CS152B Final Project: Integer Identification

Group 3
Michael Hale (004-620-459)
Matthew Nuesca (904-440-067)
Shilin Patel (904-569-866)

Introduction

This final project for CS152B was an open ended exploration of an application of FPGAs. It involved designing and building the product using the knowledge obtained over the quarter. Our goal for this project was to familiarize ourselves with the complexities of building a large multiple function system in verilog and learn more about the possibilities of programmable hardware.

Our project was an integer identifier using convolutional neural networks trained on the popular MNIST dataset for handwritten digits. The FPGA would train on the data set and store the optimal set of weights. The data set would be sent over serial from a personal computer. After training is completed, on the PC, the user can draw an image and send it over serial to be tested on the trained weights. This will then print out the correct response to a seven segment display. All controls are done without input on the FPGA, but by feeding in different sets of data from the PC, either training data or our test image data.

Setup

The set up for the board and peripherals for this project is very simple. The board simply requires the seven segment pmod to be plugged into a pmod port. This is to display the output result of our test run, using a drawn image passed from the PC. The seven segment display uses a PMOD port of the FPGA board, where implementation is specified in the seven_seg module. . To enable communication with the personal computer, an serial cable must be connected to the board and an RS232-USB adapter to the laptop (unless the laptop has a serial port for some reason). This completes the setup for large hardware components.

Most of the setup for this project included initializing modules within the FPGA itself. The RS232 Serial module, Block RAM cores, seven segment, and Fixed Point modules all are crucial building blocks for the actual convolutional neural network.

RS232 Serial Module: There are actually two separate modules for RX and TX communications. It works by instantiating the modules and providing ready bits for when data can be read or written. Once these bits are flipped, the serial communication can begin. Writing out is a simple process of assigning to an output register, while reading places read data on a wire to be moved somewhere for use. These modules are not unique to any other serial uart implementation on an FPGA board. We only had to make minor adjustments in integration with our top module when using a standard serial module.

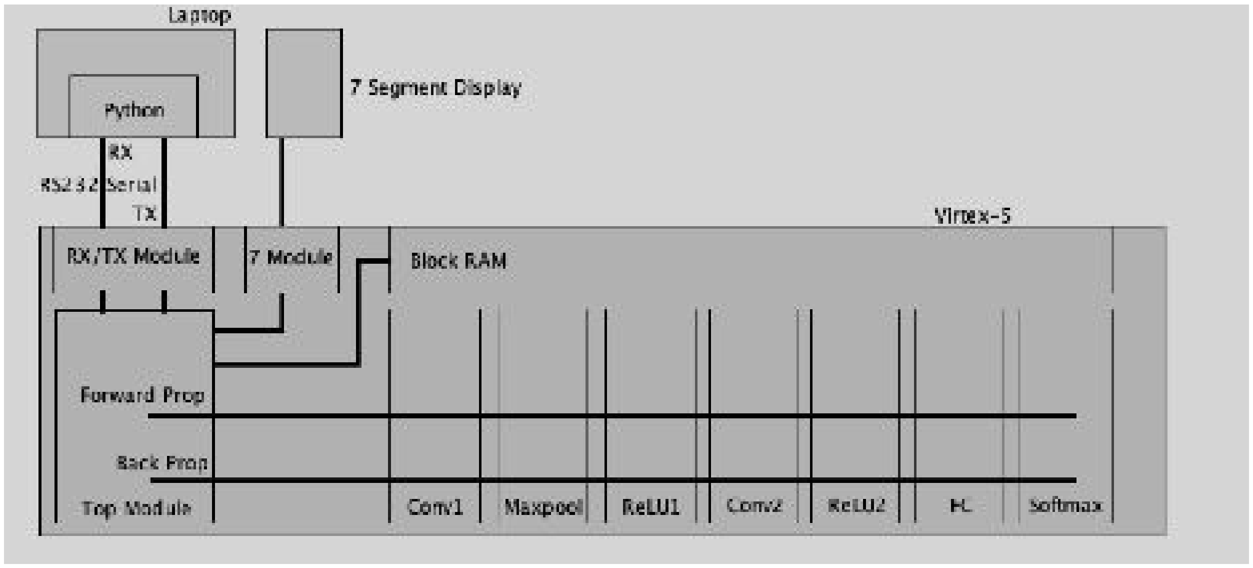
Block RAM cores: These are generated using the Xilinx core generator. For this program we generate 12 block ram cores. These are variable in size depending on the number of weights/errors for each layer of the CNN. Each of the layers(convolutional 1&2, fully connected) consists of 4 cores. They are for storing the last input to the layer, the weights for forward and backward propagation, error for backward propagation, and the updated output value.

Seven Segment Display Module: This module is rather simple. A custom clock is divided from the master clock to ensure the refresh rate of the display is proper for the user to see the shown digit. The module allows the user to input a digit in binary and this will be printed using the PMOD specification detailed the specification.

Fixed Point Modules: Fixed point modules are for performing fixed point operations. They can perform multiplication, addition, division, subtraction. These were built to take two inputs and provide a single output with the correctly formatted fixed point byte size. Fixed point numbers are represented by 32 bits where 15 bits are fractional numbers, 16 bits are integers, and 1 bit is signed. All of these calculations are done using sequential logic. One of the main uses of our fixed point modules is to calculate exponentiation for fixed point numbers. Due to the limit number of multipliers on the FPGA, our implementation consists of a state machine which latches, calculates, then combines the results. During the latch state we ensure that the value does not overflow. We then calculate both fractional and integer parts of the fixed point module and then combine them into one 32 bit piece of data.

Once these modules are set up, the CNN can be built using its own building blocks. These blocks are convolutional layers (including convolution, ReLU, and maxpool), a fully connected layer, and a softmax output layer.

High Level Understanding



This block diagram shows the overall architecture of our convolutional neural network. As described in the setup for the Virtex5-FPGA, the BlockRAM, SevenSeg, and RX/TX serial modules are instantiated for use by the top module as well as the CNN layers. The RX/TX module communicates with the laptop running a python script to send the data in the MNIST dataset to the board for the training process. For each image the forward propagation begins through each of the CNN layers. Once the forward propagation has completed, the data moves backwards adjusting weights and error values stored in the BRAM.

Python Script

On the personal computer two python scripts were created, one for training and one for testing. The training script loads the MNIST dataset of 60,000 28x28 greyscale images and attempts to connect to the serial module on the FPGA using the pyserial library. Once a connection is established and the FPGA sends a signal that it is ready to receive one image, the script sends one of the 60,000 images in the dataset along with the corresponding label value (information about which digit is drawn) concatenated to the bytearray.

For the testing portion of the python implementation the goal was to allow the user to draw a digit on a canvas and then obtain the bytearray and send it serially for testing. To do this required the use of python libraries Tkinter and Pillow which allow a canvas to be drawn upon and the resulting image saved resized and formatted into a 28x28 bytearray.

Convolutional Neural Network Implementation

Each layer of neural network works similarly in the way it interacts with data and the top module. Both convolutional and the fully connected layers are based on a state machine which contains the following states:

- Receive forward input
- Perform forward computation
- Output forward value
- Receive output error
- Perform backprop updates
- Output error to input layers

For these layers the first state will signal when it is ready to receive data. Only then will it receive valid data. The data is not passed directly between layers but rather will move through the top module which can signal the validity of the input/output data.

The convolutional layers are composed of the states listed above. Each convolutional module includes all of the convolutional, ReLU, and maxpool algorithms. The first layer received data in the dimensions of 28x28 and outputs data at dimensions 12x12. The second layer takes the 12x12 and reduces it to 10x10. Each convolutional layer performs the standard operations expected from this layer in a neural net. However, because we are implementing this on an FPGA and with only 32 bits (as opposed to 64), issues of overflow and hardware limitations become realistic. Overflow is handled properly by our fixed point arithmetic modules, however because of the inability to multiple more than 48 operations in parallel, our convolutional layer uses multiple clock cycles for each state so the multiplications will work.

The fully connected layer takes a 10 inputs and outputs 10 outputs as well. The same limitations exist for fixed point arithmetic as in the convolutional layer. These issues are resolved by our clock aware fixed point multiplication.

The softmax output function is the last layer in our CNN, it places our 10 outputs between 0 and 1 with the summation of them equal to 1. It consists of multiple states as well.

- Receive input
- Perform additions and exponentials
- Perform divisions

- Backpropagation

These states, unlike the design in the conv and FC layers, do not simply transition from the prior to the next. In the receiving input state we determine whether to transition to the computational states or to the backpropagation state. Each of the states in this softmax state machine perform the mathematical requirements of the softmax function, however there are issues with max values and overflow which are handled by the fixed point exponentiation implementation.

Challenges

This project was unfortunately plagued with issues from conception. Our first proposal was to use DDR2 SDRAM in order to host the MNIST dataset on the board itself. The idea was to simply access the SDRAM instead of constantly checking over serial for the next image to process. The process to initialize the SDRAM took over week. It began with problems on improper setting of parameters in the xilinx GUI menus. After these issues were resolved and the SDRAM module was created, we ran into the problem of the ready signal not triggering for loading data into the memory. Even after we spent hours pouring over the manual and consulting with others attempting to use the memory, we were forced to look for different methods of handling the dataset, as this module was proving to be too much trouble to initialize properly. We settled on transferring one image at a time over serial and notifying the PC, when sending an image was appropriate with a ready signal from the FPGA serial module.

Once of the requirements of the convolutional network we were building was the necessity to store inputs, outputs, weights, and error values for prolonged periods of time and have them accessible to each layer. Our first attempt was to simply store all of this data in registers, however it soon became apparent that the LUT would not be able to host all of this information. The solution to this problem was to use the core generator to make BRAM modules. Only after the BRAM was initialized and working it became clear to us that the reset pin that the core generator created did not have consistent behaviour on our Virtex5 product. This resulted in the need to reprogram the board after each test run. An inconvenience for sure, however still usable.

When implementing our CNN layers, especially in the convolutional and fully connected layers our understanding of the arithmetic components of the FPGA were put to the test. We severely overestimated the ability to do multiplication in parallel because we didn’t realize that the FPGA had 48 multipliers. Our previously built implementations of the conv and FC layers would have to be adjusted to multiply sequentially (not in parallel). The solution to this was adjusting one layer to take place over many more clock cycles than previously planned. This actually didn't have as adverse effects on our training speed as predicted. We would still be able to train on the MNIST dataset in only a few minutes.

Apart from these challenges to do with our lack of information or the complexity of a certain feature, some other challenges faced was the problem of being tied to the lab to do all of our work. Due to our reliance on the core generator and other xilinx tools, we were unable to work and simulate from our personal computers as effectively as other groups. A final odd bug we encountered while testing softmax was an iSim graphical error where the values of the waveform would become invalidated when too much data was viewed through the window. We didn't understand this was a graphical error and this led to more wasting of precious time.

In the end our final product was delivered in the form of the test implementation which completed serial, BRAM, testing(w/o softmax), and seven segment portions of the network completed. This allowed for a demo that output a predicted digit from a drawn image. However, the weights were acquired by training on a python implementation of our CNN and then loaded into the BRAM for access during the test run. Our infrastructure for training exists but unresolved errors forced us to abandon it.

Conclusion

In total, even with all of the difficulties, this project was extremely worthwhile to undergo. We learned in depth about the hardware components associated with the FPGA as well as built in tooling Xilinx provides. Some of us also had to study how CNNs work from the ground up and familiarize ourselves with the formulas and algorithms that build neural networks. Some python library exploration was a bonus experience! It was a rewarding process from start to finish even when nothing worked to the point where only some things worked. If done again, we would definitely want to avoid the time sink that dealing SDRAM was, however in an area without many resources outside of few forum posts and complex official documentation, we believe we achieved a good deal of what we proposed.