University of Toronto

ECE532H1 Digital Systems Design

Group 9 Project Report

Hardware Approach of Forward Propagation in Neural Network on FPGA

Yu Gao

Zhuojun Yu

Wales Zhou

Randy Ewing Chow

April 14th, 2021

Table of Contents

1. Project Overview

1.1 Motivation & Goals

Neural networks and deep learning currently provide probably one of the best solutions for many applications like image recognition, speech recognition and natural language processing. In traditional programming, the software is designed with little or no hyperparameters, and the whole program runs in a decided fashion. The advantage of a neural network is it does not need us to tell the computer how to deal with the problem, but by learning from observing data, it can generate its solution. Although this is a powerful tool, it requires significant computing powers and is time-consuming.

Under these circumstances, we would like to implement a neural network built on hardware able to recognize handwritten digits from the MNIST dataset. Performance and accuracy will be sent from one FPGA where the neural network is to another FPGA and can be visualized on a PC that is connected by UART.

1.2 Similar Projects

Some successful experiments of implementing machine learning on FPGA have made promising results. For example, using OpenCL[1] to design convolutional neural networks on Intel FPGA. It can classify 50k images at a speed of 500 images per second [2]. This gave us inspiration for implementing neural network forward propagation on Nexys DDR board.

There was also a group of students from the University of Birmingham doing a project on general neural network hardware architecture on FPGA [3].

1.3 Design Overview

IPs used for the block design in this project are Ethernet, UART, DMA and MicroBlaze. As shown in Figure 1, the first FPGA has NN block, DDR and DMA block.
The whole system consists of two PCs and two FPGAs. The 1st PC sends the image file and label file to the MicroBlaze on the 1st FPGA. The 1st FPGA receives the test images and test labels then saves them to the DDR3 memory, then sends all the labels to the 2nd FPGA. Then it converts one image at a time to fixed-point format and sends the image to the neural network through DMA, then the result from the neural net will be sent to the 2nd FPGA. The 2nd FPGA receives the output results from the neural network, compares them with the test labels, then sends the result to the 2nd PC. The 2nd PC runs a visualization program using Python, to display the overall accuracy and the accuracy for each digit.
The neural network is implemented with ReLU, sigmoid, vector multiplication, the top-level neural network, and the AXI-Stream wrapper.
The neural network functions in a sequential style, the data flow is from the input layer to vector multiplication, to the ReLU, to the middle layer, to the vector multiplication units, then to the output layer. The top-level neural network implements an FSM to control the data flow from the input to the output.
The AXI-S wrapper has a slave interface and a master interface. The slave interface saves one image in a buffer (size of 28x28 pixels, each pixel has 16 bits). After a full image is received, it sends all the bits in

parallel to the neural network. The master interface also uses a buffer to store the 10 digit values from the neural network.

A link to our video introducing our project: https://www.youtube.com/watch?v=85MMx2kJ0M8

1.4 Block Diagram

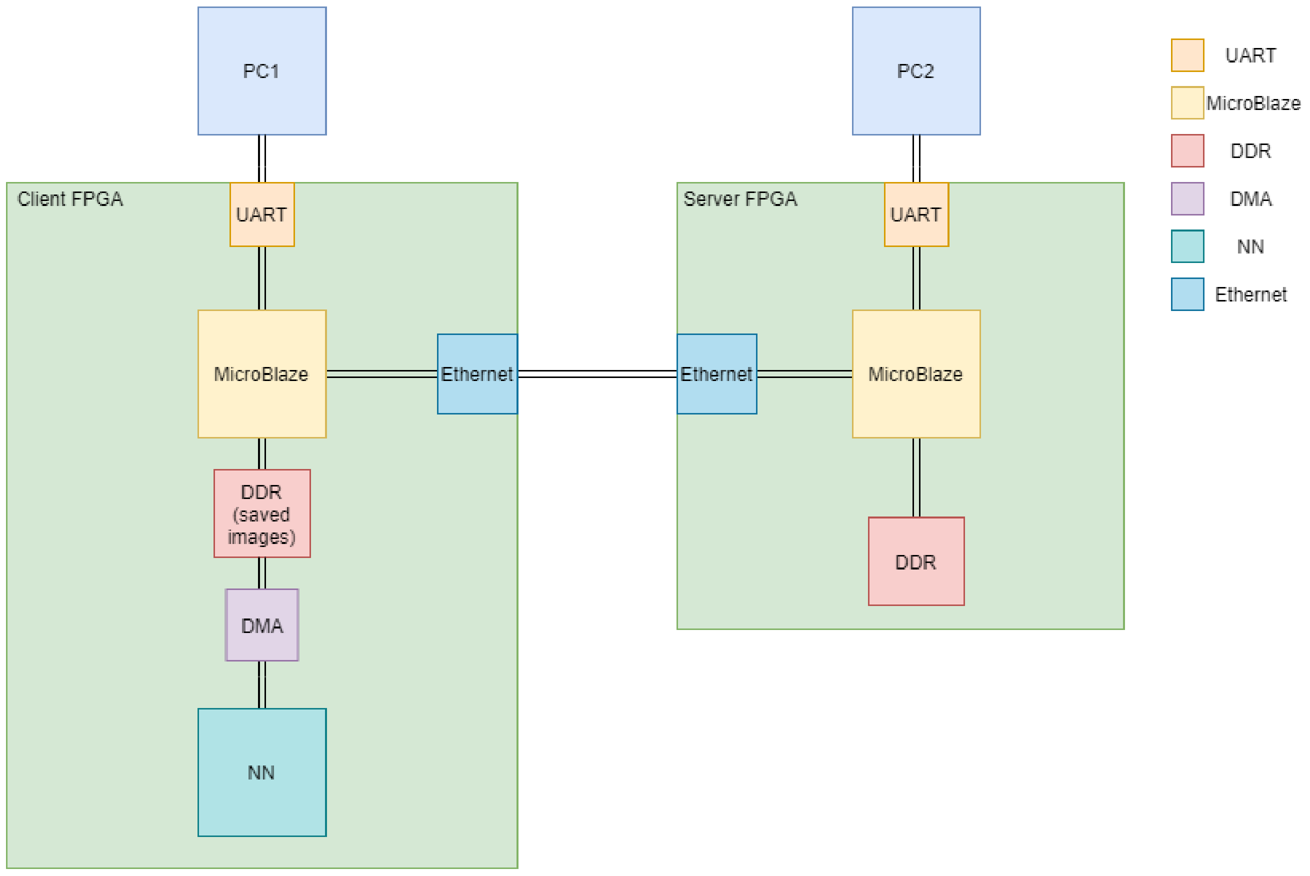


Figure 1: Full System Block Diagram

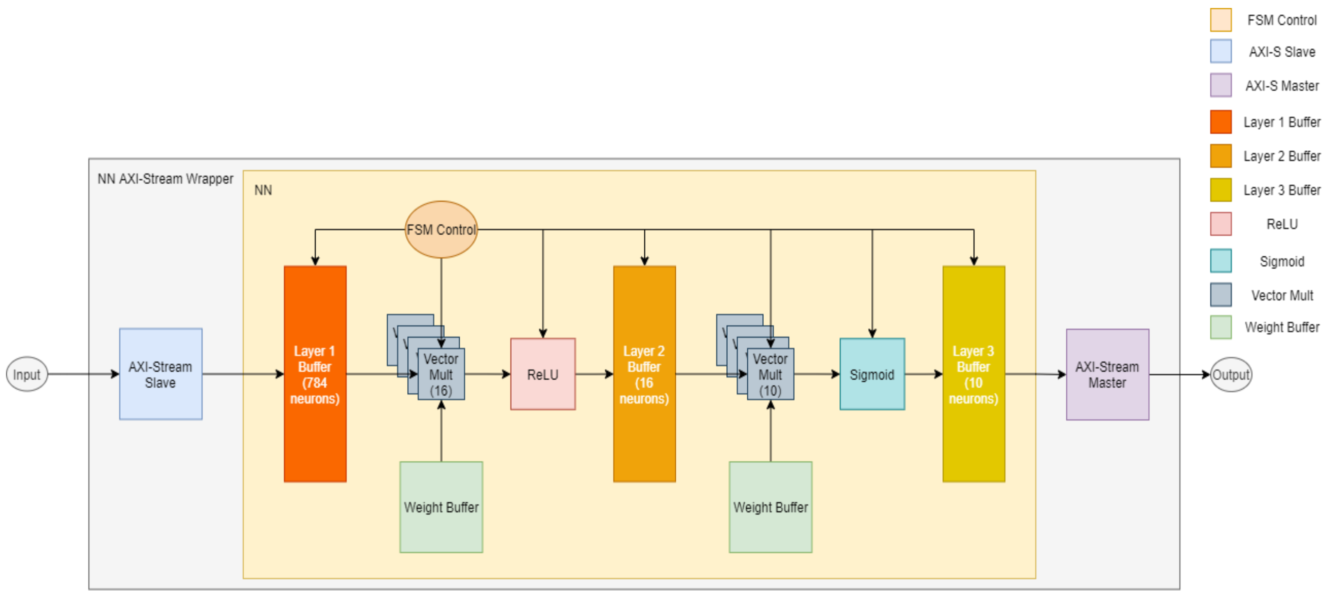


Figure 2: Neural Network Block Diagram

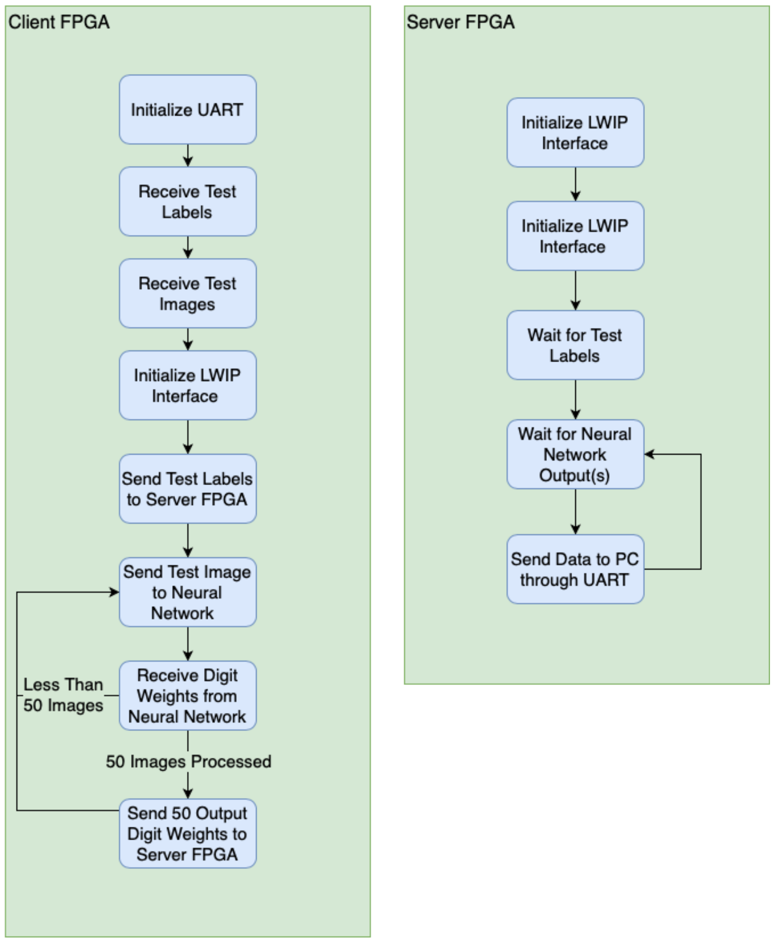
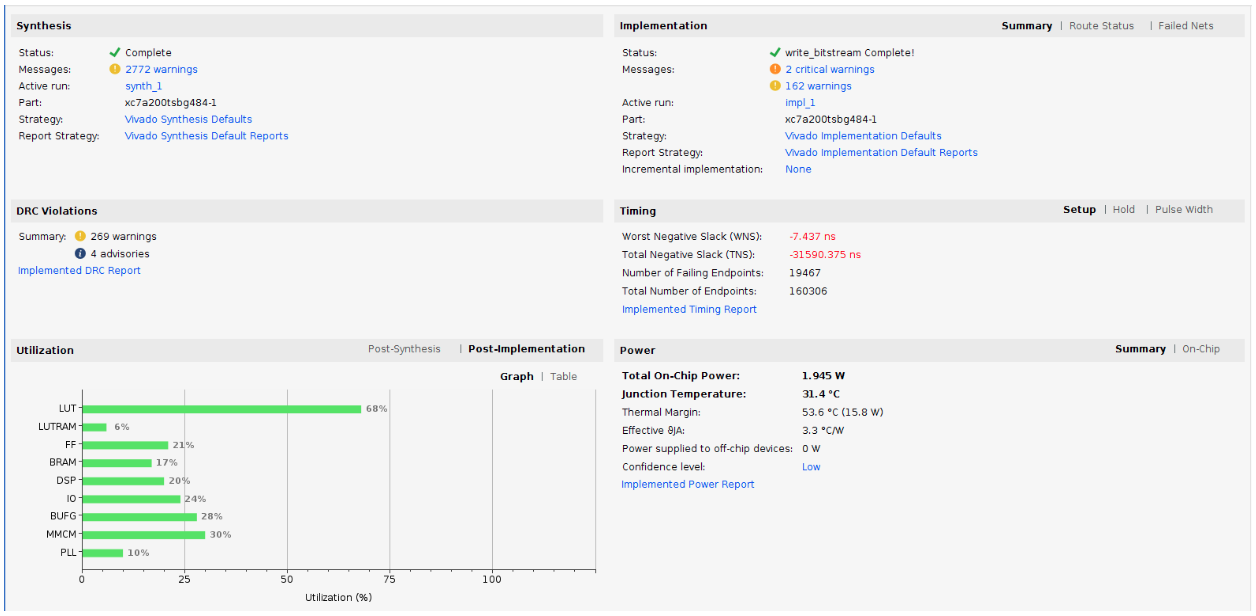

Client FPGA
Initialize UART
Receive Test Labels
Receive Test Images
Initialize LWIP Interface
Send Test Labels to Server FPGA
Send Test Image to Neural Network
Receive Digit Weights from Neural Network
Less Than 50 Images
50 Images Processed
Send 50 Output Digit Weights to Server FPGA
Server FPGA
Initialize LWIP Interface
Initialize LWIP Interface
Wait for Test Labels
Wait for Neural Network Output(s)
Send Data to PC through UART

Figure 3: Software Block Diagram

2. Outcome

2.1 Results

We successfully implemented a 784-16-10 Neural Network on Nexys Video Board, with pre-trained weights and biases. According to the waveform from simulation, processing time for each image takes around 8 microseconds at 100 MHz clock frequency. Testing images are sent to the neural network through DMA. Sending Neural Network labels & output results to the 2nd FPGA in real-time through TCP. Visualizing overall accuracy and digit accuracy in real-time using PySerial. In Figure 4 below, we can see we can fit all components within the Nexys Video board successfully with 16 neurons in the hidden layer.



Synthesis

Status:	✓ Complete
Messages:	2772 warnings
Active run:	synth_1
Part:	xc7a200tsbg484-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports

Implementation — **Summary** | Route Status | Failed Nets

Status:	✓ write_bitstream Complete!
Messages:	2 critical warnings
	162 warnings
Active run:	impl_1
Part:	xc7a200tsbg484-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Summary: 269 warnings
4 advisories
Implemented DRC Report

Timing — **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS):	-7.437 ns
Total Negative Slack (TNS):	-31590.375 ns
Number of Failing Endpoints:	19467
Total Number of Endpoints:	160306

Implemented Timing Report

Utilization — Post-Synthesis | **Post-Implementation**

Graph | Table

Resource	Utilization (%)
LUT	68%
LUTRAM	6%
FF	21%
BRAM	17%
DSP	20%
IO	24%
BUFG	28%
MMCM	30%
PLL	10%

Power — **Summary** | On-Chip

Total On-Chip Power:	**1.945 W**
Junction Temperature:	**31.4 °C**
Thermal Margin:	53.6 °C (15.8 W)
Effective θJA:	3.3 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

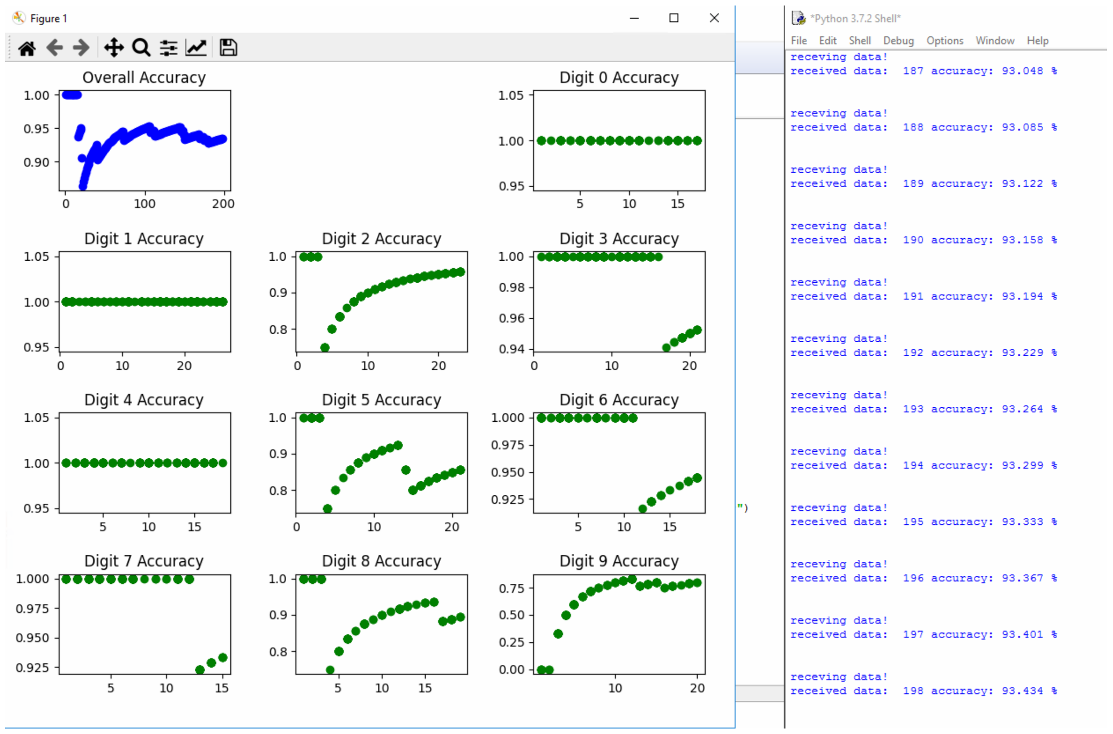
Implemented Power Report

Figure 4: Post-Implementation 784-16-10 Neural Network & Client System Summary

As shown in the table below, we can see compared to the python implementation of the neural network, the hardware implementation has lower accuracy. This is due to we are using the pre-trained weights from the python models, which use the floating-point numbers. However, in our design, we used fixed-point numbers for simplicity and efficiency. This could lead to the saturation in the fixed-point number, which further leads to the errors, and we believe that with more middle layers, this issue will be magnified. For 4 neurons in the middle layer, we can see there is a huge discrepancy between the hardware and the python model. When using 16 neurons, however, the difference reduced to 4%, which is very good considering the saturation problem.

Number of Neurons in Hidden Layer	Overall Accuracy (Hardware Implemented Neural Network)	Overall Accuracy (Python Keras Neural Network)
4	60%	86%
16	92%	96%

For each digit, we can see digits 0, 1 and 4 have the perfect accuracy, and digit 5 has the worst accuracy. This is not because the digit 5 is hard to train, it is due to not being enough digit 5's in the training dataset.



```
receving data!
received data:  187 accuracy: 93.048 %


receving data!
received data:  188 accuracy: 93.085 %


receving data!
received data:  189 accuracy: 93.122 %


receving data!
received data:  190 accuracy: 93.158 %


receving data!
received data:  191 accuracy: 93.194 %


receving data!
received data:  192 accuracy: 93.229 %


receving data!
received data:  193 accuracy: 93.264 %


receving data!
received data:  194 accuracy: 93.299 %


receving data!
received data:  195 accuracy: 93.333 %


receving data!
received data:  196 accuracy: 93.367 %


receving data!
received data:  197 accuracy: 93.401 %


receving data!
received data:  198 accuracy: 93.434 %
```

Figure 5: Python Visualization Output for 200 Images

2.2 Changes Made

Compared to our original plan, we changed the number of neurons in the second layer from 800 to 16 due to the limitation of resources available onboard.

We used pre-trained weights instead of the backpropagation implementation on MicroBlaze because the algorithm runs too slow on the soft CPU, and it would take a few weeks to train on the full test dataset.

Another thing we changed is instead of using Two FIFOs connected to the input and output side of the neural network, we used the DMA to read the data from the DDR.

2.3 Potential Future Improvements

A potential improvement can be having one server FPGA to control flow from Client FPGAs and send their request to processing FPGAs with the Neural Network. Another FPGA on the network will receive analytical data from the server FPGA and visually present it through a Python program on a PC. Having back Propagation implemented on hardware instead of using pre-trained weights and bias can also be an improvement because in this case, we can have a neural network completely on FPGA. The multiplication unit, which currently uses an accumulator for addition, can be improved with the use of an add tree, so the addition can be done with the time complexity of O(logn), instead of O(n).

3. Project Schedule

The project schedule is divided into 6 milestones as follow:

3.1 Milestone #1: Neural Network Research & Architecture Design

In the first week of the project, we were mainly focused on getting familiar with the Xilinx Vivado environment and understanding the concept of neural networks. We understand mathematical terminology of how forward and backpropagation works, how to build up neural networks mathematically. Hardware side we researched how other projects can achieve building neural networks on an FPGA. Another progress we had is studying Xilinx AXI-Lite and AXI-Stream protocol. On the software side we found out using the MNIST database will be the handiest resource for training our neural network in the future of this project.
The challenge we had this week was mainly about designing the architecture for this neural network on hardware. A question like: what should the input and output look like? In what way can we retrieve data from a buffer? Do we need to implement controllers for those buffers, too?
Another thing is the 28x28 input pictures. This network requires an input vector of length 784. we were not sure if we had enough time to build that many neurons.

3.2 Milestone #2: AXI Protocol & Concept Verification on C Code

In the second week of the project, on the hardware side, we started to build up buffers and tested how to have AXI protocol work between buffers and MicroBlaze. The software team successfully has neural network work with C code. We were able to send test pictures from the MNIST database and have recognition accuracy above 90%. This helped us understand how neural networks work. Also this week, we started planning on the second FPGA. We would like to have data visualization achieved on the second FPGA.

The challenge we encountered was some trouble testing the IP block with verification IP. There are some problems with starting the simulation.

3.3 Milestone #3: Neural Network Block Design

This week on the hardware side we implemented the subblocks of the NN system. We started with a single neuron, which consists of a matrix multiplication unit, a sigmoid unit, a weight buffer, output buffer, data buffer, and the control logic. As we moved on to design the layers. Software side we were able to send MNIST data byte by byte using TeraTerm to the MicroBlaze and then store them onto the DDR3 memory of the Video Board (DESLB).

The problem we realized this week was designed by adding neurons may not be very efficient, so I changed my method to implement the layers by moving the matrix multiplication unit to the top level, as well as the sigmoid unit, which can be reusable.

3.4 Milestone #4: Neural Network Pre-training & Activation Module Implementation

By this time, this project was at the midpoint. On the hardware side, we verified the Sigmoid and ReLU function Verilog module. Started writing the testbench for Sigmoid and ReLU, then checking the waveform - whether the sequence is exactly what we want, as well as the possible sequence conflicts with other modules when transferring data through interfaces.

The challenge encountered this week was the original Sigmoid and Relu Verilog module may not be very efficient? Thus we are trying to use some kind of SERDES process in Verilog to make it faster, this new implementation for ReLU has been tested to work well, however, Sigmoid's new parallel implementation is still under testing.

3.5 Milestone #5: Hardware & Software Integration

This week the software side started working on integrating achievement between teams. We are also Finishing verification for the VecMult and Relu module, and testing some data inputs into them.

Carefully considering the sequence, whether any enable signals to show up earlier than supposed.
Trying several implementation methods to deal with the sequence problems met.
Currently working on searching possible bugs in the post-implementation/synthesis simulation waveform.
lots of issues are found this week. We need to find the easiest and best way to communicate with AXI4-Stream Data FIFO to send data to Custom IP. We could not find memory addresses or driver commands to write and read from the FIFOs. We looked into Memory-Mapped Registers and AXI DMA to send data to Custom IP and receive output. Also tried to increase MicroBlaze performance to speed up conversions and potential backpropagation on SW addition. This required adding an Instruction and Data cache which would take up resources but also when configured, the old code did not work with the configuration.

3.6 Milestone #6: Final Test

For the last week of this project, we cleaned up some of the TCP data transfer from FPGA 1 to FPGA2. Data visualization is achieved with python. Hardware side we created an AXI4 Stream wrapper for both a Logic Function Neural Network and MNIST Neural Network. We are test methods for sending data to hardware cores through AXI Data FIFO and AXI DMA.

We had a synthesis fail during that process, then we found it was because the RAM of my PC is not enough. And when the input is 100 it passes.

4. Description of the Blocks

4.1 Custom Hardware IP

The Hardware design of the neural network consists of the following functional blocks: RelU, sigmoid, vector multiplication, the top-level neural network, and the AXI-Stream wrapper. For data format, all the data in our system is using Q4.12 fixed-point format. Each data uses 16 bits, 4 bits on the MSB side represent the integer part of the number, and 12 bits on the LSB side are the decimal part of the number. To convert the number to floating-point, simply treat the fixed-point number as an integer, which is 4096 times larger than the actual value, so divide the number by 4096.

4.1.1 Sigmoid IP Design

We referred to an article about how to design Sigmoid in FPGA[5], this article uses VHDL to design Sigmoid IP in FPGA. We learn from its original idea and complete a Sigmoid IP based on Verilog-HDL.

Furthermore, since we met out of ram trouble during our synthesis process, i.e., the synthesis process fails when we are synthesizing the project as a whole. Therefore, we add an instruction (* use_dsp = "yes" *) to the Sigmoid module to conduct it to use DSP resources in FPGA. And in order to speed up the Sigmoid module, we also alternate it to get a group of input numbers.

Since the Sigmoid function needs to do some point numbers processes. So we split up the fixed-point number, to separate the integer part and the point part. So in an input that includes 16 bits, 4bits are for integer part and 12 for the numbers after the point.

The sigmoid function also includes the e^x function that cannot be implemented in Verilog, thus we use a ram text which includes the corresponding Sigmoid output for each possible input. Whenever this module receives an input that contains 16 bits, it will send the output from the ram, like a LUT.

For the verification of this module, we use a Python script to calculate the decimal numbers we get from the waveform, which is the decimal form of the 16 bits input, then this script outputs the fixed-point number form of it. After that, another script will get this input value and output the corresponding Sigmoid output. Then we send it to the next script to transform it back to the decimal form, which can be compared with the output in the waveform, that's how we verify this module.

The testbench and test vector are made by ourselves originally.

From the simulation result below, we can see it gives us the correct result. When the input is 0, according to the sigmoid function, it should give a 0.5, which in Q4.12 is 0.5x4096 = 2048. For input of 32767, it is the max value in Q4.12, so it should saturate the sigmoid and give a value closest to 1, which is 4095.

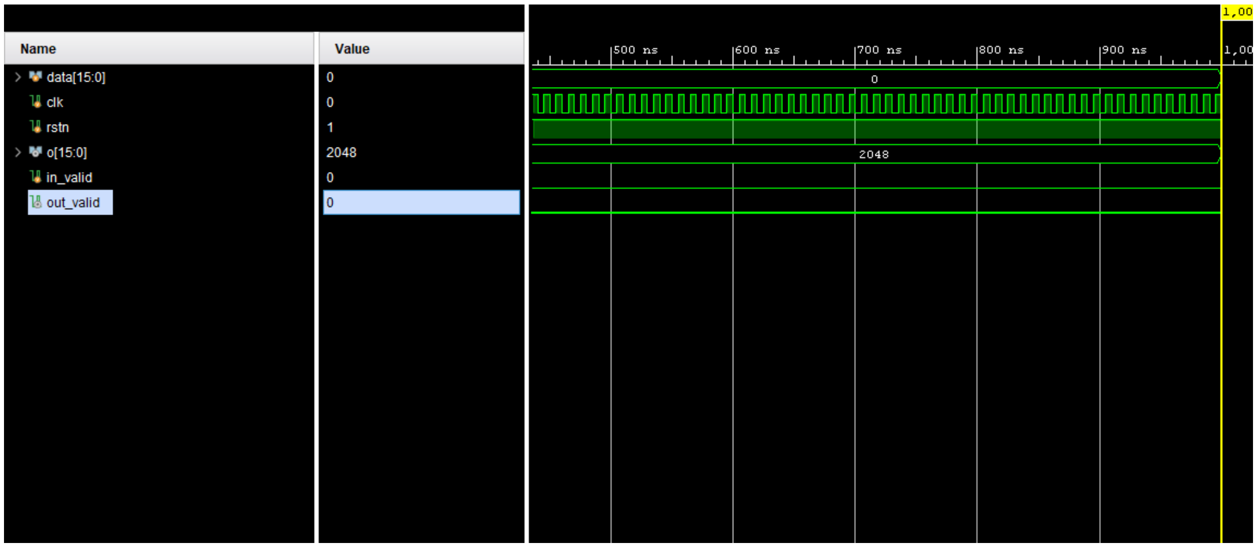


Figure 6: Simulation Result of Sigmoid with Input of 0

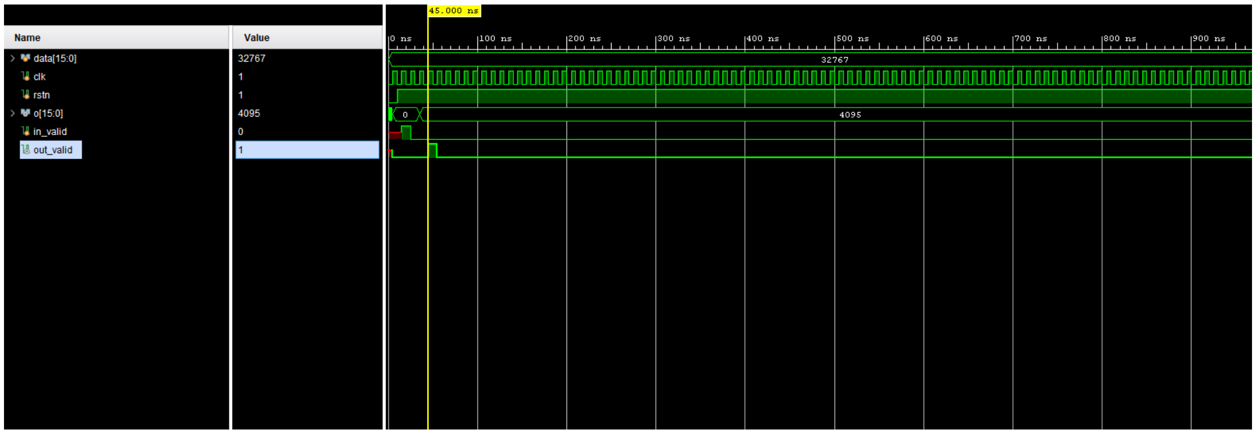
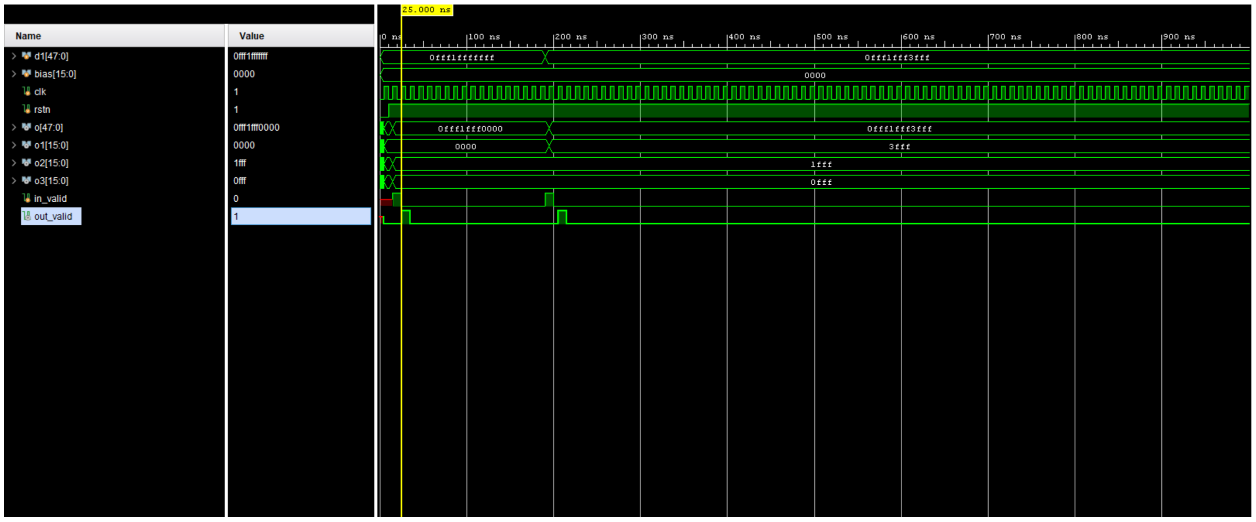


Figure 7: Simulation Result of Sigmoid with Input of 32767

4.1.2 ReLU IP Design

We implement this module to be parallel, other features of it are very simple. The basic theorem of ReLU is to make the output be 0 when input is less than 0, and output the same as input when it is larger than 0. In our design, we use generators to form several assignments which can use ternary operators to assign the output to be either the input or 0. Like Sigmoid, we use 16 bits binary codes for the input and output which can easily connect other modules inside NN. Each part for ReLU is made by ourselves independently. From the simulation result, we can see when the input values are 0x0fff, 0x1fff, 0xffff, the output values are 0x0fff, 0x1fff, 0x0000, which are correct based on the definition of ReLU.



Name	Value
d1[47:0]	0fff1fffffff
bias[15:0]	0000
clk	1
rstn	1
o[47:0]	0fff1fff0000
o1[15:0]	0000
o2[15:0]	1fff
o3[15:0]	0fff
in_valid	0
out_valid	1

Figure 8: Simulation Result of ReLU

4.1.3 Vector Multiplication

The vector multiplication unit is used to multiply two vectors together, then add all the values together. This block is used to multiply the activation of the neural network and the weight together. For our final design of the neural network, a total of 26 vector multiplication units are used (16 of them are used between the first layer and the middle layer, 10 of them are used between the middle layer and the output layer). The reason that we decided to use vector multiplication instead of matrix multiplication is that it is more flexible compared to matrix multiplication. For multiplication, I need to hardcode the row number of the matrix, so for different matrix sizes, different matrix multiplication units will be needed. The downside of this approach is that we can't manually add too many units to the top-level design, for it would cost us a lot of time. Inside the unit, a parallel multiplication will be performed on the input vectors. Since the vectors are all 16 bits, the value after multiplication will be 32 bits. The values are then stored in a 32-bit variable length (decided by the input parameters) array. After multiplication, the values are then added together by an accumulator, which adds two values together per clock cycle. To make room for a larger output value, the data is stored in a 32+log(vector size) bits register. This value, however, cannot be the final output, since it has too many bits and the output value should be 16 bits. To cut off the redundant bits, an overflow check is performed to the 31+log(vector size) to the 27th bit, to detect if any overflow occurred. If the overflow occurred, the output value then saturates to 16'b0111_1111_1111_1111 if positive, or saturates to 16'b1000_0000_0000_0000 if negative. The 12 bits from the LSB side are discarded based on the fixed-point arithmetic rule for Q4.12 format.

From the simulation result, we can see, one bias and two vectors with the size of two are fed into the module, and the two vectors multiplied and added together giving us a result of 10355, the bias has a value of -10355, which gives us the correct result of 0.

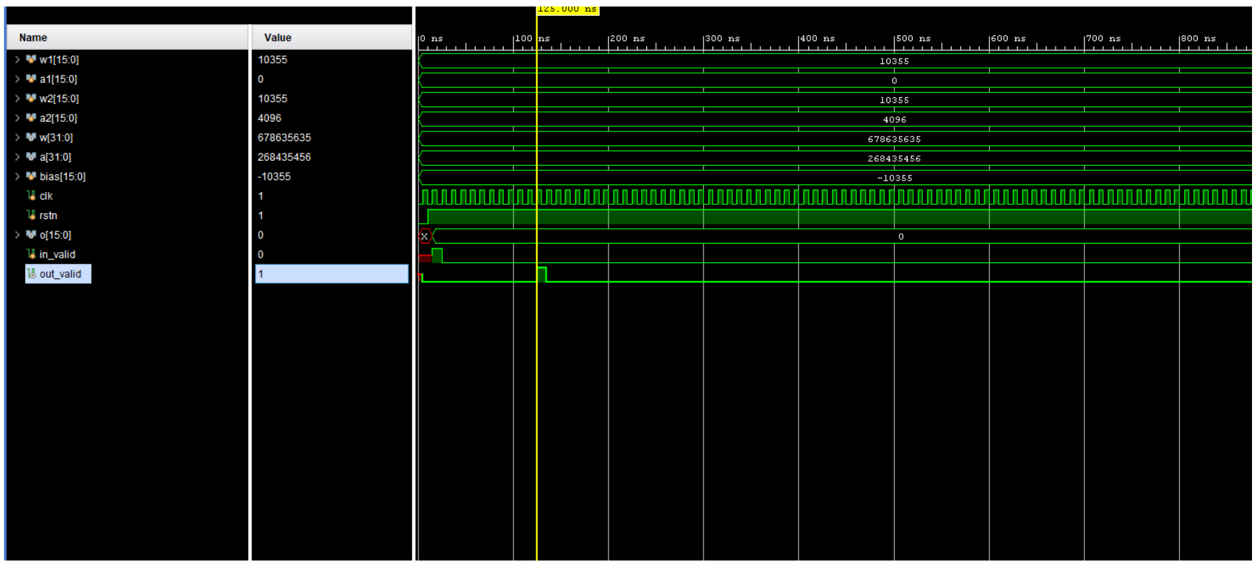
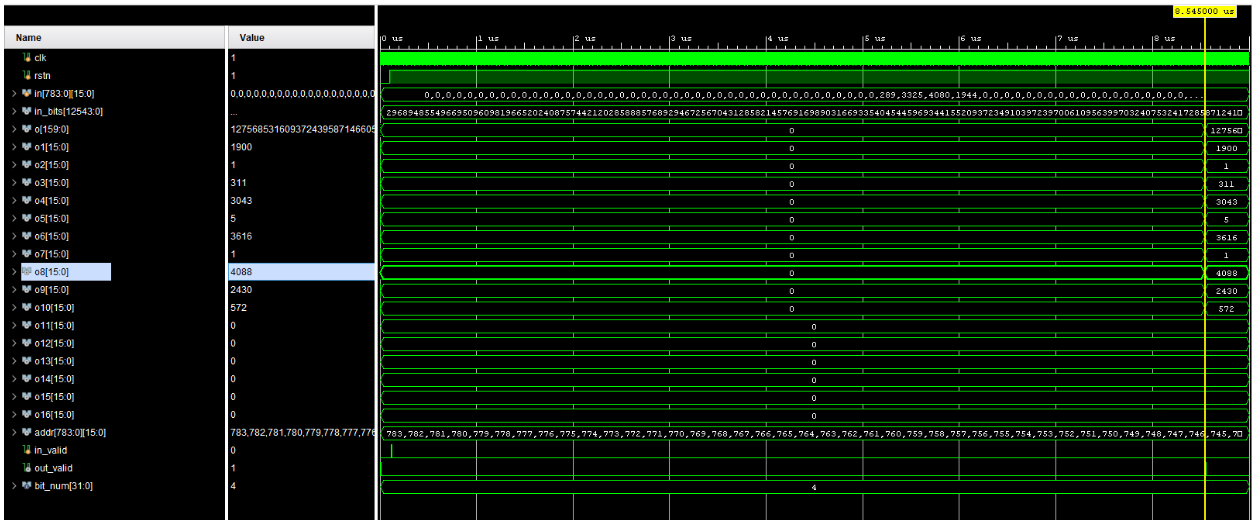

Name	Value
w1[15:0]	10355
a1[15:0]	0
w2[15:0]	10355
a2[15:0]	4096
w[31:0]	678635635
a[31:0]	268435456
bias[15:0]	-10355
clk	1
rstn	1
o[15:0]	0
in_valid	0
out_valid	1

Figure 9: Simulation Result of the Vector Multiplication Unit

4.1.4 Neural Network

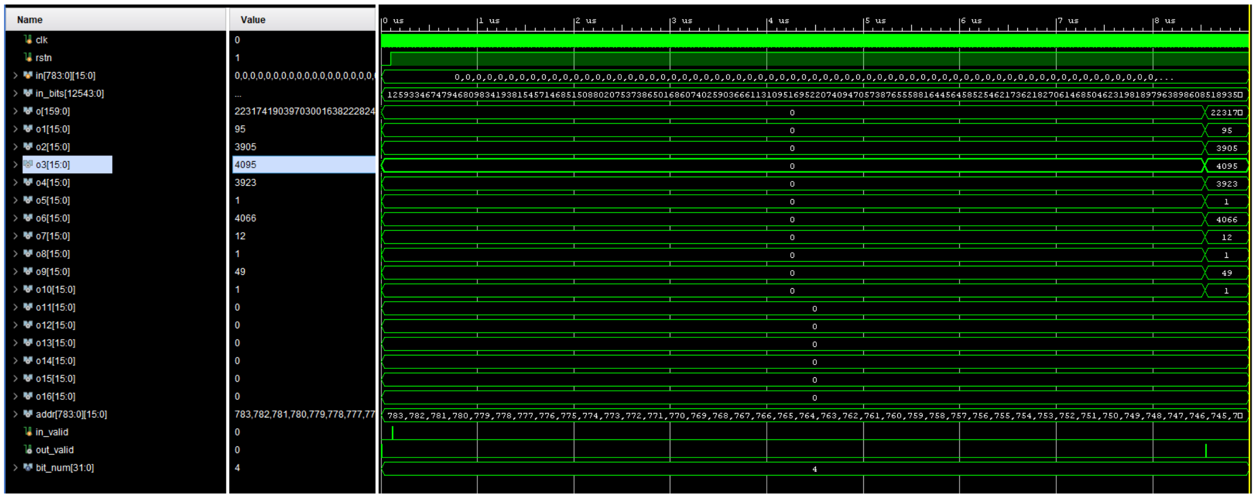
The neural network consists of three layers: the input layer, the middle or hidden layer, and the output layer. all the layers and their neurons are abstracted as storage elements (registers). The input layer reads in 784 16-bit fixed-point values, which represent the 28x28 image pixels. The 784 values are stored inside the registers. The middle layer stores 16 16-bit values, and the output layer stores 10 values, which represent the 10 digits (0-9). Between the first and middle layers, 16 vector multiplication units are used to compute the activation for each of the 16 neurons in the middle layer. The 16 values then feed into a RelU unit, which provides a nonlinear functionality to the neural network. All the 16 computed values are then stored in the middle layer. Between the middle layer and the output layer, 10 more vector multiplication units are used to compute the activations for the 10 output digits. The computed results then are fed into the sigmoid unit, which normalizes the values to a range from 0 to 1. The 10 values then are stored in the output layer, which will be the output data of the neural network. The overall architecture of the neural network can be viewed as a top-level FSM controlling the data flow through the functional units. First, the neural network reads in 784 data, then enables the first 16 multiplication units, after getting a signal indicating the multiplication is done, it enables the RelU unit, and non-linearize the data. After ReLU is done, the data is then stored in the middle layer, then the rest 10 multiplication units are enabled, after multiplication is done, the sigmoid is then enabled after the sigmoid is done, the data will be stored in the output layer. The ten values are then sent through AXI-Stream to the MicroBlaze, which will decide what is the predicted digit by finding the maximum value from the 10 values. For example, if the output data is [0.1, 0.1, 0.03, 0.5, 0.8, 0.1, 0.2, 0.9, 0.2] then the maximum value is 0.9, which corresponds to the 9th digit, which is 8. So the prediction for this image is 8.

From the simulation result, we can see when input the first image in the test file, after around 8ns, outputs ten-digit values, in which the 8th one has the highest value, so digit 7 should be the prediction, which is correct according to the label file. For the second image, the highest value is the 3rd one, which is also correct.



Name	Value
clk	1
rstn	1
in[783:0][15:0]	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
in_bits[12543:0]	...
o[159:0]	127568531609372439587146605
o1[15:0]	1900
o2[15:0]	1
o3[15:0]	311
o4[15:0]	3043
o5[15:0]	5
o6[15:0]	3616
o7[15:0]	1
o8[15:0]	4088
o9[15:0]	2430
o10[15:0]	572
o11[15:0]	0
o12[15:0]	0
o13[15:0]	0
o14[15:0]	0
o15[15:0]	0
o16[15:0]	0
addr[783:0][15:0]	783,782,781,780,779,778,777,776
in_valid	0
out_valid	1
bit_num[31:0]	4

Figure 10: Simulation Result of the First Image



Name	Value
clk	0
rstn	1
in[783:0][15:0]	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
in_bits[12543:0]	...
o[159:0]	22317419039703001638222824
o1[15:0]	95
o2[15:0]	3905
o3[15:0]	4095
o4[15:0]	3923
o5[15:0]	1
o6[15:0]	4066
o7[15:0]	12
o8[15:0]	1
o9[15:0]	49
o10[15:0]	1
o11[15:0]	0
o12[15:0]	0
o13[15:0]	0
o14[15:0]	0
o15[15:0]	0
o16[15:0]	0
addr[783:0][15:0]	783,782,781,780,779,778,777,77
in_valid	0
out_valid	0
bit_num[31:0]	4

Figure 11: Simulation Result of the Second Image

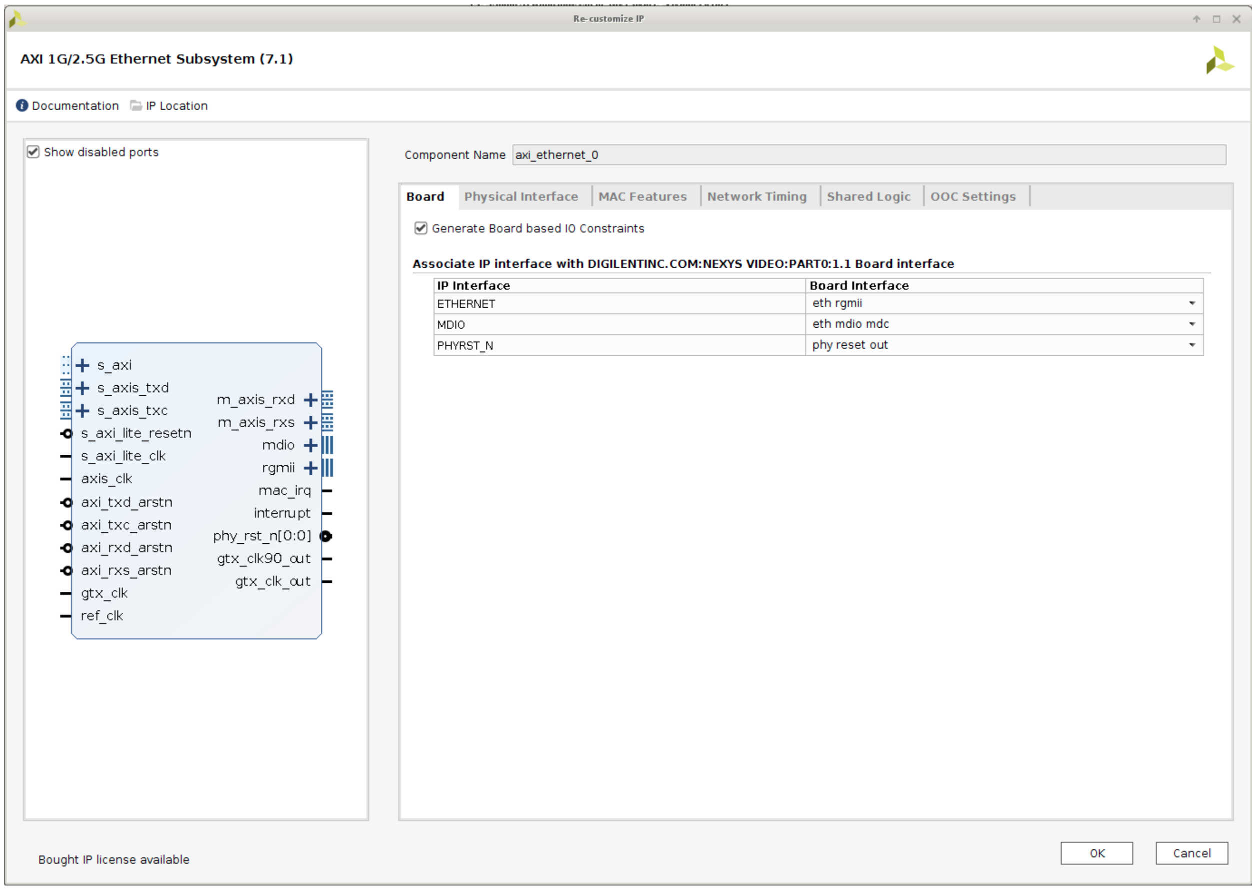
4.1.5 Neural Network with AXI-Stream Wrapper

The AXI-Stream slave is connected to the input side of the neural network, and the AXI-Stream master is connected to the output side of the neural network. The AXI-Stream slave receives 32-bit data from the DMA, so for each transaction, the slave could receive 2-pixel values. A total of 784/2 = 392 transactions are needed to transmit one image for the neural network to predict the corresponding digit. Each of the 784 values is stored inside a register array, which is used as the input data for the neural network. The master will store the 10 output data into a 10x32bits array, and after 10 values are received, the transaction to the DMA will start.

4.2 Important IP Blocks Used

4.2.1 Ethernet

The Ethernet block in our block design is the same one used in Tutorial 5 and the Warmup Demo for the course which is the AXI 1G/2.5G Ethernet Subsystem (7.1) but for the Nexys Video board. This allows for the Digilent LWIP echo server and client. There have not been any special modifications to this block in comparison to when we had configured it in the past for the Tutorial and Demo.



Component Name axi_ethernet_0

☑ Generate Board based IO Constraints

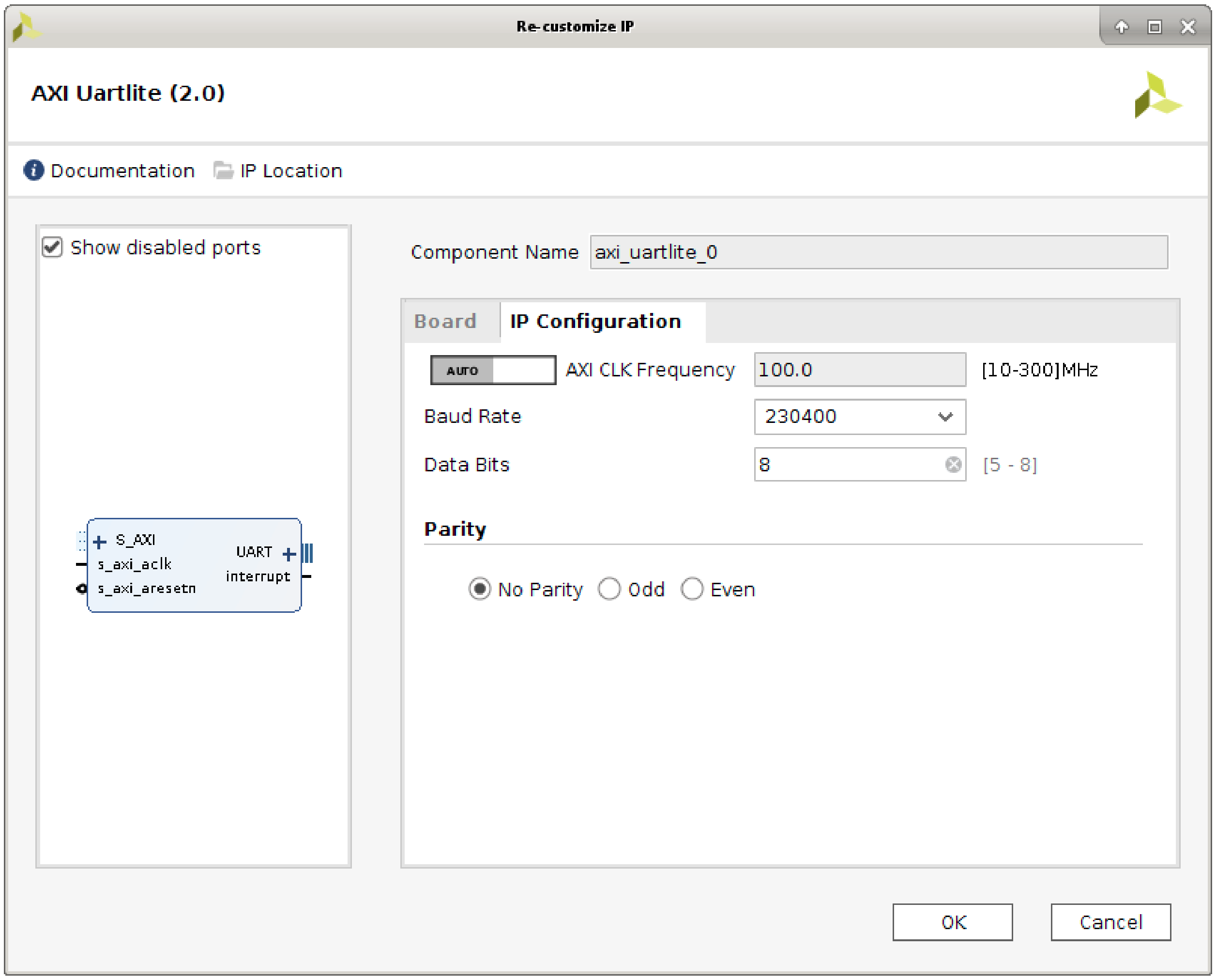
Associate IP interface with DIGILENTINC.COM:NEXYS VIDEO:PART0:1.1 Board interface

IP Interface	Board Interface
ETHERNET	eth rgmii
MDIO	eth mdio mdc
PHYRST_N	phy reset out

Figure 12: Configuration of the AXI Ethernet Subsystem

4.2.2 UART

The UART block in our design is the same one used in our Tutorials for the course which is the AXI Uartlite (2.0). This IP block has been configured with a maximum BAUD rate of 230400 with a reference clock frequency of 100MHz.



Component Name: axi_uartlite_0

AUTO AXI CLK Frequency: 100.0 [10-300]MHz

Baud Rate: 230400

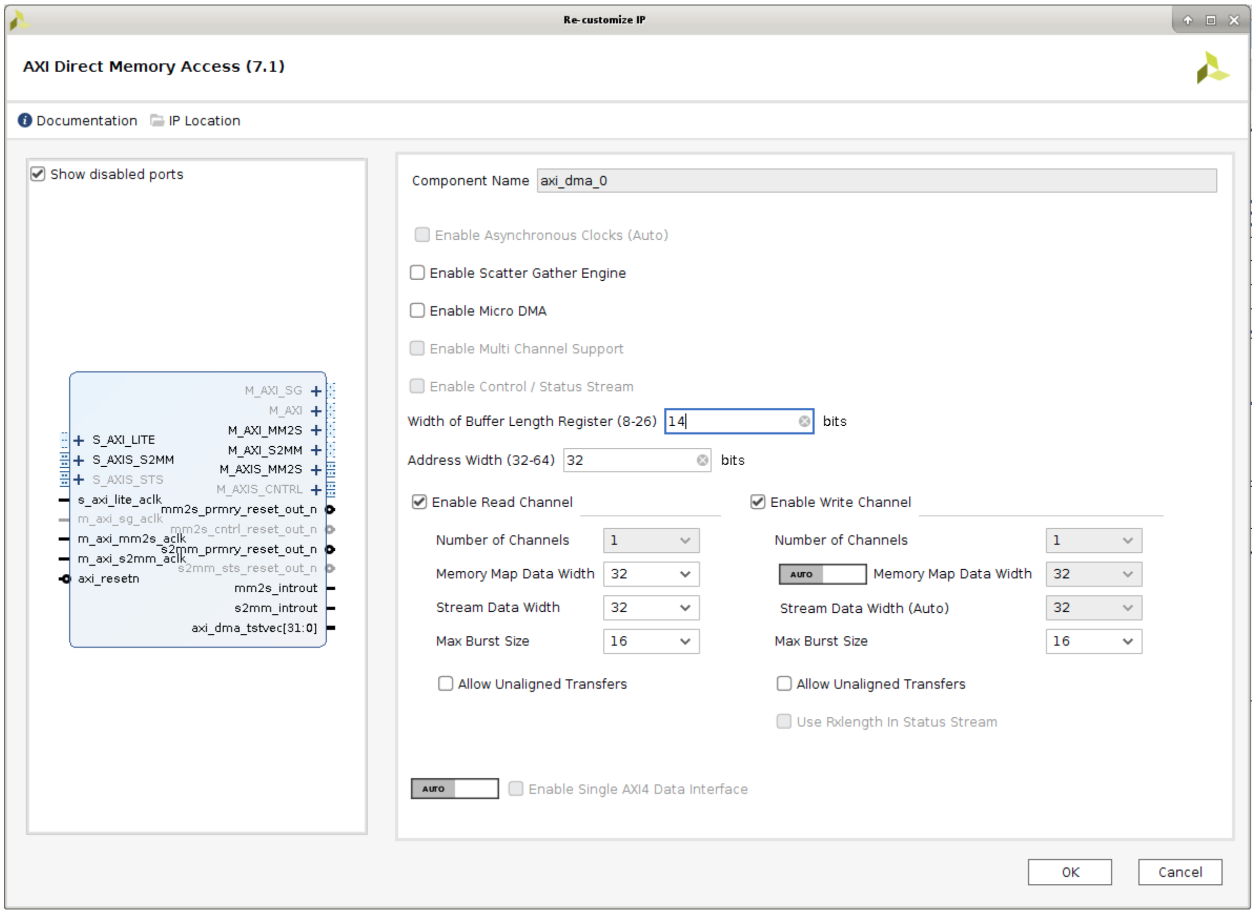
Data Bits: 8 [5 - 8]

Parity: [✓ No Parity] Odd Even

Figure 13: Configuration of the AXI UARTlite

4.2.3 DMA

The DMA block to communicate with our custom neural network hardware was determined from the help of online resources in which we use the AXI Direct Memory Access (7.1). There are no changes to the default configuration of this block.



[✓] Show disabled ports

Component Name: axi_dma_0

[ ] Enable Asynchronous Clocks (Auto)
[ ] Enable Scatter Gather Engine
[ ] Enable Micro DMA
[ ] Enable Multi Channel Support
[ ] Enable Control / Status Stream

Width of Buffer Length Register (8-26): 14 bits
Address Width (32-64): 32 bits

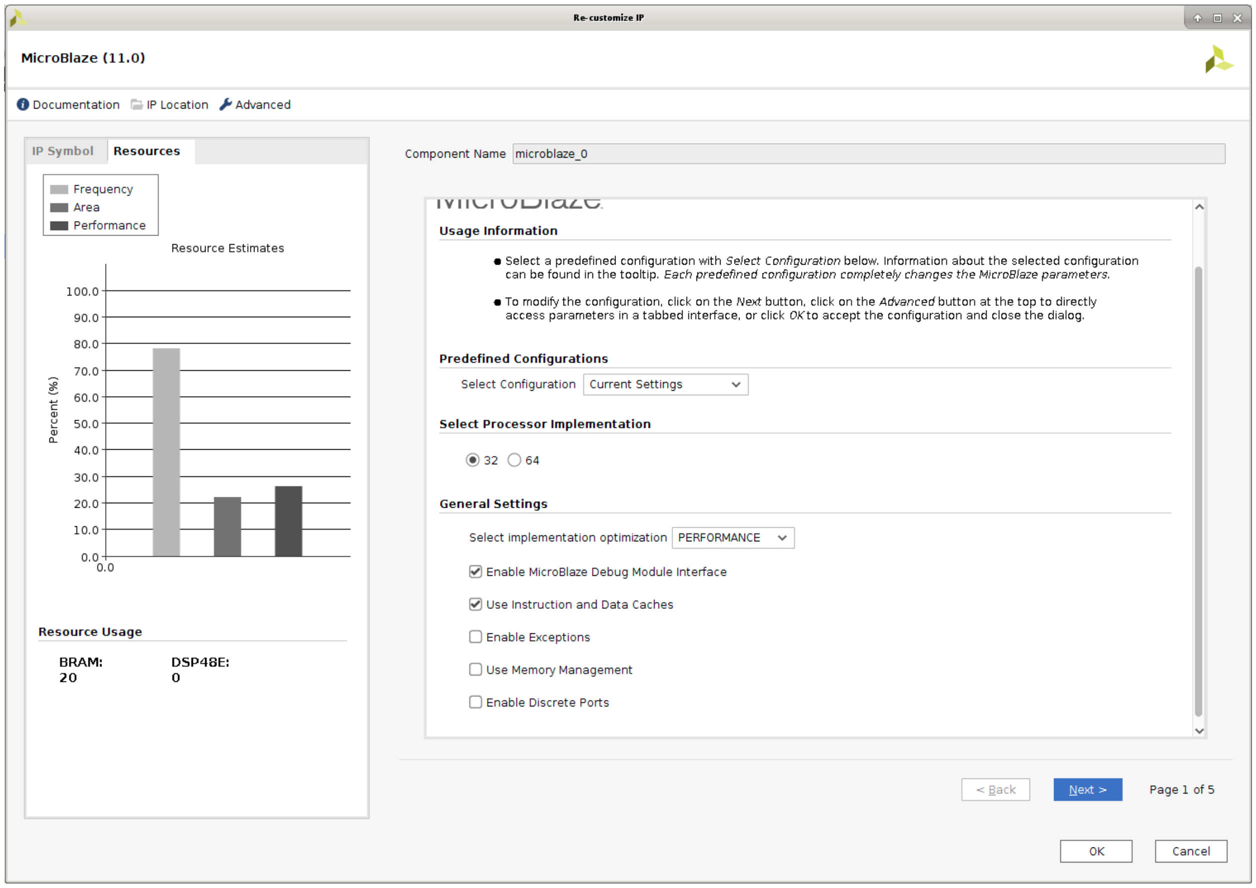
[✓] Enable Read Channel		[✓] Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	32	AUTO Memory Map Data Width	32
Stream Data Width	32	Stream Data Width (Auto)	32
Max Burst Size	16	Max Burst Size	16
[ ] Allow Unaligned Transfers		[ ] Allow Unaligned Transfers	
		[ ] Use Rxlength In Status Stream	

AUTO [ ] Enable Single AXI4 Data Interface

Figure 14: Configuration of the AXI DMA

4.2.4 MicroBlaze

The MicroBlaze (11.0) block used had the same design as the one for the Warmup Demo and Tutorial 5 in which it is configured to enable the Peripheral AXI Interface for instructions. With implementation optimization made for performance.



Component Name microblaze_0

MicroBlaze

Usage Information

- Select a predefined configuration with *Select Configuration* below. Information about the selected configuration can be found in the tooltip. *Each predefined configuration completely changes the MicroBlaze parameters.*
- To modify the configuration, click on the *Next* button, click on the *Advanced* button at the top to directly access parameters in a tabbed interface, or click *OK* to accept the configuration and close the dialog.

Predefined Configurations

Select Configuration: Current Settings

Select Processor Implementation

[✓ 32] 64

General Settings

Select implementation optimization: PERFORMANCE

- ✓ Enable MicroBlaze Debug Module Interface
- ✓ Use Instruction and Data Caches
- ☐ Enable Exceptions
- ☐ Use Memory Management
- ☐ Enable Discrete Ports

Resource Usage

BRAM: 20 DSP48E: 0

Figure 15: Configuration of the MicroBlaze

4.3 Software

4.3.1 Client Application (NN_HW_Client)

The client application leverages the client code from Tutorial 5 and the Warmup Demo. This allows us to send the neural network output to the server FPGA. The software is designed to first set up the UART connection using the Uartlite drivers and then loop to receive the testing labels from the PC and store it on the DDR3 of the Nexys board through dynamic memory allocation (MALLOC). Once all the data for the labels is received through UART, then the MicroBlaze asks for the testing images and loops until all data has been received. Once all data has been received, the Digilent LWIP interface is then initialized. After initialization, the label data is then sent over to the server FPGA. After this data has been sent, through DMA, images are sent in order from the DDR3 to the neural network IP. For 10,000 testing images, after 50 images have been processed, the classified outputs of these 50 images are sent in one packet to the server FPGA. This continues until all testing images have been processed.

4.3.2 Server Application (NN_Server)

The server code from Tutorial 5 and Warmup Demo has been leveraged for our server application on the MicroBlaze on the second FPGA through the Digilent LWIP interface. We were able to use this code to receive the test labels and neural network outputs from the first FPGA. This is done as we are waiting for an interrupt from the ethernet controller with received data. Once this data is received it is sent from the FPGA to the PC through UART which is then taken over by the Python program to visualize the data as we receive it in real-time.

4.4 Python Visualization

From the second FPGA, the output results are transferred to the second PC, which has a python program running on it. The python program listens at COM5, and whenever there is data received, it will first look for the data indicator (ASCII value of 130). This is because there are some debugging print functions in the MicroBlaze, and this debugging information will also be sent to the python program, so a data indicator is needed to indicate the following data is output results.

From the second FPGA, the expected digit and the actual digit produced from the neural network are sent to the python program. The python program then collects all the data and stores them in separate lists. The overall Accuracy is calculated by summing all the correct predictions and divided by the number of received data. The same approach is used when calculating the accuracy for each digit.

5. Description of Design Tree

On our GitHub, we have posted a README file that introduces our project as well as how to use it and what the other two directories “src” and “docs” contain. Within the “docs” directory we include directories for presentations and reports that hold the presentations and reports that were done throughout the project from the proposal. We also have a directory that holds a 5-minute video that goes over our project and demos the full system processing 10k images on both client and server-side of our system. Within the “src” directory we have the directories that hold the Vivado 2018.3 projects for the 784-16-10 Neural Network, the client system and the server system. We also have a folder that contains the files which contain 10,000 testing labels and images which will be sent to the client FPGA.

Link:
https://github.com/rewingchow1/G9_Hardware-Approach-of-Forward-Propagation-in-Neural-Network-on-FPGA

5.1 How to Use:

Client with Neural Network:

To Run Out of the Box:

1. Open Vivado 2018.3, click open project, search for the T5.xpr in Client_NN, then open it
2. Click file, then click Launch SDK
3. Once SDK is open, open main.c, change SRC_IP4_ADDR to the IP address of your FPGA. Ex, if your IP address is 1.1.22.2, then set ADDR to 22. If your IP address is not in the format of 1.1.ADDR.2, then go to line 75, change to your IP address
4. Change SRC_MAC_ADDR to the corresponding MAC address of your FPGA, go to line 74
5. Change DEST_IP4_ADDR to the corresponding IP address of your server FPGA with the corresponding port for DEST_PORT (Default: 7), go to lines 80 and 82
6. Open TeraTerm and connect the terminal to COM5, with a BAUD rate of 230400
7. Program the FPGA, then run with GDB configuration
8. When prompted in the TeraTerm terminal to send labels, click "File" then "Send file..." and locate the labels file ("test_labels_10k") and send it as a binary
9. When prompted in the TeraTerm terminal to send images, click "File" then "Send file..." and locate the images file ("test_images_10k") and send it as a binary

NOTE: Setup server before setting up the client.

To Configure Neural Network or Client System:

Neural Network:

1. Open Vivado 2018.3, click open project, search for the edit_NeuralNetHandWritten_v1_0.xpr in AXISHandWritten_16, then open it
2. Open the source files and make necessary changes
3. Synthesize and Implement
4. Once implemented successfully, export IP

5. Replace IP block in T5.xpr in Client_NN with the updated design
6. Follow next session to update the client system

Client System:

1. Open Vivado 2018.3, click open project, search for the T5.xpr in Client_NN, then open it
2. Open block diagram, and make necessary changes
3. Synthesize, Implement and Generate Bitstream for new design
4. Once bitstream has been generated, click file and export hardware including bitstream
5. Click file, then click Launch SDK
6. Make any MicroBlaze software changes to client code
7. Program FPGA with new bitstream
8. Run application configuration on Client FPGA

Server & Python Visualization:

To Run Out of the Box:

1. Open Vivado 2018.3, click open project, search for the T5.xpr in Server_NN, then open it
2. Click file, then click Launch SDK
3. Once SDK is open, open main.c, change ADDR to the IP address of your FPGA. Ex, if your IP address is 1.1.22.2, then set ADDR to 22. If your IP address is not in the format of 1.1.ADDR.2, then go to line 196 and line 251, change to your IP address
4. Change MAC_ADDR to the corresponding MAC address of your FPGA
5. Connect the SDK terminal or TeraTerm to COM5, with a serial speed of 9600
6. Rebuild the project, program the FPGA, then run with GDB
7. On the terminal, you should see information that states the server is ready to receive data
8. Once the server is ready, disconnect the terminal, open the pyserial.py file with IDLE (or other editors), and run it, you should see on the terminal it prints out listening at COM5
9. Once data is received from the client, a plot with 10 subplots should show up

To Configure Server System:

1. Open Vivado 2018.3, click open project, search for the T5.xpr in Server_NN, then open it
2. Open block diagram, and make necessary changes
3. Synthesize, Implement and Generate Bitstream for new design
4. Once bitstream has been generated, click file and export hardware including bitstream
5. Click file, then click Launch SDK
6. Make any MicroBlaze software changes to server code
7. Program FPGA with new bitstream
8. Run application configuration on Server FPGA

5.2 Repository Structure:

src

- **AXISHandWritten_16**: Includes source code for pre-trained MNIST 784-16-10 Neural Network with corresponding Vivado 2018.3 project.
- **Client_NN**: Includes system design for client FPGA along with corresponding Vivado 2018.3 project and Vivado SDK application to run client software on MicroBlaze.
- **Server_NN**: Included system design for server FPGA along with corresponding Vivado 2018.3 project and Vivado SDK application to run server software on MicroBlaze. Also includes a pyserial.py Python3 file which is used to visualize neural network performance.
- **NN_Files**: 10k testing images and labels.

docs

- **presentations**: Proposal, mid-project and final presentation for the project.
- **reports**: Proposal and final report for the project.

6. Tips & Tricks

6.1 Tips for The Design Process:

We suggested that when building a digital hardware system, it might be easier to prototype your design in software, then translate your design to hardware.

During the design process, we first prototyped our design in software (implemented a neural network with backpropagation in C), which provided us proof that this design could work, at least on a software level. Then based on this software approach, we then moved the software design to the MicroBlaze (with some modifications for compatibility reasons) and proved that this design is capable of producing accurate results with a softcore like MicroBlaze. As a final step, we integrated the hardware design of the neural network into the system.

6.2 Tips for Scalability

The scalability of your design should be considered at the beginning of the design process. This is helpful for scale your designs and could also help you debugging easier. From our experience, we first considered building a neural network for a simple logic function. This neural network only has 2 neurons in the input layer, 3 neurons in the middle layer, and 1 neuron in the output layer. When testing it, we used logic functions, such as AND, OR, and XOR, to see if the neural network can predict the correct result or not. After we proved the design is capable of giving correct results given the logic functions, we can easily scale this design to a 784-16-10 neural network, with little effort. Also, debugging the 2-3-1 neural network is much easier, since the input value is only 1 or 0, and the output value is in the range from 0 to 1.

6.3 Tips for Working with VIVADO

When working with Vivado, there might be some bugs or features causing problems, and sometimes the directory name also causes problems. Always use a directory name with no special characters (white space for example) and all letters should be English letters. Sometimes when the design fails at synthesis, and can’t find out the problems from log files or from forums, you may try creating a new design and copy your files to the new project.

7. Conclusion

A three-layer neural network is successfully implemented on FPGA with accuracy above 90%. This is accomplished by implementing a vector multiplication unit, ReLU IP and AXI-Stream wrapper. Pretrained weights are used in this neural network and backpropagation is done on MicroBlaze. Future work can be having backpropagation also implemented on hardware. A TCP/IP communication across two FPGAs set up for displaying historical accuracy so that we can visualize trends of accuracy. Since machine learning is becoming more and more popular, we believe this project can have a lot of potential.

References

[1] Intel.com, “CNN Implementation Using an FPGA and OpenCL™ Device”

https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/machine-learning.html#:~:text=Neural%20networks%20are%20inspired%20by,in%20particular%20the%20human%20brain.&text=FPGAs%20are%20a%20natural%20choice,resources%20in%20the%20same%20device.

[2] medium.com, “Building Neural Network Framework in C using Backpropagation”

https://medium.com/analytics-vidhya/building-neural-network-framework-in-c-using-backpropagation-8ad589a0752d

[3] projectf.io, “Fixed Point Numbers in Verilog”

https://projectf.io/posts/fixed-point-numbers-in-verilog/

[4] researchgate.net, “IMPLEMENTATION OF A SIGMOID ACTIVATION FUNCTION FOR NEURAL NETWORK USING FPGA”

https://www.researchgate.net/publication/224843989_IMPLEMENTATION_OF_A_SIGMOID_ACTIVATION_FUNCTION_FOR_NEURAL_NETWORK_USING_FPGA

[5] seas.ucla.edu, “FPGA Matrix Multiplier”

http://www.seas.ucla.edu/~baek/FPGA.pdf

[6] Michael Nielsen, “Neural Networks and Deep Learning”

http://neuralnetworksanddeeplearning.com/