPU (i5-4200M) and GPU (GTX 750Ti) about x19 times and x33 times respectively. Next, the power efficiency of PYNQ-Z2 is better than CPU (i5-4200M) and GPU (GTX 750Ti) about x111 times and x2 times respectively. Besides, the time taken of ZedBoard and PYNQ-Z2 to predict the image is x4 times and x6 times respectively faster compared with CPU (i5-4200M). For the BNN trained with CIFAR-10 dataset, the time taken for hardware implementation to classify an image is about x936 times faster than the software implementation. For the BNN trained with MNIST dataset, the time taken for hardware implementation to classify a digit number is about x3403 times faster than the software implementation.

Finally, we have succeeded to achieve the objectives of this project as mention in Chapter 1. We investigated a few methods and selected the most suitable methods to accelerate CNN. Besides, we optimized and analyzed the accelerated CNNs on ZedBoard and PYNQ-Z2 to achieve low power consumption and high-power efficiency. Next, we evaluated the performance of ZedBoard and PYNQ-Z2 compared with previous researches and concluded that FPGA is the best platform to accelerate CNN.

5.3 Recommendation

To enable the CNNs to achieve the most effective performance in FPGA, there are still have some areas required to be enhanced for future work.

Since there are only 70% of DSPs usage in FPGAs used for YOLOv2. To improve the power efficiency of YOLOv2 in this project, one of the methods is to increase the DSP usage of FPGA. The function of DSP can improve the ability of FPGA to compute CNN and it also will provide more channels to be processed at a given time. More channels mean that the number of memory transactions decreases and the performance capability of each layer in neural network increase.

Next, the other method can use to improve the precision results of YOLOv2 on PYNQ-Z2 is replacing the default pre-trained file with the large training dataset. When more images used as the labelled dataset would improve the training results and get high accuracy precision results.

Another possible way of improving performance would be to optimize the already existing logic and block design. Due to the reconfiguration characteristic of FPGA, the optimization of design can allow it to achieve high-speed processing and high-power efficiency.

REFERENCES

[1] D. Gschwend, “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network,” no. August 2016, pp. 1–102, 2016.

[2] “Classification of Road Side Material Using Convolutional Neural Network and a Proposed Implementation of the Network Through Zedboard Zynq 7000 Fpga,” no. December, 2017.

[3] L. Huang, Y. Yang, Y. Deng, and Y. Yu, “DenseBox: Unifying Landmark Localization with End to End Object Detection,” pp. 1–13, 2015.

[4] E. Cengil, A. Çinar, and E. Özbay, “Image classification with caffe deep learning framework,” *2nd Int. Conf. Comput. Sci. Eng. UBMK 2017*, pp. 440–444, 2017.

[5] D. Danopoulos, C. Kachris, and D. Soudris, “Acceleration of image classification with Caffe framework using FPGA,” *2018 7th Int. Conf. Mod. Circuits Syst. Technol. MOCAST 2018*, pp. 1–4, 2018.

[6] M. Komar, P. Yakobchuk, V. Golovko, V. Dorosh, and A. Sachenko, “Deep Neural Network for Image Recognition Based on the Caffe Framework,” *Proc.*

2018 IEEE 2nd Int. Conf. Data Stream Min. Process. DSMP 2018, pp. 102–106, 2018.

[7] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An FPGA-based CNN accelerator integrating depthwise separable convolution," *Electron.*, vol. 8, no. 3, 2019.

[8] Z. Ren, S. Yang, F. Zou, F. Yang, C. Luan, and K. Li, "A face tracking framework based on convolutional neural networks and Kalman filter," *Proc. IEEE Int. Conf. Softw. Eng. Serv. Sci. ICSESS*, vol. 2017-Novem, pp. 410–413, 2018.

[9] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2072–2085, 2019.

[10] Y. Tu, S. Sadiq, Y. Tao, M. L. Shyu, and S. C. Chen, "A power efficient neural network implementation on heterogeneous FPGA and GPU Devices," *Proc. - 2019 IEEE 20th Int. Conf. Inf. Reuse Integr. Data Sci. IRI 2019*, pp. 193–199, 2019.

[11] Y. Han and E. Oruklu, "Traffic sign recognition based on the NVIDIA Jetson TX1 embedded system using convolutional neural networks," *Midwest Symp. Circuits Syst.*, vol. 2017-Augus, pp. 184–187, 2017.

[12] Z. Jiao, Y. Yang, H. Zhu, and F. Ren, "Realization and Improvement of Object Recognition System on Raspberry Pi 3B+," *Proc. 2018 5th IEEE Int. Conf. Cloud Comput. Intell. Syst. CCIS 2018*, pp. 465–469, 2019.

[13] L. Stornaiuolo, M. Santambrogio, and D. Sciuto, "On how to efficiently

implement deep learning algorithms on PYNQ Platform," *Proc. IEEE Comput. Soc. Annu. Symp. VLSI, ISVLSI*, vol. 2018-July, pp. 587–590, 2018.

[14] J. H. Kim, J. Lee, and J. Anderson, "FPGA Architecture Enhancements for Efficient BNN Implementation," *Proc. - 2018 Int. Conf. Field-Programmable Technol. FPT 2018*, pp. 217–224, 2018.

[15] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," *FPGA 2017 - Proc. 2017 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, no. February, pp. 65–74, 2017.

[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once : Unified , Real-Time Object Detection," 2016.

[17] B. Leibe, J. Matas, N. Sebe, and M. Welling, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9906 LNCS, pp. VII–IX, 2016.

[18] J. Redmon and A. Farhadi, "YOLO9000: Better , Faster , Stronger," *Comput. Vis Pattern Recognitt. (CVPR), 2017 IEEE Conf.*, pp. 6517-6525, 2017.

[19] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded GPU systems," *Proceeding 2017 2nd Int. Conf. Inf. Technol. INCIT 2017*, vol. 2018-Janua, pp. 1–5, 2017.

[20] H. Nakahara, M. Shimoda, and S. Sato, "A demonstration of FPGA-based you only look once version2 (YOLOv2)," *Proc. - 2018 Int. Conf. Field-Programmable Log. Appl. FPL 2018*, vol. 2, pp. 457–458, 2018.

[21] X. Liu, M. Che, "A Parallel Architecture of AdaBoost-Based Face Detection for

Gaze Estimation,” 2011 International Conference on Multimedia Technology, ICMT ,2011.

[22] C. Zheng, M. Yang, and C. Wang, “A Real-Time Face Detector Based on an End-to-End CNN,” *Proc. - 2017 10th Int. Symp. Comput. Intell. Des. Isc. 2017*, vol. 2018-Janua, pp. 393–397, 2018.

[23] C. Chen, ZL. Chai, and J. Xia, “Design and implementation of YOLOv2 accelerator based on Zynq FPGA heterogeous platform”, Journal of Frontiers of Coputer Science and Technology, 2019.

[24] Z. Ruizhe, N. Xinyu and W. Yajie, “Optimizing CNN-Based Object Detection Algorithms on Embedded FPGA Platforms”, Springer International Publishing AG, ARC 2017, LNCS 10216, pp. 225-26, 2017.

[25] Y. Guan, H. Liang, N. Xu, W. Wang, and S. Shi, “FP-DNN : An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates,” Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2017.

[26] D. Wu, J. Chen, and A. W. Algorithm, “A Novel Low-Communication Energy-Efficient Reconfigurable CNN Acceleration Architecture,” Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL, no. 2, pp. 64–67, 2018.

[27] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, “An Automatic RTL Compiler for High-Throughput FPGA Implementation of Diverse Deep Convolutional Neural Networks.” 2017 27th International Conference on Field Programmable Logic and Applications, FPL, 2017

[28] V. Gokhale, A. Zaidy, A. Xian, M. Chang, and E. Culurciello, “Snowflake : A Model Agnostic Accelerator for Deep Convolutional Neural Networks.”, 2017

[29] M. Zhu, Y. Zhuo, C. Wang, W. Chen, Y. Xie, and S. Barbara, “Performance Evaluation and Optimization of HBM-Enabled GPU for Data-intensive Applications,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, no. 1, pp. 1245–1248, 2018.

[30] S. Aldegheri, N. Bombieri, D. D. Bloisi, and A. Farinelli, “Data Flow ORB-SLAM for Real-time Performance on Embedded GPU Boards,” IEEE International Conference on Intelligent Robots and Systems, pp. 5370–5375, 2019.

[31] H. Nakahara, T. Fujii, and S. Sato, ”A fully connected layer elimination for a binarizec convolutional neural network on an FPGA”, 2017 27th International Conference on Field Programmable Logic and Applications, FPL, 2017.

[32] Digilent, "Python productivity for Zynq," [Online]. Available: http://www.pynq.io/board.html

[33] Xilinx, "AI Developer Hub," Xilinx, [Online]. Available: https://www.xilinx.com/products/design-tools/ai-inference/ai-developer-hub.html

[34] J. Redmon, "YOLO: Real-Time Object Detection," [Online]. Available: https://pjreddie.com/darknet/yolov2/

[35] I. Kuon, and J. Rose, “Measuring the Gap Between FPGAs and ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 203-215, 2007.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “*Gradient-based learning applied*

to document recognition", Proceedings of the IEEE, 1998.

[37] D. Rongshi, and T. Yongming, " *Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS", 2019 3rd International Conference on Circuits, System and Simulation (ICCSS), 2019.*

APPENDICES A

```
import bnn

print(bnn.available_params(bnn.NETWORK_CNVW1A1))



hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,'cifar10',bnn.RUNTIME_HW)

sw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,'cifar10',bnn.RUNTIME_SW)



print(hw_classifier.classes)



from PIL import Image

import numpy as np



im = Image.open('/home/xilinx/jupyter_notebooks/bnn/pictures/deer.jpg')

im
```

```
class_out=hw_classifier.classify_image(im)

print("Class number: {0}".format(class_out))

print("Class name: {0}".format(hw_classifier.class_name(class_out)))



class_out = sw_classifier.classify_image(im)

print("Class number: {0}".format(class_out))

print("Class name: {0}".format(sw_classifier.class_name(class_out)))



from pynq import Xlnk



xlnk = Xlnk()

xlnk.xlnk_reset()
```

APPENDICES B

```
import bnn

print(bnn.available_params(bnn.NETWORK_LFCW1A1))



hw_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A1,"mnist",bnn.RUNTIME_HW)

sw_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A1,"mnist",bnn.RUNTIME_SW)



print(hw_classifier.classes)



import cv2

from PIL import Image as PIL_Image

from PIL import ImageEnhance

from PIL import ImageOps



# says we capture an image from a webcam
```

```
cap = cv2.VideoCapture(0)

_ , cv2_im = cap.read()

cv2_im = cv2.cvtColor(cv2_im,cv2.COLOR_BGR2RGB)

img = PIL_Image.fromarray(cv2_im).convert("L")



#original captured image

#orig_img_path = '/home/xilinx/jupyter_notebooks/bnn/pictures/webcam_image_mnist.jpg'

#img = PIL_Image.open(orig_img_path).convert("L")



#Image enhancement

contr = ImageEnhance.Contrast(img)

img = contr.enhance(3)

# The enhancement values (contrast and brightness)

bright = ImageEnhance.Brightness(img)

# depends on backgroud, external lights etc

img = bright.enhance(4.0)



#img = img.rotate(180)

#Rotate the image (depending on camera orientation)

#Adding a border for future cropping
```

```
img = ImageOps.expand(img,border=80,fill='white')

img



from PIL import Image as PIL_Image

import numpy as np

import math

from scipy import misc



#Find bounding box

inverted = ImageOps.invert(img)

box = inverted.getbbox()

img_new = img.crop(box)

width, height = img_new.size

ratio = min((28./height), (28./width))

background = PIL_Image.new('RGB', (28,28), (255,255,255))

if(height == width):

    img_new = img_new.resize((28,28))

elif(height>width):

    img_new = img_new.resize((int(width*ratio),28))

    background.paste(img_new, (int((28-img_new.size[0])/2),int((28-img_new.size[1])/2)))
```

```
else:

    img_new = img_new.resize((28, int(height*ratio)))

    background.paste(img_new, (int((28-img_new.size[0])/2),int((28-img_new.size[1])/2)))



background

img_data=np.asarray(background)

img_data = img_data[:,:,0]

misc.imsave('/home/xilinx/img_webcam_mnist.png', img_data)



from array import *

from PIL import Image as PIL_Image

from PIL import ImageOps

img_load = PIL_Image.open('/home/xilinx/img_webcam_mnist.png').convert("L")

# Convert to BNN input format

# The image is resized to comply with the MNIST standard. The image is resized at 28x28
pixels and the colors inverted.



#Resize the image and invert it (white on black)

smallimg = ImageOps.invert(img_load)

smallimg = smallimg.rotate(0)
```

```
data_image = array('B')


pixel = smallimg.load()

for x in range(0,28):

    for y in range(0,28):

        if(pixel[y,x] == 255):

            data_image.append(255)

        else:

            data_image.append(1)


# Setting up the header of the MNIST format file - Required as the hardware is designed for
MNIST dataset

hexval = "{0:#0{1}x}".format(1,6)

header = array('B')

header.extend([0,0,8,1,0,0])

header.append(int('0x'+hexval[2:][:2],16))

header.append(int('0x'+hexval[2:][2:],16))

header.extend([0,0,0,28,0,0,0,28])

header[3] = 3 # Changing MSB for image data (0x00000803)
```

```
data_image = header + data_image

output_file = open('/home/xilinx/img_webcam_mnist_processed', 'wb')

data_image.tofile(output_file)

output_file.close()

smallimg



class_out = hw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")

print("Class number: {0}".format(class_out))

print("Class name: {0}".format(hw_classifier.class_name(class_out)))



class_out=sw_classifier.classify_mnist("/home/xilinx/img_webcam_mnist_processed")

print("Class number: {0}".format(class_out))

print("Class name: {0}".format(hw_classifier.class_name(class_out)))



from pynq import Xlnk

xlnk = Xlnk()

xlnk.xlnk_reset()
```