Efficient implementation of Deep Neural Networks on FPGA

A Project Report

submitted by

DHANRAJ KOLI

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY &
MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
May 2019

THESIS CERTIFICATE

This is to certify that the thesis entitled **Efficient implementation of Deep Neural Networks on FPGA**, submitted by **DHANRAJ KOLI** (**EE14B091**), to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelors of Technology** and **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Janakiraman Viraraghavan
Research Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 10/5/2019

ACKNOWLEDGEMENTS

I would like to thank our group project guide Prof. Janakiraman for providing us with an amazing opportunity to work on this project. He consistently guided and supported us on our venture throughout and also allowed us to work on our timeline at our pace. I would like to thank my project co-member Yaseen for essential support.

I would especially like to thank lab mates of VLSI and CAD design lab for helping through my doubts and from providing us with essential information, cooperating and showing the ropes on how to start and approach the project.

Finally, I would also take this opportunity to thank all my friends who have been a great source of motivation and encouragement for this project.

ABSTRACT

KEYWORDS: Convolution Neural Network (CNN), ImageNet

ImageNet is an image data set consisting of over 1000 classes of images. It is inspired by a growing sentiment in the image and vision research field.

One of the ways to solve the problem of image classification of such extensive database is using Deep Neural Network's class CNN. A Convolutional Neural Network (CNN) consists of multiple layers. These layers typically consist of convolution layers, ReLU layer i.e. activation function, pooling layers, fully connected layers, and normalization layers.

In our project, our focus was to implement CNN onto hardware: Zedboard's (the device we used) FPGA. Since FPGA can run operations in parallel instead of the sequential process like CPU, there is a scope of acceleration in the speed of execution of CNN. We tried many experiments to accelerate CNN used for ImageNet dataset classification.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DNN	Deep Neural Network
CNN	Convolutional Neural Network
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuits
HLS	High Level Synthesis
VLSI	Very Large Scale Integration
HW	Hardware
SW	Software

CHAPTER 1

INTRODUCTION

1.1 Overview

The ImageNet project is an extensive visual database designed for use in visual object recognition software research. ImageNet contains more than 20,000 categories with a typical class, such as ″balloon″ or ″strawberry″ consisting of several hundred images. The database of annotations of third-party image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet.

A convolutional neural network (CNN) is a class of Deep Neural Networks, most commonly applied to analyzing visual imagery. We used it to solve the ImageNet.

CNN consists of layer typically used to extract features out of an image. These features include edges, curves, sharpness, and blur. It helps in breaking down of large scale of information into small chunks. These small chunks of information are given weights biases as per training and image recognition is done.

1.2 Motivation

At present, most of these networks are simulated by software programs. The software simulation needs a microprocessor and usually takes a long period to execute a massive number of computations involved in the operation of the network. Hence we do hardware implementations to realize such networks. This realization makes the network stand alone and operate on a real-time fashion.

Hence in our project, we implement the CNN architecture model, which can solve the ImageNet on hardware. It would explain the purpose of having broad image classification in real-time and without any software redundancy. Moreover, hardware implementation helps in speed-up of the architecture of CNN; By use of pipe-lining and decreasing memory access time.

1.3 Objectives and Scopes

There exist pre-trained model architectures provided by DARKNET community. These architectures are based on the AlexNet model, which was the first model to solve the ImageNet database.

These architectures vary as per various weight file sizes, number of operations required, and accuracy. For this project, we analyzed which model works best with more accuracy, reasonable complication in operations, and low weight MB file. We implemented the model architecture in C-code following its implementation on hardware (Zedboard).

Our objective was to move convolution function (since it takes maximum computations in the network) onto hardware and do performance analysis. Also, find ways to accelerate the hardware function.

CHAPTER 2

Making of Convolution Neural Network

2.1 Structure

The structure of any DNN in specific CNN is quite simple. It involves the use of typically three functions/operations, which makes the layers of the CNN. The depth of CNN is measured using how many hidden layers does a particular CNN has.

The first layer of CNN is the input layer, and the last is the output layer. The internal layers are called hidden layers. These in specific to image recognition carry features from an image. These features are taken from the input layer through the hidden layers until the output layer by weights. These weights are fixed as per specific model architecture cause different models vary in depths.

The operations involved in each layer includes convolution and ReLu, max pooling, and Softmax. In the upcoming sections, we will discuss this in detail. These operations at each layer are then connected to the next layer with weights so the features stored are not lost. While connecting layer to another layer, the weights are balanced out with normalization.

2.2 Explanation of CNN

There is an image which might contain anything example an object or animal etc.; we need to know how the image is, how edgy it is, where the intensity is low and where is it high, where there is sharpness and so on. So these are the features of image, these are extracted using convolution by giving weights to each pixel. This will give us an idea of what total how many parameters are there so that we have those many features. Now we need accuracy with less computation too; So we use max pooling to extract most viable information from convolution and reduce complexity.

In this fashion, we move ahead layer by layer extracting features from each pixel. In the end, when we have final features from an image, we make all the features(basically numeric representation) into an array. This array is then multiplied by weights we get probabilities. These probabilities determine how much the image resembles a particular thing. We do normalization and pick top 5 probabilities that are done by softmax function and map the top possibilities with a reference map of a specific dataset of images of which we have picked an image to recognize.

2.3 Convolution

Convolution is an operation of summing each element of the image to its local neighbors, weighted by the inputs of the kernel. Note that the matrix operation being performed convolution is not traditional matrix multiplication.

In image processing, a kernel is a small matrix. It is used for blurring, sharpening, edge detection, and more. These operations are done by a convolution between a kernel and pixels of an image.

In simple terms, convolution is done by multiplying a pixels and its neighboring pixels color value by a kernel summing the multiplied values and placing it back to the pixel where the operation has been performed. Differently sized kernels containing different patterns of numbers produce different results under convolution. The numbers stored in the kernel are the weights which are used to extract features and join it to the next layer.

Now there can be many instances wherein weights can be negative, and kernel might contain negative values. When this occurs, it might result in the next layers having negative values post convolution. While making CNN, it is must to have non-negative values, but it's not always possible there might be negative weights. So to solve this ReLu operation is used. This operation does thresholding, and it keeps positive as it is. And to negative values, it multiples with a bias term. This helps in preserving features of image for its recognition at the output layer.

2.4 Max Pooling

Max pooling is done to down-sample an input representation reducing its dimensionality and allowing only significant weighted features to pass on to the next layer.

This is done so in part to help over-fitting. It also reduces the computational cost by reducing the number of parameters. Max pooling is done by applying a max filter to (usually) non-overlapping sub-regions of the initial representation.

Let's say we have a 2x2 filter that we'll run over our input image at pixels and choose the maximum of 4 numbers. We'll have a stride of 2 meaning the filter will jump two rows and two columns each time to perform this operation reducing the dimensionality and features(parameters) for the next operation by a factor of 2.

2.5 Softmax and Output Layer

It is a function that takes a numeric array and normalizes it into a probability distribution i.e., array of probabilities. Before applying softmax, some numbers could be negative, or greater than one; and might not sum to 1; but after applying softmax, it ensures each component will be in the interval (0,1), and the components will add up to 1, thus interpreted as probabilities.

The softmax function is used in the final layer of a neural network-based classifier.

Finally, the output layer is used to map the output, i.e., the set of probabilities to see what the image is. This layer is too essential as it carries the vital information of what the features. Note, whichever data set we wish to classify we should have the index map of it. Suppose max probability comes at index 25, we should know what index 25 corresponds to from the data set we are classifying.

Example: We ran the classification of a zebra image, data set: ImageNet. Now we got probability maximum at index 44. Then we saw the index map of ImageNet at 44 its zebra. So in such a way, we validate our results, and that's how accuracy is also defined. Because predictions should at least be in top 5 probabilities of which the image is estimated.

CHAPTER 3

Model used to solve ImageNet

3.1 Background

We used a model developed by DARKNET community for our project. Darknet is an open source neural network framework written in C and CUDA. Darknet community studied the models like Alexnet, VGG-16, and ResNet and developed a simple version of these models to solve ImageNet.

AlexNet is the name of a convolutional neural network, competed in the ImageNet Large Scale Visual Recognition Challenge. The network achieved a top-5 error of 15.3 percent, more than 10.8 percentage. It was run using GPUs.

The darknet reference model is designed to be small but powerful. It attains the comparable top-1 and top-5 performance as AlexNet but with $1/10^{\text{th}}$ the parameters. Darknet uses mostly convolutional layers without the large fully connected layers at the end. It is about twice as fast as AlexNet on CPU. Darknet has also made models after studying Oxford's renowned Visual Geometry Group (VGG) model and Google's ResNet model.

Model	Top-1	Top-5	Ops	Weight
AlexNet	57.0	80.3	2.27 Bn	238 MB
Resnet 50	75.8	92.9	9.74 Bn	87 MB
VGG-16	70.5	90.0	30.94 Bn	528 MB
Darknet Reference	61.1	83.0	0.96 Bn	28 MB
Darknet19	72.9	91.2	7.29 Bn	80 MB

Table 3.1: Various models with their comparison in three parameters

Model	Top-1	Top-5	Ops	Weight
AlexNet	57.0	80.3	2.27 Bn	238 MB
Darknet Reference	61.1	83.0	0.96 Bn	28 MB
SqueezeNet	57.5	80.3	2.17 Bn	4.8 MB
Tiny Darknet	58.7	81.7	0.98 Bn	4.0 MB

Table 3.2: Less MB model file comparison with other three parameters

We initially chose darknet19, one of the reference models of DARKNET community since the accuracy was higher compared to the darknet reference model.

This model requires 20+ convolution layers and creates a very deep CNN, with many hidden layers. This made the cost in terms of size and memory to go up by a significant amount. On further analysis, we found there exist a small version of darknet with less complexity of implementation on hardware.

SqueezeNet is a DNN made with less MB weights. And when most high-quality images are 10MB or more, why do we care if our models are 5 MB or 50 MB? We want a small model that's FAST; So Darknet reference network works. It's only 28 MB, but more importantly, it's only 800 million floating point operations. The original Alexnet is 2.3 billion. Darknet is 2.9 times faster, and it's small, and it's 4 percent more accurate.

So what about SqueezeNet? Sure the weights are only 4.8 MB, but a forward pass is still 2.2 billion operations. So to balance out SqueezeNet, the community came up with Tiny darknet.

So what advantage does Tiny darknet gives us? It solves the problem of having more size, i.e., more MB weight file. It has comparable operation calls. Finally, the accuracy is also equivalent of darknet reference model hence becomes the best solution on how to solve ImageNet dataset with DNN, which has less complexity and can easily implement on hardware.

3.2 Tiny Darknet

Smaller CNN architectures offer three advantages:

(1) Less communication across servers during distributed training.

(2) Less bandwidth to export a new model from the cloud to an autonomous car.

(3) More feasible to set up on FPGAs and other hardware with limited on-chip memory. To provide all of these advantages, we propose a small CNN architecture.

It is preferable to work with Tiny darknet as:

- **More efficient distributed training.** For training, communication overhead is directly proportional to the number of parameters in the model. Smaller models require less communication, making frequent updates more feasible.
- **Feasible FPGA and embedded deployment.** Sufficiently small model could be stored directly on the FPGA instead of being bottle-necked by memory bandwidth. Deploying CNN on Application-Specific Integrated Circuits (ASICs), a sufficiently small model could be stored on-chip directly. FPGAs often have less than 10MB of on-chip memory and no off-chip memory or storage.

To make Tiny darknet, three main strategies were used in its design:

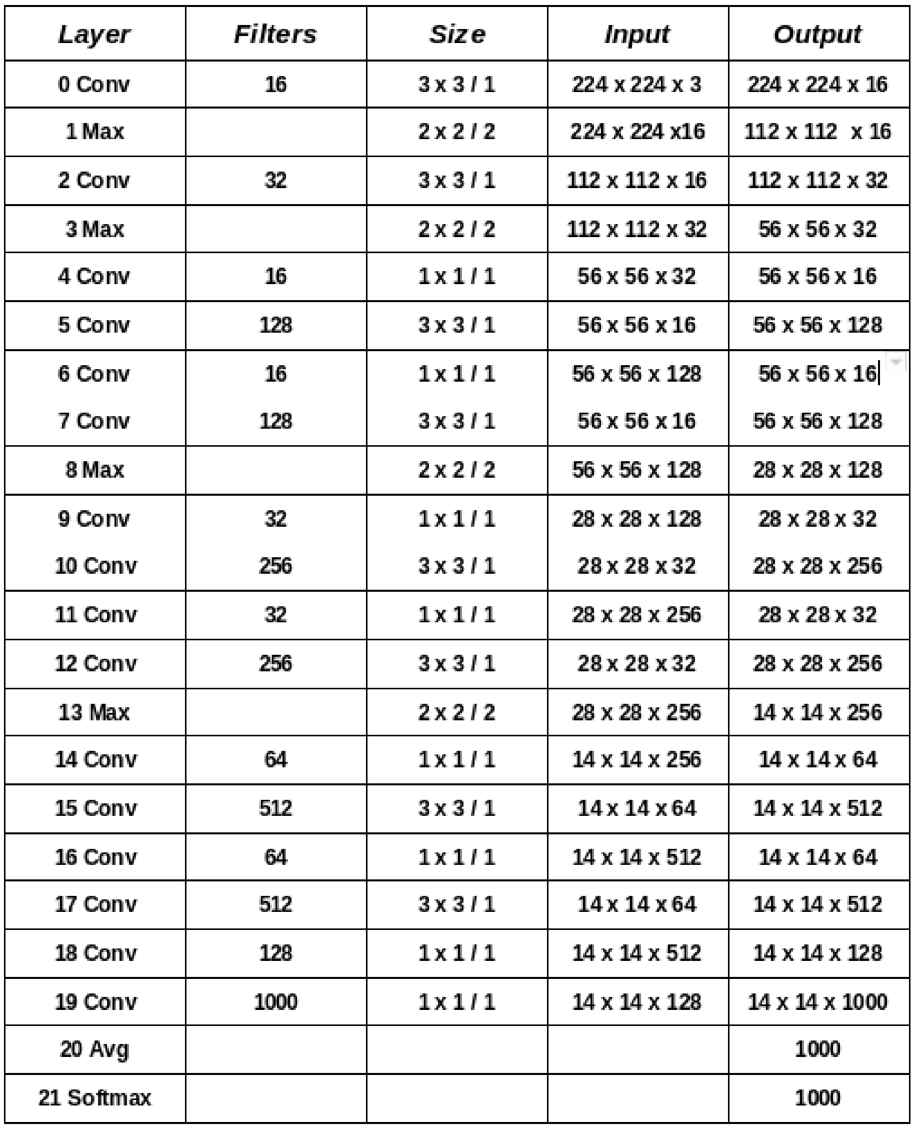
Strategy 1: Replace 3x3 filters with 1x1 filters. A 1x1 filter has 9X fewer parameters than a 3x3 filter.

Strategy 2: Decrease the number of input channels to 3x3 filters by downsampling. The total quantity of parameters in this layer is equal to (number of input channels) x (number of filters) x (3x3). So, to maintain a small total number of parameters in a CNN, it is important not only to decrease the number of 3x3 filters. The size of the input data (e.g., 224x224 images). Then the choice of layers in which to downsample in the CNN architecture is engineered into CNN architectures by setting the (stride > 1) in some of the pooling layers. Here downsampling is referred to as pooling discussed in Chapter 2

Strategy 3: No fully connected layer at the end.This reduces the number of weights significantly. (This was adjusted in training by some optimization method).

3.3 Architecture explanation

As in figure 3.1, Tiny darknet has 22 layers. The details are shown there. Convolution and max-pooling are the main layers of the architecture. 22nd layer is softmax, which we have discussed in previous chapters.



Layer	Filters	Size	Input	Output
0 Conv	16	3 x 3 / 1	224 x 224 x 3	224 x 224 x 16
1 Max		2 x 2 / 2	224 x 224 x16	112 x 112 x 16
2 Conv	32	3 x 3 / 1	112 x 112 x 16	112 x 112 x 32
3 Max		2 x 2 / 2	112 x 112 x 32	56 x 56 x 32
4 Conv	16	1 x 1 / 1	56 x 56 x 32	56 x 56 x 16
5 Conv	128	3 x 3 / 1	56 x 56 x 16	56 x 56 x 128
6 Conv	16	1 x 1 / 1	56 x 56 x 128	56 x 56 x 16
7 Conv	128	3 x 3 / 1	56 x 56 x 16	56 x 56 x 128
8 Max		2 x 2 / 2	56 x 56 x 128	28 x 28 x 128
9 Conv	32	1 x 1 / 1	28 x 28 x 128	28 x 28 x 32
10 Conv	256	3 x 3 / 1	28 x 28 x 32	28 x 28 x 256
11 Conv	32	1 x 1 / 1	28 x 28 x 256	28 x 28 x 32
12 Conv	256	3 x 3 / 1	28 x 28 x 32	28 x 28 x 256
13 Max		2 x 2 / 2	28 x 28 x 256	14 x 14 x 256
14 Conv	64	1 x 1 / 1	14 x 14 x 256	14 x 14 x 64
15 Conv	512	3 x 3 / 1	14 x 14 x 64	14 x 14 x 512
16 Conv	64	1 x 1 / 1	14 x 14 x 512	14 x 14 x 64
17 Conv	512	3 x 3 / 1	14 x 14 x 64	14 x 14 x 512
18 Conv	128	1 x 1 / 1	14 x 14 x 512	14 x 14 x 128
19 Conv	1000	1 x 1 / 1	14 x 14 x 128	14 x 14 x 1000
20 Avg				1000
21 Softmax				1000

Figure 3.1: Architecture of Tiny darknet

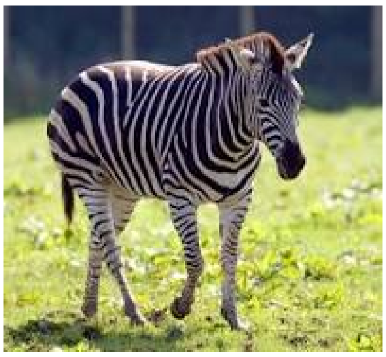
CHAPTER 4

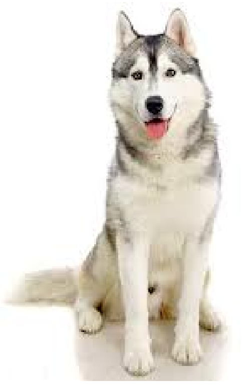
Software implementation of Tiny darknet

4.1 Software results

In the previous chapter, we understood why Tiny darknet should be used, with its complete architectural understanding. In our project, we took the architecture model file of Tiny darknet and hard-coded it in MATLAB initially. After some result analysis, we wrote our own C code based(taking parts of the code) from the main Darknet reference C code and made Tiny darknet architecture model.

Following are some snippets of the results obtained for specific images.


(a) zebra


(b) husky dog

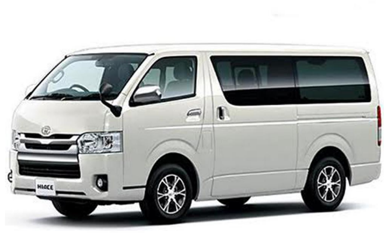
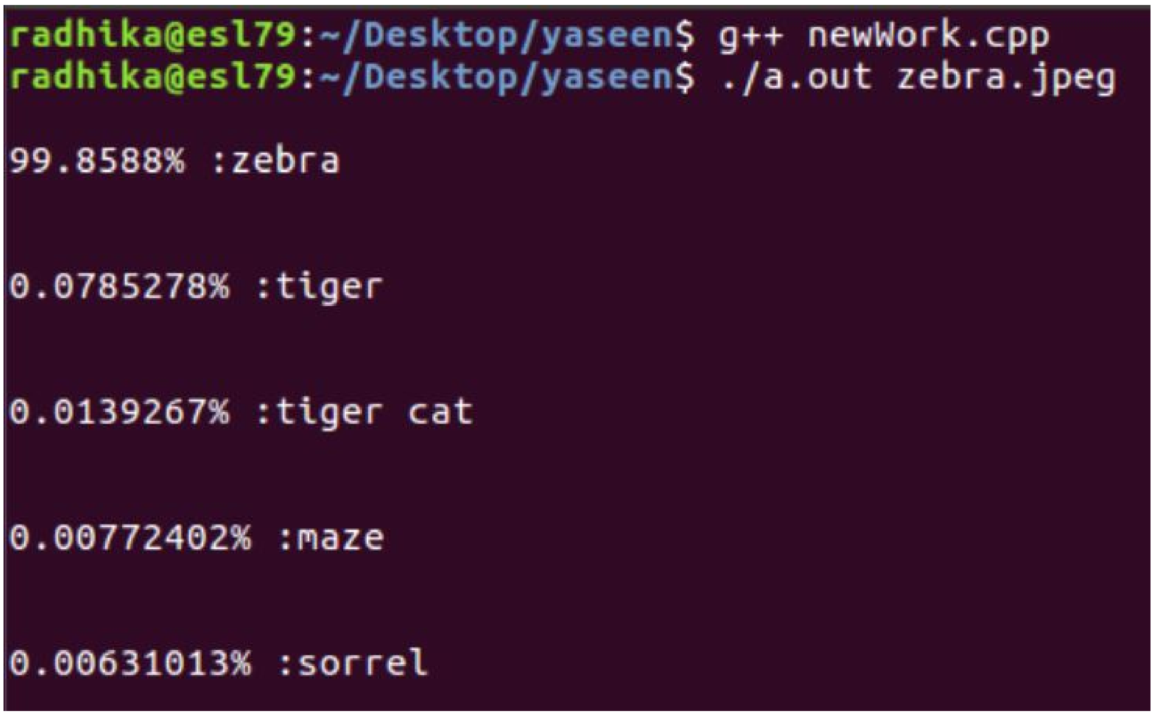

(c) van

Figure 4.1: Images given as input

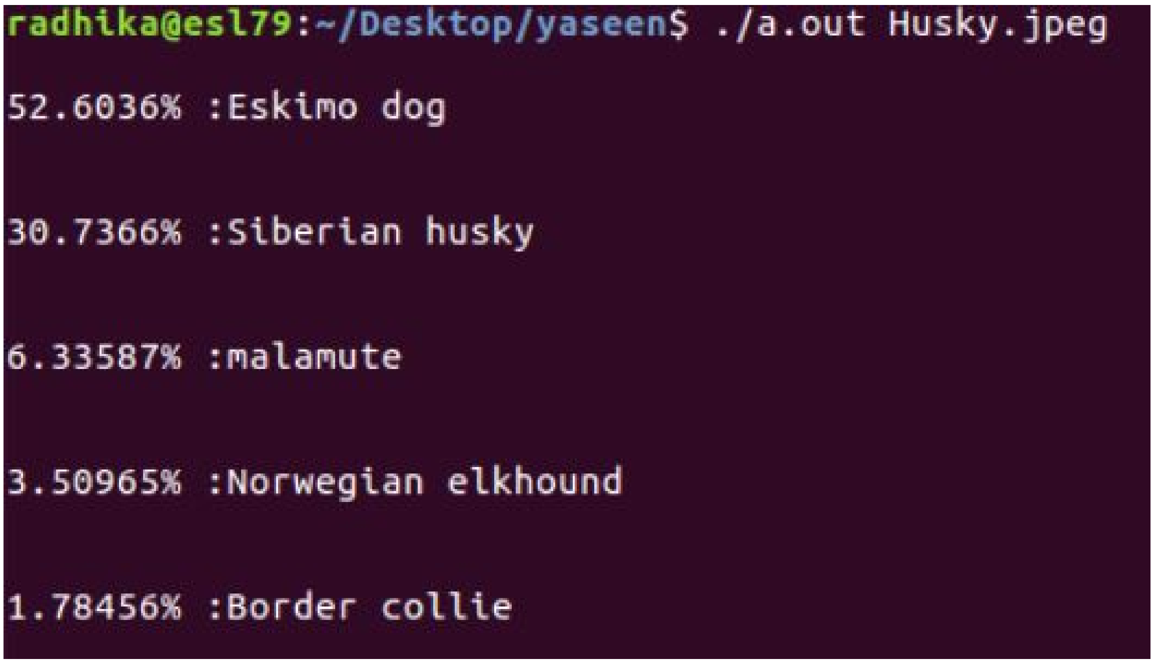
Figure 4.2: Results

(a) zebra



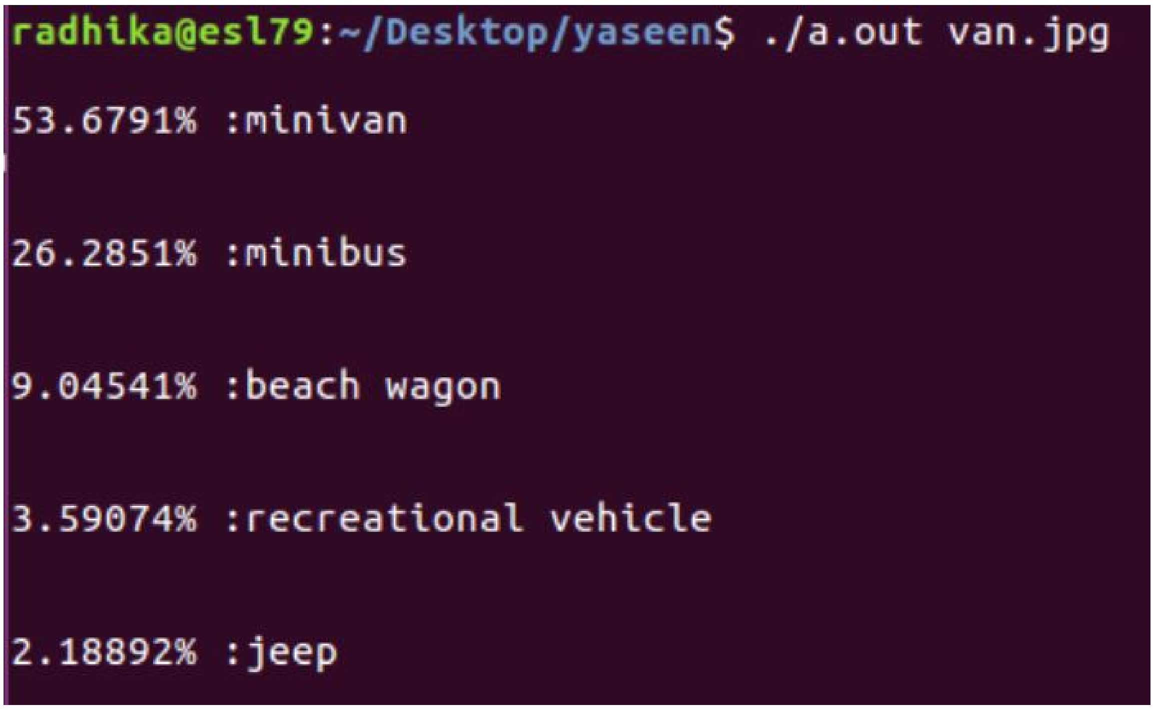
```
radhika@esl79:~/Desktop/yaseen$ g++ newWork.cpp
radhika@esl79:~/Desktop/yaseen$ ./a.out zebra.jpeg

99.8588% :zebra


0.0785278% :tiger


0.0139267% :tiger cat


0.00772402% :maze


0.00631013% :sorrel
```

(b) husky dog



```
radhika@esl79:~/Desktop/yaseen$ ./a.out Husky.jpeg

52.6036% :Eskimo dog


30.7366% :Siberian husky


6.33587% :malamute


3.50965% :Norwegian elkhound


1.78456% :Border collie
```

(c)van



```
radhika@esl79:~/Desktop/yaseen$ ./a.out van.jpg

53.6791% :minivan


26.2851% :minibus


9.04541% :beach wagon


3.59074% :recreational vehicle


2.18892% :jeep
```

CHAPTER 5

Hardware and Environment

5.1 Zedboard

Before we move on to the hardware implementation, we must understand the hardware we used in this project.

The device that was targeted for the implementation was Zedboard, which belongs to the Zynq-7000 family of SoCs. Zed-board is typically used in an academic institution as it is easily configurable for embedded system development. The configuration is done with the help of SDSoC. Zedboard has ARM A9 Cortex processor with Artix 7 FPGA included with it. It has external DDR3 memory of 512MB and on-chip BRAM. It uses the AXI3 bus interface for transferring data from PS to PL.

So whenever code is compiled on Zedboard, it will run on its processor. To use FPGA, we must specify which function to be pushed on hardware via SDSoc environment.

5.2 SDSoC and HLS

This is used to configure the hardware for embedded systems development. The Xilinx SDSoC development environment is a member of the Xilinx SDx family that provides like C/C++ programming experience including an easy to use Eclipse

IDE and a comprehensive design environment for heterogeneous Zynq All Programmable SoC. Complete with the industrys first C/C++ full-system optimizing compiler, SDSoC delivers system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming.

This tool enables us to toggle any function on hardware or software as and when required. Once we decide which functions to keep on HW and which to compile on SW, we have to build the whole code, and SDSoC will provide us with a file enable of running on Zedboard with functions to run on FPGA and processor separately.

The SDSoC environment is backed up with the HLS tool. HLS provides a detailed report of how the model code will run on hardware. How and what kind of resources the system will use on hardware. This gives deep insight on how to model our code to increase its specifications, like time and memory management.

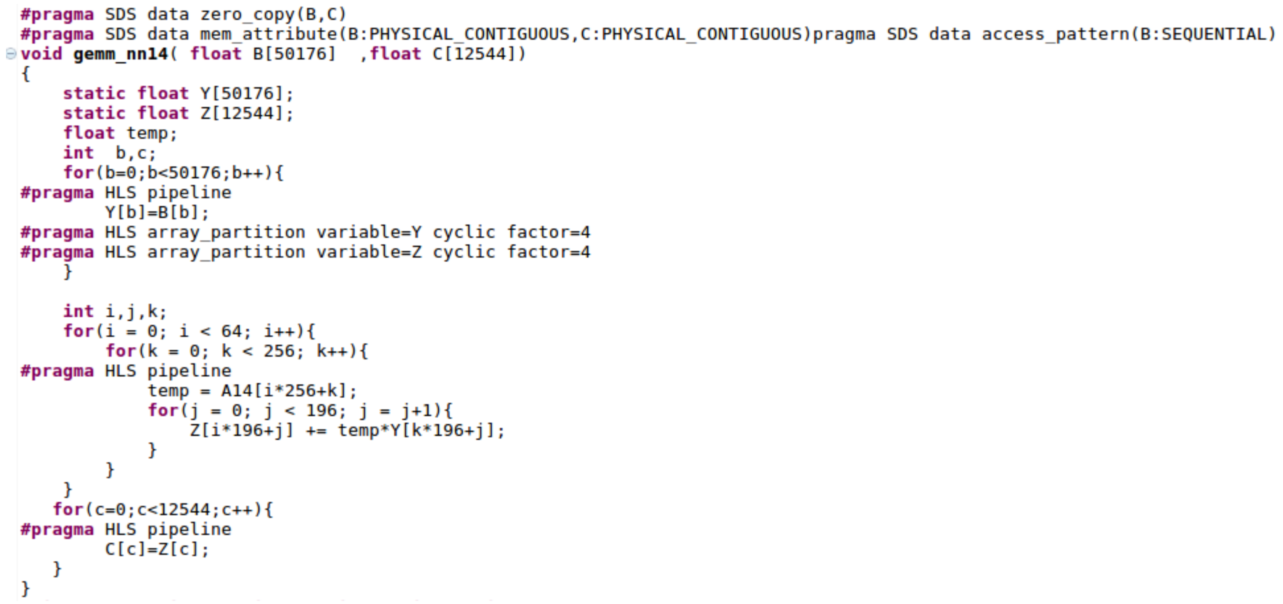
CHAPTER 6

Hardware implementation of Tiny darknet

6.1 Making a hardware function

In chapter 4, we saw the software implementation of the code. Our project aim was to move some of the computations to be done onto hardware. For this, we identified the essential function of the model: convolution and moved it in on hardware.

To achieve, we need to toggle function on hardware as in move it to FPGA of Zedboard discussed in chapter 5. Figure 6.1, is the function pushed onto hardware.



```
#pragma SDS data zero_copy(B,C)
#pragma SDS data mem_attribute(B:PHYSICAL_CONTIGUOUS,C:PHYSICAL_CONTIGUOUS)pragma SDS data access_pattern(B:SEQUENTIAL)
void gemm_nn14( float B[50176]  ,float C[12544])
{
    static float Y[50176];
    static float Z[12544];
    float temp;
    int  b,c;
    for(b=0;b<50176;b++){
#pragma HLS pipeline
        Y[b]=B[b];
#pragma HLS array_partition variable=Y cyclic factor=4
#pragma HLS array_partition variable=Z cyclic factor=4
    }

    int i,j,k;
    for(i = 0; i < 64; i++){
        for(k = 0; k < 256; k++){
#pragma HLS pipeline
            temp = A14[i*256+k];
            for(j = 0; j < 196; j = j+1){
                Z[i*196+j] += temp*Y[k*196+j];
            }
        }
    }
   for(c=0;c<12544;c++){
#pragma HLS pipeline
        C[c]=Z[c];
   }
}
```

Figure 6.1: Convolution function containing A14: Weights array, B: Previous layer array and C: Next layer array.

6.2 Hardware results

These were the results of image recognition when we ran the code on Zedboard via GTK term(terminal interface for Zedboard). **The image used is HUSKY figure 4.1 (b), for all experimentation and results.**

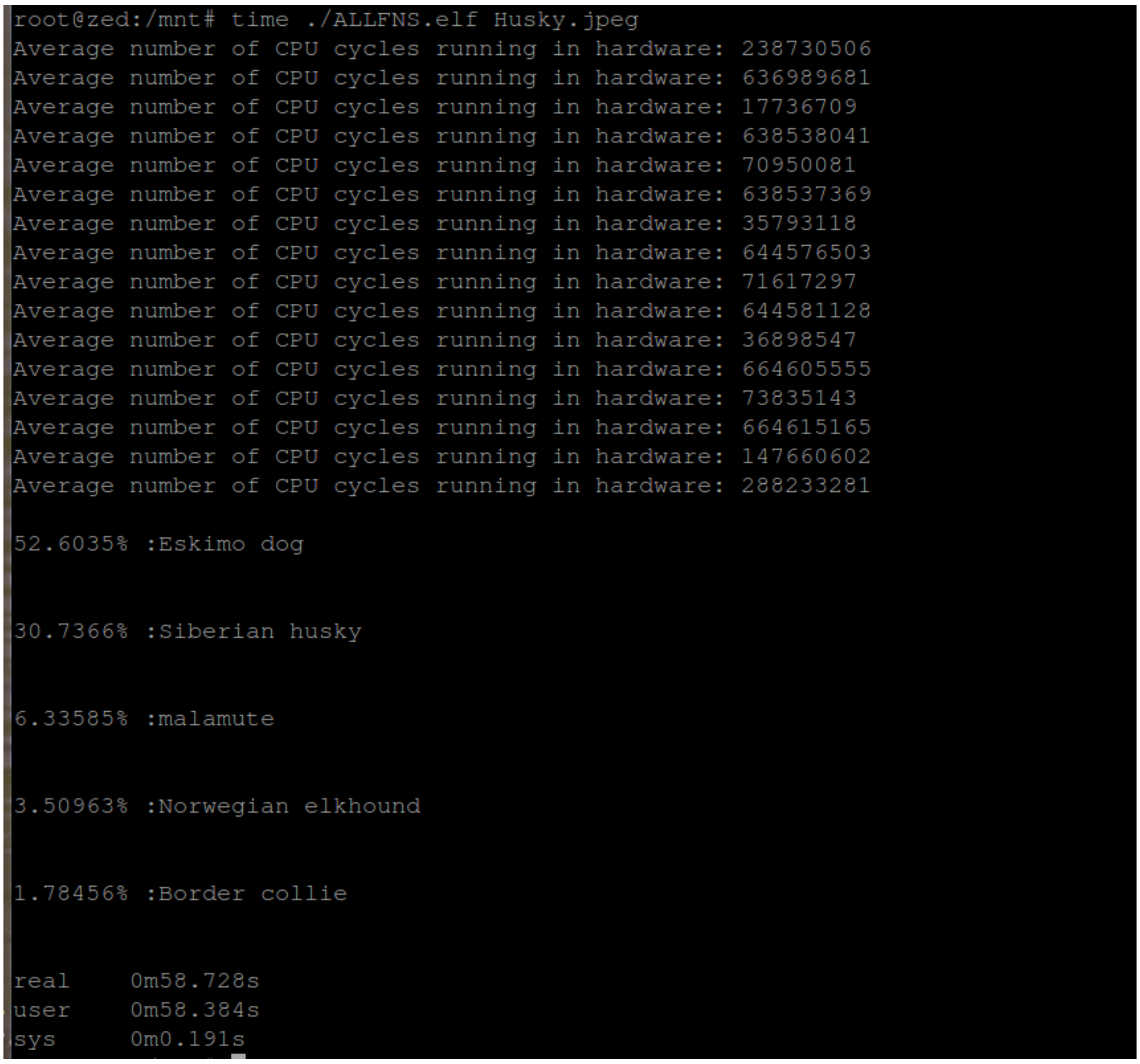
If we are running any function on the hardware and it requires memory, then the memory should be less, because we want all our hardware code to run on programmable logic. Programmable logic involves using BRAMs. *(BRAM: Block random access memory is a programmable memory attached with FPGA. It can be allocated piece by piece, each has its address and data lines and can be read/written to, all in the same clock cycles synchronously.)* The limitation is BRAMs are available in small size. If the function put on hardware requires memory than available on-chip(BRAMs), then it allocates all memory on DRAMs, which is external to FPGA. This leads to an increase in time of computation cause computation has to happen in FPGA while memory is external, and each computation requires memory transfer, which adds up to time.

Here the main observation is that when we ran our model code on Zedboard with convolution function toggled on hardware: FPGA. It requires more time to execute than compared to having run the whole model code on software. Why does this happen? The reason for this is less on-chip memory. The amount of BRAMs required to run convolution function are too high in some layers, and hence, FPGA has to rely on its external memory. This external memory transfer of data results in the extra overhead of time, which is reflected in the result.

Also, there is an HLS report in fig 6.3 which clearly describes by how much we are short of BRAMs. When there is enough BRAM space available then only we can move our arrays used in hardware to FPGA memory.

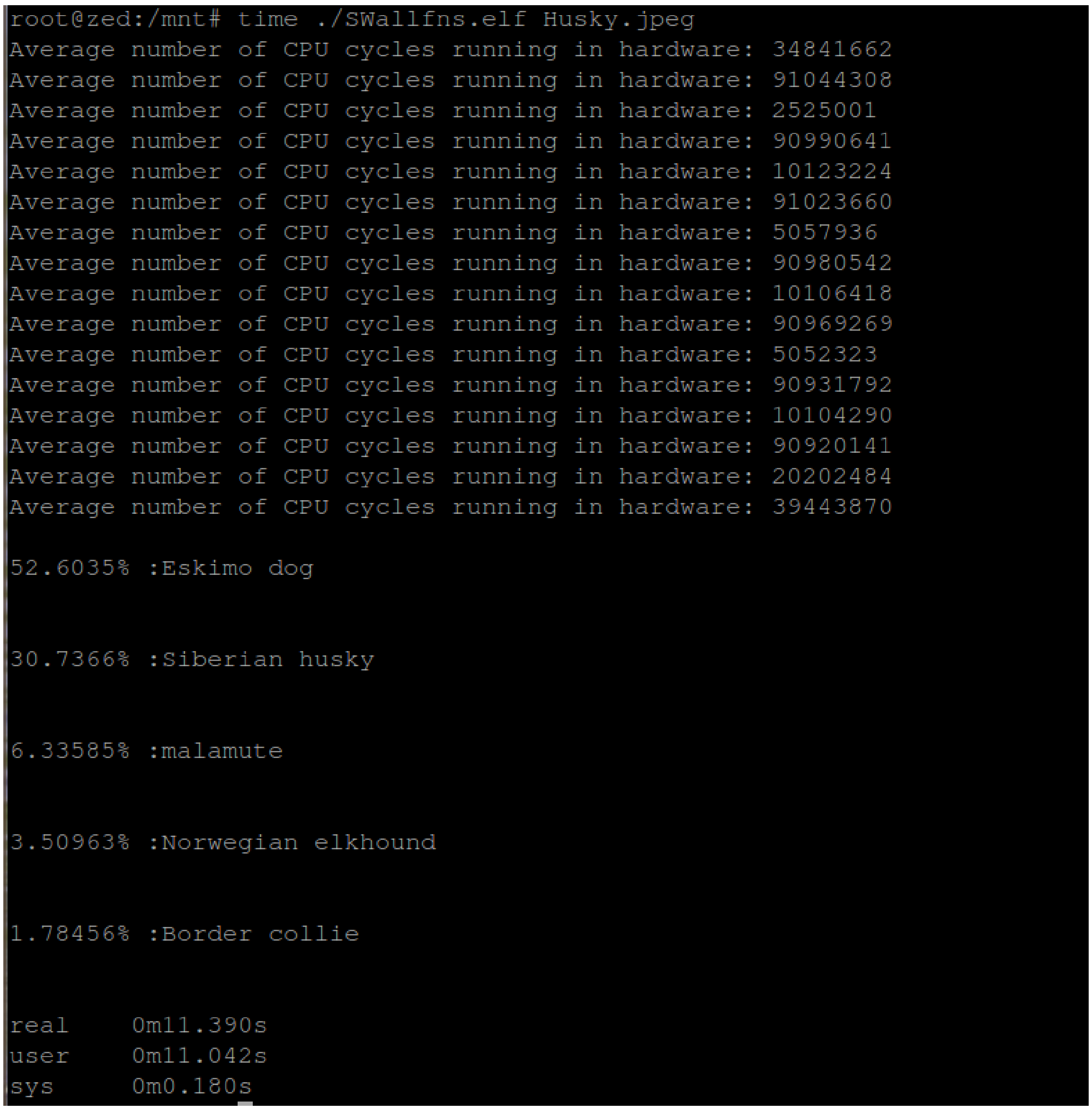
Figure 6.2: Significant difference in time taken for the code to give result

(a) Hardware analysis

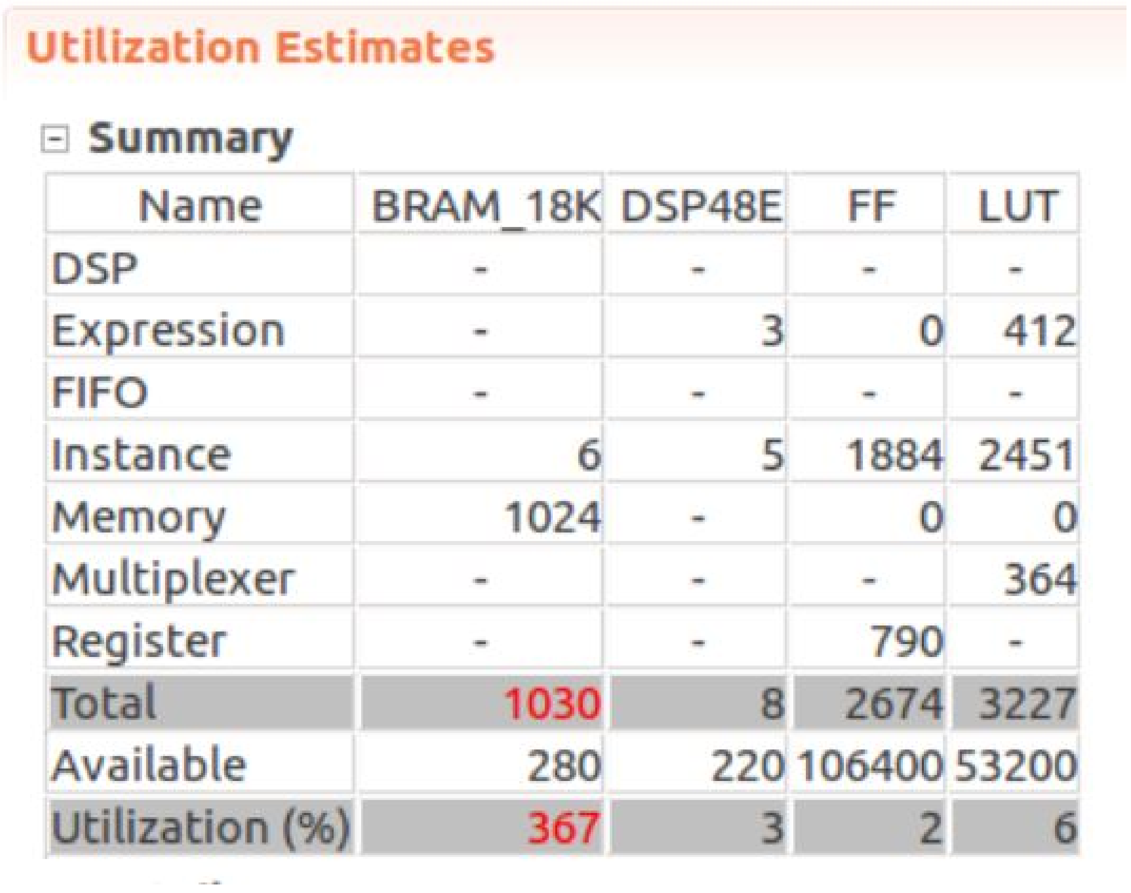


```
root@zed:/mnt# time ./ALLFNS.elf Husky.jpeg
Average number of CPU cycles running in hardware: 238730506
Average number of CPU cycles running in hardware: 636989681
Average number of CPU cycles running in hardware: 17736709
Average number of CPU cycles running in hardware: 638538041
Average number of CPU cycles running in hardware: 70950081
Average number of CPU cycles running in hardware: 638537369
Average number of CPU cycles running in hardware: 35793118
Average number of CPU cycles running in hardware: 644576503
Average number of CPU cycles running in hardware: 71617297
Average number of CPU cycles running in hardware: 644581128
Average number of CPU cycles running in hardware: 36898547
Average number of CPU cycles running in hardware: 664605555
Average number of CPU cycles running in hardware: 73835143
Average number of CPU cycles running in hardware: 664615165
Average number of CPU cycles running in hardware: 147660602
Average number of CPU cycles running in hardware: 288233281

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie


real    0m58.728s
user    0m58.384s
sys     0m0.191s
```

(b)Software analysis



```
root@zed:/mnt# time ./SWallfns.elf Husky.jpeg
Average number of CPU cycles running in hardware: 34841662
Average number of CPU cycles running in hardware: 91044308
Average number of CPU cycles running in hardware: 2525001
Average number of CPU cycles running in hardware: 90990641
Average number of CPU cycles running in hardware: 10123224
Average number of CPU cycles running in hardware: 91023660
Average number of CPU cycles running in hardware: 5057936
Average number of CPU cycles running in hardware: 90980542
Average number of CPU cycles running in hardware: 10106418
Average number of CPU cycles running in hardware: 90969269
Average number of CPU cycles running in hardware: 5052323
Average number of CPU cycles running in hardware: 90931792
Average number of CPU cycles running in hardware: 10104290
Average number of CPU cycles running in hardware: 90920141
Average number of CPU cycles running in hardware: 20202484
Average number of CPU cycles running in hardware: 39443870

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie


real    0m11.390s
user    0m11.042s
sys     0m0.180s
```


Utilization Estimates

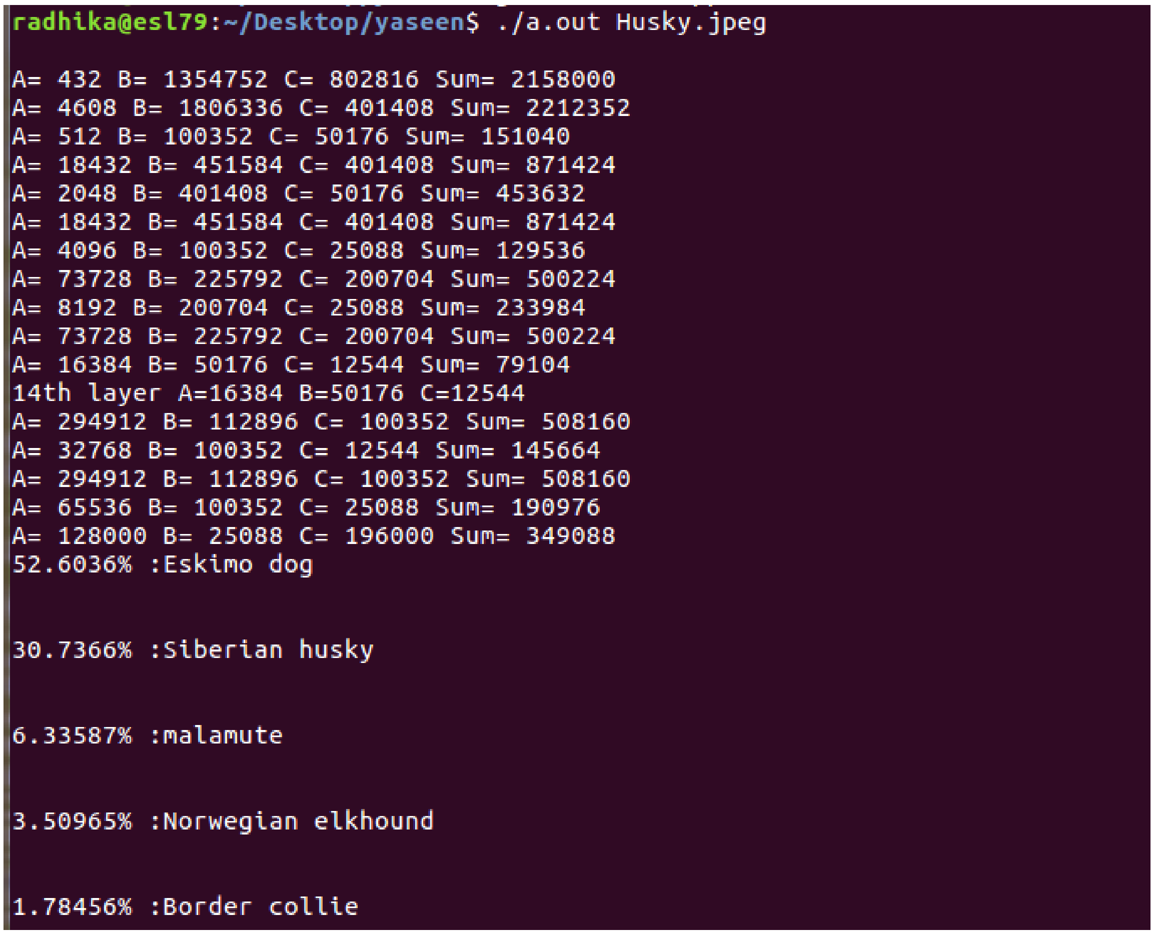
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	412
FIFO	-	-	-	-
Instance	6	5	1884	2451
Memory	1024	-	0	0
Multiplexer	-	-	-	364
Register	-	-	790	-
Total	1030	8	2674	3227
Available	280	220	106400	53200
Utilization (%)	367	3	2	6

Figure 6.3: HLS Report clearly stating that there is short of BRAM(on-chip) memory when function is ran on FPGA

6.3 Hardware optimization

Pushing the whole convolution operation on hardware implies that all convolution layers we need to go on-chip, which is a limitation. Now given our model has max pooling in some layers the size of memory required to do convolution with weights goes down with progress from layer after layer. So, we just pushed one single convolution layer which required very less memory compared to other layers on hardware. In reference to figure 6.4, we found out that 14th layer takes the least amount of memory.



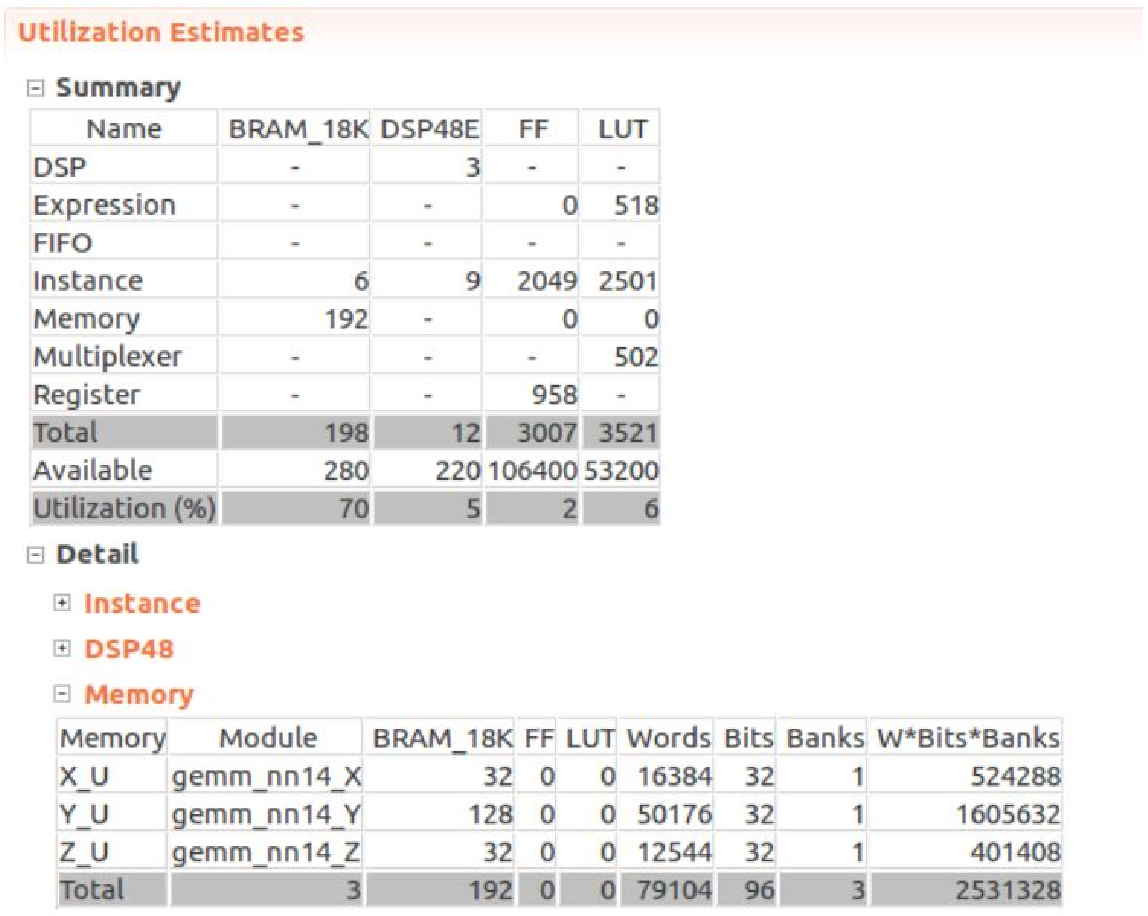
```
radhika@esl79:~/Desktop/yaseen$ ./a.out Husky.jpeg

A= 432 B= 1354752 C= 802816 Sum= 2158000
A= 4608 B= 1806336 C= 401408 Sum= 2212352
A= 512 B= 100352 C= 50176 Sum= 151040
A= 18432 B= 451584 C= 401408 Sum= 871424
A= 2048 B= 401408 C= 50176 Sum= 453632
A= 18432 B= 451584 C= 401408 Sum= 871424
A= 4096 B= 100352 C= 25088 Sum= 129536
A= 73728 B= 225792 C= 200704 Sum= 500224
A= 8192 B= 200704 C= 25088 Sum= 233984
A= 73728 B= 225792 C= 200704 Sum= 500224
A= 16384 B= 50176 C= 12544 Sum= 79104
14th layer A=16384 B=50176 C=12544
A= 294912 B= 112896 C= 100352 Sum= 508160
A= 32768 B= 100352 C= 12544 Sum= 145664
A= 294912 B= 112896 C= 100352 Sum= 508160
A= 65536 B= 100352 C= 25088 Sum= 190976
A= 128000 B= 25088 C= 196000 Sum= 349088
52.6036% :Eskimo dog


30.7366% :Siberian husky


6.33587% :malamute


3.50965% :Norwegian elkhound


1.78456% :Border collie
```

Figure 6.4: Memory analysis of each layer

In reference to fig 6.5. We observed that the time overhead down significantly when we run one layer on hardware or only on software. Since now the on-chip memory is available in abundance given size of the memory required by this layer is less.

Next, we pipe-lined the *for* loops to extract the pipelining feature of FPGA hardware. In reference 6.1, will show clearly where we used the HLS pipelining. We got the better speed up in clock cycles about fig. 6.7 in comparison to reference fig. 6.6. Then the next approach we implemented was array partitioning by a factor of 2 and 4, and the results are in figures 6.7 and 6.10.

In pipelining each clock cycle, one instruction is run in parallel. While when we do array partitioning and the pipeline, we can do simultaneously many operations in parallel. In reference to fig. 6.8


Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	-	0	518
FIFO	-	-	-	-
Instance	6	9	2049	2501
Memory	192	-	0	0
Multiplexer	-	-	-	502
Register	-	-	958	-
Total	198	12	3007	3521
Available	280	220	106400	53200
Utilization (%)	70	5	2	6

⊟ Detail

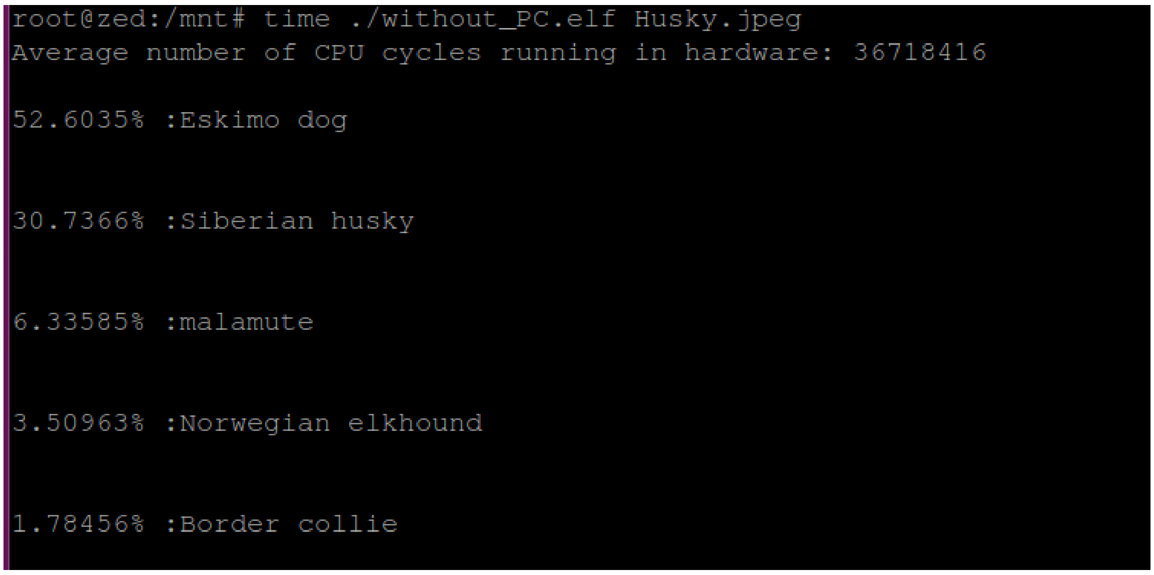
⊞ Instance

⊞ DSP48

⊟ Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
X_U	gemm_nn14_X	32	0	0	16384	32	1	524288
Y_U	gemm_nn14_Y	128	0	0	50176	32	1	1605632
Z_U	gemm_nn14_Z	32	0	0	12544	32	1	401408
Total	3	192	0	0	79104	96	3	2531328

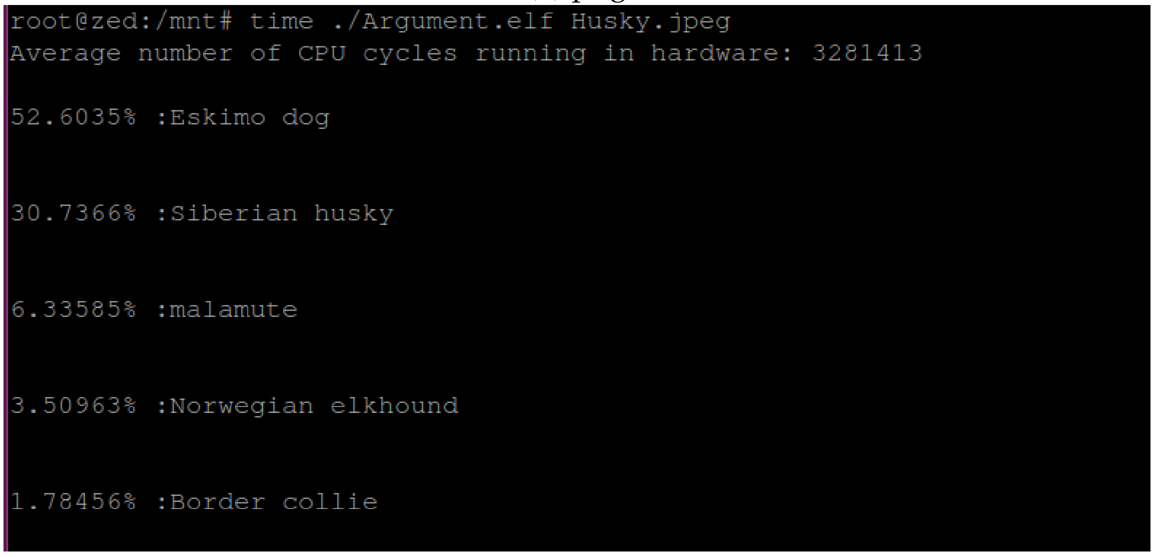
Figure 6.5: HLS Report clearly 14 layer uses BRAM(on-chip) memory optimally when function is ran on FPGA



```
root@zed:/mnt# time ./without_PC.elf Husky.jpeg
Average number of CPU cycles running in hardware: 36718416

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie
```

Figure 6.6: Hardware implementation without pipeline and use of BRAMs

(1).png



```
root@zed:/mnt# time ./Argument.elf Husky.jpeg
Average number of CPU cycles running in hardware: 3281413

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie
```

Figure 6.7: Hardware implementation with pipeline and use of BRAMs

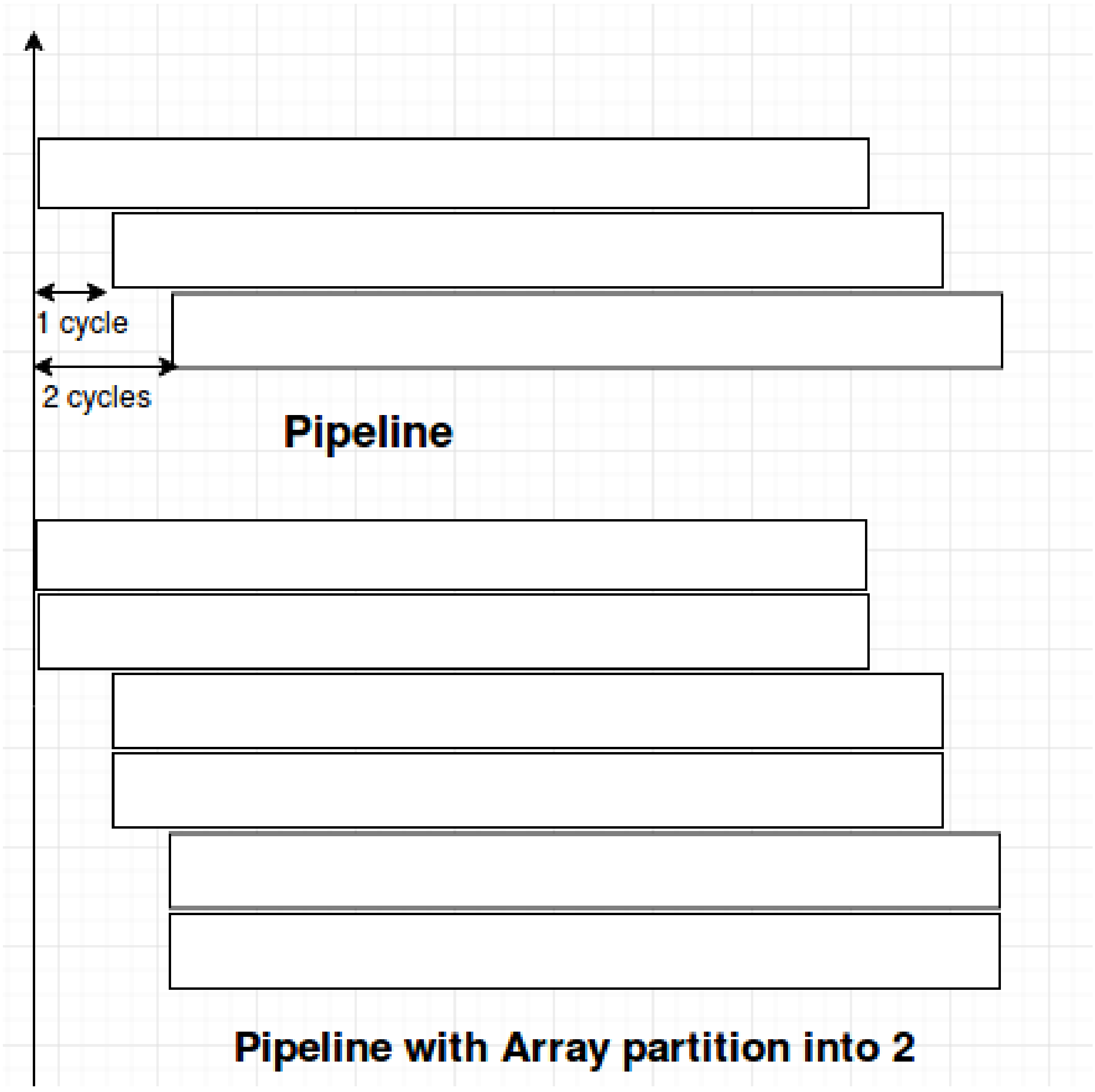
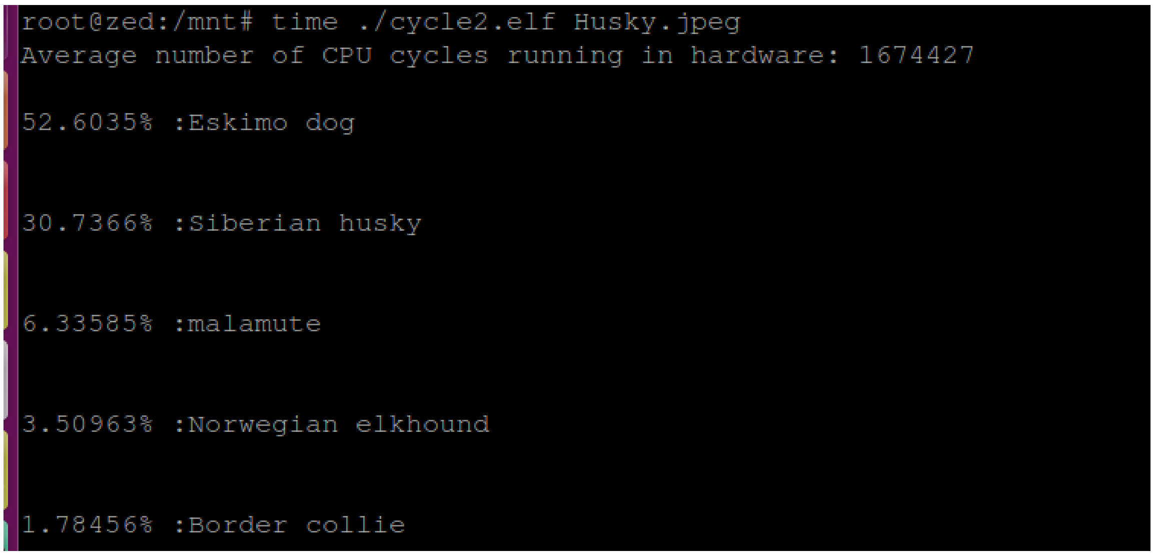
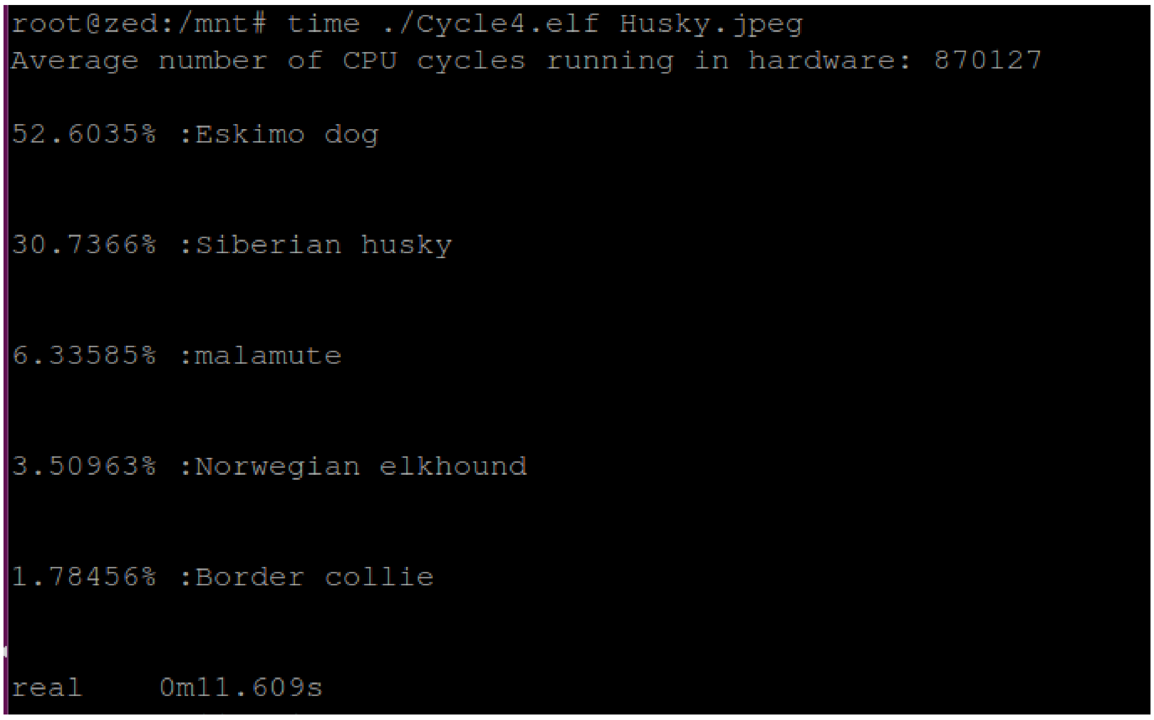


Figure 6.8: Array partitioning explained



```
root@zed:/mnt# time ./cycle2.elf Husky.jpeg
Average number of CPU cycles running in hardware: 1674427

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie
```

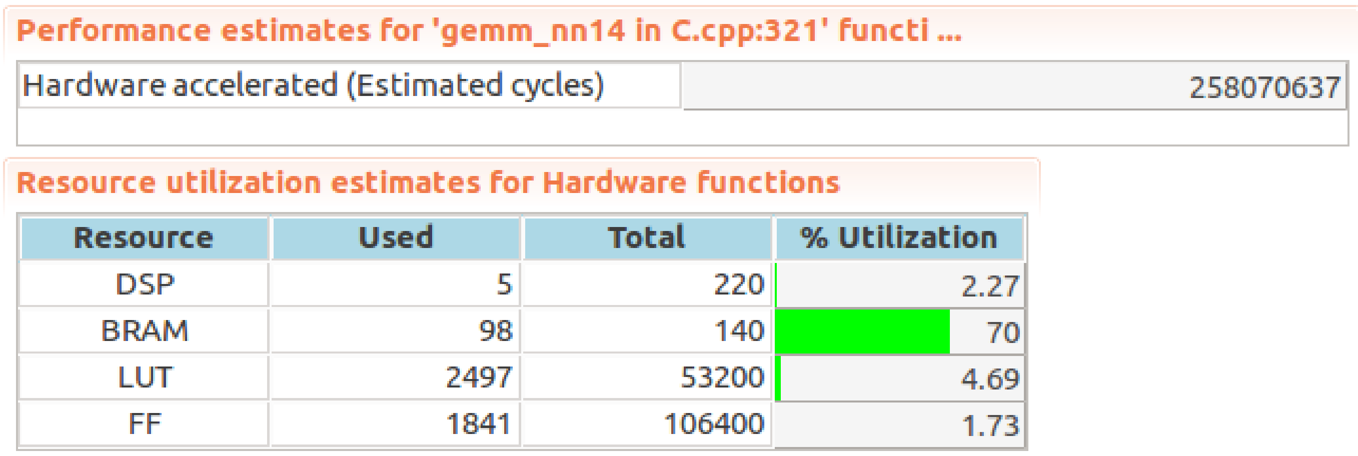
Figure 6.9: Array partitioning by factor of 2



```
root@zed:/mnt# time ./Cycle4.elf Husky.jpeg
Average number of CPU cycles running in hardware: 870127

52.6035% :Eskimo dog


30.7366% :Siberian husky


6.33585% :malamute


3.50963% :Norwegian elkhound


1.78456% :Border collie


real    0m11.609s
```

Figure 6.10: Array partitioning by factor of 4

Figure 6.11: Without pipeline resources utilization in hardware acceleration.



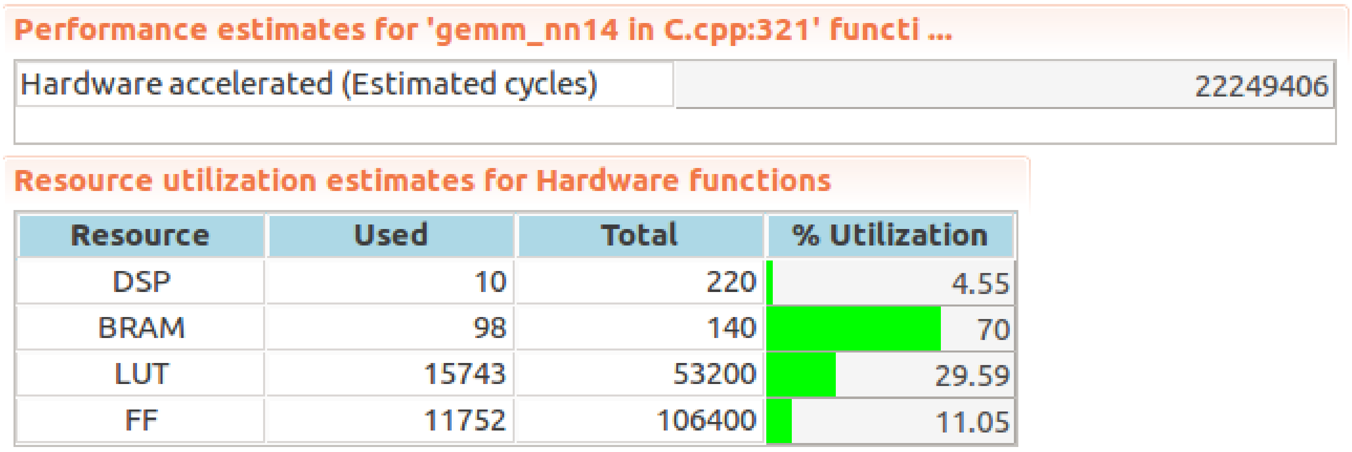
Performance estimates for 'gemm_nn14 in C.cpp:321' functi ...

Hardware accelerated (Estimated cycles)	258070637

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	5	220	2.27
BRAM	98	140	70
LUT	2497	53200	4.69
FF	1841	106400	1.73

Figure 6.12: With pipeline resources utilization in hardware acceleration.



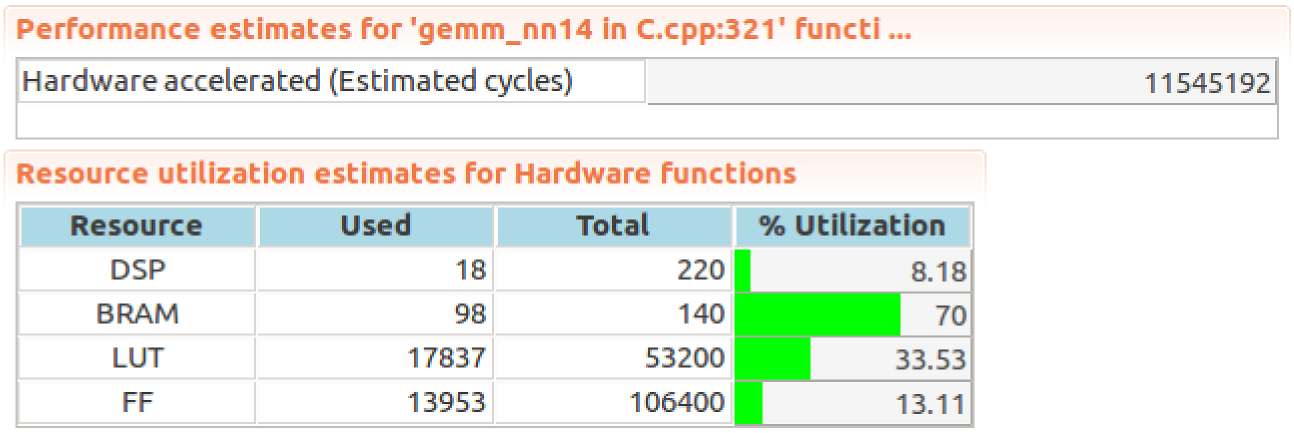
Performance estimates for 'gemm_nn14 in C.cpp:321' functi ...

Hardware accelerated (Estimated cycles)	22249406

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	98	140	70
LUT	15743	53200	29.59
FF	11752	106400	11.05

Figure 6.13: Array partitioning with factor 2 resources utilization in hardware acceleration.

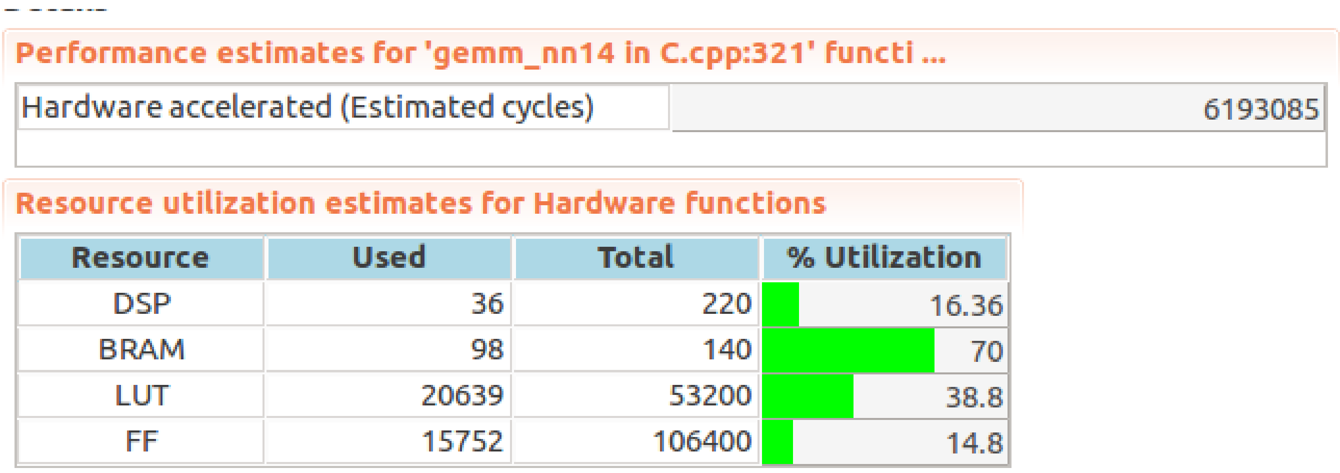

Performance estimates for 'gemm_nn14 in C.cpp:321' functi ...

Hardware accelerated (Estimated cycles)	11545192

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	18	220	8.18
BRAM	98	140	70
LUT	17837	53200	33.53
FF	13953	106400	13.11

Figure 6.14: Array partitioning with factor 4 resources utilization in hardware acceleration.


Performance estimates for 'gemm_nn14 in C.cpp:321' functi ...

Hardware accelerated (Estimated cycles)	6193085

Resource utilization estimates for Hardware functions

Resource	Used	Total	% Utilization
DSP	36	220	16.36
BRAM	98	140	70
LUT	20639	53200	38.8
FF	15752	106400	14.8

CHAPTER 7

Conclusion

7.1 Various experiments

Experiments	No. of CPU cycles running in Hardware	Time(uS)
SOFTWARE	**2579714**	**3868**
Without using Pipe-lining and BRAMs	36718416	55050
Using Pipe-lining and BRAMs	3281413	4920
Array partitioning by factor of 2	1674427	2510
Array partitioning by factor of 4	**870127**	**1304**

Table 7.1: Hardware acceleration experiment results

7.2 Summary

1. Size of BRAMs is limited; hence, we cannot put all convolution layers on FPGA.

Single layer hardware implementation is possible.

2. Array partitioning gives the best hardware acceleration.

3. Array partitioning can only be done on arrays stored on on-chip memory.

4. In our experiment, we can get **3X** acceleration on hardware.

REFERENCES

[1] N.M. Botros and M. Abdul-Aziz Dept. of Electr. Eng., Southern Illinois Univ., Carbondale, IL, USA *Hardware implementation of an artificial neural network using field programmable gate arrays (FPGA's)*
Proc. IEEE Int. Conf. on Neural Networks, vol. 2, pp. 1252-1257, 1993-March

[2] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer *SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE*. conference paper at ICLR 2017

[3] Mohammad Samragh, Mojan Javaheripi, Farinaz Koushanfar Department of Electrical and Computer Engineering, University of California San Diego *CodeX: Bit-Flexible Encoding for Streaming-based FPGA Acceleration of DNNs*

[4] *https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html*

[5] *https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks*

[6] *http://web.pdx.edu/ jduh/courses/Archive/geog481w07/Students/Ludwig$_I$mageConvolution.pdf*

[7] *https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148*

[8] *https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8*