Design and Implementation of Hardware accelerator for Deep Neural networks

ECN-300 : Lab Based Project

Presented by :

Shivanshu Raj Shrivastava
B.Tech III Year
Enrolment No. : 16116063
Department of Electronics and Communication Engineering
Indian Institute of Technology, Roorkee

Supervised by :

Dr. Brajesh Kumar Kaushik
Associate Professor
Department of Electronics and Communication Engineering
Indian Institute of Technology, Roorkee

CONTENT

1. Introduction

In computing, hardware acceleration is the use of computer hardware specially made to perform some functions more efficiently than is possible in software running on a general-purpose CPU.

Any transformation of data or routine that can be computed, can be calculated purely in software running on a generic CPU, purely in custom-made hardware, or in some mix of both. An operation can be computed faster in **application-specific hardware designed** or programmed to compute the operation than specified in software and performed on a general-purpose computer processor.

Greater RTL customization of hardware designs allows emerging architectures such as in-memory computing, transport triggered architectures (TTA) and networks-on-chip (NoC) to further benefit from increased locality of data to execution context, thereby reducing computing and communication latency between modules and functional units.

The implementation of computing tasks in hardware to decrease latency and increase throughput is known as **hardware acceleration**.

The Aim of this project is to develop a Hardware accelerator to realize a **Deep Neural Network** on FPGA for **Handwritten Character Recognition.**

2. Previous work and Motivation

The main source of motivation is the project I completed under the guidance of **DRDO** in **Inter IIT TechMeet 2018** where I developed a module through which Soldier's gear was updated so that **hand gesture communications is possible beyond range of their sight or in darkness**.

The module involved various sensors and **Raspberry Pi 3B** as a microprocessor to perform Deep Neural Network Calculations. A PCB was created to realize everything on a single chip.

I also developed a python model to do **Object recognition on Raspberry Pi** as a course project Under Prof. B.K. Kaushik.

In Both the project I realised the need of an ASIC which can replace Raspberry Pi to decrease latency and increase throughput.

In this project I tried to Implement a Hardware accelerator to Hardware accelerator to realize a Deep Neural Network on FPGA for Handwritten Character Recognition

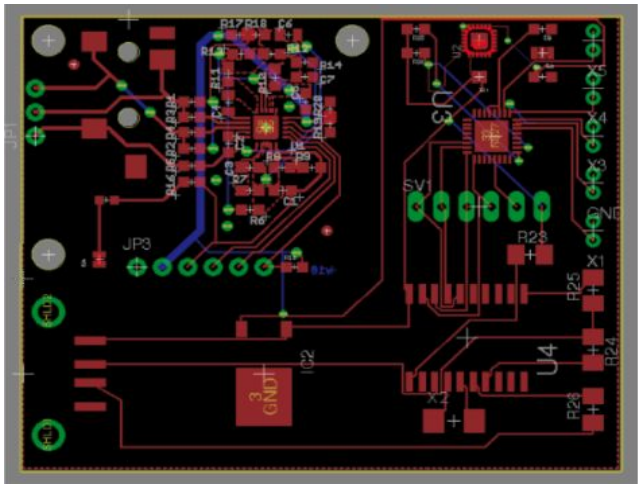

Fig. 1 - Final PCB for Technology of Soldier Support

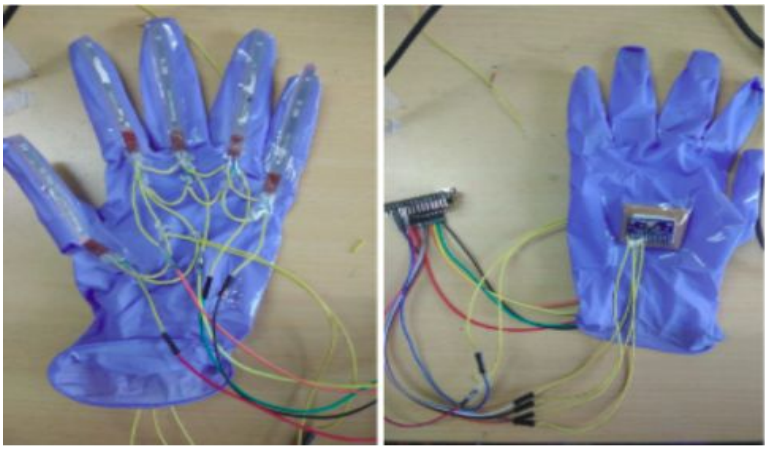

Fig. 2 - Hand Gesture Recognition Module

3. Implementation details

First A Deep Neural Network Model was created in Python and further optimized to realize it on a Hardware using Perceptron realization method on Digital Logic.

3.1. Deep Neural Network

Deep learning is a class of machine learning algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input.

learn in supervised and unsupervised manners. learn multiple levels of representations that correspond to different levels of abstraction, the levels form a hierarchy of concepts.

3.2. Feed-forward Neural Network

To reduce complexity a feed forward Neural Network was Implemented with the following implementation details.

A feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. As such, it is different from recurrent neural networks.

The feedforward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes and to the output nodes. There are no cycles or loops in the network.

Types of FNN:

Single layer perceptron (SLP)

Multi layer perceptron (MLP)

In this project MLP model is implemented whose details are provided in the next section.

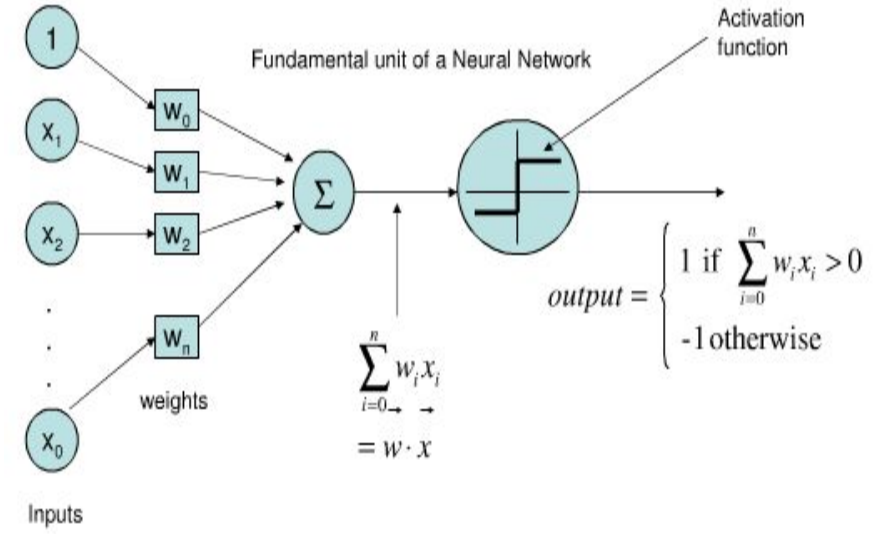

Fig. 3 Perceptron Model which is realized on hardware inside every neuron.

3.3. Hardware to Software conversation

The following property is being utilized to convert the Neural network generated into an Hardware. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits, x1

and x2. This requires computing the bitwise sum, x1⊕x2, as well as a carry bit which is set to 1 when both x1 and x2 are 1, i.e., the carry bit is just the bitwise product x1x2:

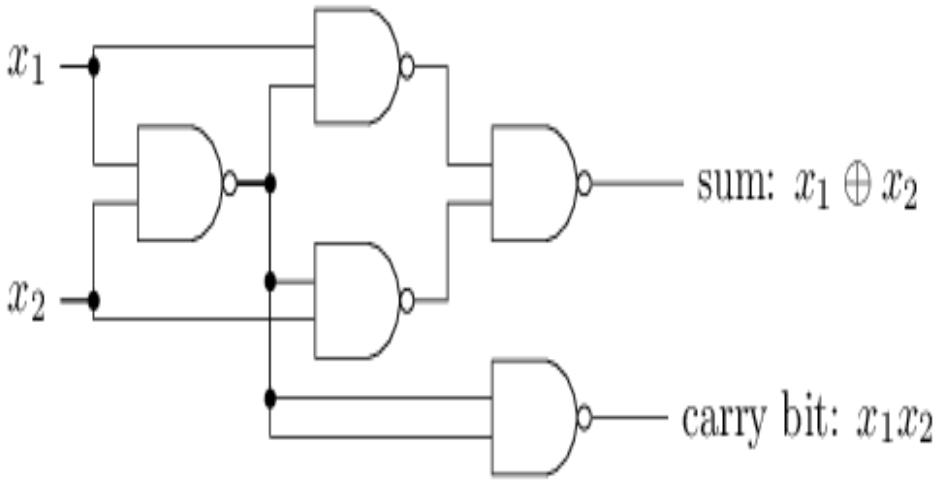

Fig. 4- Hardware Realization of Half Adder

To get an equivalent network of perceptrons we replace all the NAND gates by perceptrons with two inputs, each with weight −2, and an overall bias of 3. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:

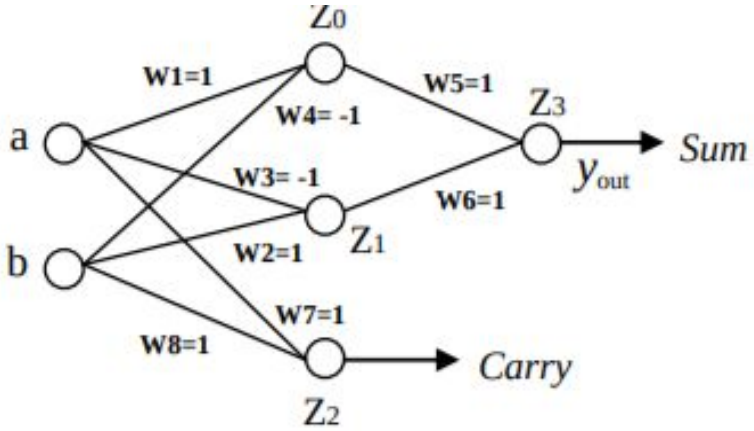

Fig. 5 - Perceptron Model of Half Adder

a	b	Z_0	Z_1	Z_2 [Carry]	Z_3 [Sum]
0	0	0	0	0	0
0	**1**	0	1	0	**1**
1	**0**	1	0	0	**1**
1	**1**	0	0	**1**	0

Fig.6 - Truth Table for Perceptron Model

3.4. The Multi-Layer Perceptron Algorithm

Initialisation:

Set all of the weights to small (positive and negative) random numbers

Training:

->Repeat: *for each input vector:

Forwards phase:

Compute the activation of each neuron j in hidden layer(s) using Eq.1 and Eq.2

->Work through the network until you get to the output layer neurons, which have activations

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \quad a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \quad h_\kappa = \sum_j a_j w_{j\kappa}$$

Eq. 1 **Eq. 2** **Eq.3**

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \qquad \delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa)$$

Eq. 4 **Eq.5**

->Backwards phase:

Compute the error at the output using Eq. 5

Recall:

->Use the Forwards phase in the training section above

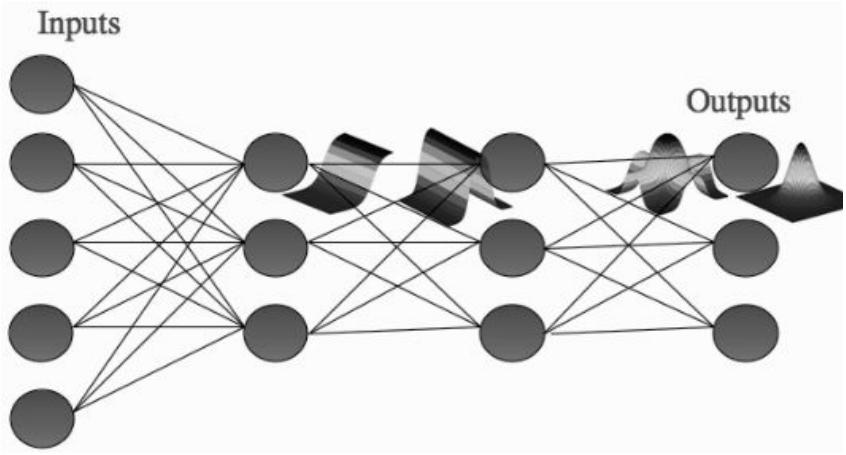

Fig.7 Graphical representation of Model training

3.5. Recognize Handwritten Digits using MLP

Developed a Neural Network and Train the model using Dataset provided by MNIST (Modified National Institute of Standards and Technology database).

Image specification: 28*28 .png, 2,40,000 training images, and 40,000 testing images

Neural Network specification:

Training input x as a 28×28=784-dimensional vector

Epoch = 30

Batch Size = 10

Used Mini Batch Stochastic for optimization

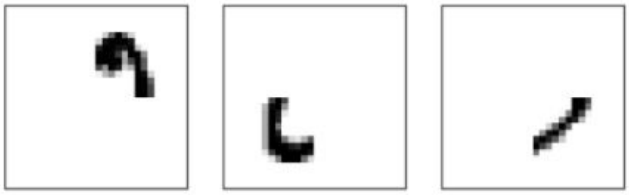

Fig. 8 Biasing of each neuron while training

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import MLP
>>> net = MLP.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Fig. 9 Script to run the training of Model

3.6. Designing the Hardware Accelerator

After repeating these steps for various network Architectures, I decided to work with 784*16*16*10 Neural Network model to utilize the fact that $16=2^4$ and to utilize 4 bits to handle each neuron. This is the output of Accuracy v/s Neurons for my Model.

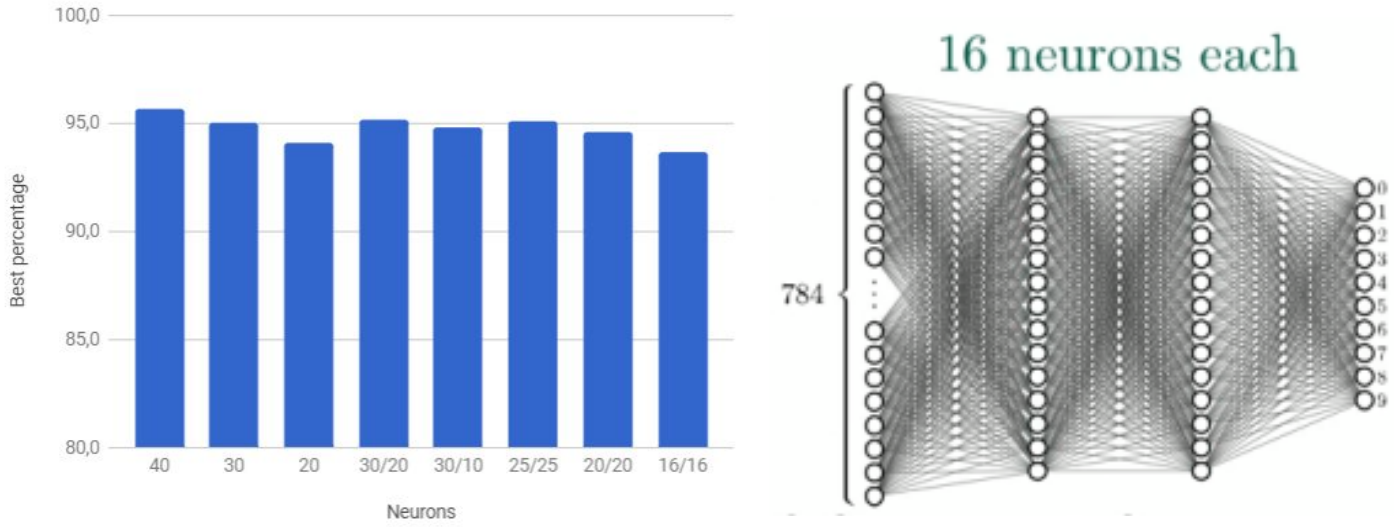

Fig. 10 Training accuracy v/s number of Neurons

After designing the Neural network in Python the next work is to implement it on FPGA.
The software used is Vivado 2017.4
Then following sequence was followed to convert the current model into Digital Circuit.
-> Export weights from python code and compile it for verilog.
-> Design perceptron on the basis of final neural network
-> Design a quantized Sigmoid function for Verilog
-> Develop different IP cores to build a Neural Network in Verilog
-> Program Each IP core in Vivado
-> Develop a Test Bench to check and verify the outputs
The task was to optimize the quantization by reducing the error in 8 bits.
Generate a lookup Table for verilog to use Sigmoid
Following is the Matlab code to quantize Sigmoid function

```
NQA = 8;
NQF = 10;
x = [-(2^NQA):1:(2^NQA-1)]'/(2^NQA);  % 256+256 point interval
y = exp(x);
yQ = round(y/max(y)* (2^NQF-1)) ;
plot(x,y)
title('e^x')
ylabel('e^x')
xlabel('x')
grid
figure
plot(x,yQ)
title('e^x - quantized 8 bit')
ylabel('e^x')
xlabel('x')
grid
```

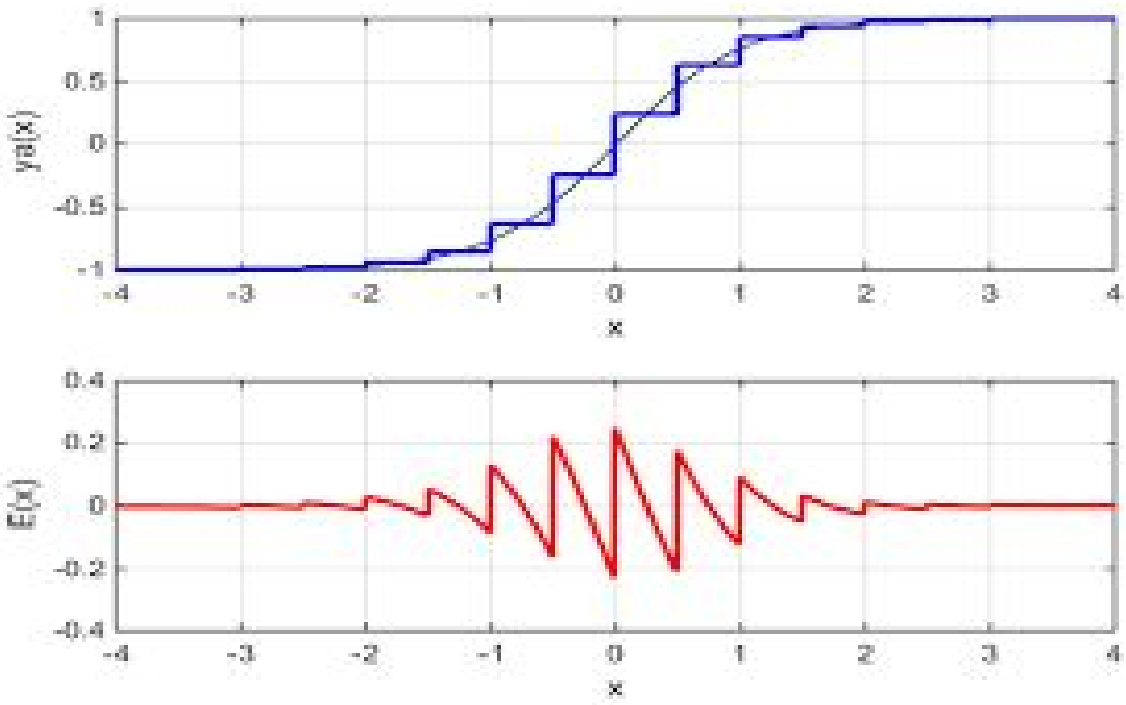

Fig. 11 - Error generated in Quantization of Sigmoid function in matlab
Manual Designing of IP and verilog programming of each IP

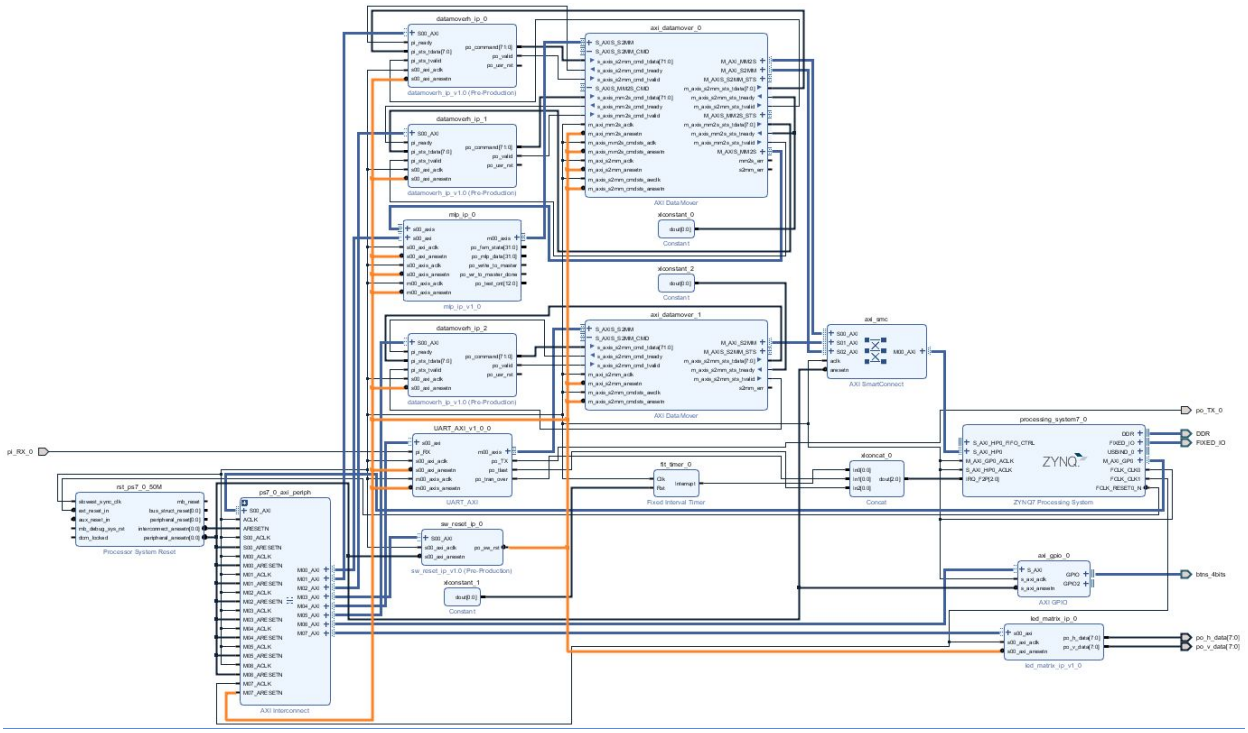

Fig. 11 - Different IP cores designed by me to realize the DNN for Character Recognition

4. Results

The Neural network created on Verilog is then tested by giving raw data from image in terms of vectors and assigning weights from the Python model. It is found that that the simulation results and output from each hidden layer were in the order of nanoseconds whereas in raspberry pi and in general purpose cpu and gpu it is found to be in order of seconds to milliseconds respectively.

4.1. RTL Simulation

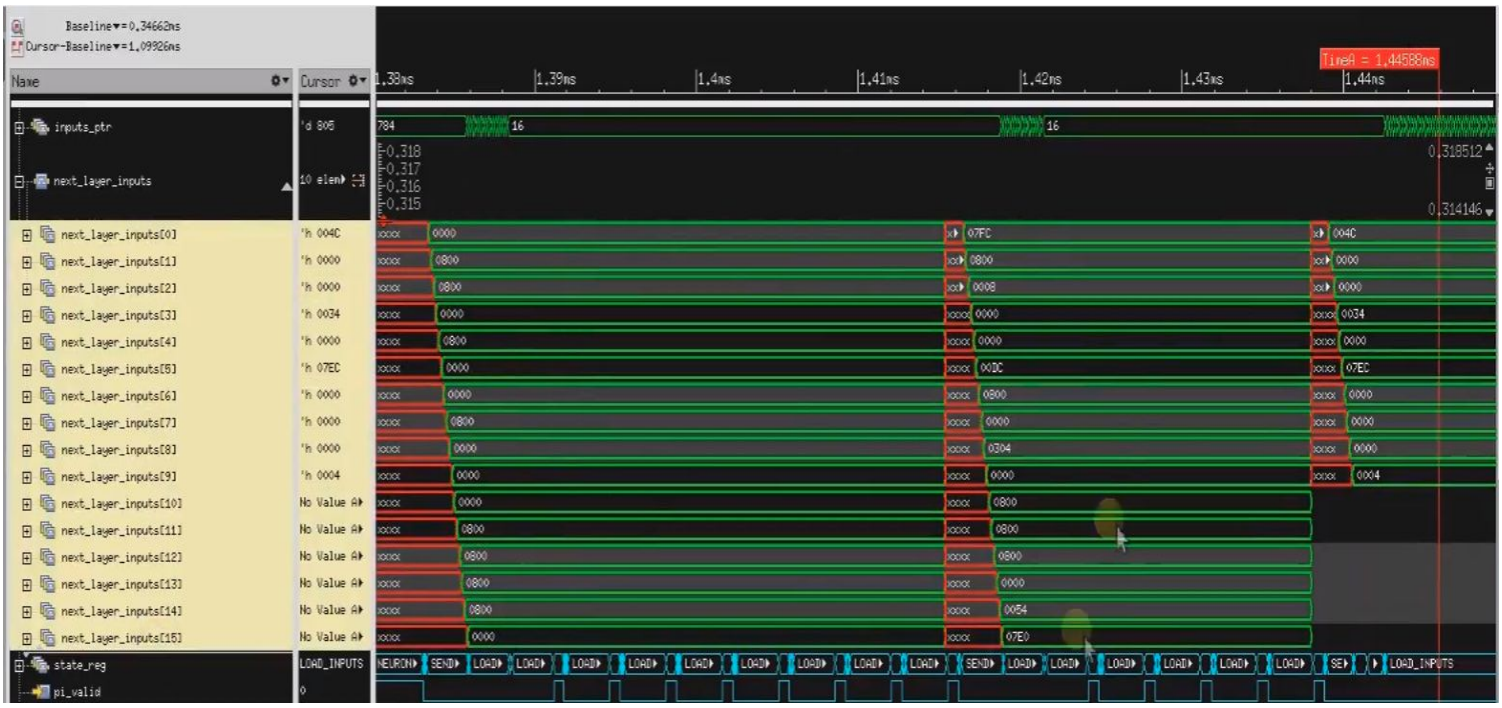

Fig. 12 - Output of Different layers and final output on giving image as an input vector to the Hardware realization of Neural Network

It is clear from the simulation result that output to the final result is maximized for the input value only. The output of each hidden layer are modified according to the weights assign by python code and eventually generates the required output.

4.2. Elaborated design

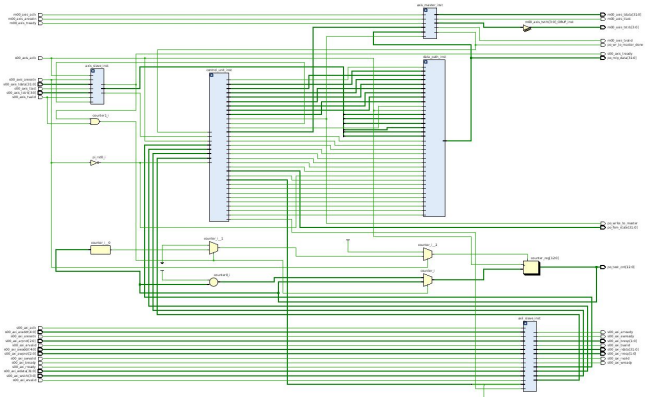

Fig. 13 - Shows 2 hidden Layers in hardware implementation with 16 neuron each

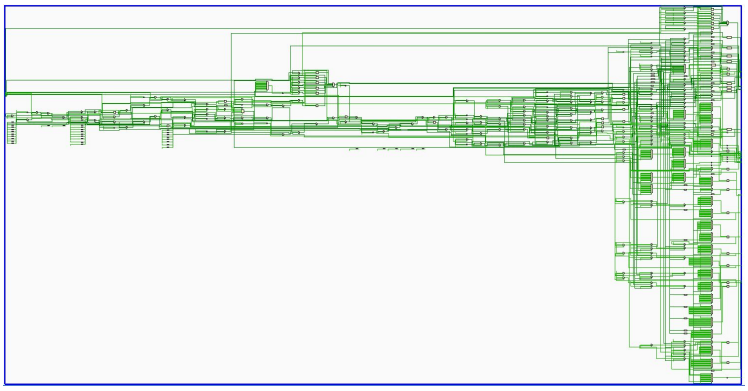

Fig. 14 - Shows Complex connections of Neural

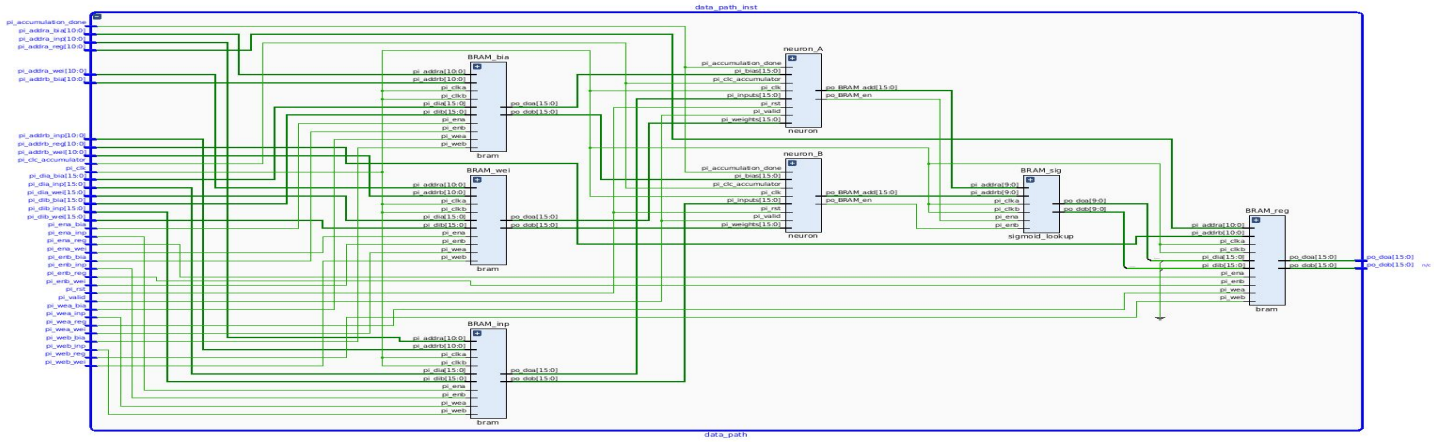

Fig. 15 - Shows the utilization of BRAM for sigmoid and weights data by different neurons

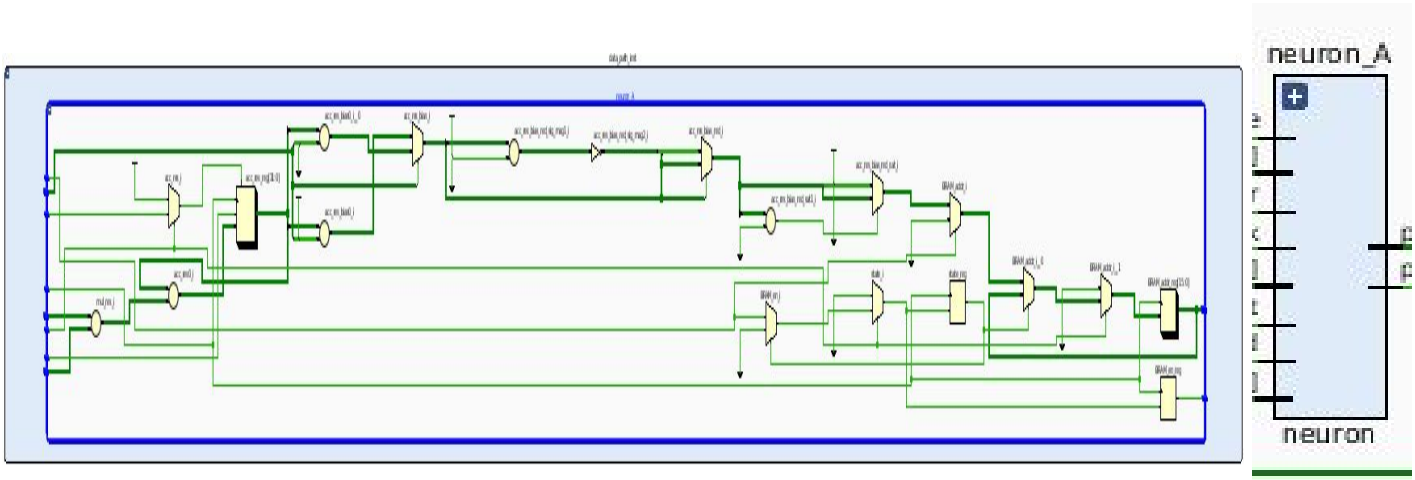

Fig. 16 - The hardware implementation of each neuron using digital logic design for a perceptron model

4.3. Comparison between Nvidia DGX-1 GPU and New Design

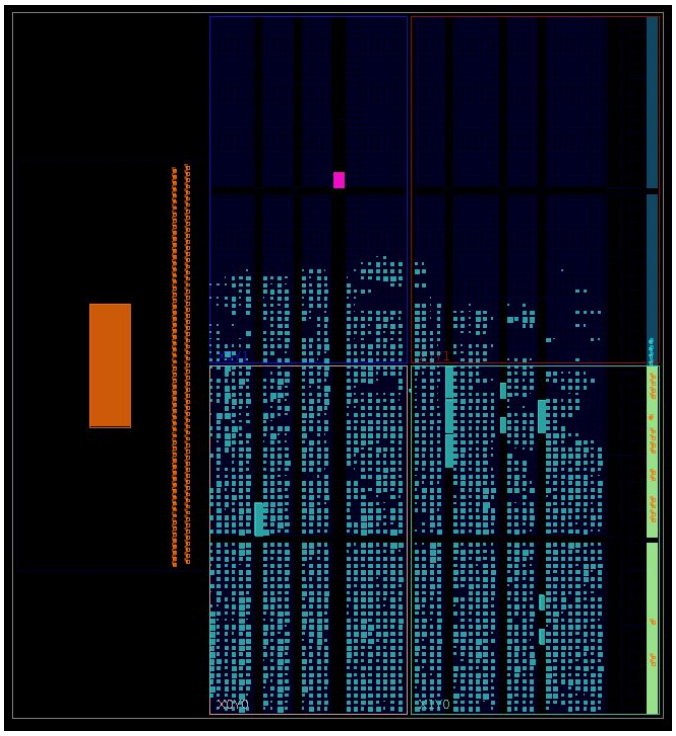

Fig. 17- FPGA configuration our DNN

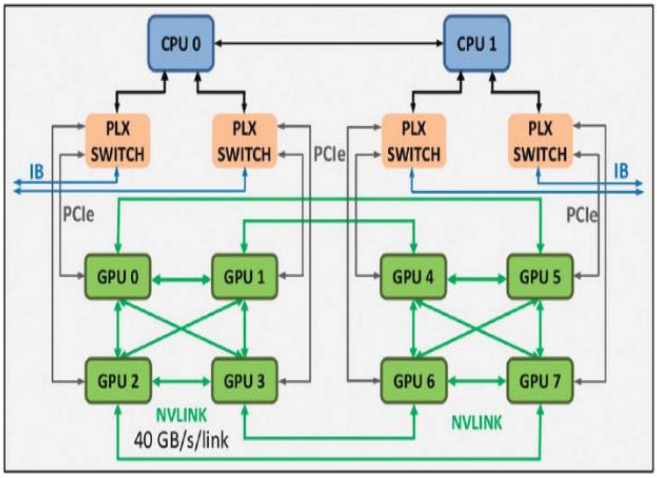

Fig. 18- Nvidia DGX-1 GPU configuration

GPU usually has thousands of cores designed for efficient execution of mathematical functions. For instance, Nvidia's DGX-1 GPU, the Tesla V100, contains 5,120 CUDA cores for single-cycle multiply-accumulate operations and 640 tensor cores for single-cycle matrix multiplication. It has been flaunting massive processing power for target applications such as video processing, image analysis, signal processing and more.

Parallel processing and Pipelining sometimes is major cause for delay in output.

Where as FPGA allows for specifying hardware description language (HDL) that can be in turn configured in a way that matches the requirements of specific tasks or applications. It is known to consume less power and offer better performance.

FPGA shows efficiency in parallel processing. And thus suitable for our work and because of efficiency in parallel processing we are getting output in nanoseconds whereas in Nvidia we are getting output in order of Milliseconds.

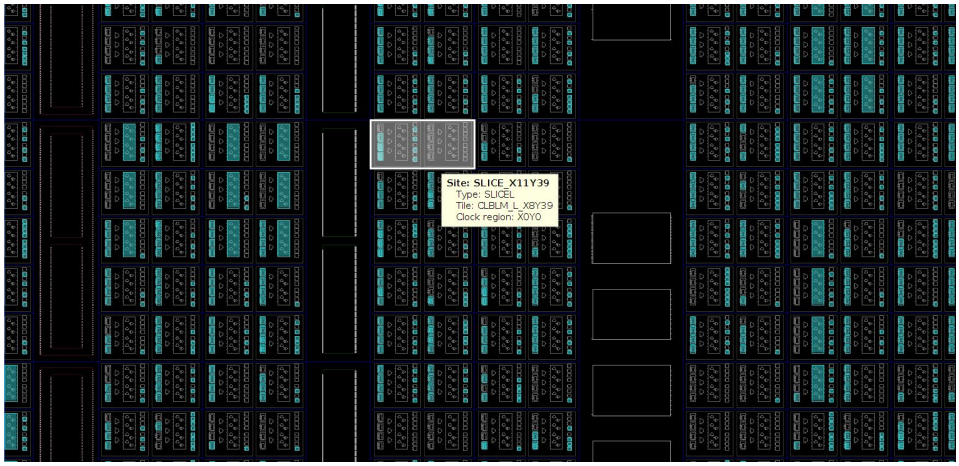

Fig. 19 - Detailed diagram of programmed FPGA to understand its hardware level functioning

4.4. Comparison to Raspberry Pi-3B

On modifying my model to run on Raspberry Pi after running the network on RPI 3B the output was obtained with a latency of few seconds whereas in our designed Hardware accelerator the output is obtained in the order of nanoseconds.

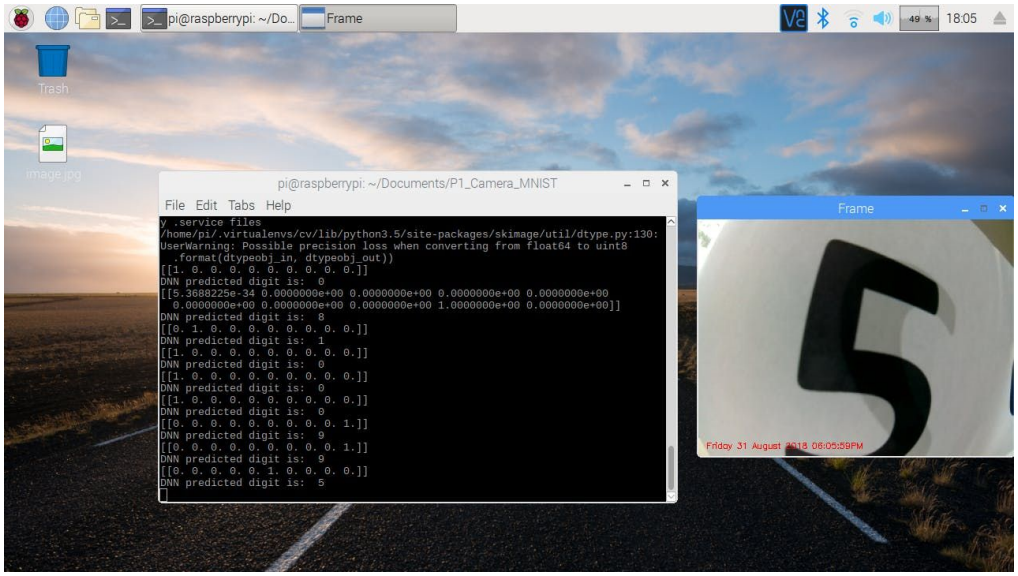

Fig. 20 - Output of our Neural Network in Raspberry pi 3B

Thus, it can be concluded that the Hardware accelerators provide both power optimization and time optimization as shown in the results obtained. In future more efficient and general purpose hardware accelerators can be designed to realize DNN as they provide efficiency in parallel processing and Pipelining.

5. References

https://ieeexplore.ieee.org/document/7450197
https://ieeexplore.ieee.org/document/8050809
https://ieeexplore.ieee.org/document/8585396
https://ieeexplore.ieee.org/document/7987595
http://neuralnetworksanddeeplearning.com/
https://www.xilinx.com/video/hardware/i-and-o-planning-overview.html
https://www.doc.ic.ac.uk/~wl/papers/17/vlsi17rz.pdf
https://daim.idi.ntnu.no/masteroppgaver/013/13656/masteroppgave.pdf
http://yann.lecun.com/exdb/mnist/
https://cs217.stanford.edu/