Final Project

**Neural Network Inference on FPGA**

ECE 6213 – Design of VLSI Circuits

The George Washington University – Fall 2022

Osama Yousuf, Joseph Riem

1. **Project Description**

In this project, we implemented inference of a feed-forward neural network on FPGA for an image classification problem. Mini-ITX boards from the ADAM Lab @ GW, as shown in Figure 1 below, were used. These boards contain a hard processor or a processing system (PS), as well as programmable logic (PL). The implementation was done in two ways – firstly through a high-level synthesis workflow, and secondly through custom RTL modules (written in SystemVerilog). Accuracy results were obtained from both neural networks (HLS and RTL) and it was found that the custom RTL network occupied less resources compared to the HLS network and achieved a similar accuracy performance. Additionally, a web camera was integrated into the inference workflow, and we successfully demonstrated how our network could classify an image taken from the camera in real-time using a server-client approach.

**Graphical user interface

Description automatically generated**

Figure 1: Picture of the boards used for this work

1. **Specification**
   1. **Problem Overview**

The problem investigated is a reduced version of the handwritten MNIST dataset. This is a dataset consisting of handwritten images of digits labelled 0 through 9 and is publicly available. The neural network is fed an image from this dataset as an input, and it attempts to classify the digit as its output. Figure 2 below captures this input-output function, with the neural network treated as a black box.

Graphical user interface, application

Description automatically generated

Figure 2: Overview of neural network inference for image classification

The MNIST dataset has 60,000 training images, and 10,000 testing images. Originally, all of these images contain a total of 28 x 28 pixels. For our project, we reduced the dataset size to 18 x 18 by center-cropping these images for purposes of achieving fast inference latency and supporting a future hardware prototype where an array of memristive devices will be used to represent network weights. Additionally, the images were binarized for simplicity i.e., input images can either have pixels that are fully black or fully white.

* 1. **Neural Network Operation**

A feed-forward network has two modes of operation: **training** and **inference**.

In the **training** phase, the network has a forward pass as well as a backwards pass. In the forward pass, the network is presented an image in the form of a simple vector. This vector gets multiplied to the internal parameters of the network (weights and biases – explained in Section 2.4 below), and produces an output vector, which is then used to draw a classification prediction. In the backwards pass (referring to the backpropagation algorithm), the internal parameters are adjusted based on the prediction which was made. In brief, the idea is as follows: Based on the quality of the prediction, a gradient matrix is computed using a loss function for each network layer or weight matrix. Once all gradients are calculated, they are scaled by a scalar hyperparameter known as the learning rate. Finally, the network weights are updated by simply subtracting from the scaled gradient. This process (of a forward pass and a backwards pass) is repeated with different images from the training dataset until either the network has been trained for a set number of iterations, or the network meets some accuracy benchmark.

In the **inference** phase, the network only goes through a forward pass, meaning that internal parameters of the network are not adjusted.

For our project, we implemented the training phase in software and only the inference phase on FPGA. Once the software weights have been trained to desired accuracy performance, they are converted to text files and are loaded in RTL code in custom-width fixed-point precision as specified.

* 1. **Network Details**

We implemented a simple 2-layer perceptron network (or a multi-layer perceptron/MLP), also known as a feed-forward network. The first layer had dimensions 324 x 10, and the second layer had dimensions 10 x 10, meaning that the architecture was 324 x 10 x 10. The HLS network was trained with biases since the HLS library used (hls4ml, more details in section 4.3) supported biases, whereas the custom RTL was trained without biases for simplicity. A ReLU function was used as a non-linear activation function, primarily because it was easy to implement on the FPGA. For the loss function, we opted to use CrossEntropy since we train the network in software and are not concerned about the ease to train the network in hardware.

* 1. **Inference Details**

Figure 3 below summarizes the inference operation in our chosen network architecture. First, the 18 x 18 input image is vectorized and multiplied with the weights of the first layer of the neural network. The output is passed to the ReLU non-linear activation function, which is then multiplied with the weights of the second layer of the neural network. The output vector is a 1 x 10 vector, and the predicted class is identified by doing a simple argmax (or softmax) operation.

Diagram

Description automatically generated

Figure 3: Details of neural network inference

1. **Task Breakdown**

We broke down the project into the following phases, so as to manage the workload in easy-to-accomplish chunks. Implementation details of each tasks are presented in Section 4.

* **Phase 0:** Implement neural network training in software
* **Phase 1:** RTL coding and simulations for neural network inference
* **Phase 2:** HLS coding and simulations for neural network inference
* **Phase 2:** Synthesis and deployment of the RTL and HLS neural networks on the Mini-ITX Zynq 045/100 FPGA boards
* **Phase 3:** Study how reducing network bit-precision impacts network accuracy in simulations (%)
* **Phase 4:** Implement inference based on live-camera feed instead of preset testing dataset

1. **Implementation**
   1. **Phase 0: Network Training in Software**

Related files are provided in the accompanying “Phase 0 – Network Training in Software” folder.

We implemented a neural network in PyTorch matching our proposed specification. Appropriate Python dependencies can be installed using the provided “requirements.txt” file via running the command “pip install -r requirements.txt”. The file “main.py” can be executed with appropriate command line arguments to train the network for either the HLS case or the RTL case, with the difference being that the HLS network has biases as well. For the custom RTL network, layers from the trained weights are exported as text files and are provided in the “data\_rtl” directory, whereas for the HLS network, the weights are dumped using PyTorch’s state dictionary functionality in a “.pt” format, provided in the “data\_hls” directory. Moreover, the center-cropped and binarized MNIST dataset is provided in the “mnist” directory.

* 1. **Phase 1: Custom RTL Network**

Related files are provided in the accompanying “Phase 1 - Custom RTL Network” folder. File “inference.v” contains our synthesizable top-level module for network inference, and “inference\_tb.v” is the accompanying testbench. The testing image dataset is provided in the “testimages.rar” archive, and this needs to be extracted and loaded in the testbench for replicating our simulation results. The flow for network inference is as follows:

* Module MM1 performs matrix multiplication for layer 1.
* Module fullReLU performs the non-linear activation function for inference on the output from MM1.
* Module MM2 performs matrix multiplication for layer 2.
* Module label produces the final classification.

We use fixed-point representation for the network weights and input images. Our testbench outputs a “label\_v{i}.txt” file, where {i} denotes the number of fractional bits in the chosen fixed-point representation (with a total of 16 bits), which has all the predictions for the input testing dataset (10,000 predictions, one on each testing image in the dataset). These output files are provided in the “outputs” folder. We wrote an additional verification script in Python “comparelabelfiles.py” to compare the predictions of the network in the “label\_v{i}.txt” files with the actual classes of each image in the testing dataset. Using 16-bit fixed point precision with 11 fractional bits, our network achieved an inference accuracy of 90.47 % on the 10,000 images in the testing dataset.

Figure 4 below shows the output verification waveforms of our testbench with signal meanings indicated on the left-most side.

A picture containing timeline

Description automatically generated

Figure 4: Testbench waveforms for our custom RTL neural network

* 1. **Phase 2: High-level Synthesis (HLS) Network**

Related files are provided in the accompanying “Phase 2 - High-level Synthesis (HLS) Network” folder.

The workflow is summarized in Figure 5. The trained network in “.pt” format from Phase 0 is used as the starting point. We utilized the “hls4ml” library for converting this Python network to synthesizable RTL code, which first converts the Python network into an equivalent C codebase where each layer is quantized using fixed point precision, and then uses Vivado HLS to convert this C network to equivalent RTL code. This conversion script is provided in the “hls\_creation\_hls4ml.py” script, and the hls4ml project folder is present in the “hls4ml\_project\_16\_8” directory (with 16 total bits, 8 of which are fractional).

A picture containing diagram

Description automatically generated

Figure 5: Flowchart summarizing the workflow for the HLS neural network

After the conversion to RTL, the hls4ml framework provides us with an AXI4-protocol compliant IP block for the RTL network which can be imported into Vivado and incorporated into a block diagram. This wrapper is located in the “hls4ml\_project\_16\_8 myproject\_prj\solution1\impl\ip” directory. We used Vivado 2019.1 for this step, and had to additionally alter some library files in the hls4ml Python library to get it to properly create an IP block which would be importable for our Mini-ITX boards. We then create a block diagram using Vivado’s IP Integrator workflow to establish connections between the Zynq hard processor (PS) and the programmable logic (PL), as shown in Figure 6 below.

Diagram

Description automatically generated

Figure 6: Vivado block diagram with the neural network (PL) linked to the Zynq processor (PS) using a DMA interface

As can be seen, the HLS network is an AXI4 peripheral with master as well as slave connections to the PS over the DMA. This means that the PS will be able to both send data to the HLS network, as well as receive data from the HLS network, both via the AXI DMA block. The complete Vivado project is provided in the “z045\_hls\_16\_8\_vivado\_2019” directory.

Finally, we synthesize, implement, and generate bitstream and hardware handoff files directly based on the block design using Vivado. Both of these files together create what is called a hardware overlay for the PYNQ operating system running on our board. In addition, we wrote a Python driver to handle the data sending and receiving from/to the PS and PL, present in “final\_pynq\_overlay/axi\_stream\_driver.py”. This allows us to dynamically load the bitstream for our design in Python code executing on the PS, cleanly send input images to the HLS neural network via a DMA send channel, and finally receive output predictions from the HLS neural network back into the PS via a DMA receive channel.

The folder “final\_pynq\_overlay” consists of the final files required for our HLS network to perform inference on the board. Files “X\_test.npy” and “y\_test.npy” contain the 10,000 testing set images and labels respectively. This folder can be pasted on to the PYNQ environment and can be directly used to perform inference on the board using the complete testing dataset. We successfully verified that we attain the exact same testing accuracy as the C-network on the board using this overlay approach (which runs the network on the FPGA) – 91.11 % using 8 fractional bits with a total of 16 bits fixed point precision.

* 1. **Phase 3: Studying Network Bit-Precision**

We studied network bit-precision in simulations and compared the network’s performance between the HLS network and our RTL network. The total number of bits was fixed at 16, 1 of which is always dedicated for the sign bit. The number of fractional bits was varied and the network accuracy over the full testing dataset was evaluated on both the networks. Section 5.1 includes detailed results and comparisons for this phase.

* 1. **Phase 4: Web Camera Integration**

Related files are provided in the accompanying “Phase 4 – Web Camera Integration” folder.

The objective with the web camera integration was to have the neural network on the FPGA perform classification on images taken in real-time from a web camera. For the implementation, we chose a web server-based approach as a workaround since our board was not correctly recognizing USB connections.

In brief, we coded a simple web server in Python using the Flask framework. Figure 7 shows a snapshot of the server. The server hosts a live camera feed and implements an endpoint “/capture\_image”. Any device connected to the Internet can connect to the server and this endpoint and the server responds with a jpeg image in real-time based on the live camera feed.

Graphical user interface, text, application

Description automatically generated

Figure 7: Snapshot of our web server hosting a live camera feed

The server files are provided in the “server/app.py” directory. The server can be started by using “python app.py” directly, as detailed in the accompanying demonstration video [here](https://www.youtube.com/watch?v=fW6_RJwT5pw). The idea is that the web server would be hosted on a separate system, decoupled from the FPGA board. The FPGA board would require an active ethernet connection, and could request an image from the server when required.

The full inference workflow is implemented in the provided “End-to-End Inference.ipynb” Python notebook file. This notebook is intended to be run on the actual Jupyter Python server running on the PYNQ operating system on the Mini-ITX Boards. In this notebook, the following steps are implemented:

* **Step 1 – Real-time Image Capture:** Here the PS connects to the web server and requests an image; the server responds with an image based on the live camera feed.
* **Step 2 – Image Processing:** The image that is received from the web server has a higher resolution than what the network expects. Moreover, it’s an image in which the MNIST digit is not required to be localized. To make the image compatible and suitable with our neural network, we perform processing on the image (thresholding, binarizing, digit localization, and squaring). Figure 8 below shows the before and after of a live image that we used in our project.

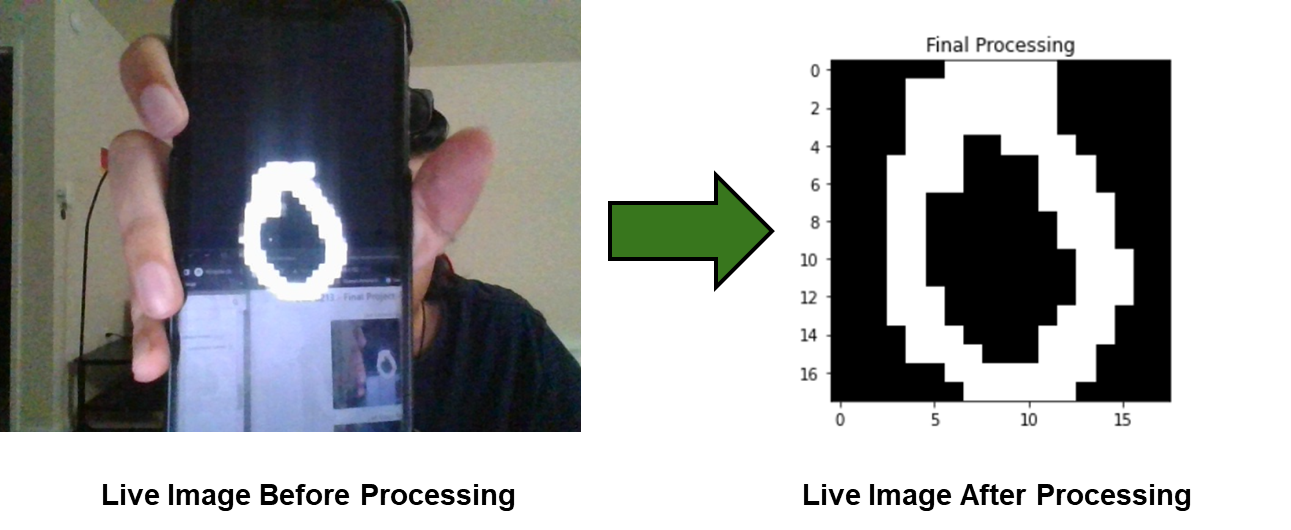


Figure 8: Example of image processing used for successful inference

* **Step 3 – Inference:** Once the image has been processed to match network dimensions (18 x 18 pixels, 16-bit each), it can then be directly sent to the neural network sitting on the PL via a DMA send channel (all 5,184 pixels). The neural network on the PL produces 10 output neurons (16 bits each, total of 160 bits). These output bits are then received back by the PS through the DMA receive channel. This sending and receiving of data is managed through the AXI stream driver as described in Section 4.2. The PS then takes the maximum of the 10 output values to determine the network’s prediction of the image. On the image shown above in Figure 8, our network correctly predicts the output class as being 0.

1. **Results**
   1. **Studying Network Bit-Precision**

Figure 9 captures the change that we saw in the inference accuracy of the network as we varied the total number of fractional bits for both the custom RTL as well as the HLS network.

Chart

Description automatically generated

Figure 9: Trends in network inference accuracy vs. fractional bits in the fixed-point representation

It can be seen that the HLS network can achieve higher accuracy, but has a tighter quantization window compared to the custom RTL network. For benchmarking, software accuracies are also plotted as horizontal lines. For the HLS network, the 32-bit floating point software benchmark was 91.14 %, and the fixed-point peak performance was 91.16 % with 6 fractional bits (1 sign bit, 9 integer bits). For the custom RTL network, the 32-bit floating point software benchmark was 90.56 %, and the fixed-point peak performance was 90.56 % with 8 fractional bits (1 sign bit, 7 integer bits).

We hypothesize that these differences come from a variety of sources, two of which are as follows: Firstly, the trained networks used in both cases were different, and so the mapped weights were different. The HLS network was trained with biases, whereas the RTL network was trained without biases. Secondly, the overflow and underflow handling is different. For the custom RTL, we max or min out weights when they exceed the max or min representable value, but in the HLS case, we used the default strategy set forth by the hls4ml framework.

* 1. **Synthesis and Implementation**

A screenshot of a computer

Description automatically generated with medium confidence

Figure 10: Comparison of synthesized HLS/RTL network designs

Graphical user interface, chart

Description automatically generated

Figure 11: Comparison of implemented HLS/RTL network designs

* 1. **Design and Power Reports**

Chart, line chart

Description automatically generated

Figure 12: Comparison of HLS/RTL network design utilization reports

Graphical user interface, application

Description automatically generated

Figure 13: Comparison of HLS/RTL network power reports

1. **Conclusions & Future Work**

In this work, we explored neural network inference on an FPGA board using high-level synthesis tools as well as custom RTL. We successfully showed that our FPGA neural network can correctly classify images coming from a live-camera feed.

The end-to-end inference results for this project were based on the HLS network which uses the IP core generated by Vivado HLS. We would like to write an AXI4 streaming interface wrapper for our custom RTL network as future work, so as to get our network running on the board and integrated with the PS through the DMA interface. In the long-term, we would like to expand this project by having neural network weights implemented on a custom ASIC daughterboard that will house a 20,000 ReRAM (resistive random access memory) device array connected to the FPGA board by a FMC connector.

1. **Appendix**
   1. **Hardware Setup**

For this project, we boot our Mini-ITX boards from the SD card mode. The SD card has an image of the PYNQ operating system which we built by following steps on the official documentation located [here](https://pynq.readthedocs.io/en/latest/pynq_sd_card.html). This PYNQ image is a combination of a root filesystem with Python pre-installed, Xilinx’s PetaLinux, as well as PYNQ Python module and classes, as highlighted in Figure 14 below. Once the image is created, we simply flashed it onto an SD card using the tool BalenaEtcher.

Once the board boots up, a Jupyter server is running by default. Additionally, a network-based file-system is also configured. Files can be transferred directly by doing an ssh into the PYNQ environment, and python notebook files (for example, the end-to-end inference project notebook described in Section 4.5 can be executed directly on the Jupyter server. The PYNQ operating system allows us the flexibility to program our FPGA board dynamically without rebooting the board.

Diagram

Description automatically generated

Figure 14: Overview of the constituents of a PYNQ image

* 1. **Demonstration**

We recorded an in-depth demonstration of the board bring up and the end-to-end neural network inference. It can be found on YouTube [here](https://www.youtube.com/watch?v=fW6_RJwT5pw): <https://www.youtube.com/watch?v=fW6_RJwT5pw>

* 1. **Miscellaneous Details**

We populated documentation for several key aspects of this project. These resources are provided separately in the accompanying “Extra Documentation” folder. These are:

* **1 – Custom Board PYNQ Instructions:** This document contains instructions for creating custom PYNQ images for our AVNET Mini-ITX Boards (7z045/7z100) based on Zynq-7000.
* **2 – hls4ml HLS NN IP to PYNQ Overlay Workflow: This** document lists steps on using the exported IP block from hls4ml (using Vivado HLS) to a block diagram in Vivado 2019.1 for a custom Xilinx FPGA Zynq board, for deployment using PYNQ.
* **3 – Custom RTL on the Mini ITX Board:** This document lists steps on how to get custom a simple Verilog (RTL) module running on the Mini-ITX Board using a direct JTAG connection.