REGION-BASED CONVOLUTIONAL NEURAL NETWORK AND IMPLEMENTATION OF THE NETWORK THROUGH ZEDBOARD ZYNQ

A Thesis

Submitted to the Faculty

of

Purdue University

by

Md Mahmudul Islam

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2019

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Lauren Christopher, Chair

Department of Electrical and Computer Engineering

Dr. Maher Rizkalla

Department of Electrical and Computer Engineering

Dr. Paul Salama

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

Head of the Graduate Program

To my parents, Mahmuda Khatun and Md Rezaul Islam

ACKNOWLEDGMENTS

I want to express my sincerest gratitude to my thesis advisor, Dr. Lauren Christopher, for allowing me to work on the topic and providing me the guidance required at every step. I am grateful to Dr. Maher Rizkalla and Dr. Paul Salama for serving on the thesis committee. I also want to thank all my lab-mates and especially to David Emerson for his collaboration.

Finally, I would like to express my earnest gratitude to my parents, my family and friends for all of the motivation and support throughout this Journey.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

ABBREVIATIONS

NN	Neural Network
CNN	Convolutional Neural Network
RCNN	Region-based Convolutional Neural Network
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HLS	High-Level Synthesis
GPU	Graphics Processing Unit
CPU	Central Processing Unit
ReLU	Rectified Linear Unit
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuits
HDL	Hardware Description Language
DSP	Digital Signal Processing
BRAM	Block Random Access Memory
LUT	Look-up Table
BSP	Board Support Package
FSBL	First Stage Boot Loader
SDK	Software Development Kit
GPU	Graphics Processing Unit
IP	Intellectual Property
SGDM	Stochastic Gradient Descent with Momentum

ABSTRACT

Islam, Md Mahmudul. M.S.E.C.E., Purdue University, May 2019. Region-based Convolutional Neural Network and Implementation of the Network Through Zedboard Zynq. Major Professor: Lauren Christopher.

In autonomous driving, medical diagnosis, unmanned vehicles and many other new technologies, the neural network and computer vision has become extremely popular and influential. In particular, for classifying objects, convolutional neural networks (CNN) is very efficient and accurate. One version is the Region-based CNN (RCNN). This is our selected network design for a new implementation in an FPGA. This network identifies stop signs in an image.

We successfully designed and trained an RCNN network in MATLAB and implemented it in the hardware to use in an embedded real-world application. The hardware implementation has been achieved with maximum FPGA utilization of 220 18k_BRAMS, 92 DSP48Es, 8156 FFS, 11010 LUTs with an on-chip power consumption of 2.235 Watts. The execution speed in FPGA is 0.31 ms vs. the MATLAB execution of 153 ms (on computer) and 46 ms (on GPU).

1. INTRODUCTION

The most robust visual processing system in the mammal is its' visual cortex [1]. This visual system was the initial inspiration for the creation of the current day Neural Network. In the coming age, autonomous systems (i.e., autonomous vehicles, animal robots or cooking assistant robots [2]) have an increased demand for machine vision. There are many variations of machine vision Neural Networks now available. One of the most common is the Convolutional Neural Network which provides more flexibility and improved accuracy for classification using the convolutional layer's parameter learning through this network's hierarchy [3]. For the automotive industry, it is becoming essential to implement these facilities within the embedded hardware for speed and cost.

1.1 Literature Review

In recent years, the demand for Computer Vision (CV) processes in Autonomous Vehicles (AV), Medical Diagnostics and Unmanned Aerial Vehicles (UAVs) have created much interest among researchers and scientists. This work is not an entirely new horizon for this type of task. Researchers have developed CNN for either the software level [4] or the FPGA hardware level, but only with one-channel grey scale images [5]. In this case, our new FPGA hardware implementation of the CNN needed the RGB images.

Before starting this thesis, we searched the web for papers or projects with hardware acceleration to understand the state of the art. The most relevant papers were CNN2ECST- a Xilinx Open Hardware Contest 2016 project [5] and ZynqNet Embedded CNN - a masters thesis report by David Gschwend [4]. ZynqNet was a highly efficient FPGA-based CNN acceleration exploration with 84.5 percent top-5-

accuracy [6]. The ZynqNet FPGA accelerator had been synthesized using high-level synthesis for the Xilinx Zynq XC-7Z045, reached 200 MHz clock frequency with a device utilization of 80 to 90 percent. However, this chip had many more resources needed compared to us. CNN2ECST, was designed by an Italian group, and similar to our goal. CNN2ECST is an CNN, used the same FPGA chip. However, It takes grey scale images and hand-drawn digits into ten classes 0,...,9. They used an USPS dataset [7] for training and testing their classification work. USPS dataset consists of 1100 images of each of the ten digits, with 1000 training images and 100 test images [8]. This project gave us an initial starting point for our work. Although it inspired us, our network design was different from them.

1.1.1 Neural Network and FPGA

Artificial Neural Network (ANN) implies a network that is based on the connections of the mammal's brain neurons implemented as a computer network [9]. Axons and Dendrites are two significant components of a neuron cell. These neurons get excited electrically. Axons perform the Neural coding depending upon the receiving signal through the Dendrites [10] and are connected to the next Neuron's Dendrites, repeating the procedure forms the Neural Network. Modern computer science invented a connection based system modeling the biological neurons as nodes [11] and performs various interactions in between network components [12]. It is done by proper usage of weights, biases and activation functions. Neural networks can perform tasks without being programmed precisely, and they improve performance through data learning. A neural network consists of several layers of nodes, like the neurons of the brain, and these are interconnected in layers. Input layers, output layers, and hidden layers are the main three layers. Nodes are determined by individual weights and biases and have a unique output. A defined activation function activates these outputs.

Field Programmable Gate Array (FPGA), on the other hand, are electrically programmable and re-programmable integrated circuits. The FPGA is composed of arrays of programmable logic blocks and four kinds of resource sharing elements. Its reconfigurability property makes it different from Application Specific Integrated Circuits (ASICs) which are not re-configurable. Previously only VHDL/Verilog was used to program and model FPGAs. However, now we can do operations in standard C/C++ language. Once programmed, a file to configure the function into the hardware, known as a bit-stream, is created and contains the resource and wiring information for the FPGA components. We created such a file for our CNN, and then we downloaded it into the FPGA board. Once we power on, our board gets configured and initiated to run according to the designed CNN function.

1.1.2 RCNN AND Matlab

RCNN stands for region-based Convolutional Neural Network. What the regular CNN does is that it captures little information through predefined sub-regions called the receptive fields within a fixed dimension image or region. Then in a later stage, this locally captured data is analyzed, and CNN Neurons perform classification. The RCNN adds a pre-processing step to identify regions of interest to pass to CNN. So RCNN is a special version of CNN. It was Krizhevsky [13] who first initiated this thought of RCNN in 2012 and eventually in 2014, Girshick [13] designed a new method of image detection (RCNN), In 2015, he also designed the model Fast RCNN. [13]

In Matlab, there are options to create a CNN network and train it with labeled image sets. We designed our network using the MATLAB dataset provided with the Faster-RCNN example. In this case, we designed a new network to fit into a specific FPGA.

1.1.3 Vivado Design Suite

The software-hardware acceleration in this research will use Vivado Design Suite as the interface to validate, design the software section and to create bitstream for the hardware. Vivado Design Suite consists of Vivado HLS, Vivado, and SDK. Vivado HLS synthesize and implement the high-level C code into IP block. Vivado and SDK will be used to draw the network's block diagram, and generate the bitstream. We have used Vivado Design Suite 15.3.

1.1.4 Petalinux

The hardware implementation requires an embedded software design. A commercial Linux distribution developed by Petalogix, operating system Petalinux, was our chosen embedded system software. It is used for microprocessors in Xilinx FPGAs as it supports ARM microprocessor. It is considered that Petalinux is useful for this CNN network implementation [14].

1.2 Organization

- Chapter 2 describes our RCNN definition in MATLAB and its output modification.
- Chapter 3 explains how the verification of the network was done.
- Chapter 4 describes the hardware implementation details.
- Chapter 5 is the summary.

2. DEFINE RCNN AND IMPLEMENTATION IN MATLAB

The first step of this thesis was to train an RCNN network with a labeled dataset and using the weights and biases for the next step in FPGA hardware. For training the already labelled dataset of "rcnnStopSigns.mat" [15] from MATLAB 2018B was used. Then we defined, designed and trained our RCNN in MATLAB.

2.1 Layers and Training Image Set

Initially, we choose to select a CNN network with three convolutional layers with filter numbers, 32,32,64 respectively. However, later we revised the filter numbers to be 32,32,16 respectively. The reason why we did not validate the earlier estimation of the design is described in detail in Chapter 3's validation section. So, our final layers for the RCNN detection and classification is as following Figure 2.1 and Figure 2.2.

```
layers = [imageInputLayer([32 32 3])
 convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 convolution2dLayer(5,16,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 fullyConnectedLayer(16)
 reluLayer()
 fullyConnectedLayer(2,'WeightLearnRateFactor',20,'BiasLearnRateFactor',10)
 softmaxLayer()
 classificationLayer()];
```

Fig. 2.1.: RCNN layers description

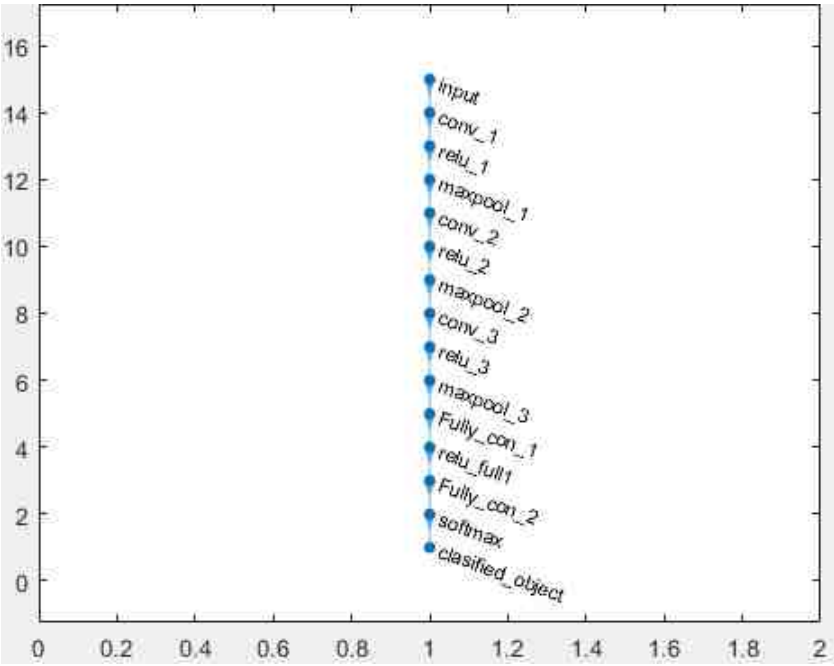


Fig. 2.2.: RCNN layers graph in MATLAB

Here we can see that we choose a network with RGB images of 32 by 32 by 3 as its image input layer. In a Network, layers start with an input layer. So, we started with the image input layer which will take the first step to take the sample into the network. After feeding the network, we started the first convolutional operation using the first convolutional layer. It takes regions from the image and convolves it with the parameter values. After the operation, to provide the non-linearity, we use the ReLU layer. ReLU stands for Rectified Linear Units [16]. It provides the network with non-linearities to better model the real world using "max" function. After ReLU, we get new features from convolved images.

We used a max pooling layer to extract the max values from each pooling square with stride 2. We repeat the process two more times to get the final features for the fully connected layer. The goal of these layers was to create unique features which will detect a particular object when it goes through the fully connected layers. After all nine layers (3 times convolution, ReLU, and max-pooling) we obtain the essential features. Now, we need fully connected layers to classify from these features. We

take two fully connected layers and a ReLU layer in between. We did this to provide non-linearity after the first fully connected layers. The second fully connected layer is defined by the number of objects to be determined. In our case, it had to be either 'stop sign' or 'background.'

Softmax layer is the next one. Softmax layer is normally used at the final fully connected layer because it emphasizes the most likely feature match by regression. So finally, we receive our class through the classification layer. Among the three convolutional layers, the first two convolution layers have 32 5 by 5 filters, whereas third one has 16 5 by 5 filters. All of them were with padding and bias learn rate factor of 2. The first fully connected layer uses 16 nodes to learn non-linear combinations of the features, and the last fully connected layer is used to produce the two class scores [17]. Parameters during the training are shown in the Figure 2.3 using MATLAB [18].

Training Options	SGDM
Mini Batch Size	32
Initial Learning Rate	1.00E-09
Max Epochs	20

Fig. 2.3.: Training parameters

The training had three steps.

- First, extracting region proposal from the labeled Data set. This phase reads the image input and identifies the feature to learn. Before any network training, there must be ground truth dataset. We had 27 sign images where stop signs were labeled with a rectangle box of four co-ordinates as the following Figure 2.4.

- Second, training our defined network to classify objects in our data. In this phase, the network gets trained according to described parameters such as the number of epochs, mini-batch size and initial learning rate.

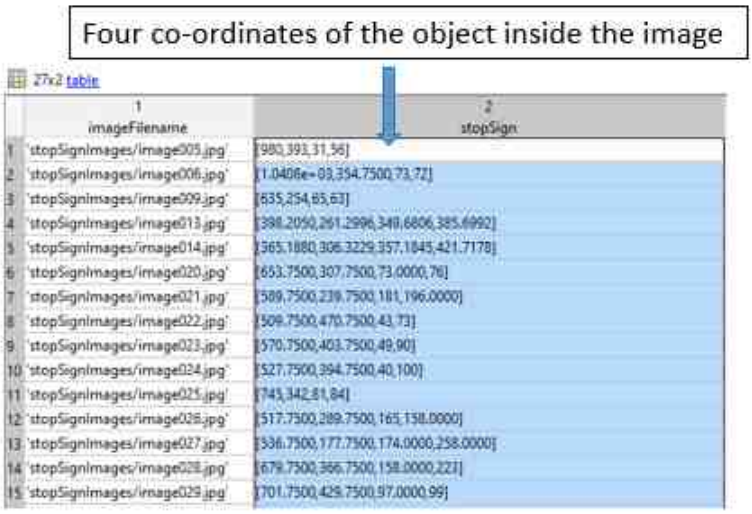


Fig. 2.4.: Labelled stop sign co-ordinates in the training images

```
Step 1 of 3: Extracting region proposals from 27 training images...done.

Step 2 of 3: Training a neural network to classify objects in training data...

Training on single CPU.
Initializing image normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:00 |       25.00% |       0.6960 |      1.0000e-09 |
|       3 |          50 |       00:00:06 |       25.00% |       0.6948 |      1.0000e-09 |
|       6 |         100 |       00:00:12 |       25.00% |       0.6947 |      1.0000e-09 |
|       8 |         150 |       00:00:19 |       25.00% |       0.6951 |      1.0000e-09 |
|      11 |         200 |       00:00:25 |       25.00% |       0.6953 |      1.0000e-09 |
|      14 |         250 |       00:00:31 |       25.00% |       0.6948 |      1.0000e-09 |
|      16 |         300 |       00:00:38 |       25.00% |       0.6953 |      1.0000e-09 |
|      19 |         350 |       00:00:44 |       25.00% |       0.6957 |      1.0000e-09 |
|      20 |         380 |       00:00:48 |       25.00% |       0.6948 |      1.0000e-09 |
|=========================================================================================|

Network training complete.

Step 3 of 3: Training bounding box regression models for each object class...100.00%...done.
```

Fig. 2.5.: 3 steps of training

- Third, training bounding box regression models for each object class. This phase detects the region inside an image where the detected object is placed and place a box around that before we get the output. We can see this box in all the images of Figure 2.6, 2.7, 2.9 and 2.10.

In our case, We display the sigle stop sign with the best match, but multiple objects can be shown and there is also the background (no stop sign). The three steps were successfully done in MATLAB as can be seen from the images.

Now it was time for testing the network with test images. We choose 8 random images from Google search [19] [20] [21] [22] [23] [24] [25] [26] and two data from the training set to feed into the network and observe the performance. The detected images in Figure 2.6 and 2.7 showed good result, using MATLAB.

Fig. 2.6.: First 6 test images output with boxes

Fig. 2.7.: Last 4 test images output with boxes

It has to be noted that before this network we tried another network. This network had similar parameters but had 64 filters in the third convolutional layer, and 64 filters in the first fully connected layer as shown in the following layer settings figure 2.8.

```
layers = [imageInputLayer([32 32 3])
 convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 convolution2dLayer(5,64,'Padding',2,'BiasLearnRateFactor',2)
 reluLayer()
 maxPooling2dLayer(3,'Stride',2)
 fullyConnectedLayer(64)
 reluLayer()
 fullyConnectedLayer(2,'WeightLearnRateFactor',20,'BiasLearnRateFactor',10)
 softmaxLayer()
 classificationLayer()];
```

Fig. 2.8.: Alternate RCNN layers description

Fig. 2.9.: First 6 test images output with alternate network

The output performance was almost the same performance of the alternate RCNN as seen in Figure 2.9 and 2.10. We will describe why we chose the smaller RCNN in the next chapter.

We also ran the test images with basic MATLAB layer architecture. The layer description are shown in Figure 2.11 and detected images is shown in Figure 2.12. Confidences are higher with this trained network.

Fig. 2.10.: Last 4 test images output with alternate network

1	1x1 ImageInputLayer
2	1x1 Convolution2DLayer
3	1x1 MaxPooling2DLayer
4	1x1 ReLULayer
5	1x1 Convolution2DLayer
6	1x1 ReLULayer
7	1x1 AveragePooling2DLayer
8	1x1 Convolution2DLayer
9	1x1 ReLULayer
10	1x1 AveragePooling2DLayer
11	1x1 FullyConnectedLayer
12	1x1 ReLULayer
13	1x1 FullyConnectedLayer
14	1x1 SoftmaxLayer
15	1x1 ClassificationOutputLayer

Fig. 2.11.: MATLAB layer orientation

Fig. 2.12.: MATLAB detection with already trained network

So, after successful classification, we moved on to the network detail where we found each CNN layer's weights and biases which will make our parameter file for the hardware. At this point, our challenge was to extract the CNN weights and biases

data. It was a challenge mainly because it was a data that cannot be extracted manually. It is a 4-dimensional matrix. We wanted row reading first, so the simple solution was to make a data modification by a simple code in MATLAB. This code reforms our data according to our requirements seen in Figure 2.13. This step marks the end of our MATLAB part by exporting weights and biases for our network's hardware.

```
[row, column] = size(M);

Length_NewMat = row*column;

NewMat= zeros(Length_NewMat, 1);

NewMat_index = 1; %initialization

for block = 1:3

    for i = 1: 32 %row

        for k = 1: 32 %col

            col_index = (block-1)*32 + k;

             val = M (i, col_index);

            NewMat(NewMat_index, 1) = val;

            NewMat_index = NewMat_index + 1;
        end
    end
end
```

Fig. 2.13.: Code for formatting matrices

2.2 Network Code and Parameter Setup

After the successful creation of the weights and biases from the trained network, we need to create the parameter file where the weights and biases will be presented as matrices to the hardware. We again formatted the file according to our requirement for input to the hardware. It creates the parameter file which will work alongside the CNN C code. This file has three big matrices of weights of the convolutional

layers and seven other small matrices which will include biases of the convolutional layers and weights and biases of fully connected layers. We also had to calculate the output after each operation of convolution and pooling layers. We calculate the output dimensions depending on input dimensions, pooling dimensions, and stride squares [17]. The second part to be prepared was the C code which represents the CNN architecture. We had to represent each layer in C language to validate the network in Vivado HLS. We wrote each layer according to layer.

```
//Load biases
    for (k = 0; k < FM_2; k++) {
        for (i = 0; i < DIMH_2; i++) {
            for (j = 0; j < DIMW_2; j++) {
                o2[k][i][j] = b2[k];
            }
        }
    }
```

Fig. 2.14.: Loading biases

```
//convolution
 Conv2:for (i = 0; i < DIMH_2; i++) {
        for (j = 0; j < DIMW_2; j++) {
            for (l = 0; l < FM_1; l++) {
                for (s = 0; s < KH_2; s++) {
                    m = i + s;
                    for (t = 0; t < KW_2; t++) {
                        n = j + t;
 cnn_label1:for (k = 0; k < FM_2;
                                k++) {
#pragma HLS PIPELINE
                            v = w2[k][l][s][t] *
                                p1[l][m][n];
                            o2[k][i][j] += v;
```

Fig. 2.15.: Loading weights and calculating with biases

Figure 2.13, 2.14, 2.15 and 2.16 shows four major operations through each convolutional, ReLU and max pooling. We can see from Figure 2.14 coding that a parameter file loads the bias values of the each layer in the convolution operation.

Then in Figure 2.15, it is shown how we have calculated operation between weights and biases, and the pragma PIPELINE is used. This pragma allows concurrent executions of operation by reducing the interval during initiation [27].We also used pragma HLS INTERFACE inside vivado HLS tool(validation operation) which specifies RTL genertation from the definition of the function [27].

```
//Relu for convolution 2
                        if ((o2[k][i][j]) <= 0){
                            o2[k][i][j] = (0.00000000000000);
                        }
                        else o2[k][i][j] = (o2[k][i][j]);
                    }
```

Fig. 2.16.: Applying ReLU (MAX(0,x) operation)

```
//Pooling
 Pool2:for (k = 0; k < FM_2; k++) {
        for (i = 0; i < PDIMH_2; i++) {
            for (j = 0; j < PDIMW_2; j++) {
                max = -HUGE_VAL;
                for (s = 0; s < PH_2; s++) {
                    m = i * PS_2;
                    m += s;
                    for (t = 0; t < PW_2; t++) {
                        n = j * PS_2;
                        n += t;
                        if (o2[k][m][n] > max) {
                            max = o2[k][m][n];
                        }
                    }
                }
                p2[k][i][j] = max;
            }
```

Fig. 2.17.: Pooling max values from the 3 by 3 square

In Figure 2.16, we can see the ReLU layer performing the MAX operation which will provide the network with the non-linearity. The max pooling layer is pooling the max out of the three by three square. This operation is shown in Figure 2.17. When the generation of the C code for network and parameter file is done, we are ready for the validation and design Phase.

3. VALIDATION, DESIGN AND BITSTREAM GENERATION

This section deals with the validation of the network with chosen FPGA. We can choose any network or circuit for given hardware, but we always have to remember that hardware has limited resources. So, even if we design a perfect network, it actually might not work within particular hardware. That is why validation is required. Once a design is validated then we move on to its block diagram design. Here we use the module created by our design along with other supporting parts and wires. If the block diagram is also validated correctly, then we move on to the bitstream generation phase. This whole process was performed using Vivado Design Suite 15.3 using HLS scripting language, as shown in section 2.2. When we complete synthesis and implementation, the tool creates a script file inside Vivado HLS that maps to hardware.

3.1 Validation

The biggest initial challenge of this whole thesis was resource optimization. This thesis work was initially a continuation of a previous student's work to improve the accuracy of a given network [28]. There was a given RCNN/CNN network of 15 layers in the previous design. It also used three convolutional layers, similar to ours. However, it had some differences regarding the third convolutional layer and first fully connected layer. It was a conventional RCNN/CNN layer configuration, and it was classifying ten classes. It is the alternate network that we have described and showed stop sign classification performance in Chapter 2. Figure 3.1 shows the layer configuration of that network.

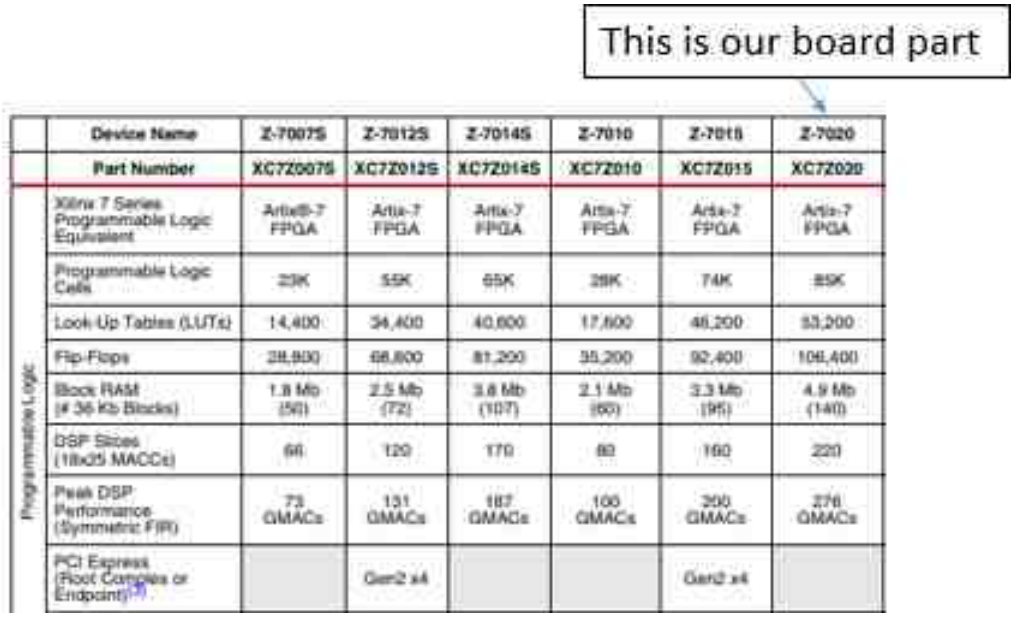
Layer	Layer Type
Image Input	32x32x3 images with 'zerocenter' normalization
Convolution	32 5x5 convolutions with stride [1 1] and padding [2 2]
ReLU	ReLU
Max Pooling	3x3 max pooling with stride [2 2] and padding [0 0]
Convolution	32 5x5 convolutions with stride [1 1] and padding [2 2]
ReLU	ReLU
Max Pooling	3x3 max pooling with stride [2 2] and padding [0 0]
Convolution	64 5x5 convolutions with stride [1 1] and padding [2 2]
ReLU	ReLU
Max Pooling	3x3 max pooling with stride [2 2] and padding [0 0]
Fully Connected	64 fully connected layer
ReLU	ReLU
Fully Connected	10 fully connected layer
Softmax	softmax
Classification Output	crossentropyex

Fig. 3.1.: Proposed 15 layers Convolutional Neural Network [28]

We choose the Zedboard development kit as our hardware platform. The resource requirements for this previous network was too high for the Zedboard. The board has a Xilinx Z-7020 FPGA chip, and its specifications are shown in Figure 3.2. We can see that it has 85k programmable logic cells, 53,200 LUTs, 106.400 Flip-Flops, 4.9 Mb BRAM, and 220 DSP slices. This 4.9 Mb BRAM is consist of 140 36Kb blocks or 280 18Kb blocks. The usable RAM is 4.9 Mb.

BRAM stands for Block Random Access Memory, and it stores data. The DSP slice is used to implement algorithmic operations (Digital Signal Processing). The DSP slice performs multiply-accumulate operation. Within the signal processing

industry, utilization DSP operations in FPGA is a common practice. LUT stands for LookUp Table. It is a table that determines how our combinational logic output behaves. FFs and LUTs can be used for creating registers.



	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs
	PCI Express (Root Complex or Endpoint)		Gen2 x4			Gen2 x4	

Fig. 3.2.: Zynq-7000 SoC Data Sheet: Overview [29]

The previous student's network could not be implemented with our chip resources. The number of parameters it would need to save was too high for the Zedboard's memory. The resources were mainly absorbed by the values of weights and biases and their products. So after the synthesis, we found that it was not a fit for our hardware. The report of the synthesis for this network is shown in Figure 3.3.

We can see that it has exceeded the BRAM utilization by 59 percent more than maximum usage. This view is just the summary, but we investigated in detail and saw that some resources could be reused but it will hamper the overall latency. The first two vertical images in Figure 3.4 shows that it improved the resource usage by 11 percent from 159 to 148 but it is still too high. Then remaining within this usage, we tried to increase the latency using pipelining. It improved the latency but exceeded the estimated clock. Next, we used another option to increase memory. We created

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.58	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
15896497	15924177	15896498	15924178	none

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2180
FIFO	-	-	-	-
Instance	-	92	5201	7022
Memory	446	-	73	4
Multiplexer	-	-	-	1884
Register	-	-	2784	71
Total	446	92	8058	11161
Available	280	220	106400	53200
Utilization (%)	159	41	7	20

Fig. 3.3.: Basic C synthesis result of the proposed network

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1571
FIFO	-	-	-	-
Instance	-	92	5201	7022
Memory	416	-	73	4
Multiplexer	-	-	-	1781
Register	-	-	2251	-
Total	416	92	7525	10378
Available	280	220	106400	53200
Utilization (%)	148	41	7	19

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1932
FIFO	-	-	-	-
Instance	-	92	5201	7022
Memory	416	-	73	4
Multiplexer	-	-	-	1840
Register	-	-	2519	22
Total	416	92	7793	10820
Available	280	220	106400	53200
Utilization (%)	148	41	7	20

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.58	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
141324417	141352097	141324418	141352098	none

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	19.50	1.25

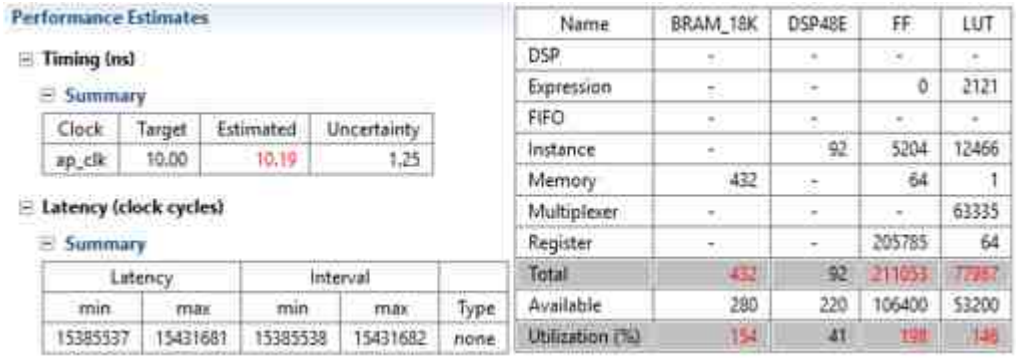
Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
55706097	55733777	55706098	55733778	none

Fig. 3.4.: C synthesis result using all loops pipelining and no pipelining with possible resource reusing

registers using available FFs and LUTs. This only increased a few 100s even using all the available FFs and LUTs. This also was limited to 14k BRAM and exceeded the clock constraint as shown in Figure 3.5.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.19	1.25

Latency (clock cycles)

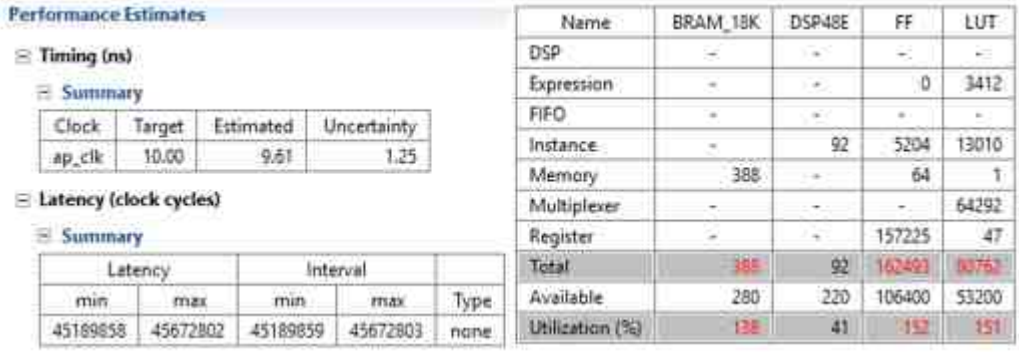
Summary

Latency		Interval		
min	max	min	max	Type
15385537	15431681	15385538	15431682	none

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2121
FIFO	-	-	-	-
Instance	-	92	5204	12466
Memory	432	-	64	1
Multiplexer	-	-	-	63335
Register	-	-	205785	64
Total	432	92	211053	77967
Available	280	220	106400	53200
Utilization (%)	154	41	198	146

Fig. 3.5.: C synthesis result after array partitioning

Next, we combined these two processes and reused BRAMs and partitioned an array (making new register) to maximize the usage. However, even after that, it could only reach 138 percent from 148 percent.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.61	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
45189858	45672802	45189859	45672803	none

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3412
FIFO	-	-	-	-
Instance	-	92	5204	13010
Memory	388	-	64	1
Multiplexer	-	-	-	64292
Register	-	-	157225	47
Total	388	92	162493	90762
Available	280	220	106400	53200
Utilization (%)	138	41	152	151

Fig. 3.6.: C synthesis result after array partitioning and BRAMs sharing

This analysis was the reason why the previous student's network, described in Chapter 2, was not selected. Finally, we were forced to modify our network, although a new board with Xilinx Z-7045 could be a future improvement [29]. At this point, we redesigned the network which made a reduction in the final convolutional layer and fully connected layer. This design fits inside the available resources as shown in Figure 3.7. The memory allocation, performance and resource profiles were also validated as shown in Figures 3.8, 3.9 and 3.10.

Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.58	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
13644866	13675618	13644867	13675619	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2045
FIFO	-	-	-	-
Instance	-	92	5204	7026
Memory	220	-	201	28
Multiplexer	-	-	-	1844
Register	-	-	2751	67
Total	220	92	8156	11010
Available	280	220	106400	53200
Utilization (%)	78	41	7	20

Fig. 3.7.: C synthesis result after network redesign and profiles

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
b1_U	cnn_b1	1	0	0	32	32	1	1024
b2_U	cnn_b2	1	0	0	32	32	1	1024
b3_U	cnn_b3	0	32	8	16	32	1	512
buf_U	cnn_buf	1	0	0	480	32	1	15360
indexes_U	cnn_indexes	0	9	3	15	9	1	135
l1_U	cnn_l1	0	64	8	16	32	1	512
l2_U	cnn_l2	0	64	1	2	32	1	64
lb1_U	cnn_lb1	0	32	8	16	32	1	512
lw1_U	cnn_lw1	8	0	0	2304	32	1	73728
lw2_U	cnn_lw2	1	0	0	32	32	1	1024
o1_U	cnn_o1	64	0	0	32768	32	1	1048576
o2_U	cnn_o2	16	0	0	7200	32	1	230400
o3_U	cnn_o3	2	0	0	784	32	1	25088
p1_U	cnn_p1	16	0	0	7200	32	1	230400
p2_U	cnn_p2	4	0	0	1568	32	1	50176
p3_U	cnn_p3	1	0	0	144	32	1	4608
lin_U	cnn_p3	1	0	0	144	32	1	4608
w1_U	cnn_w1	8	0	0	2400	32	1	76800
w2_U	cnn_w2	64	0	0	25600	32	1	819200
w3_U	cnn_w3	32	0	0	12800	32	1	409600
Total		20	220	201	28	93553	617	20	2993351

Fig. 3.8.: C synthesis memory allocation analysis

It is noticeable that most of the BRAMs are getting absorbed for saving first convolution layer output values (01_U) and weights values of the second convolutional layer (w2_U). As a result, the first pooling layer also absorbs many BRAMs. It is shown in Figure 3.8.

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
cnn	-	13644866~13675618	13644867 ~ 13675619	-	-
Loop 1	no	960	-	2	480
Loop 2	no	34912	-	1091	32
Conv1	no	3594336 ~ 3625088	-	112323 ~ 113284	32
Pool1	no	317824	-	9932	32
Loop 5	no	8256		258	32
Conv2_L_cnn_label1	no	8388000	-	233	36000
Pool2	no	69504	-	2172	32
Loop 8	no	1056	-	66	16
Conv3_L_cnn_label2	no	1199520	-	153	7840
Pool3	no	6464	-	404	16
Reshape	no	416	-	26	16
Lin1	no	23088	-	1443	16
Lin2	no	324	-	162	2
Class1	no	8	-	4	2
Class2	no	64	-	32	2
Class3	no	100	-	50	2
Pred	no	16	-	8	2

Fig. 3.9.: C synthesis performance profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
cnn	220	92	8156	10935						
I/O Ports(14)					88					
Instances(11)	0	92	5204	7026						
Memories(20)	220		201	28	617			20	93553	2993351
Expressions(320)	0	0	0	2037	2305	2127	590			
Registers(290)			2751		3102					
FIFO(0)	0		0	0	0			0		
Multiplexers(110)	0		0	1844	1349			0		

Fig. 3.10.: C synthesis resource profile

Figure 3.9 shows us that, because of those two high values, trip count at the second convolution inner loop is the highest. When we tried to use a Dataflow architecture, we noticed a decrease in latency from 13644866 to 8216258, but it then exceeded the resources again as shown in figure 3.11.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.73	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
8216258	8216258	8216259	8216259	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	-	-	-	-
Instance	116	112	10269	15042
Memory	208	-	384	20
Multiplexer	-	-	-	4
Register	-	-	3	-
Total	324	112	10656	15067
Available	280	220	106400	53200
Utilization (%)	115	50	10	28

Fig. 3.11.: C synthesis result after network redesign with DATAFLOW pragma

After a successful C synthesis, we created the script to run the C simulation and run C/RTL co-simulation in Vivado HLS, this produces a hardware module named cnn_0. A few snippets of the operation such as pipelining the convolutional layers, successfully finishing and exporting RTL to the Vivado and log messages are given below.

```
@I [SCHED-11] Starting scheduling ...
@I [SCHED-61] Pipelining loop 'cnn_label0'.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 1, Depth: 13.
@I [SCHED-61] Pipelining loop 'cnn_label1'.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 1, Depth: 13.
@I [SCHED-61] Pipelining loop 'cnn_label2'.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 1, Depth: 13.
@I [SCHED-11] Finished scheduling.
@I [HLS-111] Elapsed time: 0.519 seconds; current memory usage: 138 MB.
@I [HLS-10]
```

Fig. 3.12.: Pipelining three inner loops of conv layers

We can notice in Figure 3.12, that three inner loops of the convolution operation are successfully applied with pipelining with the labels cnn_label0, cnn_label1 and cnn_label2 respectively. Now, with a quick look at Figure 2.12 we see our second convolutional layer inner loop was declared as cnn_label1. Figure 3.13 is the confirmation of that loop getting pipelined.

```
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'cnn'.
@I [WVHDL-304] Generating RTL VHDL for 'cnn'.
@I [WVLOG-307] Generating RTL Verilog for 'cnn'.
@I [IMPL-8] Exporting RTL as an IP in IP-XACT.
```

Fig. 3.13.: CNN RTL generation

Next, we see the successful messages during the finishing of RTL generation at Vivado HLS, generating hardware language of these RTLs for Vivado and exporting it as an IP block for use with an embedded microprocessor.

During the simulation, Vivado HLS generates core modules, implements BRAMs, synthesizes and simulates. Figure 3.14 describes the generation of synthesis and simulation targets for various DSPs.

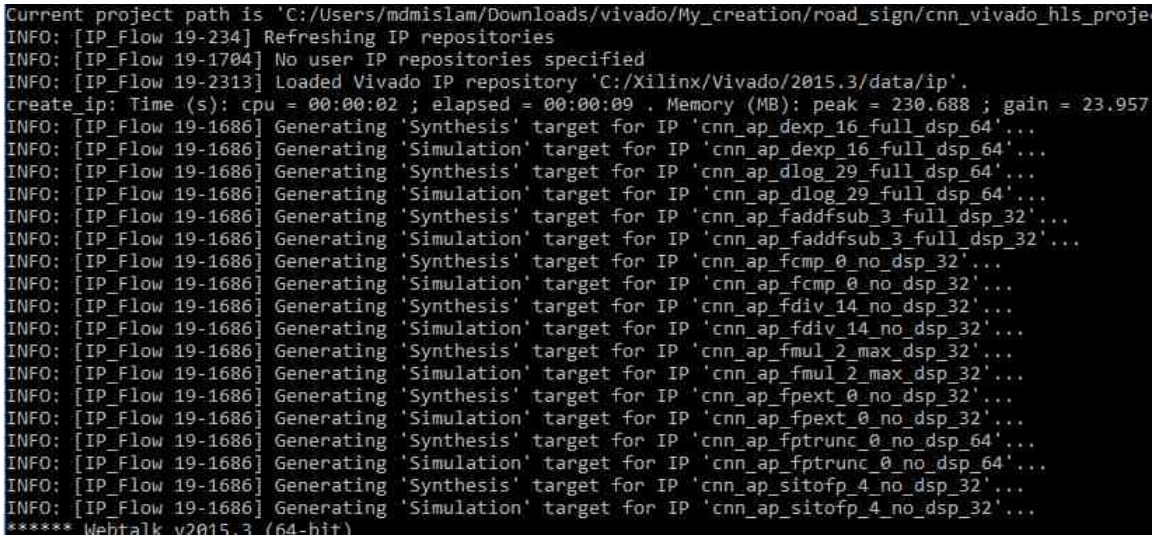


Fig. 3.14.: RTL simulation

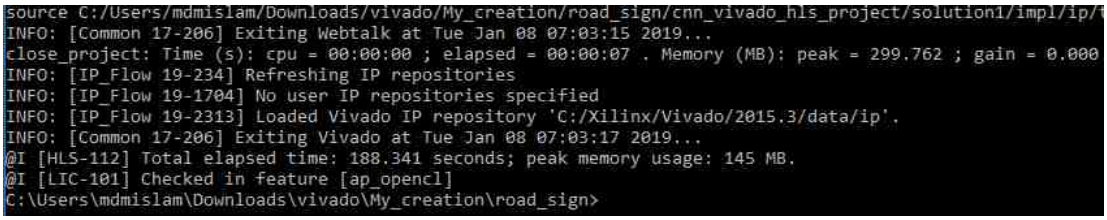


Fig. 3.15.: Finishing message

It is also noticeable from Figure 3.14 that memory usage increased to 230 MB during this phase. This last Figure 3.15 shows that, after successful compilation, Vivado HLS uploads this IP block in the Vivado IP repository. So, later Vivado will use it for designing the block diagram of the system. The total time taken for creating hardware block for our device was 188.341 seconds, and the peak memory usage was 145 MB on desktop computer.

3.2 Designing Block Diagram and Bitstream Generation

After the successful creation of an IP block from Vivado HLS, we move on to the Vivado tool to create the block diagram within the actual FPGA.

Fig. 3.16.: RTL IP block of our CNN

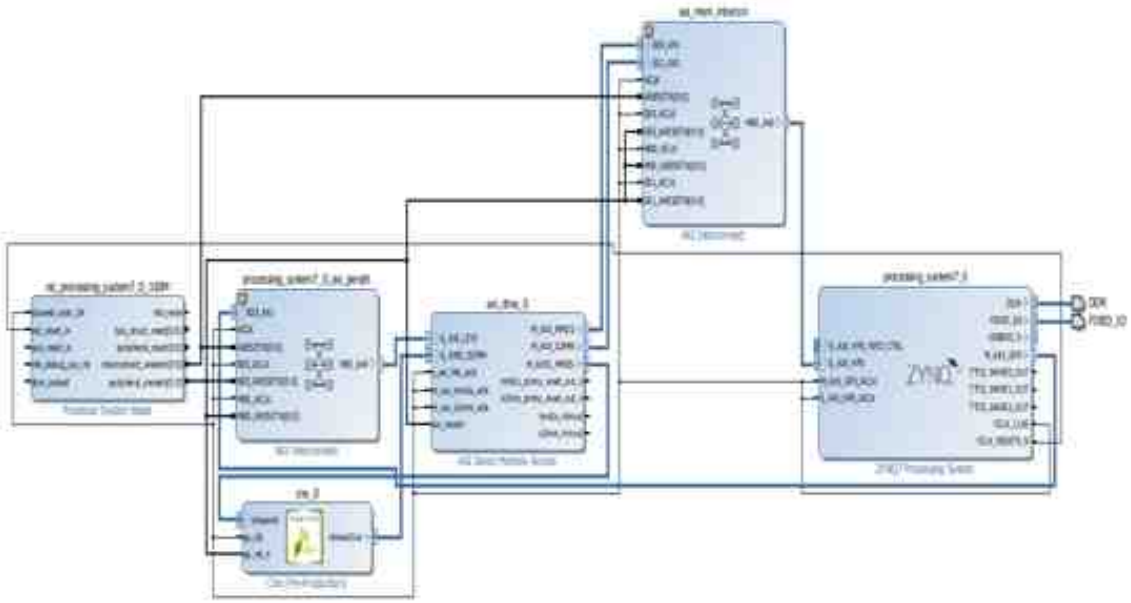


Fig. 3.17.: Block Diagram with cnn_0, DMA, ARM processor

The module cnn_0 was added to the hardware library. Upon importing it to the block diagram, we can see it in figure 3.16. ARM core processing_system7_0 is the processing system related to Zedboard zynq 700. We started our Vivado part by creating a Vivado project at the same directory of the completed Vivado HLS direc-

tory and importing the library path. Then we started designing a block diagram by importing this Zynq processing system. Next, we apply the block diagram automation command with the board preset. Then, we required an interfacing block, and to connect our block to the microprocessor AXI-bus was the standard choice. we added an AXI_DMA cell to make the interface in between the processing system AXI-bus and our module. Then we select and connect the MASTER and SLAVE using the peripheral system and the AXI_DMA. After all the interfaces are ready, we import our module cnn_0 and connect the streamIN and streamOUT ports with the DMA. Once our total connection is complete, the block diagram looks like as above in Figure 3.17. When the diagram is complete we launch the implementation run. When the implementation is done, we get all the details regarding the successful design.

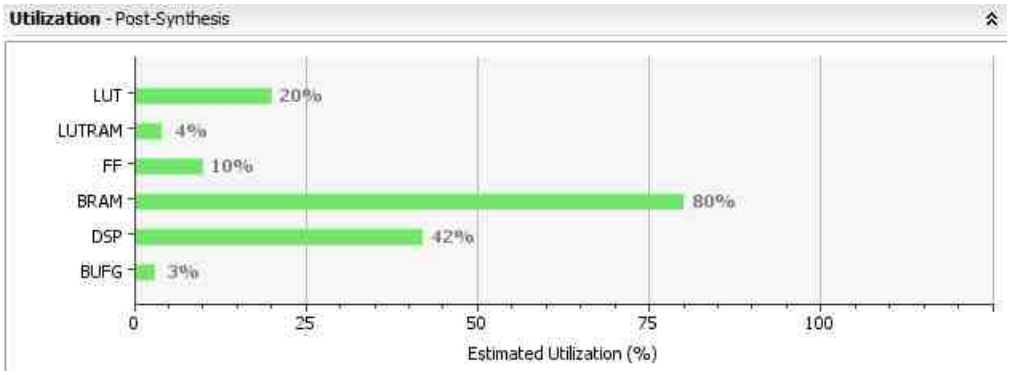


Fig. 3.18.: Post-synthesis summary graph

Now we can see the post-synthesis and post-implementation utilization estimation's graphical views as in Figures 3.18 and 3.19. We can see a 2 percent drop in LUTs usage after the implementation. Total power summary is provided in Figure 3.20 where we can see total on-chip power consumption will be 2.235W where 92 percent (2.050W) will be used for dynamic power and 8 percent for static power.

The last analysis on this phase was the timing analysis of setup, hold and pulse width timing where none of them have falling endpoint or negative slack as shown in Figure 3.21.

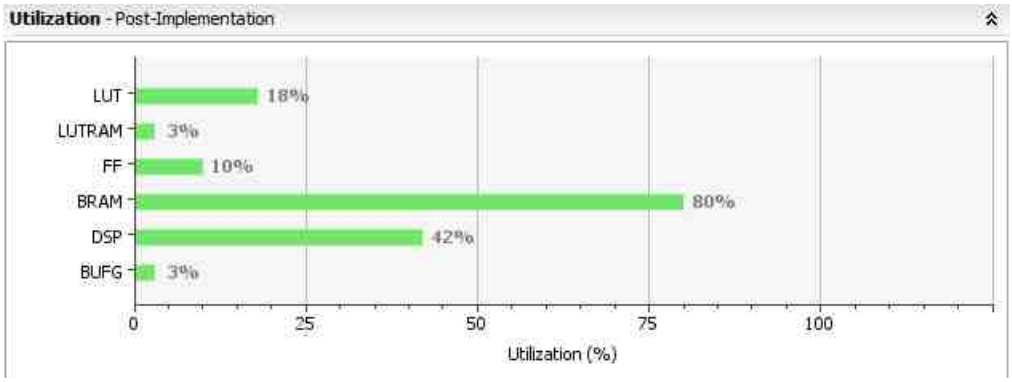


Fig. 3.19.: Post-implementation summary graph

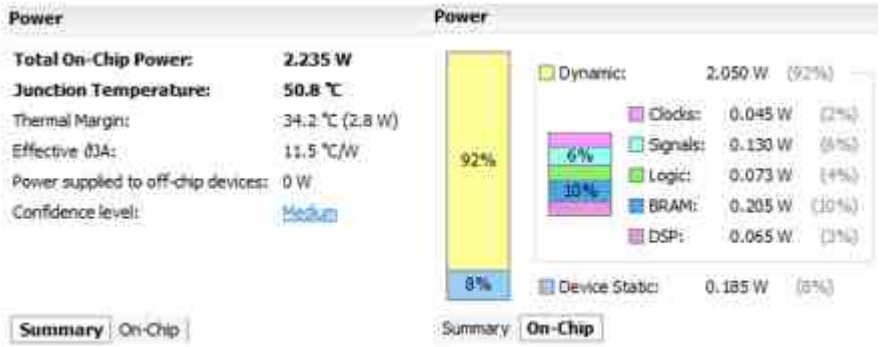


Fig. 3.20.: Power summary

Fig. 3.21.: Timing report

So, we click on the device to see the final implemented design of the device and it is shown in Figure 3.22. It provides a graphical representation of how the blocks of FPGA are used in real-life.

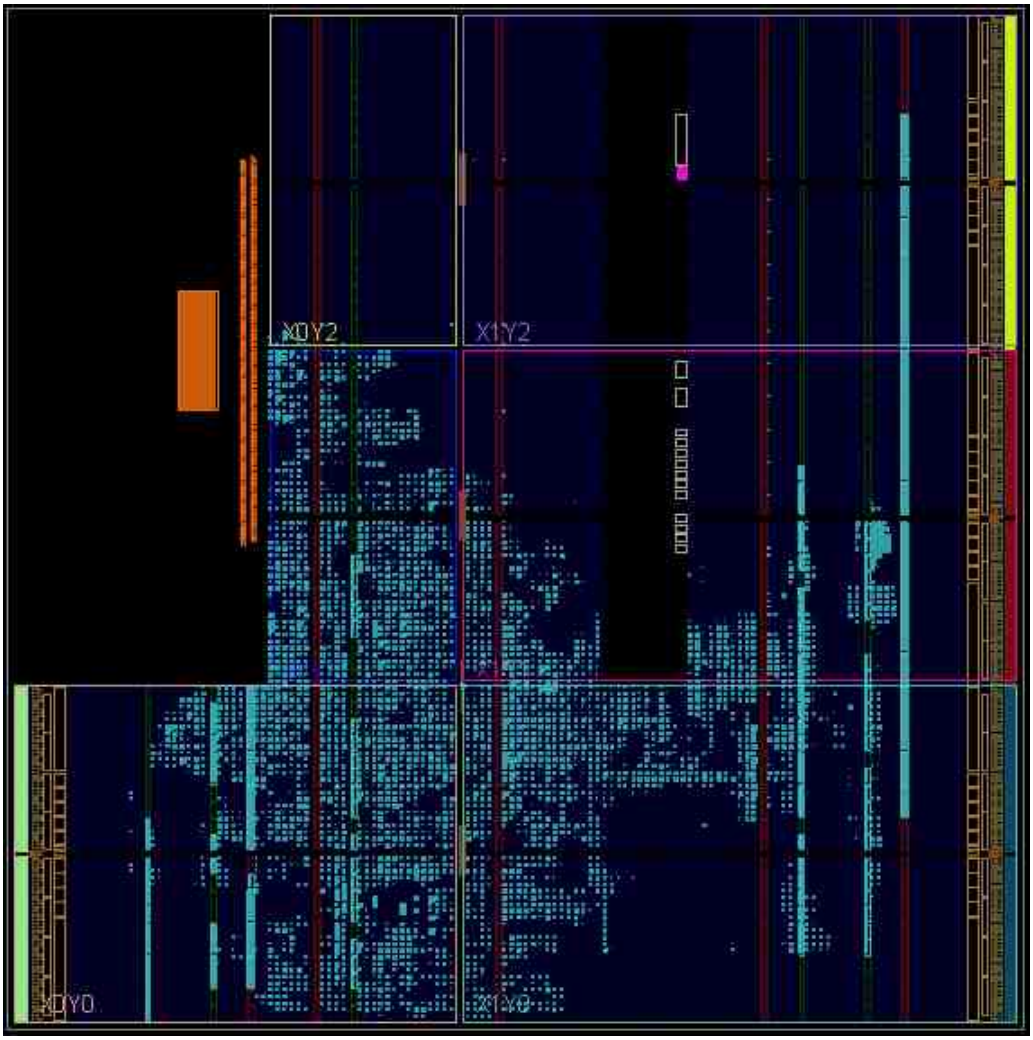


Fig. 3.22.: Device, showing used blocks on chip.

The most important thing that we obtained from this phase was the hardware bitstream programs the FPGA hardware. We exported the bitstream to the Vivado SDK, where it will have two ways to go into the hardware as described in the next chapter.

4. SOFTWARE AND HARDWARE IMPLEMENTATION IN PETALINUX

4.1 Choosing Platform and Data Modification

Now that we have obtained the bitstream from Vivado design suite we can move on the next stage which is to implement the network into the hardware. We could achieve that in two ways. One is to do it from Vivado design toolset SDK on the ARM micro processor, but it would constrain us to limited set of operations [30]. The second option is to use an actual operating system(OS) platform that has the driver availability for the hardware and can be implemented in Zedboard. Although we have used this first option (BOOTGEN utility) previously in our course, we chose the second way for more flexibility. We chose an embedded Petalinux as it looked like to be the most convenient way to get into the Zedboard hardware. It provides the Board Support Packages (BSP) for various embedded hardware (FPGA). First, we created the environment to run the hardware operation successfully. We made Linux 16.04.3 as our OS to start the operations. The next step was to install and run Petalinux successfully from a Linux platform on our desktop computer.

Minimum desktop workstation requirement for any computer [31] to install PetaLinux Tool are:

- 8 GB RAM (recommended minimum for Xilinx tools).
- 2 GHz CPU clock or equivalent (minimum of 8 cores).
- 100 GB free HDD space.
- compatible OS (for us - Ubuntu Linux 16.04.3 (64-bit)).

We installed Petalinux [32] [31] and ran successfully only after installing all the tools below in Figure 4.1 with the Ubuntu platform. After installing all the tools, our desired Petalinux is entirely ready for the next phase.

Ubuntu 16.04.3	Ubuntu 16.04.3
tofrodos_1.7.13+ds-2.debian.tar.xz	-
iproute2 4.3.0-1ubuntu3	xvfb (2:1.18.3-1ubuntu2.3)
gawk (1:4.1.3+dfsg-0.1)	git 1.7.1 or above
-	make 3.81
screen	net-tools
pax	
gzip	libncurses5 -dev
libglib2.0-dev	
	tftpd
xterm	i386/zlib1g-dev/1:1.2.8.dfsg-2ubuntu4-dev
autoconf	
libtool	
tar:1.24	libssl -dev
unzip	
texinfo	flex
zlib1g-dev	bison
gcc-multilib	libselinux1
build-essential	gnupg
libsdl1.2-dev	wget
	diffstat
	chrpath
	socat

Fig. 4.1.: Prerequisite tools for the desired environment [31]

Now the next thing we require is a Board Support Package (BSP) for the specific hardware. The good news was that the BSP for Zedboard is already available at Petalinux for various FPGA boards. We just had to use it with our bitstream with minor changes. Next step was to create and compile our program at Petalinux OS. Due to the scarcity of open source instruction and explanation, it was hard to find the commands. But we eventually found [33] it and applied it to the application to run on board. Few of the frequently used commands are seen below:

- petalinux-create -t apps –template install –name mylib –enable
- petalinux-create -t apps –template c++ –name myapp –enable
- petalinux-build
- petalinux-build -c rootfs
- petalinux-build -c myapp -x do-install

After creating the application each time we had to compile and build it. The Device tree of the network is imported from the Vivado generated HW folder. The information about how the FPGA resources will be used, and how the hardware will be designed is passed to the Petalinux BSP from Vivado using this folder and accordingly creates the FSBL, Devicetree in BSP. After building this, we also had to build a BOOT file for the SD card that will go into the FPGA with the command in Figure 4.2.

```
don@don-ThinkPad-E550:~/boot/avnet-digilent-zedboard-2018.2$ petalinux-pac
kage --boot --fsbl images/linux/zynq_fsbl.elf --fpga project-spec/hw-descr
iption/design_1_wrapper.bit --uboot
```

Fig. 4.2.: Commands for generating bootfile from our hardware bitstream

```
csvwrite('test6.csv',x5test)
M = csvread('test6.csv')
imageformatting
fileID = fopen('test6','w');
fwrite(fileID,NM);
fclose(fileID);
fileID = fopen('test6','r');
h=fread(fileID)
```

Fig. 4.3.: Commands in MATLAB to manipulate images into binary files

Once it is done, we mount the SD card and format it to pass the files to the FPGA's ARM microprocessor and Boot file. This files will go inside the FPGA and into the hardware. The RCNN uses a Region of Interest (ROI) preprocessing step which selects potential candidates for CNN classifier. Our CNN in hardware takes the ROI candidates as the input. We selected these candidates from the region pre-processed in MATLAB. To feed the images into the hardware, we need binary arrays. Again, we used MATLAB to manipulate the image data to form binary files, and in our case, we used little-endian (64 bit) format. Figure 4.3 is a snippet of the final few commands of the image creation.Once we have the binary image files to feed it into the CNN network, we are ready for the final step. We run the hardware CNN with the test images.

4.2 Classification Result in Hardware

So, at this final stage, we used an SD card containing Petalinux OS root file system, boot file, and the test images. We connect it with our FPGA through the SD card port with pin settings for SD card boot shown in Figure 4.4.

Fig. 4.4.: Demonstration of FPGA setup

We log in the Petalinux using root user and run the Petalinux using our hardware. We connect to the interface from a desktop computer using connecting software terminals such as Tera term or Putty.

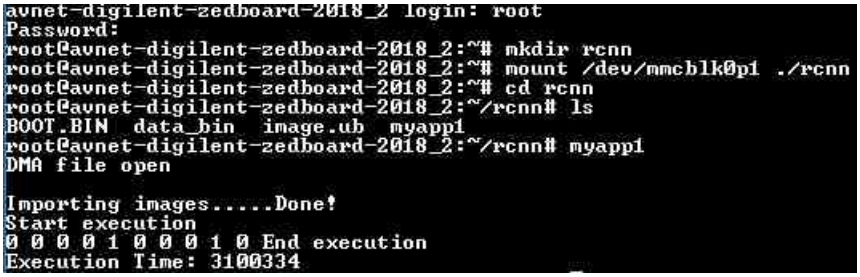


Fig. 4.5.: Run process into the petalinux and classified screenshot

Once we log in using the terminal, we run our compiled application which would uses the hardware to classify objects. It successfully classified all the selected images. Then we put 8 new test images and two arbitrary images to the network. All were correctly classified. Notice that, in Figure 4.5, "0" stands for detecting road sign and "1" stands for the background.

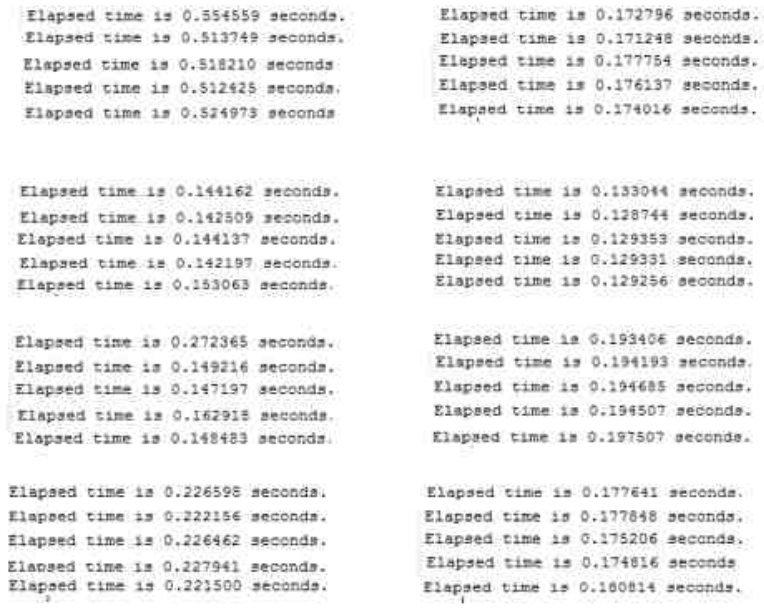


Fig. 4.6.: MATLAB time for executing first 8 test images

So, this matches our expectation as our network should classify the test images just like the results in MATLAB software and the goal is obtained. Let's look at the timing comparison between the hardware and MATLAB. We run each test image five times in MATLAB to get its detection time shown in Figures 4.6 and 4.7. Elapsed times for all ten test images in MATLAB are 0.522, 0.144, 0.176, 0.226, 0.175, 0.130, 0.192, 0.178, 0.122 and 0.292. One images took around 300-400 ms extra time due to image complexity.

```
Elapsed time is 0.120561 seconds.
Elapsed time is 0.124160 seconds.
Elapsed time is 0.117896 seconds.
Elapsed time is 0.116915 seconds.
Elapsed time is 0.127532 seconds.


Elapsed time is 0.292073 seconds.
Elapsed time is 0.287144 seconds
Elapsed time is 0.287972 seconds.
Elapsed time is 0.285781 seconds.
Elapsed time is 0.293766 seconds.
```

Fig. 4.7.: MATLAB time for executing last 2 test images

Average time in MATLAB was 0.216 seconds or 216 ms. Additionally, we tested on a computer with better specifications and with an installed GPU(NVIDIA TITAN XP). Average timing for this is 153 ms for the CPU and 46ms when GPU is used. Following Figure 4.8 shows the timing calculation.

	CPU						GPU						
Image No.	1st run	2nd run	3rd run	4th run	5th run	Average CPU	1st run	2nd run	3rd run	4th run	5th run	Average GPU	Improvement
1	0.377	0.382	0.384	0.385	0.384	0.382	0.104	0.106	0.104	0.105	0.104	0.104	3.7x
2	0.105	0.104	0.107	0.106	0.108	0.106	0.032	0.032	0.032	0.032	0.031	0.032	3.3x
3	0.099	0.1	0.099	0.099	0.101	0.1	0.032	0.033	0.031	0.032	0.033	0.032	3.1x
4	0.16	0.16	0.157	0.165	0.16	0.16	0.046	0.045	0.046	0.046	0.045	0.046	3.5x
5	0.121	0.122	0.121	0.118	0.12	0.12	0.036	0.037	0.038	0.037	0.036	0.037	3.2x
6	0.09	0.091	0.091	0.091	0.09	0.091	0.03	0.03	0.03	0.033	0.03	0.03	3x
7	0.14	0.14	0.141	0.139	0.141	0.14	0.041	0.042	0.042	0.041	0.041	0.041	3.4x
8	0.128	0.128	0.129	0.128	0.126	0.128	0.038	0.038	0.038	0.038	0.038	0.038	3.4x
9	0.086	0.087	0.086	0.086	0.087	0.085	0.028	0.029	0.028	0.029	0.029	0.029	2.9x
10	0.212	0.214	0.213	0.214	0.213	0.213	0.086	0.083	0.084	0.088	0.088	0.086	2.5x
Overall						0.153						0.046	3.3x

Fig. 4.8.: MATLAB time for execution at enhanced CPU and at GPU

Note that although usage of GPU has increased the speed, at the same time, the power consumption has also increased, as shown in Figure 4.9 and Figure 4.10. In the FPGA it took 3100334 ns or 3.1 ms for all ten images. For a single image, it took 0.31 ms. The power consumption was only 2.235 watts.

Fig. 4.9.: Power consumption at enhanced CPU

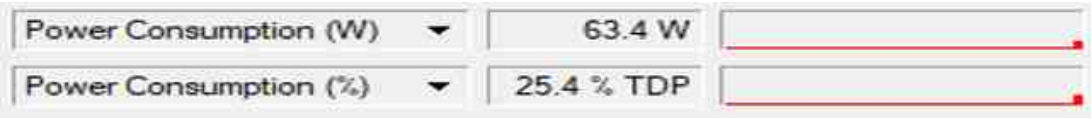


Fig. 4.10.: Increased power consumption at GPU

	ZynqNet[4]	CNN2ECST[5]	RCNN our work	Enhanced CPU	GPU(NVIDIA TITAN XP)
No. of NN Nodes	18	6	15	N/A	N/A
BRAM (No of 18k used)	996	18	220	N/A	N/A
DSP (used)	739	97	92	N/A	N/A
FF	137,000	7590	8156	N/A	N/A
LUT	154,000	10135	11010	N/A	N/A
Speed (to complete)	45 s	0.53ms	0.31ms	153 ms	46 ms
Power	7.80 W	2.01 W	2.24 W	19.5 W	63.4 W
Xilinx FPGA Chip	XC 7Z045	XC 7Z020	XC 7Z020	N/A	N/A

Fig. 4.11.: Comaprison among our network, ZynqNet [4] and CNN2ECST [5]

In Figure 4.11, we compare our result to the closest research. First, the FPGA chip, Zynq XC-7Z045 has a higher number of resources, and ZynqNet has absorbed almost all resources. Also, it consumed 7.80 watts while FPGA accelerator is running. The second one, CNN2ECST was a small network consisting of only one convolutional layer. The resource allocation for this network is also shown in Figure 4.11. It consumed 2.009 watts as on-chip power. In Figure 4.12, we also compare our network

	CMSIS-NN(CIFAR-10)[4]	RCNN our work
No. of Conv_layer	3	3
1st FC_layer Neurons	64	16
2nd FC_layer	10	2
Speed (to complete)	99 s	0.31ms
Power	7.97 W	2.24 W
Processing System (PS)	ARM Cortex - M7	ARM- Cortex - A9

Fig. 4.12.: Comaprison among our network and CMSIS-NN [34]

to an NN which ran in ARM Cortex-M7 [34]. These two tables show that our design is more complex than the CNN2ECST, but less complex than the ZynqNet, and has good performance and power for our task even after comparing to a network run by ARM Cortex-M7 processor. Although the ROI and bounding box parts were not performed in FPGA, it is still a significant time and power usage drop compared to CPU and GPU. This result can help in next-generation machine learning, especially in the automotive industry.

5. SUMMARY

5.1 Conclusion

In this research, our goal was to implement a hardware-software acceleration of an RCNN network. We have designed an RCNN from scratch in MATLAB and used data manipulation to perform the CNN part in hardware (FPGA) eventually successfully. This research contributed to the state of the art in two ways: First, we created and trained the network in MATLAB. Second, we have successfully taken an RCNN from software into the Hardware using Petalinux which has seen much improvement in timing. It achieved FPGA maximum utilization of 220 18k_BRAMS, 92 DSP48Es, 8156 FFS, 11010 LUTs with an on-chip power consumption of 2.235 Watts. We met the clock timing with 0 failing endpoint and 0 negative slack. Classification of images in FPGA is reduced to 0.31 ms from 153 ms in CPU and 46ms in GPU. This result can help in next-generation machine learning, especially in the automotive industry since autonomous driving needs embedded, fast implementations.

5.2 Future Works

Even though we have achieved the basic goals in this work, there are still many areas to improve the work. A few possible improvements:

- Larger training data-set: More images used as the labeled data-set makes better training and more robust performance.
- Implement (ROI) preprocessing for RCNN in the ARM: RCNN holds a difference from the CNN because of its region of interest (ROI). This part was done in MATLAB, but in future, that can be done in the ARM part of the FPGA.

- Compare: The design can be done on both with the bare metal application and with Petalinux OS implementation and observe the differences.
- Alternate Network: Continue to improve the CNN, and compare performance.
- Multi-classification: Extend current bianry classifier to more objects.

REFERENCES

REFERENCES

[1] D. H. Hubel and T. N. Wiensel, *"Receptive fields and functional architecture of monkey striate cortex"*. Department of Physiology, Harvard Medical School, 1968, vol. 195, no. 1.

[2] "Top 10 robots artificial intelligence," December 2017, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.designboom.com/technology/top-10-robots-artificial-intelligence-12-14-2017/

[3] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, *Flexible, high performance convolutional neural networks for image classification.* International Joint Conference on Articial Intelligence, 2011.

[4] D. Gschwend, *ZynqNet: An FPGA-accelerated embedded Convolutional Neural Network*. Swiss Federal Institute of Technology Zurich, 2016.

[5] ""CNN2ECST at Xilinx OpenHW Contest 2016," June 2016, Last Date Accessed: 01-16-2019. [Online]. Available: https://bitbucket.org/necst/cnnecst-benchmark

[6] "ImageNet: what is top-1 and top-5 error rate?" June 2015, Last Date Accessed: 01-16-2019. [Online]. Available: https://stats.stackexchange.com/questions/156471/imagenet-what-is-top-1-and-top-5-error-rate

[7] J.J. Hull, "A database for handwritten text recognition research," *IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: 16 , Issue: 5 , May 1994)*, pp. 550–554, May 1994.

[8] "Handwritten Digits," 2015, Last Date Accessed: 01-16-2019. [Online]. Available: https://cs.nyu.edu/ roweis/data.html

[9] J. L. McClelland, D. E. Rumelhart, and G. E. Hinton, *Parallel distributed processing: Explorations in the micro structure of cognition"*. Cambridge, MA: USA: MIT Press, 1986, vol. 1.

[10] S. J. Thorpe, *"Spike arrival times: A highly efficient coding scheme for neural networks"*. Paris: USA: MIT Press, 1990, vol. Parallel Processing in Neural Systems.

[11] P. H. Winston, *"Artificial Intelligence"*. Boston, MA: USA: Addison Wesley Longman Publishing Co., Inc., 1992, vol. 3rd Edition.

[12] G. Luger and W. A. Stubblfield, *"Artificial Intelligence: Structures and Strategies for Complex Problem Solving"*. Redwood City, California: Benjamin/Cumming Publishing, 1993, vol. 2nd Edition.

[13] J. Zou and R. Song, "Microarray camera image segmentation with faster- rcnn," *2018 IEEE International Conference on Applied System Invention (ICASI)*, pp. 1–4, April 2018.

[14] R. N. de Souza, D. N. Muniz, and A. V. da Silva Fidalgo, "Ethernet communication platform for synthesized devices in xilinx fpga," *2011 IEEE EUROCON - International Conference on Computer as a Tool*, pp. 1–4, 2011.

[15] "Object detection using faster r-cnn deep learning," R2018B 2019, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.mathworks.com/help/vision/examples/ object-detection-using-faster-r-cnn-deep-learning.html

[16] S. Woo and C. L. Lee, "Decision boundary formation of deep convolution networks with relu," *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 1–4, October 2018.

[17] A. Karpathy, "Cs231n convolutional neural networks for visual recognition," *Github*, 2018.

[18] "Trainrcnnobjectdetector," R2018B 2019, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.mathworks.com/help/vision/ref/ trainrcnnobjectdetector.html

[19] "Collection of stop signs," 2018, Last Date Accessed: 01-16-2019. [Online]. Available: http://clipart-library.com/stop-signs.html

[20] "Road sign shapes signify level of danger," July 2017, Last Date Accessed: 01-16-2019. [Online]. Available: http://thenewswheel.com/road-sign-shapes-signify-level-of-danger/

[21] "Stop sign at crossroads. rural road. exit onto the main road. main road. dangerous road. traffic signs stop," Last Date Accessed: 01-16-2019. [Online]. Available: https://www.dreamstime.com/stock-photo-stop-sign-crossroads-rural-road-exit-onto-main-road-main-road-dangerous-road-traffic-signs-stop-image81284724

[22] "Stop signs with flashing lights installed at chardon twp intersection where two teens were killed," June 2017, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.news5cleveland.com/news/local-news/oh-geauga/stop-signs-with-flashing-lights-installed-at-chardon-twp-intersection-where-two-teens-were-killed

[23] "Ohio woman loses lawsuit over foliage growing near stop sign," June 2018, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.news-herald.com/news/ohio/ohio-woman-loses-lawsuit-over-foliage-growing-near-stop-sign/article_ff463163-00d0-5926-9691-ea7ec8fa24eb.html

[24] "Sign bases," 2019, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.dornbossign.com/rubberform-sign-base-w-plastic-round-pole/

[25] "Running a stop sign in Nevada ," 2017, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.shouselaw.com/nevada/traffic/running-stop-sign

[26] "Laminated poster road warning stop red sign traffic stop sign poster print 24 x 36," 2019, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.bhg.com/shop/home-comforts-laminated-poster-road-warning-stop-red-sign-traffic-stop-sign-poster-print-24-x-36-pb3533a9e1ab516dae87ed6cb12b7c7b3.html

[27] *UG1188:SDAccel Development Environment Help for 2018.3*. Xilinx, Inc., December 2018, vol. v2018.3.

[28] T. Rahman, *"Classification of Road Side Material using Convolutional Neural Network and A Proposed Implementation of the Network through Zedboard Zynq 7000 FPGA"*. Indianapolis, Indiana: Purdue University, December 2017, MSECE Thesis.

[29] Xilinx, *"Zynq-7000 SoC Data Sheet: Overview"*. Xilinx, Inc., July 2018, vol. v1.11.1.

[30] "Creating a zynq boot image for an application," 2013, Last Date Accessed: 01-16-2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/SDK_Doc/tasks/sdk_t_create_zynq_boot_image.htm

[31] Xilinx, *PetaLinux Tools Documentation*. Xilinx, Inc., June 2018, vol. v2018.2.

[32] *PetaLinux SDK User Guide: Installation guide*. Xilinx, Inc., November 2018, vol. v2013.10.

[33] Xilinx, *PetaLinux Tools Documentation*. Xilinx, Inc., June 2018, vol. v2018.2.

[34] L. Lai and N. Suda, "Enabling deep learning at the lot edge," *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, January 2019.