Department of Electrical and Electronic Engineering

M.Sc. in Advanced Microelectronic Systems Engineering

Energy Proportional Object Recognition with Convolutional Neural Networks

Author:

Mohamad Musab M. Asad

xs18319

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in Advanced Microelectronic Systems Engineering in the Department of Electrical and Electronic Engineering.

September, 2019

Abstract

Real-time classification systems pose a major challenge due to its low latency and high accuracy requirements. Recently, Convolutional Neural Networks (CNNs) have grown extremely due to their effectiveness at complex image recognition problems, however, they require high computation power and storage complexity. Extreme quantization of CNN weights and activation values introduce Binarized Neural Networks (BNNs) models. These models have a minor accuracy reduction, though they can be implemented efficiently in hardware platforms.

FPGA-based neural network accelerators have been implemented to surpass GPU platform in speed and energy efficiency. This project presents a power efficient FPGA-based real-time classification system, where the inference network is based on BNN implemented specifically for high speed applications. The aim of the project is to achieve the maximum classification throughput while consuming the minimal power by controlling the clock frequency of the FPGA. The evaluation results of the system show a maximum throughput of 500 FPS, which is limited by the camera speed, while consuming 0.545 W. Moreover, the classification accuracy of the system is improved from 68% to 81.2% by applying the proposed windowing filter, which averaged the decision based on the previous frames. Finally, comparing to other literature, the design shows a *12* times improvement on power efficiency while classifying frames with the camera speed.

Acknowledgements

I would first like to thank my supervisor Dr.Jose Nunez-Yanez for his constant support throughout this project. Dr.Jose was always available whenever I ran into a problem or I get stuck, he consistently tries to break the problem up and assist me with approaching the project.

In addition, I would like to thank my family for their wise counsel and sympathetic ear. You are always there for me.

Finally, my completion of this project could not have been accomplished without the support of my friend. To Obada, Yasien, Ibrahim – thank you for allowing me time away from your research and time.

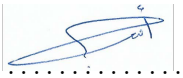
Declaration and Disclaimer

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Postgraduate Programmes and that it has not been submitted for any other academic award.

Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted from the work of others, I have included the source in the references/bibliography.

Any views expressed in the dissertation are those of the author.

The author confirms that the printed copy and the electronic version of this thesis are identical.

SIGNED: 

DATE: 29/9/2019

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence. 1
ASIC Application-Specific Integrated Circuit. 10
AXI Advanced Extensible Interface. 13

BNNs Binarized Neural Networks. 2

CNNs Convolutional Neural Networks. 1
Conv Convolutional Layer. 4
CPU Central Processing Unit. 1

DSP Digital Signal Processing. 7

FC Fully Connected. 5
FINN Framework for Fast, Scalable Binarized Neural Network. 9
FLOP/s Floating Point Operations per Second. 8
FMs Feature Maps. 4
FPGA Field Programmable Gate Array. 1
FPS Frames Per Second. 2

GPU Graphical Processing Unit. 1

HLS High-Level Synthesis. 7

IoT Internet of Things. 1

LUTs Look-Up Tables. 9

MVTU Matrix-Vector-Thresholding-Unit. 9

NNs Neural Networks. 1

OCM On-Chip Memory. 6

P Processing Elements. 18, 38
PCIe Peripheral Component Interconnect Express. 7
PL Programmable Logic. 7
PLL Phase Locked Loop. 20
PS Processing System. 7

ROI Region of Interest. 11

RTL Register Transfer Level. 7

S SIMD Lanes. 18, 38

SoC System-on-Chip. 7

SRAMs Static Random-Access Memory. 7

SSH Secure Shell. 14

SWU Sliding Window Unit. 9

Chapter 1

Introduction

With the increasing demand of applications related to image recognition, the amount of data processing has witnessed an exponential growth. Due to the emergence of the Internet of Things (IoT) concept, the amount of data generated daily is approximated to be 2.5 quintillion bytes at the current pace according to Forbes [1]. The increasing data volume and the advancement in computation capability of the current hardware leads to more development in the field of Artificial Intelligence (AI), particularly in deep learning and Neural Networks (NNs) [2].

Neural networks can be broken down into many categories, each exploits a different set of powerful techniques based on machine learning to tackle complex problems. NNs have a wide range of applications in the area of autonomous cars, field drones, surveillance cameras, and medical imaging diagnostic equipment. Particularly, Convolutional Neural Networks (CNNs) have proven their effectiveness in various fields related to computer vision including image classification, medical image analysis, and natural language processing [3].

In addition, the complexity of NNs is increasing with the increase of applications complexity. Taking industrial robots as an example, in order to effectively make use of them, a real-time processing system is required. To achieve this, the high computation and storage capability of neural network inference is needed [4]. Furthermore, the demanding computational power of NNs makes its implementation on CPU platforms inefficient. Therefore, GPU is the most widely used platforms for processing NNs due to its high computation ability, and the simplicity lied on the use of advanced frameworks.

Recently, FPGA-based accelerators design is becoming a research scope. Due to its capability for achieving high-parallelism and using NNs computation properties to eliminate the need for additional logic. The models of NNs can be modified to be hardware friendly without affecting the accuracy. Therefore, FPGA can be the next solution to overcome issues of GPU platforms i.e. speed and energy efficiency.

1.1 Scope and Objectives

Although visual recognition has improved as a result of the dramatic advancement in CNN [5], their usage in real-time task is considered limited due to limitation in resources, training and optimization techniques [6]. CNN models require high computation and memory

overhead because they have tens to hundreds of millions of parameters. Moreover, their requirements for billions of floating-point operations prevents deep CNN from embedded applications [7].

To address this issue, researches have proposed different methodologies to make CNN more hardware friendly such as Binarized Neural Networks (BNNs). This model has proven to be very efficient as it uses binary weights and binary internal representations instead of floating-point numbers. Additionally, it enables high computational performance, low power consumption and low latency; while offering the flexibility and scalability required for accelerating larger and more complex networks [8]. This project investigates how to integrate a web camera system with an FPGA-based binary CNN accelerator to perform real-time object recognition. The available hardware is a ZedBoard which is a heterogeneous platform including two ARM A9 processors and the Xilinx Zynq FPGA logic. Thus, a part of the system code will be executed in the ARM host processor, while the compute-intensive part which is the BNN itself will be accelerated on the FPGA chip.

The first aim of this project is to enable the system to classify images streamed from a camera with very low latency. The classification rate should be equal to the number of Frames Per Second (FPS) captured from the camera to achieve real-time classification. Moreover, the design is optimized in terms of throughput by using Xilinx SDSoC pragmas. This allows the execution of the hardware implementation of the BNN and the image capture process to run in parallel. Furthermore, processing latency of the captured frames before inputting them to BNN is investigated in order to reduce the processing time of the BNN.

The second aim is to test the network performance in term of accuracy and power consumption. At the end, the network should maintain its classification accuracy with the required throughput. Based on the proposed design, experiments with a different approach of high-frame-rate windowing techniques is investigated to improve the accuracy of the network. Thus, the decision will not consider one frame for classification, rather multiple frames will be considered to take the most probable window size. The classification energy is proportional to the required throughput, which is adapted by controlling the clock frequency of the FPGA. Finally, experiments are conducted to study the optimal parameters of the system to maximize the accuracy.

In the ideal design, the frame is captured by the camera and then processed by the processing unit. After that, the hardware on the FPGA inference those frames and classifies the objects. The overall classification latency should be equal to the frame capture latency of the camera. The network is already trained, and weights are calculated for CIFAR-10 dataset that consists of 10 classes such as deer, dog, car, etc.

1.2 Thesis Outline

The outline of this thesis is organized as the following:

- Chapter 2: This chapter describes the theoretical background of CNNs and BNNs, then a review of related work regarding BNN and its implementation in hardware

classification system is discussed.

- Chapter 3: The methodology followed to implement the system is discussed in this chapter. The proposed solution to improve the throughput and the accuracy is provided as well.
- Chapter 4: This chapter describes the implementation steps for the system hardware and software functions in detail.
- Chapter 5: Several experiments are conducted, and their results are evaluated in this chapter. The factors affecting the system performance are also discussed.
- Chapter 6: Conclusions about this thesis are presented, and a direction for possible improvements for this work is provided.

Chapter 2

Background

This chapter discusses the basic concepts of CNN and its computation load followed by a comparison between CNN and BNN. After that, FPGA-Based design of BNN accelerators in terms of computational unit optimization is reviewed. Finally, a complete real-time image classification system based on BNN hardware implementation is addressed.

2.1 Convolutional Neural Networks (CNNs)

A CNN is a classifier that takes multi-dimension images as an input, extracting their features and generating Feature Maps (FMs) in order to determine the probability of objects in the input and classify them to classes.

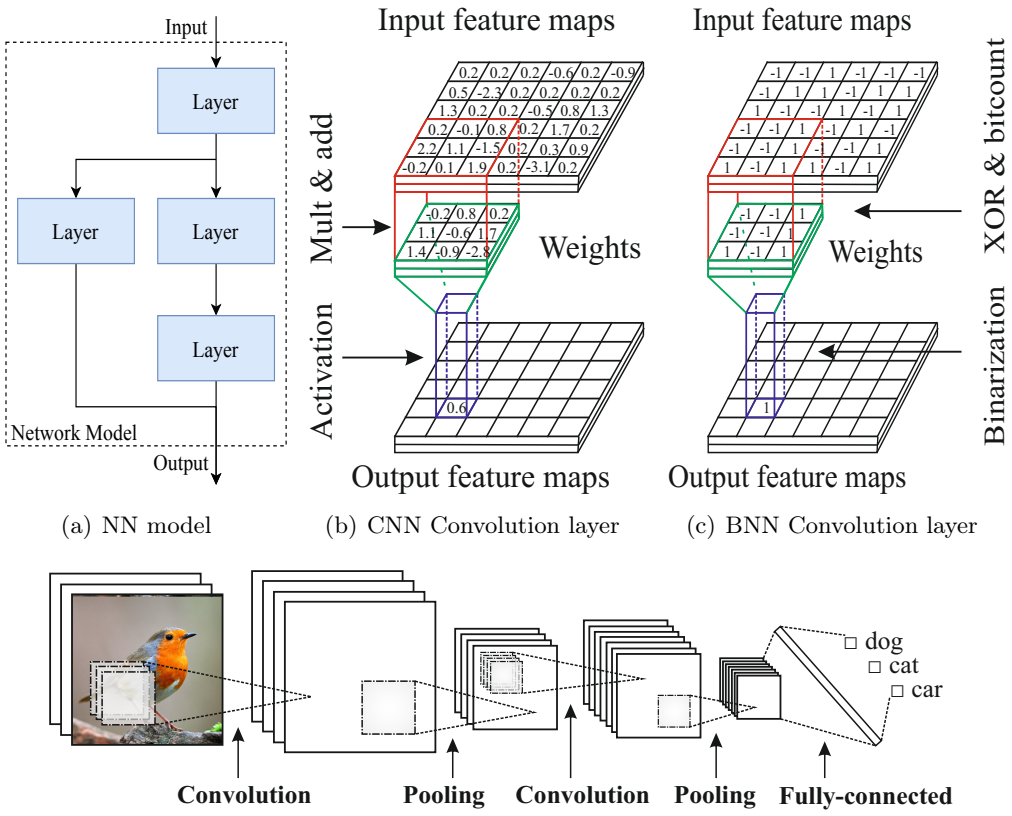
The basic feed-forward structure of NN can be modeled as a directed graph shown in Fig.2.1(a) [4]. Each layer processes the data from the input or the previous layer, it then produces data to be fed into the next layer or the output. The input/output of each layer is called the activations. Meanwhile, the parameters related to each layer are called weights which are calculated in the *training* phase. Typically, NN models are trained offline by using a pre-classified set of data. After that, they can be used for inference, which means predicting a new set of data. The training phase is out of the scope of this work, where it is assumed that all parameters are calculated.

A typical CNN model is composed of three basic layers shown in 2.1(d) and are described below:

Convolutional Layer (Conv) is the key attribute that characterises CNN from other NN models. This layer extracts a 2D FM of each input image or FMs from the previous layer by applying a convolution operation with $k \times k$ weight filter (kernel). The result is then passed through a non-linear activation function to generate one FM. The mathematical equation of this layer is shown below:

$$Y_j = f(\sum_i X_i * K_{ij} + B_j) \tag{2.1}$$

Where i is the number input FMs each convolves with $k \times k$ filter to generate j output FMs after the addition of the activation bias B. The activation function helps to identify non-linear features to the input data by applying a non-linear mathematical operation such



(a) NN model (b) CNN Convolution layer (c) BNN Convolution layer

(d) CNN layers configuration

Figure 2.1: Convolutional neural network model with insight on Conv layer in comparing to binary convoultion . Figures reproduced from [4, 9]

as sigmoid, tanh, or rectified linear unit (ReLU) [10]. Fig.2.1(b) shows the Conv layer with ReLU function to generate the output FMs. The borders of the input maps are usually padded with zeros to generate an output with similar size to the input.

Pooling layer is used to reduce the resolution of the output FMs or simply down sample the output. This decreases the number of parameters and the computation load in the next layer. Max polling divides an image into small windows and takes the maximum value of each window. Another common function is Average Polling which takes the average of the window instead of the maximum value.

Fully Connected (FC) layer is mainly the classification layer where neurons are fully connected to the next layer. This layer takes the last FM as an input after flattening it to a vector, then generates the output such as a class score [11].

2.1.1 Computation Overhead of CNN

Increasing the accuracy of CNN models comes with the cost of high computation and storage complexity. Table 2.1 summarizes the total number of parameters, operations (addition or multiplication) and the accuracy on ImageNet dataset for different advanced CNN architectures [12].

Improving CNN accuracy while maintaining its computational complexity is a challenging task for CNN developers. This task is compromised in GoogLeNet and ResNET models.

Table 2.1: Accuracy and workload of state-of-the-art NN accelerators design

	AlexNet [5]	**VGG-19** [13]	**GoogLe-Net** [14]	**ResNet-152** [15]	**Mobile-Net** [16]	**Shuffle-Net** [17]
Year	2012	2014	2015	2016	2017	2017
layers	8	19	22	152	-	-
Total workload (MACs)	724M	39G	1.58G	22.6G	1.1G	0.27G
Total Parameters	61M	144M	6.99M	57M	4.2M	2.36M
Top-1 Accuracy	61.0%	74.5%	68.9%	79.3%	70.6%	67.6%

Recent models such as MobileNet and ShuffleNet dramatically decrease the number of parameters that suffer from accuracy reduction.

Accelerating CNN network on hardware usually considers only Conv and FC layers as they take most of the computation load as shown in Fig. 2.2. The figure shows that the Conv and FC layers require more than 99% of the network parameters and operations. In addition, non-linear and FC layers have no data reuse while pooling layer has low data reuse. In contrast, data reusing is high in Conv layers because the kernel weights are needed for each FM of the image, making it an obvious candidate for hardware acceleration.

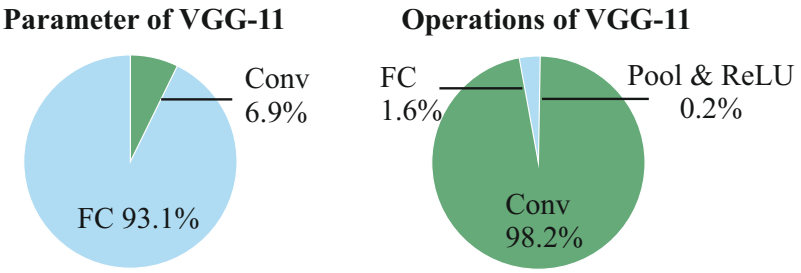


Figure 2.2: VGG11 model computation load and parameters of different layers, Conv and FC layers dominate other layers. Figure reproduced from [4].

2.2 Binary Neural Networks (BNNs)

The binary neural network is an extreme quantized model of CNN which uses binary weights and activations for the Conv layer as shown in Fig. 2.1(c) [18]. Reducing precision has two main advantages:

- The convolution operation in equation 2.1, can be simplified by realising the mathematical multiplication with XNOR between two $k \times k$ bit-vectors and a pop-count [19]. This simplification makes the convolution hardware-friendly as it can be implemented very efficiently in FPGA.

- Using binarized weights and activation for FMs and fully connected layer reduce their memory size. This also an advantage of FPGA designs as they are limited to On-Chip Memory (OCM) size or off-chip memory bandwidth.

Layers order and data types of both CNN and BNN are shown in Fig.2.3. Layers order of CNN is the same as Equation2.1, while pooling for BNN is done directly after convolution. In

BNN, batch normalization layer is introduced to reduce information loss during binarization. This layer performs a linear shift and scale to balance the input distribution to have a zero mean and a unit variance. This reduces quantization error comparing to random input distribution.

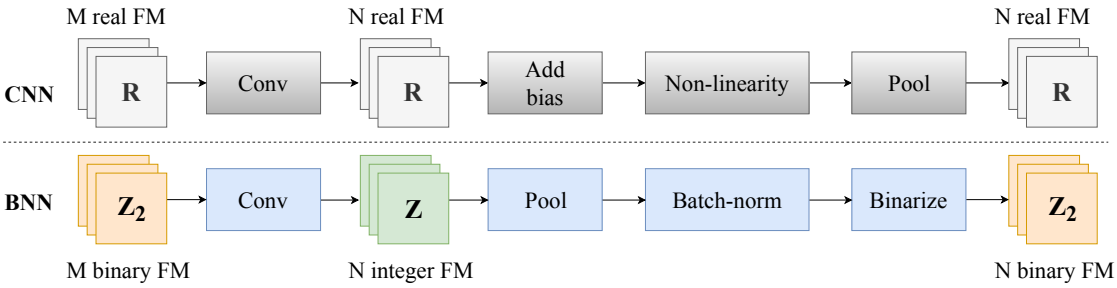


Figure 2.3: Layers order comparing between CNN and BNN. Figure reproduced from [20]

2.3 FPGA-Based Accelerator

Choosing the right platform to implement CNNs is essential due to its high computation load. The FPGA-based platforms can provide a solution to accelerate CNNs algorithms. The main advantage of FPGAs is that only the necessary logic can be accelerated in the logic fabric, while GPU, CPU, and DSP platforms implement software and hardware separately. A typical architecture of FPGA-based design consists of a CPU and FPGA logic connected through PCIe connection. The state-of-the-art System-on-Chip (SoC) platform integrates both the Processing System (PS) and Programmable Logic (PL) in the same chip. Both PS and PL have their own memory; however, they can access each other memory through the bus interface. Neural networks accelerators are usually designed in PL and controlled by the PS through software.

There are many challenges and limitations related to FPGA CNNs accelerators design. One trade-off is between controllability level and developing time of FPGA programming languages.

Recently, High-Level Synthesis (HLS) approaches decreasing the developing time and hiding complex details by using a high-level of abstraction arose. The design flow of C++ CNN classifier system is shown in Fig. 2.4(a), where Vivado HLS tools are used to translate C++ code into RTL code. Although, using HLS simplifies algorithms acceleration by pipelining and partitioning, the resulting design may not be mapped efficiently on FPGA. On the other hand, RTL coding can be more hardware efficient, though it requires more developing time and experiences with both CNN models and FPGA architecture.

The OCM of FPGA consists of registers and SRAMs. This is considered as a limiting factor because it is not enough to save all CNN model parameters as shown in Fig. 2.4(b). The figure shows that AlexNet model requires around 250MB to save all 60M parameters based on 32-bit floating point representation [21]. This cannot be stored even in a modern FPGA where it can implement up to 50MB on-chip SRAM [4]. Solving this issue requires the use of an external memory, which dominates the power consumption of the system and

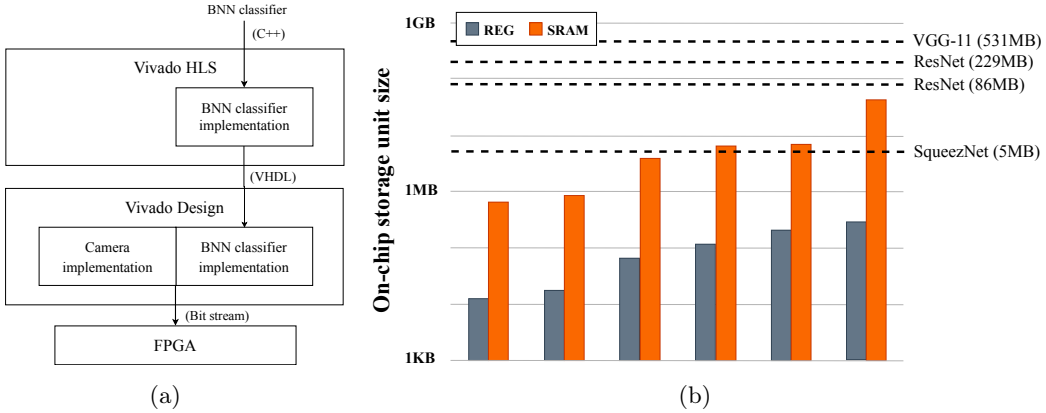


Figure 2.4: (a) FPGA accelerator of an image classification system based on Vivado HLS. (b) NN models memory requirements. The chart shows different FPGA chips memory resources which are not enough to store all NN models parameters [8].

limits its speed to the memory bandwidth.

The computational resources are another limiting factor. The state-of-the-art FPGAs can achieve up to 10 TFLOP/s (Floating Point Operations per Second) because it can implement up to thousands of DSP units. However, this computation capacity decreased to 20 GFLOP/s for low-end FPGAs like Xilinx XC7Z020, which is not enough for real-time processing and classification.

2.4 Related Work

Different hardware approaches have been used in modern FPGA-based BNN accelerator design to achieve high performance and decrease power consumption. The following review will discuss the using of low bit-width for the computation unit of BNN, and an evaluation of real-time classification systems based on BNN.

2.4.1 FPGA-Based BNN Design

A direct method to scale down the computational unit size is by reducing the parameters' bit-width. This can be realized by quantization techniques which allow replacing floating-point units with fixed-point units. BNN models are characterized by their extreme narrow bit-width which can be based on 1 bit or 2 bits. Throughout literature, there are many designs which exploit the benefits of this reduction and focus on computation units for linear quantization [7, 8, 22, 23, 24, 25]. Heterogeneous computation units are used in the designs of [19, 26].

The model used in this project is based on the design proposed by Umuroglu et al. [8]. They estimated the performance of BNN model comparing to other fixed-point CNN models by using a roofline model which estimates the performance depending on memory bandwidth, peak computational performance and arithmetic workload [27]. They concluded that BNN is 16 times faster than fixed-point models. However, achieving the same accuracy on MNIST dataset required 2 to 11 times more operations and parameters comparing to an

8-bit fixed-point networks [28].

Subsequently, they introduced FINN (a Framework for Fast, Scalable BNN) accelerator adopted on heterogeneous streaming architecture which uses separate computation engine for each layer and communicates through the on-chip data stream. The engines are pipelined enabling the next engine to start computing once the output of the previous engine is generated. In addition, all parameters of BNN are stored on OCM due to the compact BNN network which uses a fully binarize model (use only 1-bit for all input/output activations and weights). The structure of their design implemented the basic BNN properties where batch-normalization was used prior to activation. The activation function uses the sign function implemented by using thresholding, which requires only a few Look-Up Tables (LUTs). After that, max-pooling layer was designed using an OR operation.

The Conv layer in their design consists of two units; the Sliding Window Unit (SWU) described in [29] and the Matrix-Vector-Thresholding-Unit (MVTU). SWU was used to reduce the convolution operation to matrix multiplication and generate the image matrix from input FMs. MVTU was adapted to compute matrix multiplication using a different column vector from the image matrix each time by using only XNOR and pop-count operations. The design was implemented by using HLS coding level to automate loops pipelining and unrolling. The experimental results of their design showed high classification throughput (up to 12.3 M image/s), high accuracy, and low power consumption.

2.4.2 BNN Review Evaluation

The performance of the mentioned designs is compared with FINN design in terms of speed and energy efficiency. The terms used for comparison are as follow:

- *Speed* is defined as the number of operations per second measured in Giga operations Per Second (GOPS).
- *Power consumption* is defined as the total power consumed by the design measured in Watt (W).
- *Throughput* is defined as the image classification speed measured in frame per second (FPS).

Using the reported values of area (LUTs, BRAMs), speed, and power. The efficiency η of the design can be calculated by using the following equations:

$$Area\,\eta = \frac{\text{Speed}}{\#\text{LUTs}} \ \text{(GOPS/K LUTs)} \tag{2.2}$$

$$Memory\,\eta = \frac{Speed}{\#BRANs} \ \text{(GOPS/BRAMs)} \tag{2.3}$$

$$Power\,\eta = \frac{Speed}{Power} \ \text{(GOPS/W)} \tag{2.4}$$

The results are summarized in Table 2.2. It is notable that each design has used different framework structure and the network is trained and tested against different datasets. For a

reasonable compression, this evaluation will focus on the results obtained from CIFER-10 dataset and the reported top-1 accuracy.

The implementation in [26] reported an outstanding speed of 40.77 TOPS by using *a heterogeneous system* which allows accelerating only the BNN hardware friendly parts. In addition, the design accomplished the highest power efficiency of 849.38 GOPS/W with ImageNet dataset.

FINN design in [8] achieved the highest throughput of 21.9K FPS for the cost of reducing the classification accuracy by 7% in comparison to the design in [19]. The other designs in [3, 24, 25] are compared to FINN design and the following trade-off can be interpreted:

- The classification accuracy in [3] was increased by 8% for the cost of decreasing the throughput to 12K FPS. The power efficiency was also improved by a factor of 1.72.
- A compromise between accuracy and throughput is considered in ReBNet [24]. By adding three residual layers the accuracy of their design increased by 7% while decrease throughput to 2K FPS.
- Removing the internal fully connected layers in [25] increased the memory efficiency by a factor of 1.55 with almost no effect on the system accuracy.

A fair comparison with [7] is not possible because it uses ImageNet dataset.

Table 2.2: The performance results of BNN designs

Accelerator	**Speed (SOPS)**	**Power (W)**	**Area η**	**Memory η**	**Power η**	**FPS**	**Top-1 Acc.**
Umuroglu et al. [8]	2466	11.7	53.3	13.26	210.73	21.9K	80.10%
Zhao et al in [19]	208	4.7	4.43	2.21	44.21	168	87.73%
Yonekawa et al. [22]	461	22	9.61	0.67	20.95	31.48	80.16%
Fraser et al. [23]	14814	41	37.7	8.17	361.32	12K	88.70%
Ghasemzadeh et al. [24]	-	-	-	-	-	2K	86.96%
Duncan et al. [26]	40770	48	35.45	-	849.38	-	71.3%
Nakahara et al. [25]	329	2.3	22.71	20.59	143.25	420	81.80%
Jiao et al. [7]	410	2.26	9.32	3.89	181.51	106	-

2.4.3 FPGA-Based Real-Time Classification Systems

Utilizing BNN as a classifier in image processing applications has expanded after Courbariaux et al [30] proposed BinaryConnect, which is a method of training binary network during the forward and backward propagation. It was proven that the accuracy of BNN is comparable to advanced NNs in image processing systems.

The applicability of implementing BNN on hardware platforms i.e. CPU, GPU, FPGA, and ASIC was examined by Nurvitadhi et al. [31]. The experimental results demonstrated the advantages of implanting BNN on FPGA in terms of power efficiency and computation capability. Compared to CPU/GPU platforms which demand the use of external memory, BNN can be stored in OCM and accelerated more efficiently on FPGA. Many FPGA implementations were proposed [8, 19, 23, 26, 32] demonstrating the ability to achieve high performance/watt rate .

Subsequently, BNN was used to realize an image application system as presented by Mazare et al. [33]. A real-time pattern recognition system based on FPGA was realized. The system consists of an image acquisition module to import images from a camera, a processing unit to convert the images to monochrome format, and a BNN that extracts features and recognize patterns. The resolution of images is processed and reduced by a factor of 10 by reducing the size of BNN. Overall, the system was reported to perform a few tens of FPS though image resolution was very low.

Jokic et al. [34] introduced an FPGA-based 20k FPS real-time image recognition using a BNN called BinaryEye. The data flow of the system is shown in Fig.2.5. An image sensor was employed to take images, then the images are stored in the line-buffer. Following this, ROI block was used to identify the Region of Interest (ROI) from the taken image and process it rather than having to process the whole image. This tackles the issue of having to employ a large classifier. The extracted regions are then binarized for binary-input classifier which is a BNN based on FINN design [8]. The BinaryEye design was oriented to achieve classification throughput close to camera speed, which is 20K FPS. The implementation was reported to achieve the required throughput with only one-quarter of BRAMs and less than 50% of FPGA logic. The design was tested for the handwritten digit (MNIST dataset) with a classification accuracy of 98.4% (similar to FINN accuracy for MINST dataset [8]).

Quantized NN is also used in other applications such as object detection as cited by Pruber et al. [35]. A real-time object detection in a video live processed by the heterogenous Zynq UltraScale+ platform was designed. This implementation was based on Tiny YOLO topology which uses bounding box and class predictions to localize objects [36]. Tiny YOLO was able to achieve up to 244 FPS on GPU [37]. Additionally, the open-source Darknet was used as a training and inference framework. Vivado HLS was then employed to translate the implemented C code to RTL level [38].

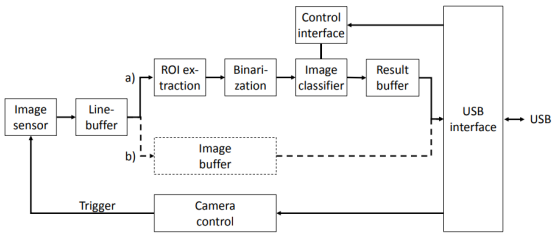


Figure 2.5: BinaryEye block diagram. Path a. data-bath for classification on FPGA board. Path b. direct transferring of images [34]

Various implementations of quantized NN provided that quantizing input/output layers is very sensitive. Hence, an 8-bit fixed-point representation for these layers was utilized in this project. On the other hand, the internal layers were shown to be less sensitive to quantization, thus FINN-based implementation for the internal layers were used. Their design was capable to achieve a throughput up to 21.9K FPS with adequate accuracy of 80.1% for CIFAR-10 dataset.

Chapter 3

Methodology

This chapter discusses the designed system methodology which was pursued to achieve the required classification throughput. The chapter starts by providing a general description of the proposed system. Then, a further description of the hardware and software resources used to accelerate the throughput and enhance the accuracy is presented. Finally, the implementation work plan is discussed.

The basic structure of the system is based on an image sensor with a specified image acquisition speed and an NN classifier. For real-time systems, the typical speed of the classifier should be close to that of the sensor speed. However, even with a well-designed NN, the speed of the system could be limited to the external memory bandwidth, which is referred to as *memory-bottleneck*. This requires extensive access to the external memory which reduces the system performance due to the limited memory bandwidth and its associated power consumption [39].

One possible solution to address this issue is by reducing the volume of the processed data. This is done by filtering, extracting ROI or resizing the image, which reduce the required computation and the number of parameters of the NN. Consequently, the system achieves higher power-efficiency and throughput.

The general system blocks are shown in Fig.3.1. As it can be seen, the image sensor is the starting point and it is connected directly to the FPGA-board through a USB interface. Then the captured images are passed through many intermediate steps on both processing system (PS) and programmable logic (PL). The functions of the system can be divided into three blocks as following:

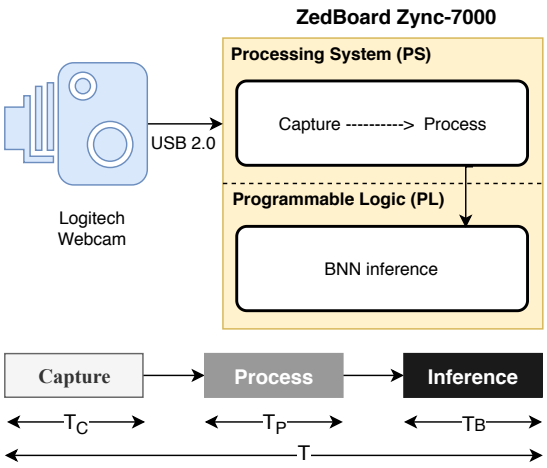


Figure 3.1: Block diagram of the proposed system using Zedboard

- **Capture:** Includes capturing the frame from the camera and saving it to a matrix by using the `OpenCV` library. Executed on SW.
- **Process:** Matches the captured frame to the required BNN form. Executed on SW
- **Inference:** The classification is done by using the BNN. Executed on HW

3.1 Hardware and Software Resources

The hardware and software programs used in this project are discussed briefly. This includes the platform and the interface protocols between the hardware components, as well as the libraries and operating system used for the created platform.

3.1.1 Hardware

In this project, the Logitech C160 webcam is used as image sensor attached to the Xilinx Zync-7000 SoC ZedBoard. The board contains 7 series Xilinx FPGA and is equipped with a double ARM Cortex-A9 processors to create SW/HW powerful designs. Further details of the product specification can be found at Xilinx website [40]. The board is also equipped with the required I/O interface to connect the camera via USB 2.0 OTG port as shown in Fig.3.2

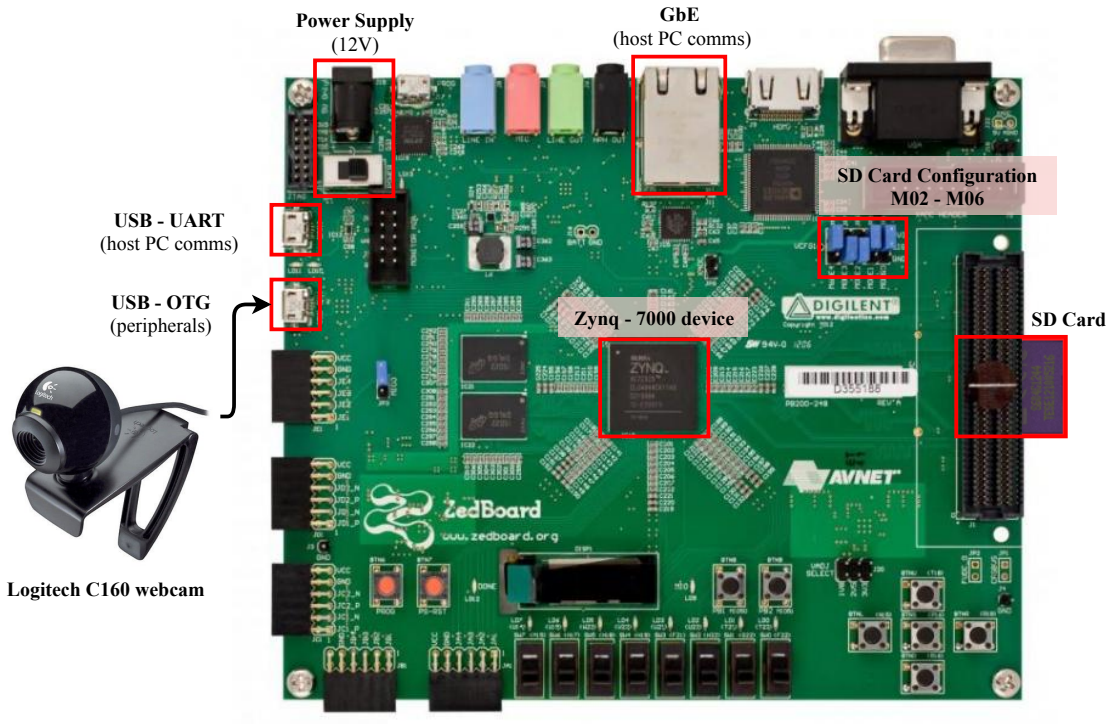


Figure 3.2: The hardware resources which consist of web camera and Zedboard

The clock frequency of the processors is fixed to 666.67 MHz, while the PL clock frequency can be programmed to span from 10 to 200 MHz. The PS and the PL are connected through the Advanced Extensible Interface (AXI) high performance bus interface, which provides 1200 MB/s bandwidth. The number of images which is needed to be processed in the PL is up to 500 image/second, where each image has $32 \times 32 \times 3 = 3072$

bytes. This requires a $500 \times 3072 = 1.536$ MB/s transfer rate. Thus, the communication between PL and PS is adequate for the required transfer rate.

3.1.2 Software

The system is running on PYNQ-Linux which is an open-source operating system developed by Xilinx to design embedded systems applications with SoC. The BNN overlay is designed and compiled on a host computer using SDSoC and Vivado. The SDSoC platform is configured to run on Linux OS and the following files are generated after compilation:

- BOOT image (`BOOT.bin`)
- kernel image (`uImage`)
- devicetree blob (`devicetree.dtb`)
- rootfs

The generated files are then copied to an SD card as described in Parth Parikh blog [41]. The board jumpers are configured to run the SD card as shown in Fig.3.2.

Moreover, the compiled BNN hardware function is in `libkernelbnn.a` file. The function can be called after including `kernelbnn.h` header in the `main.c`. The software functions can be modified and then compiled directly on the board. A list of the used compiler and libraries is shown below:

- *Linux GCC 5.2.1:* to compile other C/C++ codes.
- *SDSCC/SDS++ compilers:* to compile and link C/C++ source files into an application-specific hardware/software system on chip implemented on a Zynq-7000.
- *OpenCV 2.4.9:* used for frame processing.
- *OpenMP 4.0:* used for SW pipelining.

Communication over SSH

The communication between the host computer and the board is done over Secure Shell (SSH) protocol. WinSCP software is used to configure the IP address of both the host (the computer) and the client (the board). In addition, files can be transferred using SFTP file transfer protocol and graphical applications can be forwarded to the host screen by using X11 forwarding over SSH. The Xming program is used to display the client graphical applications on the host screen.

3.2 Pipeline Approach

The overall system latency is the sum of latency of the three blocks. Achieving the maximum throughput can be done by fully pipelining the blocks as shown in Fig.3.3. The implementation code of the approach is shown in Appendix.B.

The system has two parts: a *software part* which includes Capture and Process blocks. These blocks are executed in parallel using SW pipelining. The second part is the *hardware part* which can be run in parallel with the software part by using HW pipelining. The two parallelism techniques have been used and are discussed below:

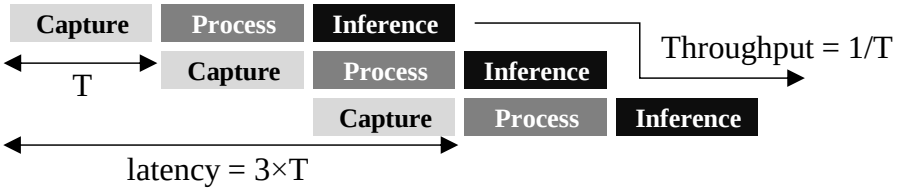


Figure 3.3: The system functions pipeline diagram

- **SW Pipelining:**

The software parts include Capture and Process blocks. Since the two blocks are independent, they can be executed in parallel. Additionally, the Zedboard is equipped with dual processors: P0 and P1. Forking the program and executing each block in a different processor can be done using OpenMP library. Capturing camera frames is executed on P0 while processing the *previous* frame is performed on P1 as shown in Fig.3.4. After capturing the first frame: *frame[0]*, P0 will capture $frame[i]$ while P1 process $frame[j = i - 1]$. The program is then joint before sending the processed frames to the PL.

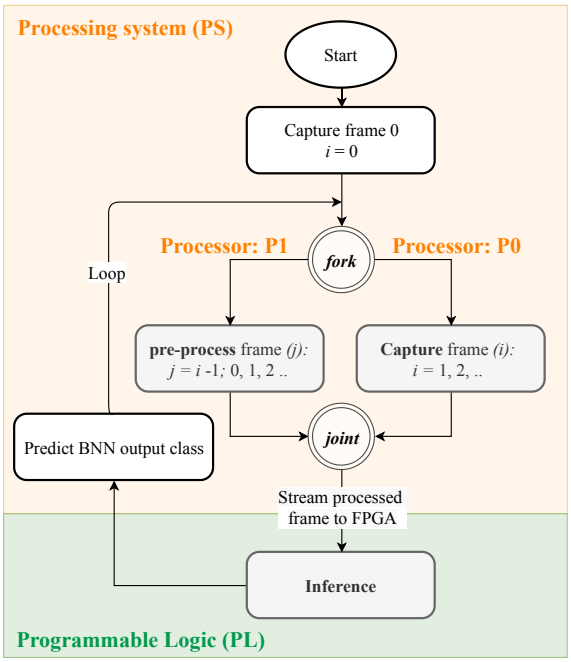


Figure 3.4: OpenMP fork-joint diagram used in SW pipelining

- **HW Pipelining:**

Using SDSoC pragma pairs `"async/wait"` allow asynchronous execution of the FPGA and the host processor. The pragma pairs provide parallelism between Capture-Process blocks and Inference block. The `async(ID)` pragma is then called before calling the hardware function as shown in the List.3.1.

```
for (i = 0; i < stop; i++) {
    if(frame[0]):
        #pragma SDS async(1)
        Call_FPGA (frame[0]);
    else:
        #pragma SDS wait(1)
            #pragma SDS async(1)
        Call_FPGA (frame[i]);
}
```

Listing 3.1: async/wait call

A wait function is generated after calling the hardware function, which makes the host processor idle until the hardware output is generated and sent back to the buffer. Using `async(ID)` pragma, the compiler does not generate wait function after calling the hardware function. Instead, after transferring the inputs to hardware, the program returns

immediately and continues with the main program, until the `wait(ID)` pragma is called to check the hardware output and waits if it is not ready. The ID of the pragma of the wait should be same as async to guide the compiler to wait for the hardware output of similar `async(ID)`.

The latency of the system is T which is the period of the slowest block as shown in Equation.3.1:

$$T = max(T_C, T_P, T_B) \tag{3.1}$$

The throughput of the system is then $1/T$. By making T_P and T_B less than T_C (defined in Fig.3.1). The classification throughput will be very close to the camera FPS allowing real-time classification of the captured frames.

3.3 Accuracy Measurement and Improvement

The accuracy of the system depends on many aspects. Starting from the neural network itself which plays the main role in the system accuracy. The BNN architecture is based on [8]. However, the network is clocked with different frequencies comparing to their design, and the input images of the network are generated from a streaming camera, which degrades the accuracy of the system. Another factor that affects the accuracy is the processing functions in the Process block. The interesting regions from the captured images are extracted with different sizes. Then the extracted regions are resized, and the pixels values are modified. The accuracy of the system is affected by the algorithms and parameters of these functions.

Improving the accuracy of the system can be controlled by choosing the optimal parameters for the Process block. The accuracy of the BNN itself does not change as the layer configuration is fixed. In addition, the accuracy can be improved by filtering the output results using windowing technique discussed in section .4.3. One problem related to accuracy measurement is the difficulty of identifying the correct classified frames from the camera streaming. To overcome this issue, an experiment setup with specific details are followed as discussed in section 4.4.

The work plan of this project can be summarised as shown in Fig.3.5. The starting point was the Inference block which is implementing the BNN network and testing it in terms of latency, power consumption, and accuracy for various clock frequencies. Then, the processing functions were implemented and optimized in terms of its latency. After that, a pipelining technique is implemented in order to maximize the system throughput. Finally, a set of experiments were conducted to evaluate the system performance and optimize its parameters.

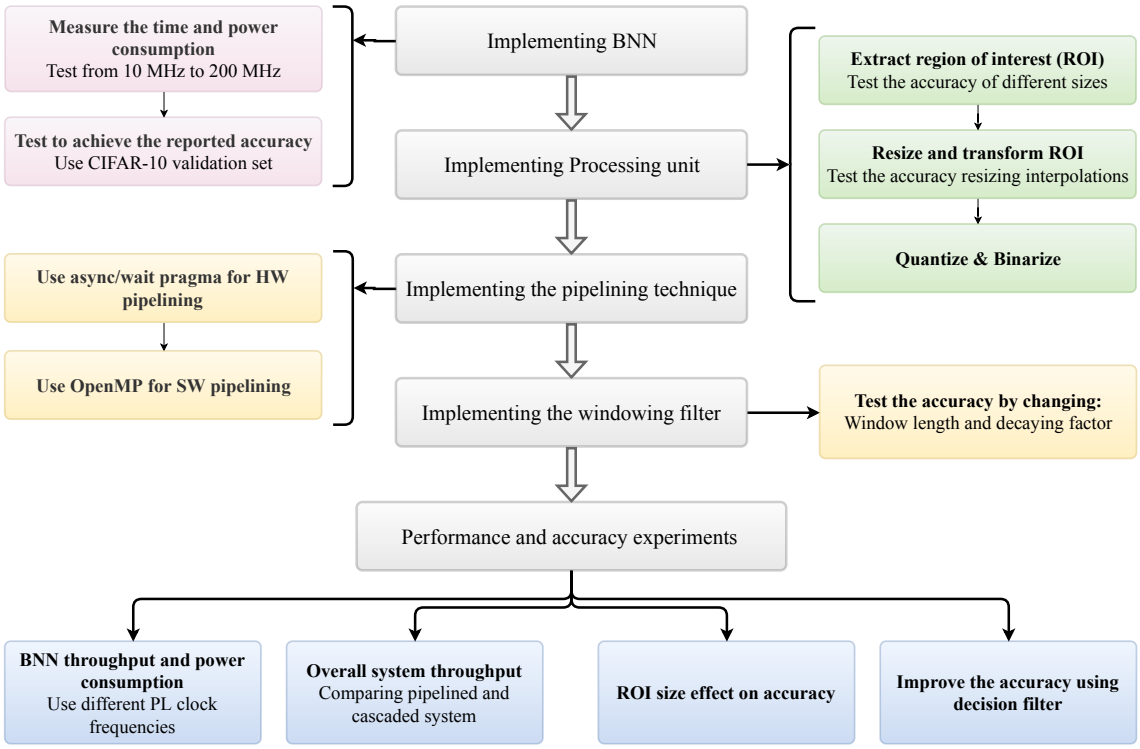


Figure 3.5: The work plan followed in the implementation and evaluation

Chapter 4

Implementation

This chapter discusses the implementation of the proposed system. Starting from synthesising BNN on the FPGA which is, as aforementioned, done in the Inference block. Following that, the implementation of the system Process block is discussed.

4.1 BNN configuration

The architecture of the BNN comprises of six Conv layers and three FC layers as shown in Table 4.1. The computing engine of the convolutional layers is the MVMU, which consists of an array of Processing Elements (P) represent the hardware synapses, and SIMD Lanes (S) per P representing the neurons as shown in Fig.4.1. The weight matrix of the synapses is stored in the OCM and multiplied with the input stream image. The number of P and S are configurable to control the throughput of the design. By making the number of P and S equal to the synapses and neurons respectively, a full parallelized design would be achieved capables of classifying images at the clock rate. However, the hardware resources of FPGA are limited which require time-multiplexing (folding) for the synapses and neurons of BNN. This approach controls the folding of matrix-vector product which can be controlled according to the required FPS. Folding directly affects the system area and power consumption at the expense of increasing classification latency.

Matrix-vector products can be folded by controlling the partitioning parameters which is the number of P and S of the MVTU. Multiplying $X \times Y$ matrix requires $X/P \times Y/S$ clock cycles to complete one multiplication. Where: X/P and Y/S represent neuron folding (F_n) and synapses folding (F_s), respectively. For a given FPS and FPGA memory resources, the parameters P and S should be calculated accordantly. The total fold can be obtained using Equation.4.1:

$$F = F_n \times F_s = \frac{f_{clk}}{FPS} \tag{4.1}$$

The throughput of the system is determined by the layer with the highest latency. By maximizing P and S, the maximum throughput can be achieved. In this design, the values of P and S are reported in [8] and described in Table.4.1. The design is built by using Vivado 2018.2 as shown in Fig.4.2. The BNN HLS IP core is generated first and exported as IP core. The synthesized report of BNN is shown in Table.4.2. It can be observed that

Table 4.1: Layers configuration of FINN model

Layers	IFM	OFM	PE (P)	SIMD (S)
3x3-conv-64	3	64	16	3
3x3-conv-64	64	64	32	32
Pooling 4x4	-	-	-	-
3x3-conv-128	64	128	16	32
3x3-conv-128	128	128	16	32
Pooling 4x4	-	-	-	-
3x3-conv-256	128	256	4	32
3x3-conv-256	256	256	1	32
FC-64	256	512	1	4
FC-64	512	512	1	8
FC-64 (no activation)	512	10	4	1

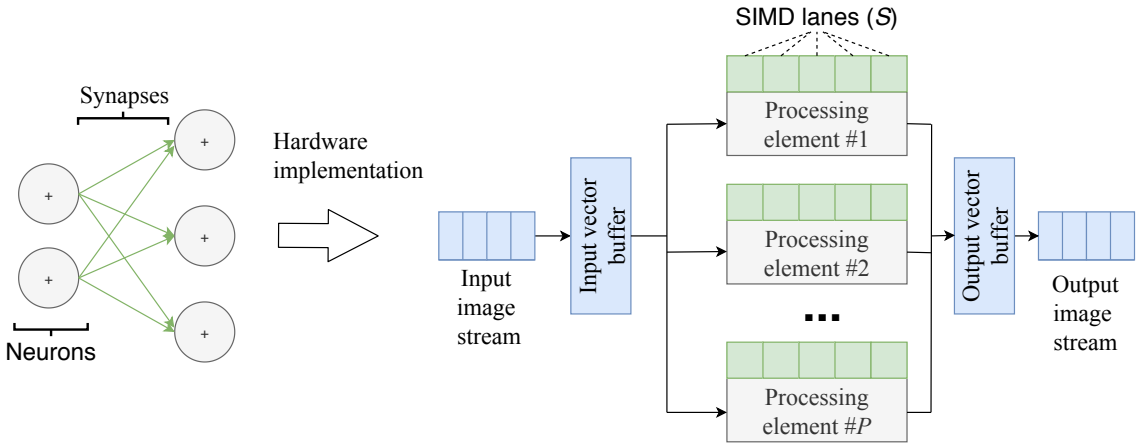


Figure 4.1: The MVTU block diagram, which is the processing unit of the Conv layer. Figure is regenerated from [8]

BNN consumes the majority of BRAMs making almost no room for other IP cores. The network is tested against a validation set of CIFAR-10 giving an accuracy of 78.5% with a classification speed of 430 image/s at FPGA clock of 100 MHz. More details about running the network on PYNQ-Linux environment and the testbench code for the results can be found in Appendix.A.

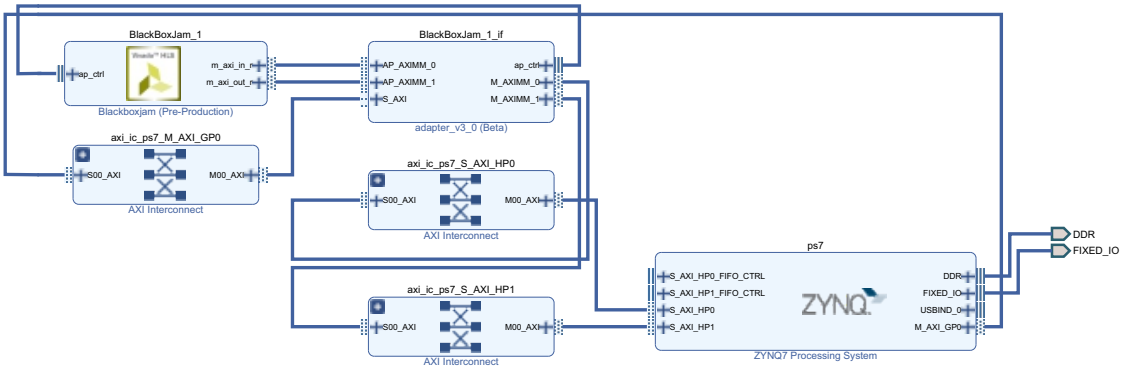


Figure 4.2: IP cores diagram of BNN using the default configuration

Table 4.2: BNN implementation utilization report on Zynq-7000: XC7Z020

Resource	Utilization	Available	Utilization %
LUT	25890	53200	48.67
LUTRAM	2361	17400	13.57
FF	36115	106400	33.94
BRAM	132	140	**94.29**
DSP	32	220	14.55
MMCM	1	4	25.00

4.1.1 Input/Output format

The input layer of BNN accepts special images data format. The Processing unit transforms the input image from RGB to the input format. Moreover, the input of BNN is an 8-bit unsigned integer number that represent the image pixels in the range [-1 to 1]. The input layer is the only one that accepts non-binary numbers by replacing XNOR-pop-count in the MVTU with a regular multiply-add.

The Final layer of BNN is an FC layer without the activation function to provide non-binarized output. Thus, the output is a vector that returns the confidence score of each CIFAR-10 class. The index with the highest score represent the most probable class, and the larger this number is, the more features are extracted from the image, therefore the higher the probability of that class.

The score class is converted to probabilities by using SoftMax as an activation function. SoftMax represent the output as a vector of probability distributions of a list of the potential outcomes by using Equation.4.2:

$$P_i = \frac{e^{X_i}}{e^{(\sum_{n=1}^{10} X_n)}} \tag{4.2}$$

Where X_i represents the output of i class. The output class range of BNN is between [300 to 400], so when it rises to exponent the output is infinity. Therefore, the output is normalized, then multiplied by 10 to make the score range between [1 to 10]. The function code is discussed in details in Appendix.B.

4.1.2 Control FPGA Clock frequency

The clock configuration of the PL is controlled by the Clocking Wizard IP. The clock source of PL is from Phase Locked Loop (PLL) I/O which provides 1000 MHz as an input of Clocking Wizard IP. Fig.4.3 shows the clock signals from the clock generator which generate four asynchronous PL clock signals to control different PL blocks.

The BNN IP is connected with CLK_OUT0 from the Clocking Wizard. The frequency of this signal can be controlled dynamically while the system is running by controlling the division factor of the output clock. The factors can be controlled through `FPGA0_CLK_CTRL` register to generate an output clock in the range between [10 to 200 MHz]. The register specifications are as follows:

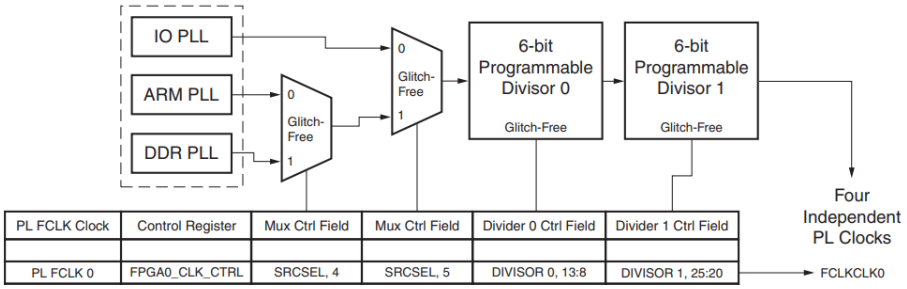


Figure 4.3: PL clock generation

- Description: PL Clock 0 Output control
- Absolute Address 0xF8000170
- Reset Value 0x00101800

The controlling bits are shown in Table.4.3. In addition, more details about the library used to map the hardware clock register address to software address and the used division factors can be found in Appendix.B.

Table 4.3: PL clock 0 control register

Name	bits range	Mode	reset value	description
DIVISOR1	25:20	rw	0x1	Generates the required clock frequency provided by the divisor by dividing the source clock. Second cascade divide
DIVISOR0	13:8	rw	0x18	Generates the required clock frequency provided by the divisor by dividing the source clock. First cascade divider.

4.2 Processing unit

The Processing unit converts the input image to BNN input form. The original input image is in RGB format where each pixel is presented by an 8-bit unsigned integer. The conversion involves the operations discussed bellow. The implemented code is discussed in details in Appendix.B.

4.2.1 Extract ROI and Resize functions

The camera capture frames with speeds of 160 FPS @ 320×240 resolution and 500 FPS @ 176×144. The overall system performance depends on the accuracy of the BNN as well as the extracted ROI. The extracted region in this project is fixed in the center of the frame with different sizes as shown in Fig.4.4.

The extracted region is then resized to 32×32 resolution, which is the resolution that is accepted by the BNN. Resizing the image is done by using built-in `OpenCV` function

`resize()`. The function resizing the frame based on different interpolation typologies. Three different interpolation methods are considered which are:

- INTER_LINEAR - a bilinear interpolation
- INTER_AREA - resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moire free results.
- INTER_CUBIC - a bicubic interpolation over 4x4 pixel neighborhood

The three interpolation typologies are evaluated by resizing a set of images and feeding them to the network. The results accuracy and the resizing latency is recorded as shown in Table.4.4.

Table 4.4: OpenCV resizing interpolation methods accuracy and latency results

Interpolation	**Accuracy**	**Latency** (μs)
INTER_LINEAR	82.6%	80
INTER_AREA	85.2%	330
INTER_CUBIC	83.1%	110

The INTER_AREA latency is around three times other typologies. While the accuracy of the three typologies are comparable. According to the table, INTER_CUBIC provide the best balance in terms of accuracy and resizing delay. Thus, it is used in the system.

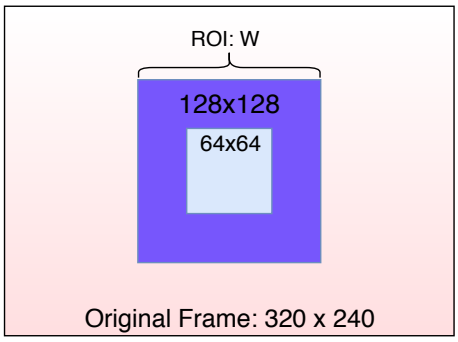


Figure 4.4: Extracting ROI from the original frame, where W is the width of the ROI

4.2.2 Reshape Vector and Transform Values

After resizing the required ROI, the output image is a 3D array of 32x32 size with three channels: red, green, and blue. This matrix needs to be reshaped as shown in Fig.4.5, where the matrix is converted to a 1D vector. The reshaping latency is 100 μs which is considered negligible, this is because the function just change the data location in the memory.

The initial pixel values in the input vector are 8-bit unsigned integers [0 to 255] and are normalized to values [-1 to 1] inclusive. The normalized values are of the float type which is 32-bit. The transformation function is shown in Equation.4.3; where $scale\_max = 1$, $scale\_min = -1$:

$$T(u) = scale\_min + (scale\_max - scale\_min) * u/255 : \tag{4.3}$$

The transformation consists of a simple division operation. Thus, its latency on the PS is negligible and does not need to be accelerated on FPGA.

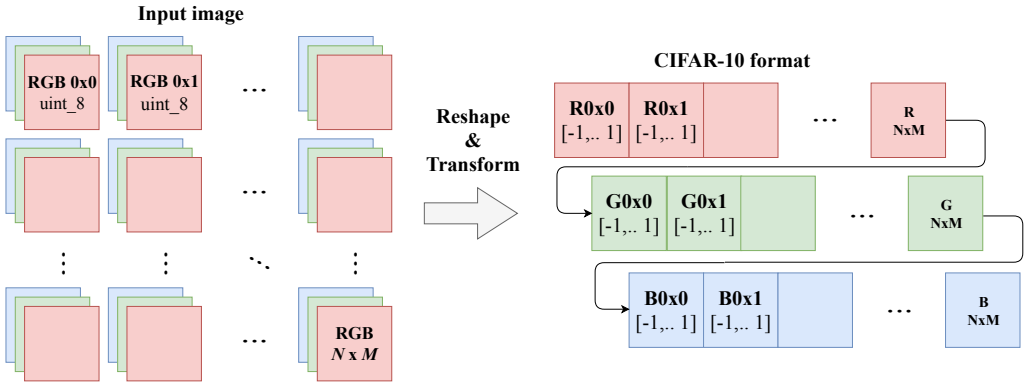


Figure 4.5: Image reshaping & values transforming: The input image is a 3D of size $H \times L \times D$ and the values of each pixel are an 8-bit unsigned integer. The CIFAR-10 format is 1D of size $1 \times (H \times L \times D)$ with floating values between -1 and 1.

4.2.3 Quantize and Binarize

The normalized vector requires to be converted from a float to binary format by quantizing the values. Quantizing is done by using the following SDSoC data type:

- **ap_fixed <W,I,Q,O>:**
  This data type bounded the float type by using a fixed width for the integer part for a specified total width as shown in Fig.4.6:

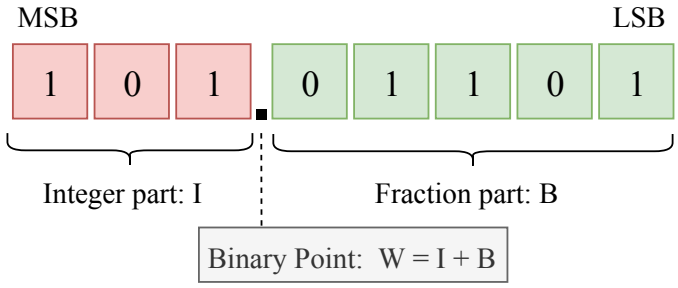


Figure 4.6: Quantizing the image values using SDSoC `ap_fixed`

Where:

  - W: Word length in bits
  - I: Number of bits used to represent the integer value (the number of bits above the decimal point)
  - Q: Quantization mode which dictates the behavior when greater precision is generated. This can be defined by the smallest fractional bit in the variable used to store the result. `AP_TRN`: Truncation to minus infinity
  - O: The number of saturation bits in wrap modes. `AP_SAT`: Saturation

The BNN requires an 8-bit width input, so the transformation is shown in Equation.4.4:

$$ap\_fixed < 8, 1, AP\_TRN, AP\_SAT > \tag{4.4}$$

The High-Level Synthesis ap_fixed type is used to define an 8-bit variable with 1-bit representing the numbers above the decimal point and 7-bits representing the value below the decimal point. The variable is specified as a signed integer. The specified quantization mode is truncated to a minus infinity, and the saturation mode is used for overflow.

- **ap_int<N>:**
  The input of BNN needs to be unsigned integer numbers. This is done by using $ap\_int < N >$ data type, where N is a bit-size from 1 to 1024. This type defines an arbitrary precision integer data, allowing to interpret the fixed value as an integer.

The latency of this block is 20us which is very small. However, each frame has $32 \times 32 \times 3 = 3072$ pixels, which requires 3072 calls. Therefore, this function is called exhaustively. After profiling the Process unit using `gprof` discussed in section 4.4, this function is proved to be the bottleneck of the Process block. However, it does not requires acceleration because its latency is still smaller than the capturing frames latency.

4.3 Classification Filter

Classifying frames from the camera fluctuates over multiple frames. The system would occasionally fail to classify the same view due to the noise introduced by the camera. Filtering the BNN output results would improve the accuracy and stabilize the output by using the windowing technique. This technique does not only depend on the current frame but also on the previews frames. The classification results over the current frame and a specific number of the previous frames are averaged to give the result of the current frame.

Windowing approach is shown in Fig.4.7 where the number of previous frames is the window length (L). The decision is based on the average of the weighted probabilities of N previous frames. The weights are exponentially decayed which makes the majority of decision dependent on the current frame. The decaying factor and window length are controlled to improve the accuracy. Increasing the window length improves the accuracy, however, it decreases the response for fast-changing objects. This is because the window needs to be filled with the new frames to offset the previous decision.

The previous N classifications are stored in an $N \times C$ matrix, where C is the number of classes. In the case of CIFAR-10 C is 10. The current class probabilities are weighted according to the decaying constant λ as shown in Equation.4.5:

$$w(N) = e^{-\lambda N} \tag{4.5}$$

The $row(i)$ from $N \times C$ matrix is multiplied with $W(i)$. Then the rows are averaged to generate the filtered probabilities vector of 10 classes over L frames. The output decision is the maximum class probability as shown in the Equation.4.6:

$$output = max[\sum_{N=0}^{L-1} row(i) \times w(i)] \tag{4.6}$$

The implementation is very fast in comparison to the other computational bottlenecks of the system, so it will not show any advantage in being accelerated on the FPGA, even if the implementation is fully parallelized.

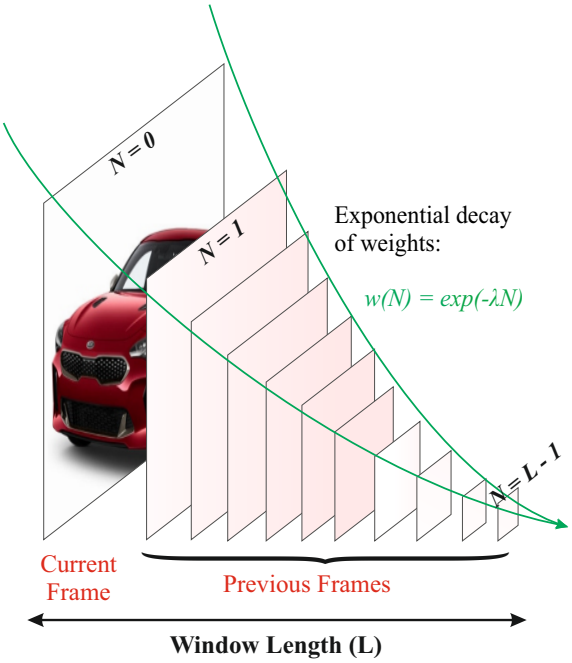


Figure 4.7: BNN decision filter using windowing technique.

4.4 Experiments Setup

The next phase of the project is to evaluate the performance in terms of throughput, power, and accuracy. A set of experiments and measurements are conducted to evaluate each parameter of the system.

Starting by measuring the latency of each function in the system. This is done by profiling the programme using Linux `gprof`, which provides information about the number of calls and the latency of each function. Using this information, slow functions can be identified and optimized. More accurate measurements are done manually by using `chrono` C++ library to use a high-resolution clock. A timer is set after each function and the difference between them gives the latency in μs resolution.

Measuring the power consumption of the FPGA is not possible in the Zedboard because it does not provide access to the internal power rails of the FPGA. Therefore, Vivado implementation report is used to estimate the power consumption with changing PL clock frequency.

The accuracy measurements are conducted for different ROI sizes and filter parameters. The system setup is shown in Fig.4.8, where the camera captures a known object such as a car. The system then runs for a specified number of frames X and the accuracy can be specified by calculating the percentage of the frames which the system can identify as a car from the total X frames i.e. 10000 frames. The object is placed in front of the camera such that is fit in the extracted ROI size. The orientation of the object is also fixed to evaluate only the effect of the region size. This experiment is done multiple times with different objects and the average of their results is reported. The filter parameters evaluation is done in a similar way, but the classification is done for the original frame without extracting ROI in order to evaluate only the filter effects.

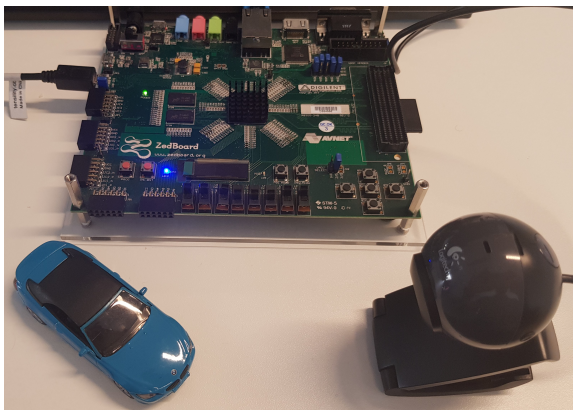


Figure 4.8: Accuracy experiment using small objects

Chapter 5

Evaluation

This chapter presents the designed system performance in terms of throughput, power consumption and accuracy. The evaluation starts by analysing the throughput and the power of the system. Following this, an experimental stage is conducted in order to obtain the optimized parameters that results in improving the overall accuracy.

5.1 Throughput and Power Evaluation

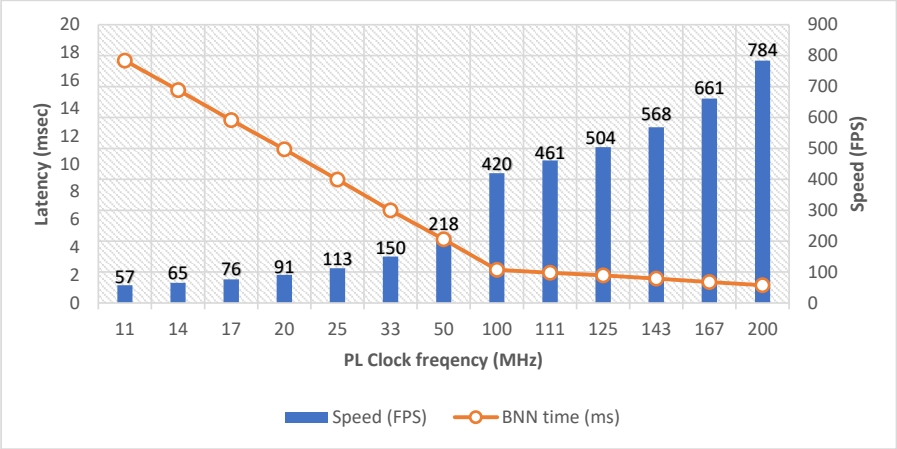
As aforementioned in section 3.2, the throughput of the system is defined as the number of frames that the system can classify per second. Achieving the maximum throughput can be done by pipelining the system functions. The latency of each block is measured, and the results are reported in Table.5.1, where the two camera resolutions are considered. The camera capture-latency increases with higher resolution, whereas the Process functions latency remains constant. This is because the extracted ROI is resized first to 32x32 pixels before processing it, making it *resolution-independent*.

Table 5.1: The latency of each block in the system

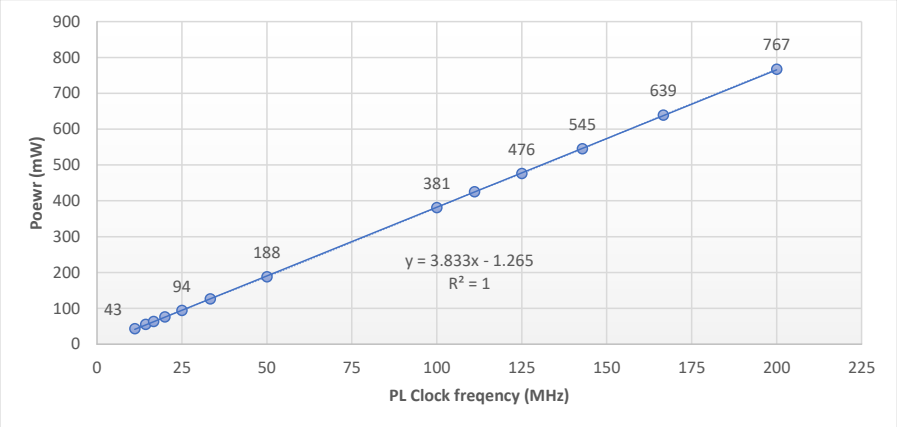
System Function	**Resolution**				**Note**
	320×240		176×144		
Capture	$6.25ms$	$160FPS$	$2.00ms$	$500FPS$	Depends on Resolution
Process	$1.54ms$	$650FPS$	$1.54ms$	$650FPS$	Constant
Inference	$2.22ms$	$450FPS$	$2.22ms$	$450FPS$	@ 100 MHz PL clock

The maximum capturing speed is 500 FPS, which should be the lowest part of the system. The Process unit achieved a higher speed of 650 FPS, which satisfies pipelining constraint. The inference speed is fixed for specific PL clock frequencies. Increasing the clock frequency decreases the inference latency though it increases the power consumption of the system. The inference speed and latency are measured for different PL clock frequencies as shown in Fig.5.1(a), and the power consumption is simulated on Vivado 2018.2 as shown in Fig.5.1(b).

Fig.5.1(a) shows that the classification speed is almost doubled when doubling the clock frequency of the PL. The frequency range that can be controlled is between 10 and 200



(a) Inference latency



(b) Inference power consumption

Figure 5.1: Classification throughput (a) and power consumption (b) of BNN for difference clock frequencies.

MHz, which makes the BNN speed range between 57 up to around 800 FPS. Achieving the maximum throughput for the pipelined system can be done by satisfying the following condition:

$$\text{BNN latency } (T_B) < \text{camera latency } (T_C) \tag{5.1}$$

Controlling the latency is done by setting the division factors in the PLL register as discussed in section 4.1.2. Choosing the highest frequency will always satisfy the condition, however, it will also consume the maximum power as shown in Fig.5.1(b). Thus, the frequency is chosen just above the condition imposed by Equation. 5.1 to achieve the maximum throughput while consuming the minimum amount of power. The simulated power of the clock frequency, in the range 10 to 200 MHz, can be estimated using the linear model shown in Equation.5.2:

$$Power = 3.833 \times freq - 1.265 \tag{5.2}$$

This shows that the power consumption is also roughly doubled by doubling the clock

frequency.

The overall performance of the pipelined system is shown in Table.5.2, which shows that the throughput is limited by the camera capturing speed. The maximum throughput is 500 FPS for 144p and 160 FPS for 240p. The cascaded system is the serial execution of the system, which is used to compare the throughput with the pipelined one. The results show that the throughput is enhanced by 2.66 and 4.44 for 144p and 240p resolution respectively. The speed of the cascaded system can be increased by clocking the PL with the maximum clock frequency of 200 MHz leading to 209 FPS and 110 FPS for 144p and 240p respectively. Although the cascade system throughput is improved, the power consumption of the FPGA increased to $0.767W$ comparing to $0.545W$ and $0.188W$ for the pipelined system.

Table 5.2: Throughput results of the pipelined system comparing to the cascading one

Camera Res.	**Capture**	**Process**	**Inference**		**Power**	**Throughput**	
			PL Clock	*Speed*		*Cascade*	*Pipeline*
176x144	500 FPS	650 FPS	144 MHz	560 FPS	0.545 W	188 FPS	486 FPS
320x240	160 FPS	650 FPS	50 MHz	218 FPS	0.188 W	36 FPS	158 FPS

The energy consumed by the system is given by the equation:

$$E = P \times t \tag{5.3}$$

Equasion 5.3 shows that the energy is proportional to both power consumption and latency. The system parameters can be controlled according to the required power consumption and latency as shown in Fig.5.2. The figure shows the latency-power relationship for different clock frequency.

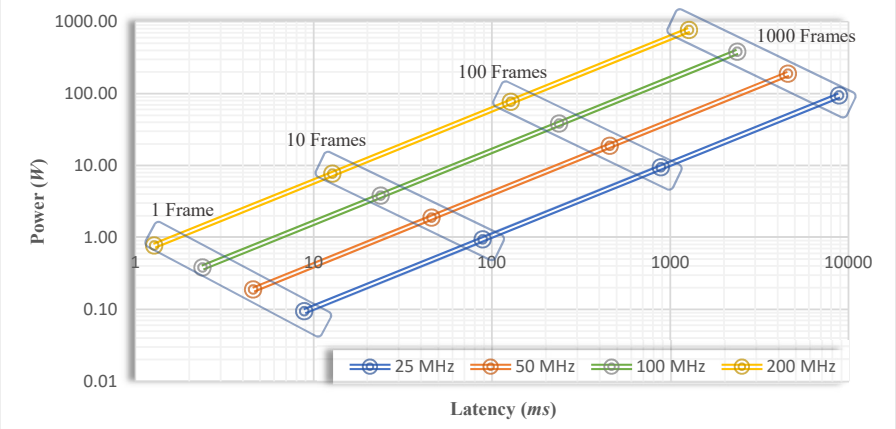


Figure 5.2: Latency-Power diagram for different PL clock frequency

5.2 Accuracy Parameters Evaluation

The system accuracy is the second consideration in this project. As discussed in section 3.2, the accuracy of the BNN itself is 78.5% for the validation set of CIFAR-10. However,

the accuracy of real-time streaming is much lower because of other factors that affect the accuracy such as the orientation of the object, its position in the extracted ROI, and the noise introduced from the camera. This section evaluates the effects of the extracted ROI size and the filter parameters discussed in section 4.3. The aspects that affect accuracy are studied in the following two experiments:

- **ROI Size**: to study the size of ROI (W) discussed in Fig.4.4
- **Filter parameters**: to study the decaying factor (λ) and window length (L) discussed in Fig.4.7

5.2.1 ROI Size

The goal of this experiment is to examine the accuracy improvement with changing ROI sizes. The sizes are chosen to be a factor of 32 for a better resizing results while processing. The accuracy in this experiment is measured without a filter because only the ROI width (W) is examined. The setup of this experiments is discussed in section 4.4 where the object is placed in a fixed distance from the camera with a fixed orientation. Then the system captured 10000 frames for various classes, where each class is measured separately. The accuracy of the classes is then calculated by taking the percentage of correctly classified frames from the 10000 frames. The reported accuracy is the average of all classes.

The accuracy is shown in Fig.5.3 where the original frame is the full-size frame. Generally, choosing a larger ROI can still classify the object but with a lower accuracy comparing to an appropriate sized rectangle. As shown in Fig.5.3, the 128 × 128 rectangle size provides the maximum accuracy of 81% for 240p resolution, showing an improvement by 5% comparing to the original frame. On the other hand, 64 × 64 rectangle size provide shows a 14% improvement comparing to the original frame for 144p, where the accuracy is maximized at 69%.

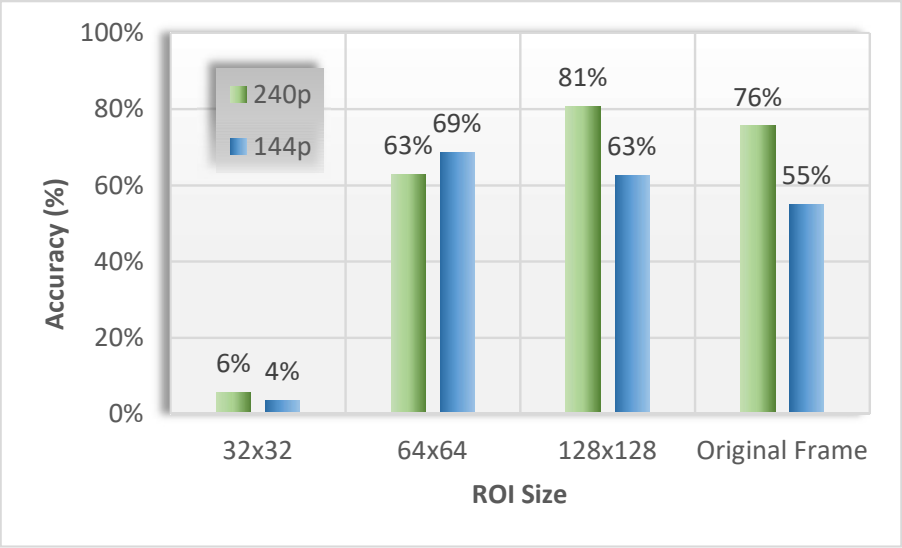


Figure 5.3: The effect of ROI size on accuracy

The ROI size effects on accuracy can be interpreted by the extracted features from the

object in the frame. The dominant features of the object is the edges. If the edges are not extracted within the ROI, the network would not classify the object correctly. Therefore, the extracted ROI should cover the entire object, and minimize the outside boundaries. If the extracted ROI are too small relative to the object, the accuracy performance of the overall system will be significantly degraded as shown in 32×32 size.

5.2.2 Filter parameters

Finding the best decaying parameter and window length is evaluated by classifying the entire frame without extracting ROI when the camera resolution is set to 320×240. The filter should smooth out the decision fluctuation of the output. Thus, the view is held constant to eliminate the effects of object orientation and the noise introduced by the moving objects. The experiment setup is similar to the extracting ROI, where 10000 frames is classified for various classes. The measurements are conducted for the window length $L = 4, 8, 16, 32, 48, 64$ and decaying factor $\lambda = 0.1, 0.2, 0.4, 0.6$.

The measurements accuracy without a filter was in the range of (65% to 75%). After applying the filter, the accuracy improved up to 97% as shown in Fig.5.4. The optimal parameters for this experiment is $L = 16$ and $\lambda = 0.4$. The accuracy improvement with the filter is around 20% depending on the chosen parameters. Moreover, the accuracy of the system with a large decaying factor of 0.6 shows a slight improvements of around 5% comparing to the accuracy without the filter. This is because increasing the decaying factor will decrease the dependency of the previous frames making the majority of the decision depends on the current frame. Thus, the fluctuation of the current frame will not be totally filtered out. On the other hand, decreasing the decaying factor to less than 0.1 makes the decision mainly depends on the previous frames, which will affect the speed of updating the decision of the new frame.

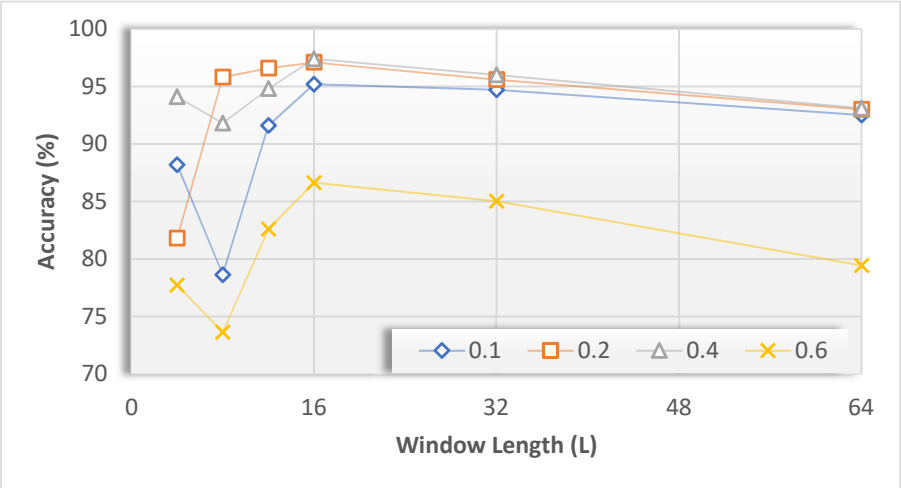


Figure 5.4: Filter length and decaying factor effects on accuracy

The time required to update the window output depends on the FPS of the system as

well as the window length as shown in the equation below:

$$t_{window} = N/FPS \tag{5.4}$$

This window time should be minimized in order to improve the system responsiveness to fast-changing objects. This can be achieved by decreasing the window length or increasing the FPS of the system. In fact, the accuracy of the system is lower for a smaller window though the system response is faster due to the less time requires to offset the previous decision. Finally, choosing the window size for moving objects depends on the object speed. Prior information and further processing of the frames such as frame subtraction and object localization are required to determine the optimal window length of the filter.

5.3 Optimal System Parameters

To summarise, the overall system is configured with the optimal parameters. For the camera a resolution of 320×240 is chosen, where the ROI size is set to 128×128 and the filter parameters are $L = 16$ and $\lambda = 0.4$. Fig.5.5 shows the output on the host screen. For this setup, the system accuracy is tested while applying movements and rotation to the object to evaluate the actual accuracy with noise and movement inputs. Overall, the throughput of the system is equal to the camera speed which is 160 FPS, where the PL clock is configured to 50 MHz consuming 0.188 W. Moreover, the overall accuracy is 81.2% comparing to the accuracy without the filter which is 68.5%.

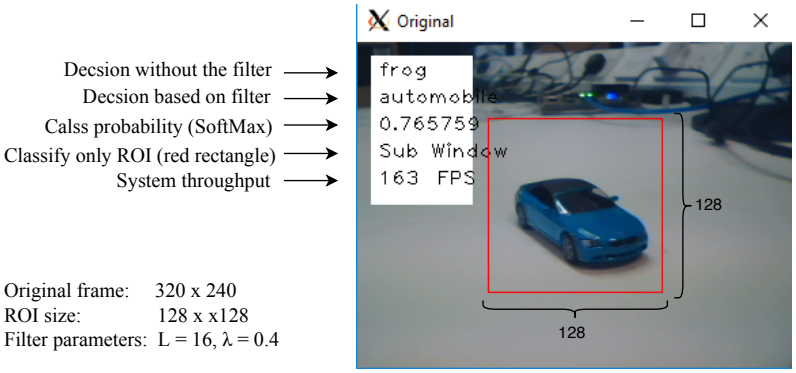


Figure 5.5: Example of the output screen on the host computer

The evaluation results of the system is difficult to be compared with other literature designs. This is because most designs throughout literature evaluate the accuracy of the BNN only and do not take into account streaming frames from a camera. As result, factors that affect the accuracy such as noise are not considered in their accuracy evaluation. A reasonable comparison in terms of accuracy is shown in Table.2.2. The FINN [8] accuracy is 80.1%. Since the design is based on FINN implementation. The accuracy of the system is limited to the BNN accuracy, where the achieved accuracy in this project is slightly higher, which shows that the accuracy of this design is almost saturated. The power consumption of this design is 0.181 W, comparing to 2.3 W in [8], providing a system that is 12 times more power efficient.

Chapter 6

Conclusion and Future Work

In conclusion, BNNs have the potential to provide an energy-efficient image classifications for hardware platforms. They can by fully implemented on logic fabric and stored on OCM. This will enhance the system speed and power consumption, which are considered the key design concepts for this project. The level of abstraction used in this design was HLS, the configurability achieved with HLS-based designs not only simplifies and automates the development process, but also helps to investigate the design space and to implement functions more efficiently on FPGA.

For the implemented system, the design depended on [8] model due to the presented reasons that demonstrate its capability of achieving a decent accuracy of 78.5% on CIFAR-10 and high-power efficiency with a throughput up to 800 FPS, which is sufficient for real-time classification.

The overall implementation achieved classification throughput was close to the camera speed which is 500 FPS. This demonstrated the advantage of using SW/HW pipelining. Moreover, it allowed clocking the FPGA with lower clock frequency which showed an improvement in terms of power consumption. The accuracy of the system has improved significantly up to 81.2% comparing to 68% showing the advantage of using the windowing technique. The reported accuracy in this project is comparable with full-precision neural networks making BNN is a promising solution for high performance and low power classification systems.

6.1 Future Work

Many aspects in this project could be extended and studied further. The following future work give ideas to improve the performance and robustness of the system:

- Using adaptive ROI extraction methods to localize the object in the image and allow classifying multiple objects in the same frame.

- Use object localization to get information about the scene. This information can be used to improve the power consumption of the system, i.e. identify the object speed to adapt the classification speed. This would permit to use a lower clock frequency

and classify fewer frames for the slow object to preserve more power while maintaining the accuracy. In contrast, higher clock frequency should be used for fast-objects.

- Implementing Process functions on FPGA would improve the throughput, but this required larger FPGA area, or a less complex BNN by sacrificing the network accuracy.
- Training the system and extend it for larger datasets such as CIFAR-100 and ImageNet.

Bibliography

[1] Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read, 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[4] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of FPGA based neural network accelerator. *CoRR*, abs/1712.08934, 2017.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[6] Seyyed Hossein HasanPour, Mohammad Rouhani, Mohsen Fayyaz, Mohammad Sabokrou, and Ehsan Adeli. Towards principled design of deep convolutional networks: Introducing simpnet. *CoRR*, abs/1802.06205, 2018.

[7] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang. Accelerating low bit-width convolutional neural networks with embedded FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017.

[8] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[9] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A 7.663-tops 8.2-w energy-efficient FPGA accelerator for binary convolutional neural networks. *CoRR*, abs/1702.06392, 2017.

[10] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. *CoRR*, abs/1807.08596, 2018.

[11] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 16–25, New York, NY, USA, 2016. ACM.

[12] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[13] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[17] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017.

[18] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[19] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 15–24, New York, NY, USA, 2017. ACM.

[20] Sparsh Mittal. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Computing and Applications*, Oct 2018.

[21] A. Shawahna, S. M. Sait, and A. El-Maleh. FPGA-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[22] H. Yonekawa and H. Nakahara. On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 98–105, May 2017.

[23] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. Scaling binarized neural networks on reconfigurable logic. *CoRR*, abs/1701.03400, 2017.

[24] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. Resbinnet: Residual binary neural network. *CoRR*, abs/1711.01243, 2017.

[25] H. Nakahara, T. Fujii, and S. Sato. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017.

[26] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong. High performance binary neural networks on the Xeon+FPGA platform. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017.

[27] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[29] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 10 2006.

[30] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015.

[31] E. Nurvitadhi, D. Sheffield, , A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, Dec 2016.

[32] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072 – 1086, 2018.

[33] A. Mazare, L. Ionescu, A. Lita, and G. Serban. Real time system for image acquisition and pattern recognition using boolean neural network. In *2015 38th International Spring Seminar on Electronics Technology (ISSE)*, pages 292–295, May 2015.

[34] P. Jokic, S. Emery, and L. Benini. Binaryeye: A 20 kfps streaming camera system on fpga with real-time on-device image recognition using binary neural networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–7, June 2018.

[35] Thomas B. Preußer, Giulio Gambardella, Nicholas J. Fraser, and Michaela Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. *CoRR*, abs/1806.08085, 2018.

[36] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[37] Jonathan Pedoeem and Rachel Huang. YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers. *CoRR*, abs/1811.05588, 2018.

[38] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[39] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[40] Joseph Redmon. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productadvantages. `http://pjreddie.com/darknet/`, 2013–2016.

[41] Joseph Redmon. P. parikh, pynq linux on zedboard, https://superuser.blog/pynq-linux-onzedboard/, 2017. `http://pjreddie.com/darknet/`, 2013–2016.

Appendix A

Run and Test BNN for CIFAR-10 Validation Set

This appendix explains the code used to map the BNN to FPGA on PYNQ-Linux OS. Then the testbench used to test the network inference against the validation set of CIFAR-10 is listed.

The BNN is already synthesised and the bitstream file is generated on the host using SDSoC. To run the network on the Zedboard, the initial step is to program the FPGA by loading the bitstream file `kernelbnn.linked.bit.bin` to the PL by using the bash command:

```
$ echo kernelbnn.linked.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

The next step is to initialize the network layers and load their parameters. The function `load_parameters` loads the configuration of each layer including Processing Elements (P), SIMD Lanes (S), the wights and threshold values as shown in List.A.1.

```
extern "C" void load_parameters(const char* path)
{
#include "config.h"
FoldedMVInit("cnv-pynq");
network<mse, adagrad> nn;
makeNetwork(nn);
        cout << "Setting network weights and thresholds in accelerator..."
    << endl;
        FoldedMVLoadLayerMem(path , 0, L0_PE, L0_WMEM, L0_TMEM);
        FoldedMVLoadLayerMem(path , 1, L1_PE, L1_WMEM, L1_TMEM);
        FoldedMVLoadLayerMem(path , 2, L2_PE, L2_WMEM, L2_TMEM);
        FoldedMVLoadLayerMem(path , 3, L3_PE, L3_WMEM, L3_TMEM);
        FoldedMVLoadLayerMem(path , 4, L4_PE, L4_WMEM, L4_TMEM);
        FoldedMVLoadLayerMem(path , 5, L5_PE, L5_WMEM, L5_TMEM);
        FoldedMVLoadLayerMem(path , 6, L6_PE, L6_WMEM, L6_TMEM);
        FoldedMVLoadLayerMem(path , 7, L7_PE, L7_WMEM, L7_TMEM);
        FoldedMVLoadLayerMem(path , 8, L8_PE, L8_WMEM, L8_TMEM);
}
```

Listing A.1: Load BNN parameters and values

The network is ready to used for inference. Thus, the function `inference_test` test the given dataset as shwon in List.A.2.

```
extern "C" unsigned int inference_test(const char* path, unsigned int
    results[64], int number_class, float *usecPerImage,unsigned int img_num
    )
{
    FoldedMVInit("cnv-pynq");
    network<mse, adagrad> nn;
    makeNetwork(nn);
    std::vector<label_t> test_labels;
    std::vector<vec_t> test_images;
    parse_cifar10(path, &test_images, &test_labels, -1.0, 1.0, 0, 0);
    float usecPerImage_int;
    testPrebuiltCIFAR10<8, 16>(test_images, test_labels, number_class,
    img_num);
}
```

Listing A.2: BNN inference test function

The function calls a series of functions that Process the images before inputting them to the network. When the images are ready, the hardware function is called with the `kernelbnn` as shown in List.A.3.

```
// The HW function definition
int kernelbnn(
ap_uint<64> * in, ap_uint<64> * out, bool doInit,
unsigned int targetLayer, unsigned int targetMem,
unsigned int targetInd, ap_uint<64> val, unsigned int numReps,unsigned int
    psi, unsigned int pso);

// The function call from the testPrebuiltCIFAR10 function
auto t1 = chrono::high_resolution_clock::now(); // Start: time counter to
    calculate inference speed
kernelbnn((ap_uint<64> *)packedImages, (ap_uint<64> *)packedOut, false, 0,
    0, 0, 0, img_num,psi,16);
auto t2 = chrono::high_resolution_clock::now(); // End time counter
```

Listing A.3: BNN hardware function call

The main programme is shown in List.A.4, which call the functions discussed previously and print the output accuracy to `accuracy.txt`.

```
main(int argc, char *argv[])
{
  unsigned int results[64];
  int number_class = 10;
  float usecPerImage;
  printf("Hello BNN\n");
  myfile.open ("accuracy.txt",std::ios_base::app);

  if (argc < 4) {
          printf("not enough arguments 1: path to params 2: path to image
    data 3: image number to infer\n");
    exit(1);
```

```
    }

  unsigned int img_num = atoi(argv[3]);
  deinit();
  load_parameters(argv[1]); // argv[1] paramaters argv[2] images
  printf("Done loading BNN\n");

  ok = 0, failed = 0;
    inference_test(argv[2], results, number_class, &usecPerImage, img_num);
  cout << "Succeeded " << ok << " failed " << failed << " accuracy " <<
    100.0*((float)ok/(float)img_num) << "%" << endl;
  printf("Done infering BNN\n");
        float accuracy_value = 100.0*((float)ok/(float)img_num);
        myfile << accuracy_value << ",";
        myfile.close();
}
```

Listing A.4: The `main` function of testing BNN

The programme is compiled directly using the `makefile`, then the generated application can be run using the command:

```
  $ ./BNN ./params/cifar10/ ./test/test_batch.bin 1000
```

where the first input is the location of binary files that have the trained values of the network. The second input is the source of the dataset file, which includes array of 8-bit unsigned integers of 10000 labeled images. The third input is the number of requested images from the file. The output is the accuracy of the input images, where each image is classified and compared with its label.

The inference latency and FPS of the network is measured as shown in List.A.5. The latency is calculated by taking the difference between $t1$ adn $t2$ in List.A.3.

```
auto duration = chrono::duration_cast<chrono::microseconds>( t2 - t1 ).
    count();
usecPerImage = (float)duration / (count);
cout << "Inference took " << duration << " microseconds, " << usecPerImage
    << " usec per image" << endl;
cout << "Classification rate: " << 1000000.0 / usecPerImage << " images per
     second" << endl;
```

Listing A.5: Calculate BNN inference latency

Appendix B

System Code Implementation

This appendix includes the code of the main functions of the system. The system has three blocks mainly: Capture, Process, and Inference. The code of each block and its related functions are discussed.

Initially, the programme sets the FPGA clock register discussed in section 4.1.2 as shown in List.B.1. The code maps the hardware register address to software address can be used by the OS. The register value is an application input which should be in HEX format as shown in Table.B.1. The input value is saved to `clk_reg_value` variable.

```
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define HW_ADDR_GPIO 0xF8000170 // absolute address: 0xF8000170
#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

//configuration of PL clocks
cout << "Starting PL clock configuration: " << endl;
int memfd;
void *mapped_base, *mapped_dev_base;
off_t dev_base = HW_ADDR_GPIO; //GPIO hardware

memfd = open("/dev/mem", O_RDWR | O_SYNC);
if (memfd == -1) {
  printf("Can't open /dev/mem.\n");
  exit(0);
}
printf("/dev/mem opened for gpio.\n");
// Map one page of memory into user space such that the device is in that
    page, but it may not
// be at the start of the page.
mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
    dev_base & ~MAP_MASK);
if (mapped_base == (void *) -1) {
```

```
  printf("Can't map the memory to user space.\n");
  exit(0);
}
printf("GPIO mapped at address %p.\n", mapped_base);
// get the address of the device in user space which will be an offset from
    the base
// that was mapped as memory is mapped at the start of a page
mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
int* pl_clk = (int*)mapped_dev_base;
cout << "Current PL clock configuration: " << hex << *pl_clk << endl;
clk_reg_value = strtol(argv[4], NULL, 16); // atoi(argv[4]);
*pl_clk = clk_reg_value;
```

Listing B.1: Setting FPGA clock register to control the inference latency

Table B.1: BNN clock register configuration

Register Value (HEX)	Output Clock (MHz)	Register Value (HEX)	Output Clock (MHz)
0x00A00900	11.1	0x00A00100	100.0
0x00A00800	12.5	0x00100900	111.1
0x00A00500	20.0	0x00100800	125.0
0x00A00400	25.0	0x00100700	142.9
0x00A00300	33.3	0x00100600	166.7
0x00A00200	50.0	0x00100500	200.0

The next step is to map the BNN to FPGA and load its parameters as discussed in Appendix.A. The camera then initialized and set to the required capture resolution as shown in List.B.2.

```
#define FRAME_WIDTH 320   //176 //320 //640
#define FRAME_HEIGHT 240  //144 //240 //480

// Open defult camera
VideoCapture cap(0 + CV_CAP_V4L2);
if(!cap.open(0))
{
   cout << "cannot open camera" << endl;
   return 0;
}
cap.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
cap.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);
```

Listing B.2: Initializing the camera

The Capture and Process blocks are based on `OpenCV` library and executed in parallel using `OpenMP` library to increase the throughput as discussed in section 3.2. The Process functions are explained in section 4.2. The code is shown in List.B.3.

```
// Capture the first frame to fill the first pipeline block
if (frameNum == 0)
{
  cap >> curFrame;
}
// fork the program
```

```
#pragma omp parallel sections
{   // Execute the Capture block on P0
  #pragma omp section
  {
    cap >> capFrame;
    curFrame = capFrame;
  }
  // Executte the Process block on P1
  #pragma omp section
  {
    if(smallWindow == 0)
    {   // Process the total frame
      cv::resize(curFrame, reducedSizedFrame, cv::Size(32, 32), 0, 0, cv::
    INTER_CUBIC );
      flatten_mat(reducedSizedFrame, bgr);
      vec_t img;
      std::transform(bgr.begin(), bgr.end(), std::back_inserter(img),
        [=](unsigned char c) { return scale_min + (scale_max - scale_min) *
     c / 255; });
      quantiseAndPack<8, 1>(img, &packedImages[0], psi);
    }
    else
    { // Process only the required ROI
      Rect R(Point((FRAME_WIDTH/2)-(WINDOW_WIDTH/2), (FRAME_HEIGHT/2)-(
    WINDOW_HEIGHT/2)), Point((FRAME_WIDTH/2)+(WINDOW_WIDTH/2), (
    FRAME_HEIGHT/2)+(WINDOW_HEIGHT/2)));
      Mat bnnInput;
      bnnInput = curFrame(R); //Extract ROI

      cv::resize(bnnInput, reducedSizedFrame, cv::Size(32, 32), 0, 0, cv::
    INTER_CUBIC );
      flatten_mat(reducedSizedFrame, bgr);
      vec_t img;
      std::transform(bgr.begin(), bgr.end(), std::back_inserter(img),
        [=](unsigned char c) { return scale_min + (scale_max - scale_min) *
     c / 255; });
      quantiseAndPack<8, 1>(img, &packedImages[0], psi);
    }
  }
} // The program joint again here
```

Listing B.3: Capture and Process blocks code

The next block is the Inference. The async/wait pragma are used to run the hardware function in the background as discussed in section 3.2. The pragmas are compiled with the hardware function code as shown in the comment in List.B.4.

```
// Call the hardware function with async only
kernelbnn((ap_uint<64> *)packedImages, (ap_uint<64> *)packedOut, false, 0,
    0, 0, 0, count,psi,pso,1,0);
if (frameNum != 1)
{   // Call the hardware function with wait pragma first, then async is
    execcuted
```

```
  kernelbnn((ap_uint<64> *)packedImages, (ap_uint<64> *)packedOut, false,
    0, 0, 0, 0, count,psi,pso,0,1);
}
```

Listing B.4: Call the BNN and run it in the background

The output of BNN is copied to a buffer and the probability of each class is calculated for each frame using the SoftMax function discussed in section 4.2 as shown in List.B.5.

```
template<typename T>
vector<float> calculate_certainty(std::vector<T> &vec, int certainty_spread
    )
{
  // Normalise the vector
  std::vector<float> norm_vec = normalise(vec);

  float sum = 0;
  for(auto const &elem : norm_vec)
    sum += exp(elem*10);
  if(sum == 0){
    std::cout << "Division by zero, sum = 0" << std::endl;
    return -1;
  }
  for(int i=0; i<10;i++)
  {
    norm_vec[i] = exp(norm_vec[i]*10) / sum;
  }
  return norm_vec;
}
```

Listing B.5: SoftMax function to calculate the classes probabilities

The decision is based on the windowing filter discussed in section 4.3. The probabilities of N frames are stored, then the output of the current frame is the maximum number of the averaged previous classes as shown in List.B.6.

```
// Calculate classes probabilities based on SoftMax function
certainty = calculate_certainty(class_result, certainty_spread);
// Save the probabilities of N (windowLength) classes
resultsHistory.insert(resultsHistory.begin(), certainty);
if (resultsHistory.size() > windowLength)
{
  resultsHistory.pop_back();
}
// The current output is the maximum index of the weighted sum of current
    and previous outputs
std::vector<float> adjustedResults(number_class, 0);
for(int i = 0; i < resultsHistory.size(); i++)
{ // From most recent frame to oldest
  for(int j = 0; j < number_class; j++)
  {
    adjustedResults[j] += (historyWeights[i] * resultsHistory[i][j]);
  }
}
```

```
int adjustedOutput = distance(adjustedResults.begin(),max_element(
    adjustedResults.begin(), adjustedResults.end()));
vector<float> propability = calculate_certainty(adjustedResults,
    certainty_spread);
float max_result = *max_element(std::begin(propability), std::end(
    propability));
cout << "adjustedOutput = " << classes[adjustedOutput] << endl;
```

Listing B.6: Decision filter based on the previous frames

The window decaying factors are calculated as shown in List.B.7.

```
// pre-calculate window length, the parameters can be as an input of the
  application
int windowLength = 16;
float lambda = 0.4;
std::vector<float> historyWeights;
std::vector<std::vector<float> > resultsHistory; // All its previous
  classifications
historyWeights.resize(windowLength);
// Pre-populate history weights with exponential decays
for(int i = 0; i < windowLength; i++)
{
  historyWeights[i] = expDecay(lambda, i);
}
std::cout << "history weights = "; print_vector(historyWeights); std::
  cout << std::endl;
```

Listing B.7: Window decaying factors

Finally, the application is compiled using `makefile`, and the generated output application is `BNN`. The application can be run using the command:

```
$ ./BNN <mode> <expected class> <number of frames> <PL clock control> <
  ROI size>
```

The input parameters are:

- *mode:*
  - hw: run the program with hardware pipelining only, used to test the SDSoC pragmas
  - sw: run the program with sw/hw pipelining
- *expected class:* label the expected objects to calculate the accuracy, the input is from 0 to 9 representing the objects as shown in Table.B.2.

Table B.2: CIFAR-10 Classes

Class	Input number	Class	Input number
airplane	0	dog	5
automobile	1	frog	6
bird	2	horse	7
cat	3	ship	8
deer	4	truck	9

- *number of frames:* the program will run for fixed number of frames and then automatically stop.
- *PL clock control:* is the division factors to control PL clock frequency, the factors used are listed in Table.B.1.
- *ROI size:* is the size of extracted ROI. If the input is 0, the total frames will be classified, otherwise the input is only one dimension of the ROI i.e. 128 pixels.

As an example, the code below run the fully pipelined program for 2000 frame and calculate the accuracy by for the automobile class. The PL frequency is 100 MHz, and the extected ROI is 128×128.

```
$ ./BNN sw 1 2000 00A00200 128
```

Fig.B.1 shows the output view on the host computer of running the previous code.

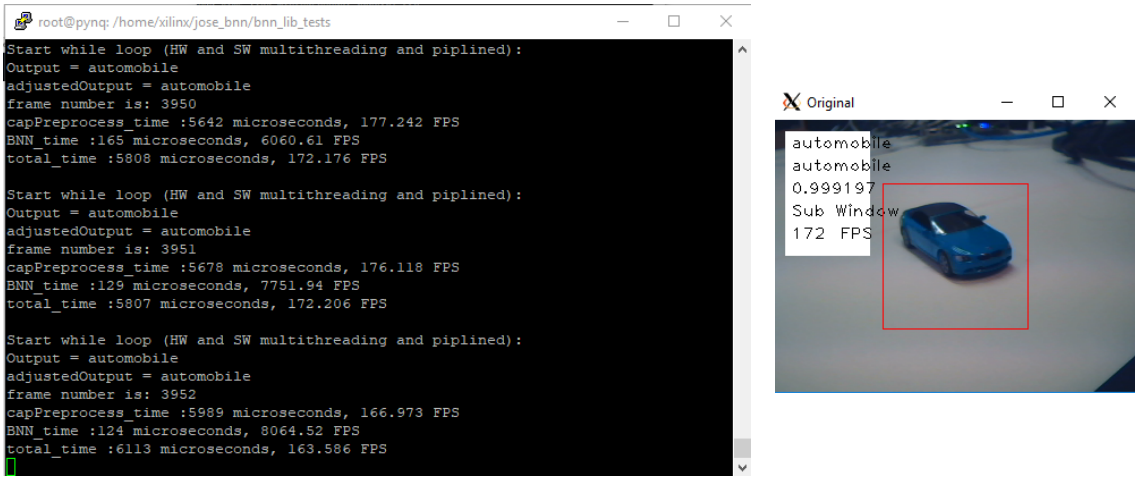


Figure B.1: Demonstration of the system output