Final Report

Eric Bower and Tyler Zhang

## What we did:

**Multiprocessing**

For this project, we were able to find two methods that brought the runtime down below twelve seconds. One method was using multiple processes running in parallel and the other method was storing every layer on the FPGA.

To be able to do parallel processing on the pynq, we used python's multiprocessing library. This library is very similar to multithreading with one major difference. Rather than using threads, it uses processes therefore effectively bypassing the Global Interpreter Lock which prevents multiple threads from running simultaneously.

First, we had to divide the data and labels into arrays for each process to use. The following code shows this implementation:

```
numProcesses = 2

# Split the data and labels into numProcess arrays for each process to use
### i.e. for two processes, split the data into two arrays so each process take half of the entire data
# Limit of 4 processes for the pynq
# 2 Processes yields the fastest runtime
data1 = []
data2 = []
data3 = []
data4 = []
data = [data1,data2,data3,data4]
labels1 = []
labels2 = []
labels3 = []
labels4 = []
labels = [labels1,labels2,labels3,labels4]

num = numProcesses
for i in range(num):
    data_index1 = int(len(test_imgs)/num)*num - ((num-i)*int(len(test_imgs)/num))
    data_index2 = int(len(test_imgs)/num) + i*int(len(test_imgs)/num)
    labels_index1 = int(len(test_labels)/num)*num - ((num-i)*int(len(test_labels)/num))
    labels_index2 = int(len(test_labels)/num) + i*int(len(test_labels)/num)

    if(i==num-1):
        data_index2+=1
        labels_index2+=1

    data[i] = test_imgs[data_index1:data_index2]
    labels[i] = test_labels[labels_index1:labels_index2]
```

Code Snippet 1: Dividing data and labels into smaller arrays

This code simply divides the data equally into numProcesses arrays. For instance, using two processes would result in the data being split into two arrays. We did this using loops instead of manually so we could quickly change how many processes we wanted to run to determine which number would result in the shortest runtime. We then called the evaluate function using these new arrays.

Next, to create the processes, we used the multiprocessing.Process class. The following image shows the code for creating these processes:

```
def evaluate(net, data, labels, numProcesses):
    corrects, wrongs = 0, 0

    jobs = []

    recv_end, send_end = mp.Pipe()

    for i in range(numProcesses):
        jobs.append(mp.Process(target=evaluate_mp, args=(net, data[i], labels[i], send_end)))
        jobs[i].start()

    for i in range(numProcesses):
        recv = recv_end.recv()
        corrects += recv[0]
        wrongs += recv[1]
    send_end.close()
    recv_end.close()

    for proc in jobs:
        proc.join()


    return corrects, wrongs
```

Code Snippet 2: Evaluate function using multiple processes

Creating new processes is relatively straightforward. The argument ‘target’ in mp.Process is the function you want the process to run. The ‘args’ argument is the arguments you want the process to pass to the function before it runs. To receive the return value of the function a process is running, we used the multiprocessing Pipe class. This class provides an input and output of a ‘pipe’ in which you can send and retrieve data. The function that each processes calls is the same function as was given to us with one small edit:

```
            wrongs += 1

    send_data = [corrects, wrongs]
    pipe.send(send_data)
```

Code snippet 3: Provided evaluate function with change

Instead of returning the corrects and wrongs, we had to send an array of these to the pipe. Looking at code snippet 2, you can see that we used the recv() function to get the values that are sent to the pipe. The recv() function is blocked until something on the pipe is sent so it waits for a process to send data before trying to read it. After receiving the data, we call the join() function to make sure the processes have all finished.

**Hardware Acceleration**

Another approach was storing all layers of the neural network on the FPGA. We started by using the given Weights_Extract.ipynb file to generate floating point encodings for the weight matrices and bias values. Next, we created modules for dot, bias, and tanh similar to the existing ones in the bd_fpga_wrapper. Each module is specific to the sizes of its inputs and outputs. For example, dot_80_40.sv declares a dot module with an 80x40 weight matrix. Similarly, bias_40.sv declares a bias module with the corresponding bias vector of size 40, and tanh_40.sv calculates the LUT values for 40 inputs. Each of SystemVerilog files are encapsulated in a Verilog file that handles the AXI Stream interface, labeled with an “axis_” prefix.

We added these modules as well as floating-point modules to the block design to create the layers of the neural network, as shown below.

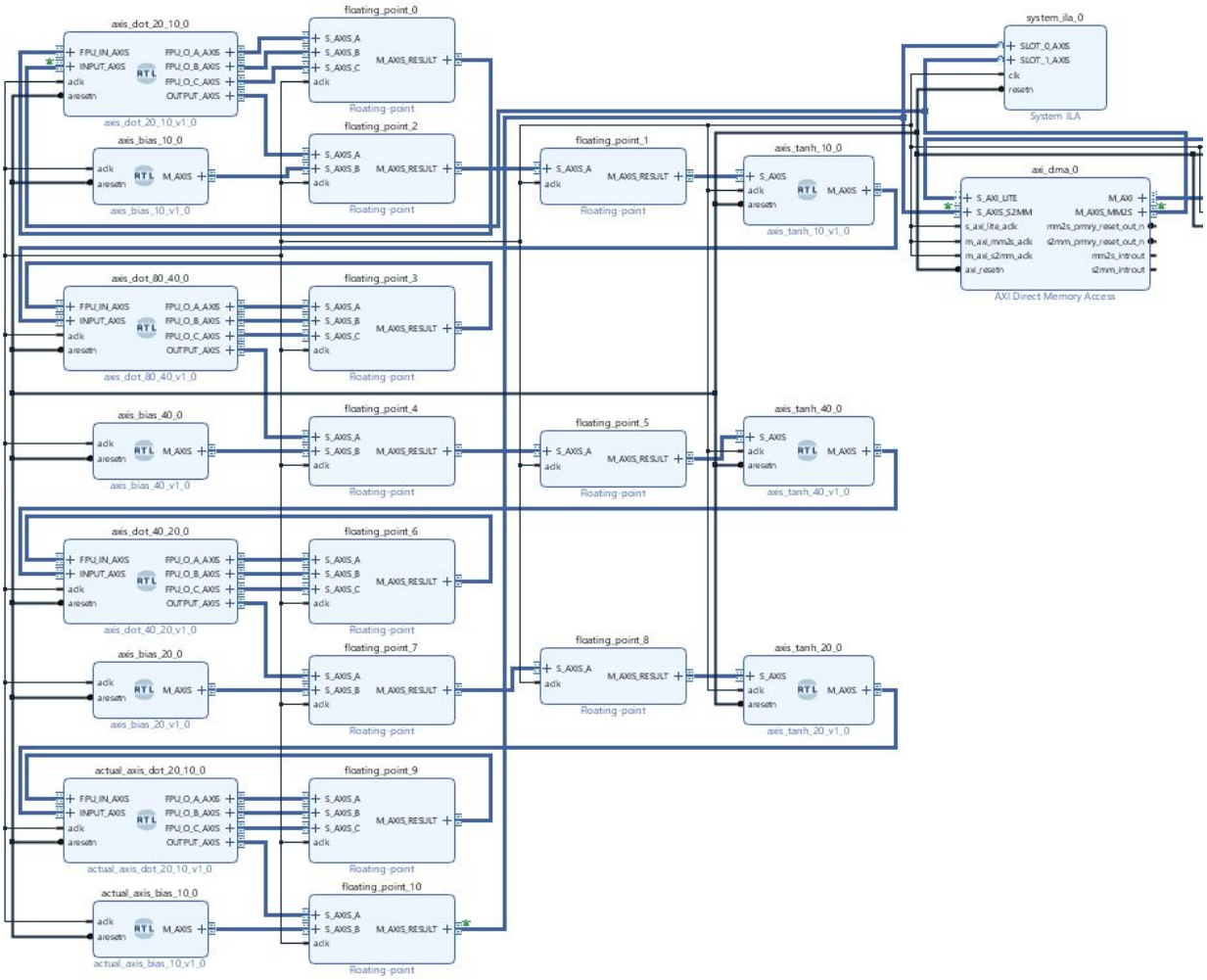


Figure 1: Block design showing the layers connected to the DMA.

Note that in the diagram above, the module labeled “axis_dot_20_10_0” actually stores the 784x80 weight matrix of the first layer, not the 20x10 weight matrix of the last layer, because renaming modules is difficult. The same idea holds for the axis_bias_10_0 module.

To use the bitstream in Python, I created a network object with one custom layer that interacts with the hardware. The layer initializes the overlay for the Pynq and input/output buffers. The layer also defines a propagation function that sends and receives data from the DMA. The code for this is shown below:

```
from pynq import Xlnk
from pynq import Overlay

class hardwareLayer(Layer):
    def __init__(self, bit, input_sz, output_sz):
        self.overlay = Overlay(bit)
        self.dma = self.overlay.axi_dma_0

        xlnk = Xlnk()
        self.input_buffer = xlnk.cma_array(
                                shape=(input_sz,),
                                dtype=np.float32)
        self.output_buffer = xlnk.cma_array(
                                shape=(output_sz,),
                                dtype=np.float32)

    def forward_propagation(self, input):
        # use this for first layer
        np.copyto(self.input_buffer, input)

        self.dma.sendchannel.transfer(self.input_buffer)
        self.dma.recvchannel.transfer(self.output_buffer)

        self.dma.sendchannel.wait()
        self.dma.recvchannel.wait()

        #output expects a [1,output_sz] matrix (not vector)
        return self.output_buffer.reshape(
                                1, len(self.output_buffer))
```

```
hw_net = Network()

# 784 and 10 are hardcoded in bitstream
hw_net.layers.append(hardwareLayer('./verilog/bitstreams/784_10.bit', 784, 10))
```

Code Snippet 4: Creating the hardware accelerated network.

To use the hardware accelerated network, we simply call the evaluate function with the defined hw_net object.

## What works, what doesn't:

Both multiprocessing and storing every neural network layer on the FPGA achieved the desired run time and accuracy, but multiprocessing was ultimately faster than using the FPGA. Several tests of using multiprocessing actually increased the runtime, which was unexpected, and we ultimately found that using two processes resulted in the quickest runtime. One way to create processes was to set the process creation to 'spawn' rather than 'fork'. At first, we used 'spawn', but this greatly increased the runtime (effectively adding 20 seconds per added process). In hindsight, though, this was apparent as the documentation said that 'spawning' rather than 'forking' processes was significantly slower on some machines.

In terms of hardware, we initially attempted to replace individual layers with FPGA calculations. We considered the possibility that doing just the first layer on the FPGA would be beneficial because the first layers perform calculations on more numbers. However, this resulted in slower performance compared to software because of the amount of array manipulation and the fact that the system still needs to send and receive large arrays via the DMA.

## Who did what:

- Eric wrote the multiprocessing code. He also wrote about multiprocessing in this paper.
- Tyler created the modules and edited the block design to extend the Verilog code to work for all layers. He also adapted the code to work with the new block design and wrote about hardware acceleration in this paper.

## Analysis:

Our final runtime was around 8.2 seconds with each run varying from 8.1-8.3 seconds. This runtime was achieved by using multiprocessing. We arrived at our final design for multiprocessing essentially through trial and error. The multiprocessing module provides several different ways of creating processes, and we tested almost all of them. We learned that multiprocessing in Python has many caveats and it doesn't always work as well as you might think. One frustrating thing with multiprocessing was its behavior in jupyter notebooks. There were often times when processes would simply not run and we could not figure out why. To fix this, we moved code into a python file, which caused problems with loading the network using pickle. Eventually, we found our final design which worked in jupyter notebooks as well as decreasing the runtime. If we could do this project again, we would definitely read the documentation more carefully, as this would have saved us time focusing on a method that never would have worked.

The FPGA calculated the results in about 11.3 seconds. We started off by replacing software layers one by one, but we gained speedups the more layers we stored on the FPGA. This led us to storing all layers on the FPGA. We completed both multiprocessing and FPGA acceleration in parallel, but after comparing the two we learned that multiprocessing was ultimately faster (and easier). There seemed to be more ways to make parallelized code faster, but not as many ways to speed up hardware accelerated code without multiple FPGA's. A thing we would do differently if we were to re-do the hardware acceleration is to take more advantage of the ILA. We initially had accuracy issues because we forgot to specify the correct accuracy in our float-to-fixed module before tanh.