ECE532 - Group Report

Accelerating Object Classification Using FPGAs

Hammad Mohiudin
Mehdi Mousavi
Ahanaf Rakin
Farid Chalabi

Group 3
April 14, 2021

Table of Contents

1. Overview

1.1 Project Motivation

Machine learning has been gaining traction and as a result, models are getting more complicated. To run these models in optimal time, expensive hardware such as GPUs might be required. GPUs have high power consumption and may not be feasible for some applications.

For this project, the team wanted to run a machine learning model on a FPGA. By doing so, this would use much less power than a GPU, while still having adequate performance. The team believes this type of application could be used to detect and classify objects in vehicles.

1.2 Project Goal

The initial goal of the project was to send a video from the DESL machine and run the SSD-MobileNet model to identify and display different objects on a VGA display. However, due to the complexity of the model, the team has shifted to using a simpler model.

The goal of this project is to run the LeNet5 model trained for classifying traffic signs on a FPGA. The user will be able to select a traffic sign image from the DESL machine and the classification of the sign, along with the chosen image, should be displayed on a VGA display.

1.3 System Block Diagram

The system consists of five main components as shown in Figure 1: DESL Computer, Nexys Video Board, Nexys DDR Board, FPGANet, and VGA Display.

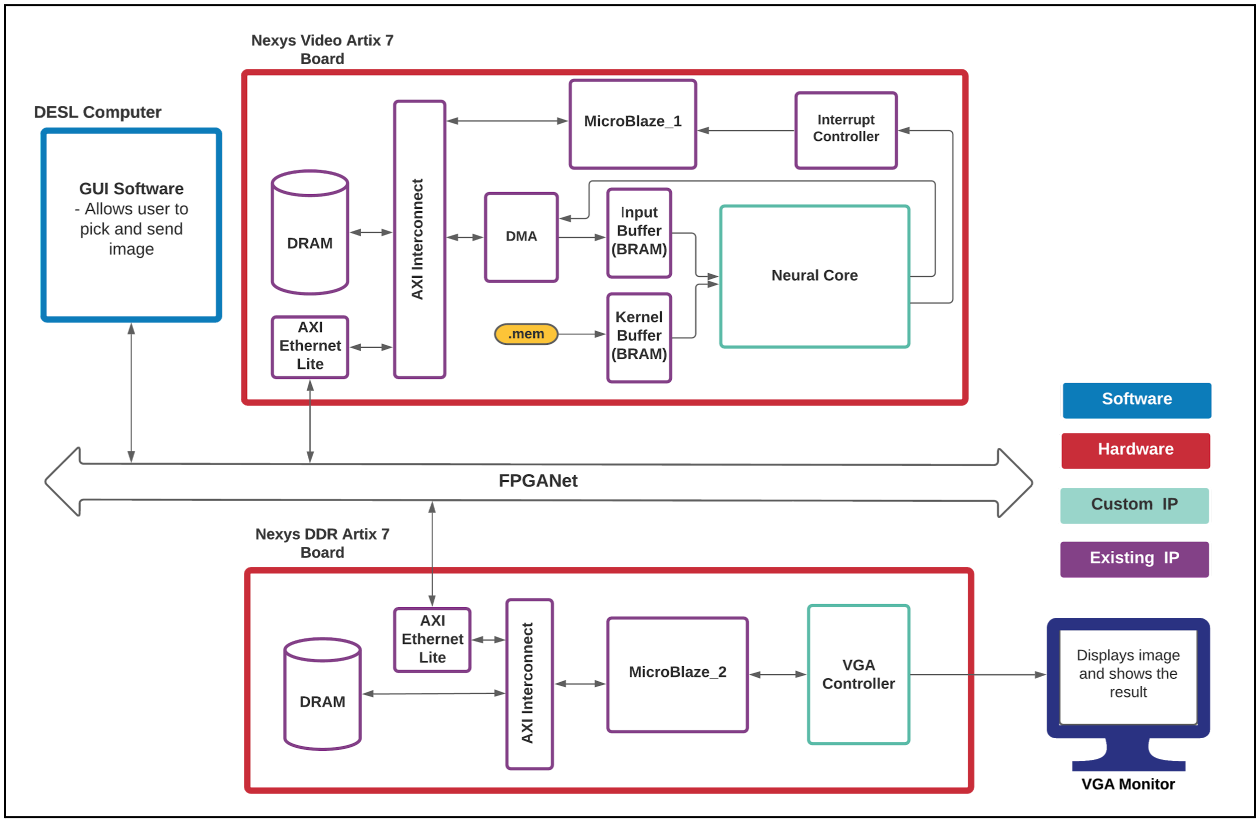


Figure 1: System-level Block Diagram of the Design

DESL Computer

This computer hosts a python server and runs a GUI that allows users to select images, establish network connections with the two FPGAs, and send images to the FPGAs for processing. A 32x32 grayscale image is sent to the Nexys Video board. The DESL computer receives an image classification from the Nexys Video board. The DESL computer sends a 180x180 grayscale image and the classification predicted by the Video board to the Nexys DDR board.

Nexys Video Board

The FPGA on this board is responsible for performing the computations required by the Neural Core subsystem. Once the image is stored in the DDR DRAM, it is passed through the Neural Core IP using the DMA Controller. The classification output from Neural Core is then sent to the DESL computer using the Ethernet subsystem and MicroBlaze_1.

Nexys DDR Board

The FPGA on this board is responsible for preparing the image and classification received from the DESL computer for the Xilinx TFT Controller, which displays both of them on the VGA monitor.

FPGANet Network

The subsystems described above are connected together through TCP connections over the FPGANet network. More specifically, the python server on the DESL computer communicates with the Ethernet subsystem on the Video board, as well as the Ethernet Lite IP on the DDR board through FPGANet.

VGA Display

The TFT Controller IP sitting on the DDR board is responsible for displaying the image and its classification on the VGA monitor.

1.4 Brief Description of IPs

This section lists the Xilinx IPs used along with a description of how they fit in the design.

Nexys Video Board

```
"design_tree": {
  "microblaze_0": "",
  "microblaze_0_local_memory": {
    "dlmb_v10": "",
    "ilmb_v10": "",
    "dlmb_bram_if_cntlr": "",
    "ilmb_bram_if_cntlr": "",
    "lmb_bram": ""
  },
  "microblaze_0_axi_periph": {
    "xbar": "",
    "s00_couplers": {},
    "s01_couplers": {},
    "m00_couplers": {},
    "m01_couplers": {},
    "m02_couplers": {},
    "m03_couplers": {},
    "m04_couplers": {},
    "m05_couplers": {}
  },
  "microblaze_0_axi_intc": "",
  "microblaze_0_xlconcat": "",
  "mdm_1": "",
  "clk_wiz_1": "",
  "rst_clk_wiz_1_100M": "",
  "axi_uartlite_0": "",
  "axi_timer_0": "",
  "axi_smc": "",
  "axi_dma_0": "",
  "axi_ethernet_0": "",
  "axi_ethernet_0_dma": "",
  "mig_7series_0": "",
  "rst_mig_7series_0_100M": "",
  "neural_core_0": ""
}
```

Nexys DDR Board

```
"design_tree": {
  "microblaze_0": "",
  "microblaze_0_local_memory": {
    "dlmb_v10": "",
    "ilmb_v10": "",
    "dlmb_bram_if_cntlr": "",
    "ilmb_bram_if_cntlr": "",
    "lmb_bram": ""
  },
  "microblaze_0_axi_periph": {
    "xbar": "",
    "s00_couplers": {},
    "s01_couplers": {},
    "s02_couplers": {},
    "m00_couplers": {},
    "m01_couplers": {},
    "m02_couplers": {},
    "m03_couplers": {},
    "m04_couplers": {},
    "m05_couplers": {},
  },
  "microblaze_0_axi_intc": "",
  "microblaze_0_xlconcat": "",
  "mdm_1": "",
  "clk_wiz_1": "",
  "rst_clk_wiz_1_100M": "",
  "axi_uartlite_0": "",
  "mig_7series_0": "",
  "rst_mig_7series_0_81M": "",
  "axi_tft_0": "",
  "RGB_SPLIT_0": "",
  "RGB_SPLIT_2": "",
  "RGB_SPLIT_1": "",
  "mii_to_rmii_0": "",
  "axi_ethernetlite_0": "",
  "axi_timer_0": ""
}
```

Figure 2: Design tree of top level Vivado projects on each board (from: <design>.bd)

IP Name(s)	Description
microblaze_0 (*) ***_local_memory** ***_axi_periph** ***_axi_intc** ***_xlconcat, mdm_1**	**MicroBlaze processor** is used in the design to perform the following tasks: receive TCP packets from DESL computer, store packets in DDR DRAM, Setup interrupts (video board), Send classification to DESL (video board), Draw image and text (ddr board)
clk_wiz_1 (*) **rst_*_100M**	**Clock generator:** divides the system clock into different frequencies to be used by different components: 25/200/100 MHz → TFT/DRAM/everything else **Reset Generator:** synchronizes the various reset signals with the appropriate clocks. It also produces reset signals with correct polarity (active-low/high)
axi_uartlite_0	The **AXI UartLite** IP was used by the MicroBlaze processors to print messages on the SDK serial terminal.
axi_timer_0	The **AXI Timer** is used by the Ethernet subsystem and Ethernet Lite IPs to handle TCP communications.
axi_smc	The **AXI SmartConnect** IP is used to connect AXI master and slave IPs.
axi_ethernet_0 (*) ***_dma**	The **AXI Ethernet Subsystem** is used on the Video board to handle TCP communications. This **AXI DMA Controller** IP is used to interface the Ethernet subsystem with the DDR3 DRAM.
mig_7series_0 (*) **rst_*_100/81M**	The **Memory Interface Generator (MIG)** IP is used to handle the communication with the DDR DRAMs.
neural_core_0	**Neural Core** is a custom hardware IP core that performs the object classification. It sends the result of the classification to the MicroBlaze via an interrupt.
axi_dma_0	This **AXI DMA Controller** IP is used to interface the Neural Core with the DDR3 DRAM.
axi_tft_0 **RGB_SPLIT_0/1/2**	**AXI TFT Controller** IP is used to handle the communication between memory and VGA monitor. The **Slice** (RGB_SPLIT_*) IPs are used to truncate the 2 least significant bits of each of the pixel colour output ports because the board only supports up to 12 bit RGB (4 bits for each R/G/B)
axi_ethernetlite_0 **mii_to_rmii_0**	The **AXI Ethernet Lite** IP is used on the DDR board to handle TCP communications.

Table 1: IP blocks used in the design

2. Outcomes

2.1 Results

Due to a pivot in our original project idea, the functional requirements had to be updated midway into the project. The updated functional requirements better reflect the specifications that the final design had to meet. We were successful in completing all updated functional requirements with a few exceptions as can be seen from Table 2. Acceptance criteria and final timing can also be seen from Tables 3 and 4 respectively.

Functional Requirements			
#	**Original**	**Updated**	**Status**
1	Implement MobileNet + SSD architecture on FPGA for object detection.	Implement the LeNet-5 architecture on FPGA for image classification.	Complete
2	Train a software model for MobileNet + SSD and extract trained weights.	Train a software model for LeNet5 and extract trained weights.	Complete
3	Implement a custom Neural Core IP to perform pointwise, depthwise and standard convolutions on an input video frame using software trained weights.	Implement a custom Neural Core IP to perform standard convolutions on a grayscale image using software trained weights.	Complete
4	Implement a TCP server/client to send input video stream frame by frame from the DESL machine to the FPGA.	Same as original but now sending grayscale images back to back.	Partially Complete (Requires restarting TCP client on the video board for successive runs)
5	Implement direct communication between FPGA 1 and FPGA 2.	No change.	Incomplete
6	Display video stream with detected object label and bounding box on a VGA monitor.	Display image with class label on a VGA monitor.	Complete
7	GUI to upload user selected video feed to FPGAs.	GUI to upload user selected images to send to FPGAs.	Complete

Table 2: Functional requirements of project

Acceptance Criteria			
#	**Original**	**Updated**	**Status**
1	System should recognize objects in a video feed correctly.	System should correctly classify images given random image samples of traffic signs.	Complete
2	System should draw bounding boxes around detected objects correctly on VGA.	System should display images with a prediction label on the VGA.	Complete
3	System should display the video with bounding boxes at a minimum of 5 FPS.	System should allow back to back images to be sent, processed and displayed.	Incomplete

Table 3: Acceptance criteria for project

Final Timing	
Operating Clock Frequency	100 MHz
Max Clock Frequency (Fmax)	102 MHz
Neural Core Latency	80 us

Table 4: Final timing of hardware

Our final software model achieved a validation accuracy of 93% (Appendix A), however we were unable to get an accuracy measurement from our final hardware implementation as there was no way to pass a large number of images to the system back to back. This was because the client had to be restarted for successive runs, and it would take too long to manually run the model on a large enough number of samples to get the true accuracy. However, we were able to run our system on a few random images of traffic signs and get the correct predictions to be displayed on the VGA along with the original image. Figures 4-6 show the system output for the given input images.

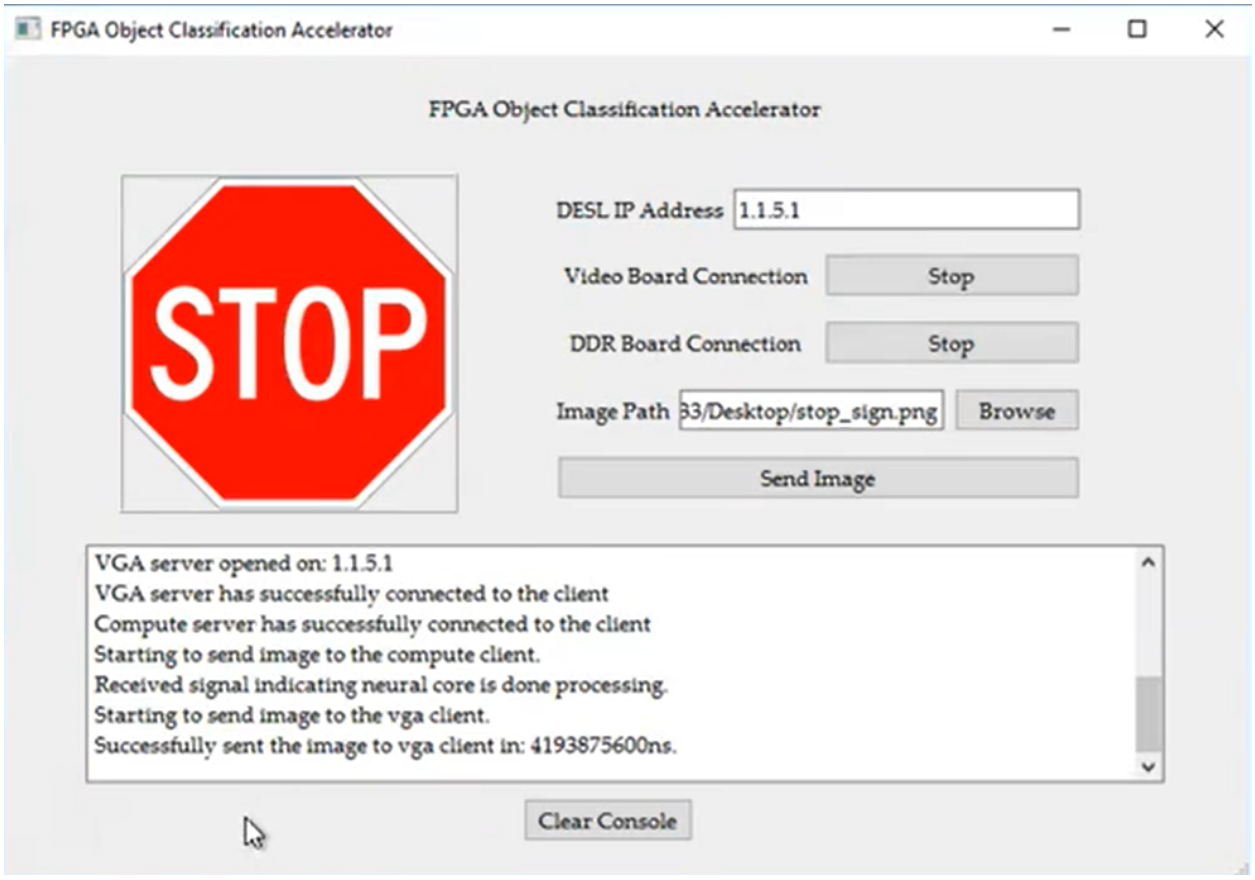
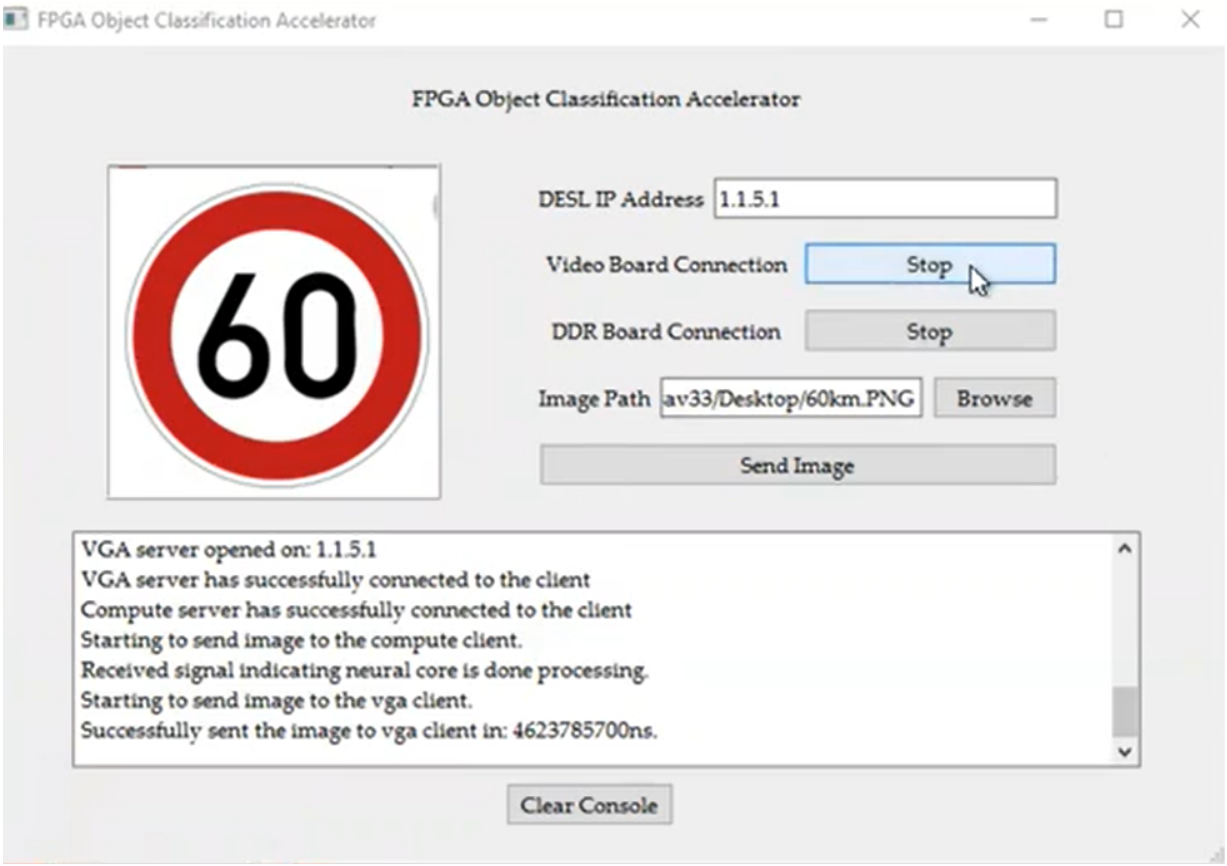


Figure 4: Correct display and classification of a stop sign image



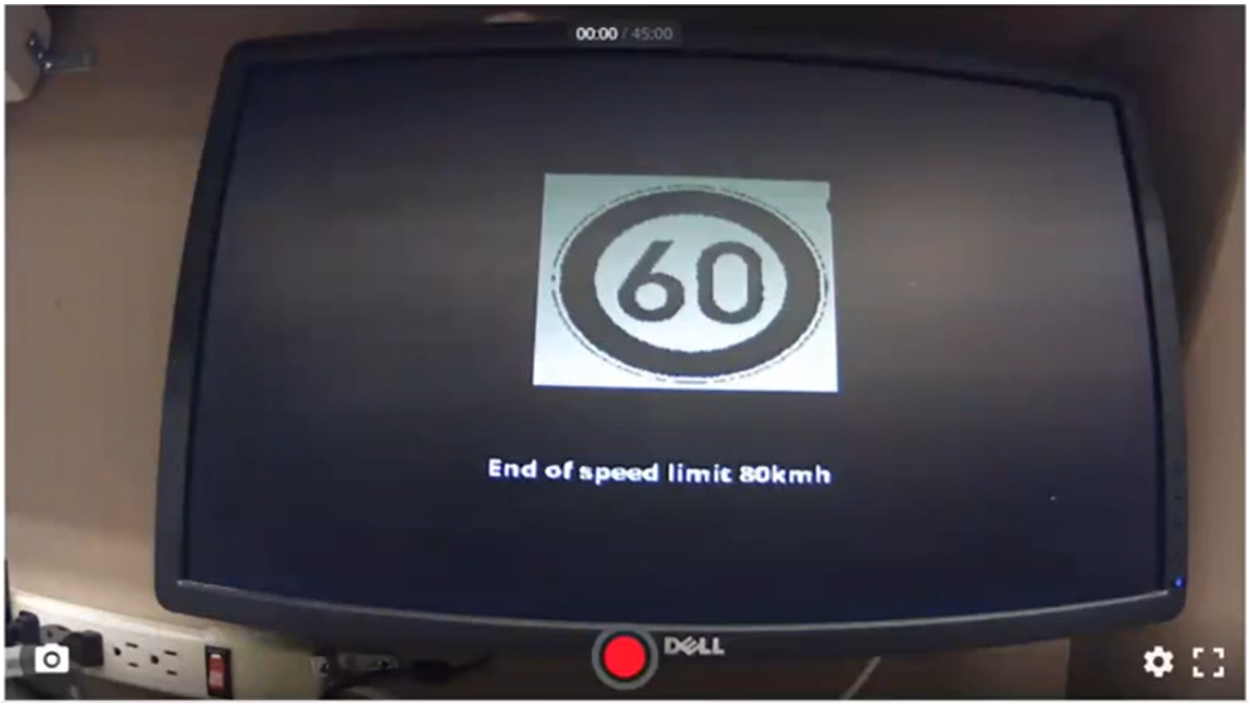


Figure 5: Incorrect classification of the 60km/h sign as 80km/h, but displayed correctly

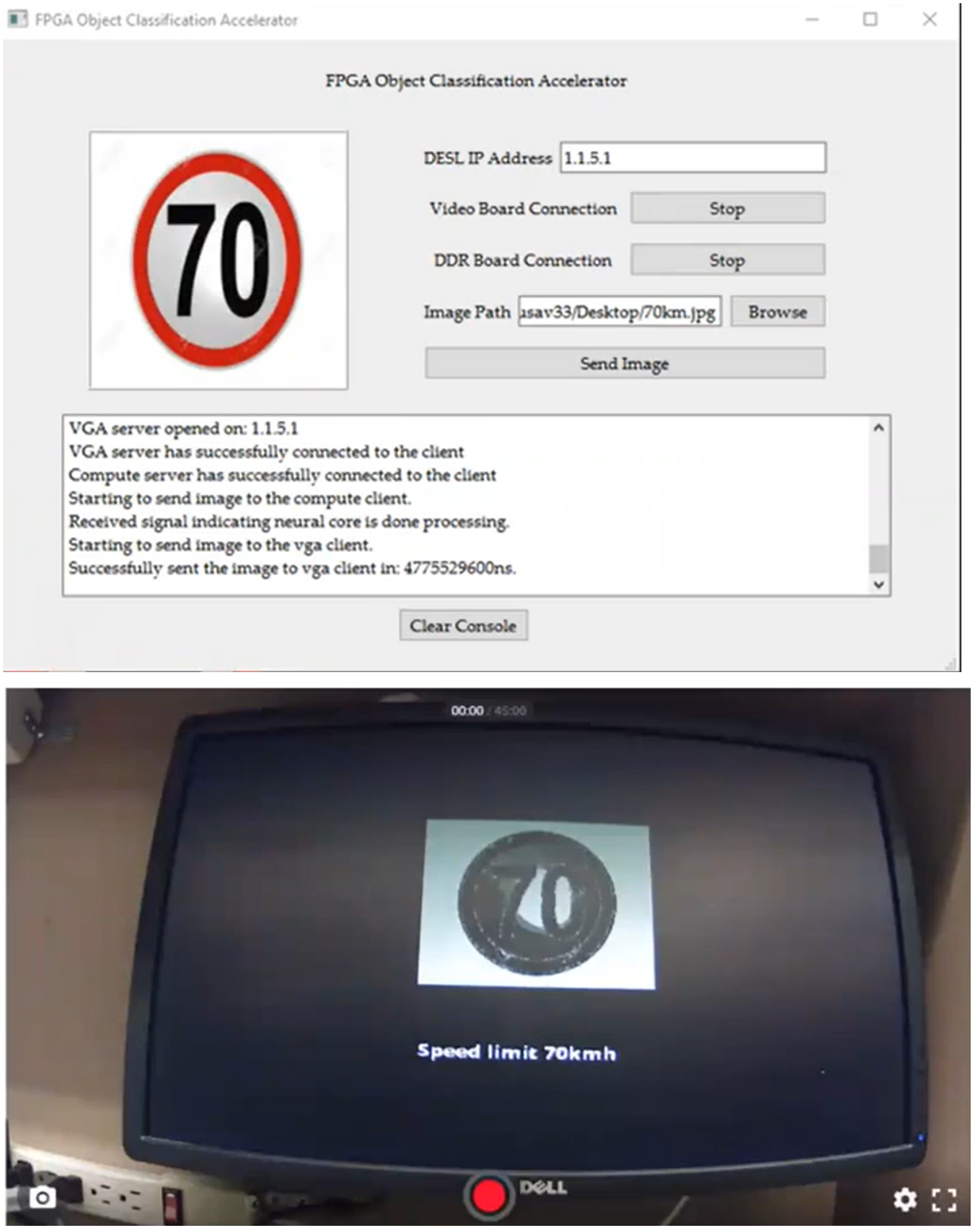


Figure 6: Correct classification and display of the 70 km/h sign

Resource Utilization

The final resource utilization for our system can be seen in Figure 3 below. Our initial implementation of the neural core ip exceeded the resources available on the Nexys Video board. However after optimizing the system by pipelining some convolution blocks, we were able to achieve much lower resource utilization. The Nexys DDR board was only used to output the images to the VGA and hence had low resource utilization.

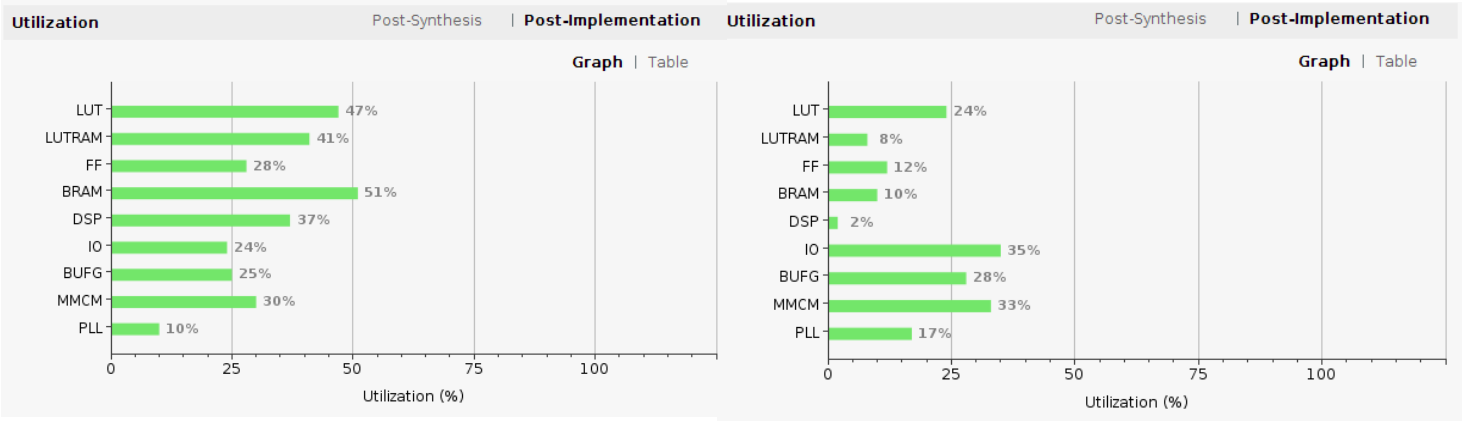


Figure 3: Resource utilization for Nexys Video board (left) and Nexys DDR board (right)

2.2 Future Improvements

A future improvement to this project would include enabling direct communication between both boards to allow the final prediction to be directly sent to the displaying board independent of the DESL machine. This would require figuring out how to maintain multiple TCP connections from one machine to another and may require multithreading.

Another potential improvement would be fixing the bug that prevents sending multiple images from GUI to the FPGAs without rerunning the client application on the video board. The most likely source of this bug is the interrupt handler code in the client software.

Also, now that we know how to pipeline our design to reduce our resource utilization, we could also implement a larger image classification network to improve accuracy. This would require training a new software model and extracting its weights. The only hardware component that would need to change would be the neural core to add the necessary layers and update the dimensions. As all the source files are parameterized, changing the dimensions of the layers will be a simple task.

3. Project Milestones

#	Planned	Actual
1	Software Model of project • Run full project in software using PyTorch on our own computer	**Completed.** The team was able to run a PyTorch model for SSD-MobileNet.
2	MicroBlaze system with Memory • Setup microblaze system with memory and run model on CPU • Run the model using the CPU	**Completed with changes.** The team realized that running the model on the Microblaze CPU was not feasible. The team was able to send an image and store it in the DDR. However, there was an issue with large packets not sending properly. This issue was fixed by milestone 4.
3, 4	Neural Core Implementation • Send an image file file to the Neural Core • Display the image on VGA • Put the weights in memory, organize them so Neural Core can read them • Retrieve weights from memory	**Completed with changes and delays.** Since the model changed, the team had to train the LeNet-5 model. This added some delay to finishing. The weights were also no longer received from memory but loaded in from a mem file. Displaying an image on a VGA was completed.
5	Communication between FPGAs • Send data using the network to other FPGA • Send an image to the DDR board from the video board	**Completed with changes.** Data is not sent between the FPGAs but rather sent via the DESL machine. The DESL machine also sends the image to the DDR board instead of it being sent from the video board. The GUI for python was also completed at this time.
6	Full system working (hardware + software) • Get video to work • Additional features	**Changed.** Video was no longer required after changing the model. This milestone was focused on integrating the system.

Table 4: Planned tasks vs actual completed tasks for each milestone

4. Description of the Blocks

This section describes the technical details of all major hardware and software blocks. This includes Xilinx IPs, custom hardware blocks, and software systems used/built to interact with the hardware systems.

4.1 Software Blocks

TCP Client

The TCP client code runs on the Microblaze for both the Nexys Video and DDR boards. This section of code is responsible for handling TCP connection with the DESL. It is also used to receive images or send information back to the DESL. When image data is received, it gets saved in DDR memory. On both machines, there is a counter for the number of bytes received and this counter gets reset after all functionality is completed, allowing the user to send another image.

Video board:

It is expected that 2048 (32*32*2) bytes are received for the entire image. Once the entire image is received, we set an interrupt, which starts running the neural core. When the computation is completed, indicated by an interrupt from the neural core, a TCP message is sent to the DESL machine.

DDR board:

It is expected 64800 (180*180*2) bytes are received for the entire image. Once the image is received, a message is sent to the DESL, and the DESL machine sends the classification. After the classification is received, the DDR board starts to draw the image along with the classification on the VGA.

DESL Python Code (GUI + TCP Server)

The DESL machine runs a python program that allows the user to the FPGA boards using ethernet. A GUI has been created which can be found in Figure 4. The GUI with the following features:

- Allows the user to set IP for the DESL machine
- Starts a thread to listen for connections from either the video or DDR board
- Allows user to pick an image
- Console to print out messages depending on TCP packets sent/received
- Sends the image to the video board

The python code sets up connections with the video and DDR boards. The user picks an image to be sent. When the “Send Image” button is pressed, this image is scaled to 32x32, turned into grayscale and then gets sent to the video board via TCP. The DESL then waits until the computation is complete in the video board. Once the computation is complete, the DESL receives the classification. The DESL then transforms the selected image into 180x180 grayscale and sends it to the DDR board. When the image is successfully sent, the classification is sent to the DDR board.

VGA

The VGA code runs on the Microblaze for the DDR board. It is responsible for drawing the received image along with the classification for that image. Images are stored in DDR memory. To draw the image, the TFT controller is pointed to the address in DDR where the image is stored. This address is then used by the TFT controller to read from and write to the VGA monitor. For the classification text, there are image arrays saved on the stack. Based on the classification, the correct image is loaded and drawn by writing to the TFT addresses, similar to the image.

ML Model Training & Weight Extraction

The software code for the machine learning model was written in Python using the PyTorch ML framework. A custom model was developed based on the LeNet-5 architecture as shown in Figure 3. The dataset used for training was obtained from the GitHub repo of a Udacity course which uses the German Traffic Sign Recognition Benchmark (GTSRB) dataset. After training, PyTorch Quantization was used to quantize the final weights to UINT8 and a custom script was developed to store the quantized weights to a .mem file. These mem files were used by the hardware to initialize the memory blocks storing the weights.

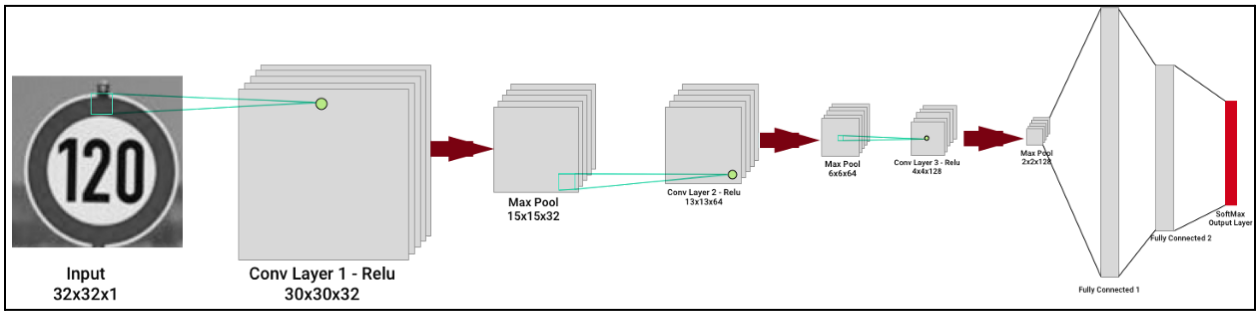


Figure 7: Modified LeNet-5 Architecture [1]

4.2 Hardware Blocks

Neural Core

Neural Core is a custom hardware IP that implements a variation of the LeNet-5 convolutional neural network (CNN) designed for classifying traffic signs with a high accuracy. This CNN consists of two major components: Convolution Layer and Fully Connected Layer. To interface with the AXI DMA controller, the Neural Core is implemented as an AXI Stream IP. Figure 8 shows the final block diagram of this core as it appears in a Vivado block design.

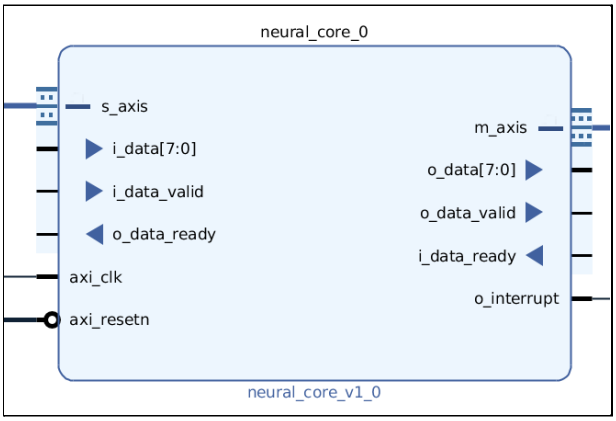


Figure 8: Vivado block diagram of Neural Core

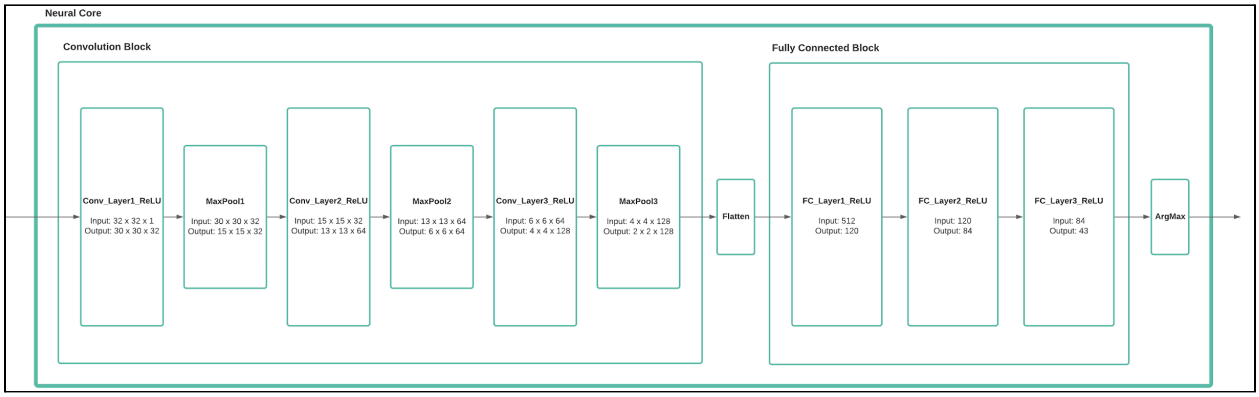


Figure 9: Neural Core Block Diagram

Convolution Block

Convolution Block consists of 3 instances of Convolution Layers and 3 instances of Max Pool Layers (Figure 9). What makes each instance different are parameters such as the Channel, Stride and Kernel size, Input and Output dimensions. These values are passed to these instances during the generation.

Convolution Layer

Convolution Layer consists of N instances of Convolution Layer Channel that will be outputted. Layer 1 has 32 channels, while Layer 2 and 3 have 64 and 128 channels respectively.

Each Channel in the layer is a pipeline of Buffer Block that was specifically designed for generating 3x3 grid data from stream, Convolution Unit block that will perform Convolution on that 3x3 grid data and ReLU activation Block (Figure 10).

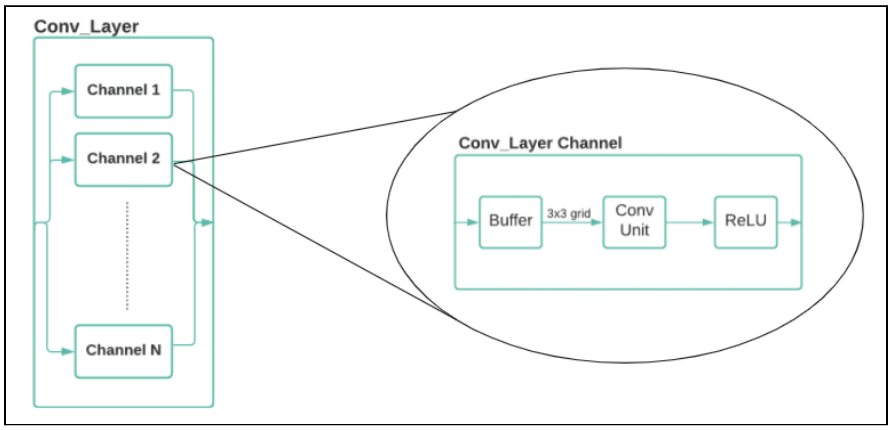


Figure 10: Convolutional Layer Block Diagram

Buffer Block

Buffer Block is a design used to convert a stream of data to a data type that is in a form of a grid. This block has been parameterized in terms of the internal row size, kernel size, and Stride. As this block was used both in Convolutional Layer Channel and Max Pool Channel blocks, each generated instance was configured to meet the specifications of each block. To be more specific, the instance of Buffer Block in Convolution Layer Channel was configured to have Kernel Size of 3x3, and Stride of 1, while for Max Pool Channel there were set to 2x2 with Stride of 2.

Figure 11 describes how this block was implemented for Kernel size of 3x3 and Stride of 1. The block consists of 4 rows (Row0, Row1, Row2, Row3) where Row0 is a shift register. As the stream data is fed to the block, Row0 starts to fill up (Figure 11.a). As soon as Row0 is full, all the data is shifted to Row1 (Figure 11.b). This process continues till all of Row1, Row2 and Row3 hold valid data. As soon as there are 3 valid rows to output, the block starts to output 9 data points shown in the 3x3 grid in Figure 11.c. The next cycle, the output window is shifted by 1 element and the next 3x3 grid of values are outputted (Figure 11.d).

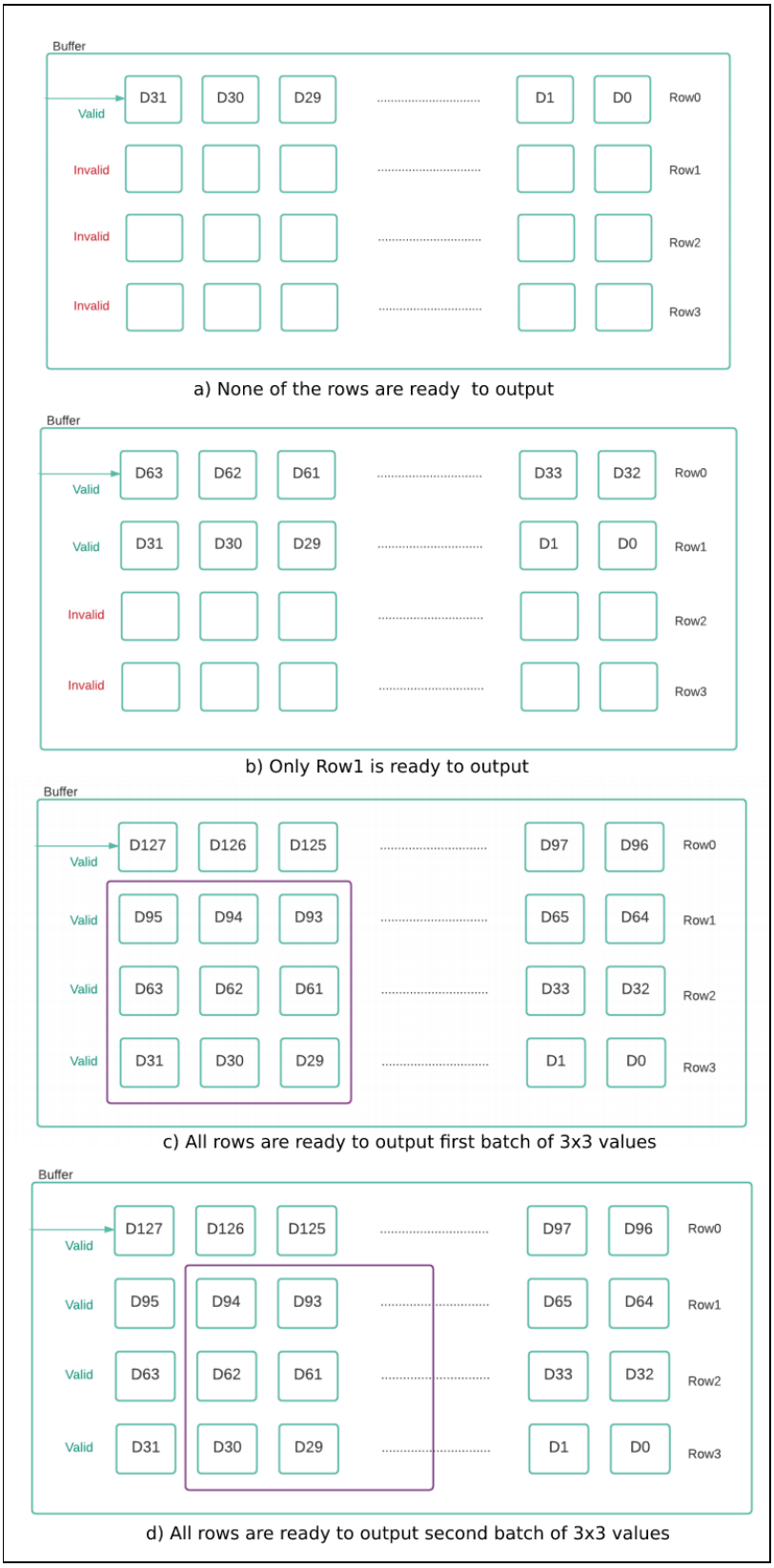


Figure 11: Possible states of Buffer Block

Convolution Unit

Convolution Unit Block performs 9 multiplication between 9 pairs of input data and kernel weights in the first clock cycle. In the following clock cycles, these results were summed up in a pipelined manner as described in Figure 12.

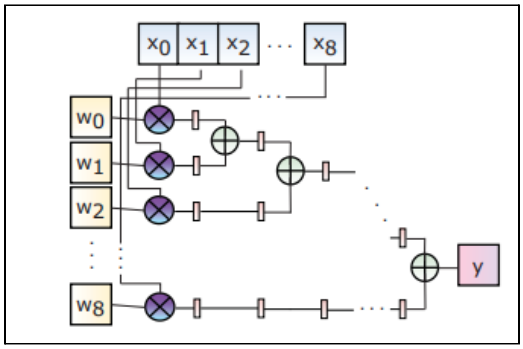


Figure 12: Pipelined design of Convolution Unit [2]

Max Pool Layer

Max Pool Layer consists of N instances of MaxPool Channel that will be outputted. Layer 1 has 32 channels, while Layer 2 and 3 have 64 and 128 channels respectively.

Each Channel of Max Pool is a pipeline of Buffer Block that was specifically designed for generating 2x2 grid data from a stream, and MaxPool Unit (Figure 13).

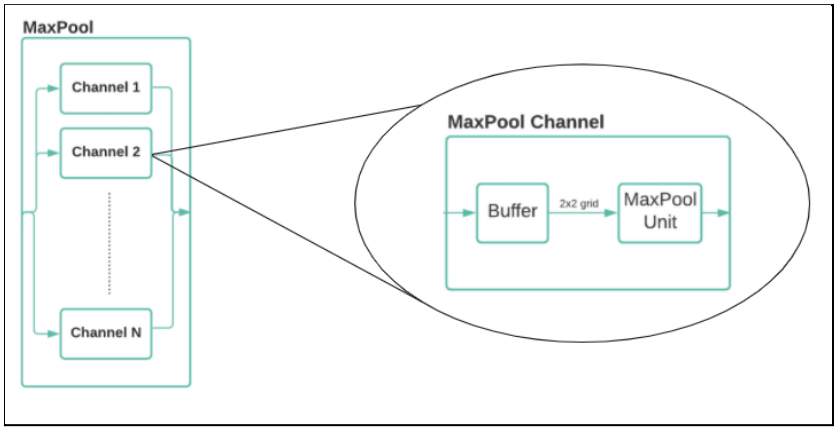


Figure 13: Max Pool Block Diagram

Max Pool Unit

This block compares 4 input values and outputs the largest one as described in Figure 14.

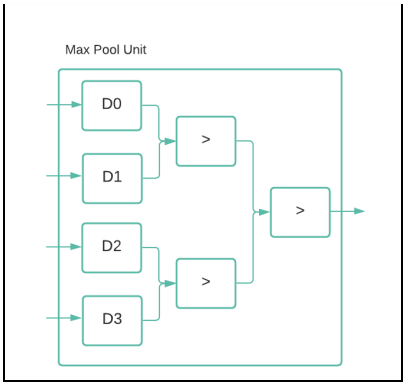


Figure 14: Max Pool Unit Block Diagram

Fully Connected Block

The fully connected block in this CNN consists of 2 hidden layers and output layer with softmax activation function. In the top level, a simple state machine is used to shift the output of the convolution block to each of the layers appropriately. The multiply-accumulate process in the layers are pipelined, meaning all the neurons in a layer do not receive all inputs at the same time. The inputs are shifted into each neuron at every cycle, which is then multiplied by the corresponding weights and then accumulated.

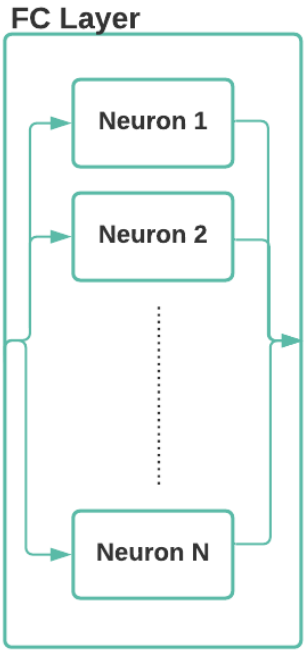


Figure 15: Architecture of a fully connected layer

Layer & Neuron Architectures

Hidden layer 1 consists of 120, layer 2 consists of 84 and the output layer consists of 43 neurons.

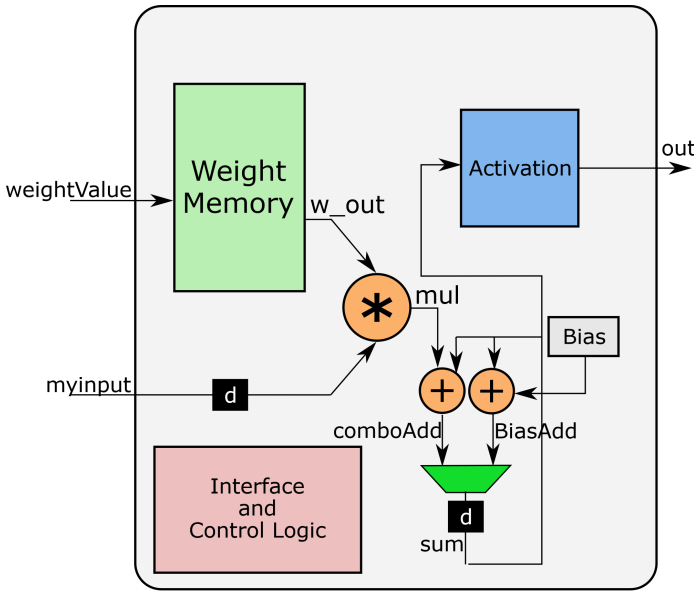


Figure 16: Architecture of a single neuron

A neuron is a pipelined module as shown in Figure 16. Each neuron contains a local (ROM) memory that stores all the weights required. At every clock cycle, a pixel value is fed into a neuron and is multiplied by its weight with a 1-cycle delay. This delay is to accommodate for the fact that all neurons do not multiply all their inputs with all their weights in one cycle. After the multiplication, that result is accumulated with the previous multiplications and this process is continued until all the inputs are received and processed by a given neuron. After that, the bias is added to the accumulated value and the result is passed to an activation module. Layers 1 and 2 use the ReLU activation function whereas the output layer uses the softmax activation function.

Xilinx AXI Interrupt Controller

The Interrupt controller is used in the design to capture and serve various interrupt requests from neural core, DMA controllers, uart, timer, and ethernet subsystem.

Xilinx AXI Direct Memory Access (AXI DMA)

The design uses two DMA controllers to eliminate the overhead of a memory-mapped design architecture.

The first DMA controller is used by the AXI Ethernet subsystem to handle TCP communications required by the Video board. Due to an option in the Ethernet subsystem, the Vivado automation tool instantiated this DMA to be used by the Ethernet subsystem.

The second DMA controller is used by the Neural Core IP. It enables the Neural Core to be implemented as an AXI Stream interface and directly communicate with the external DDR memory. Using a DMA to transfer the images to the Neural Core allowed for a much faster interface than one that could be achieved by involving the MicroBlaze or an AXI Lite or Burst interface. This is because the DMA and an AXI streaming interface is capable of transferring one image word every clock cycle, provided there is no back-pressure from the DMA controller.

Xilinx AXI 1G/2.5G Ethernet Subsystem + AXI Ethernet Lite

To establish network communications between different components of the design, Xilinx's Ethernet subsystem and Ethernet lite IPs were used. These two IPs provided the necessary hardware platform needed by the Lightweight IP (LWIP) TCP/IP stack.

On the Video board, a LWIP client was implemented using a MicroBlaze processor. This client has the ability to communicate with the python server on the DESL machine through the Ethernet subsystem hardware.

On the DDR board, a similar LWIP client was implemented using a MicroBlaze processor and it also has the ability to communicate with the same python server.

For more detail on the LWIP clients, refer to section 4.1, subsection "TCP Client".

Xilinx AXI Thin Film Transistor (TFT) Controller

The AXI TFT controller is used in the design to handle the communication between the external DRAM memory and the VGA monitor. The MicroBlaze on the DDR board is used to run the software that controls the TFT IP. The software stores the image and the classification text in DRAM. It then initializes the TFT controller's read buffer to be the address in the DRAM where the data was stored. The TFT controller then reads from this memory location and displays it on the VGA monitor.

5. Description of Your Design Tree

Figure 17 shows the directory structure of the final design. Since the directory structure generated by Vivado can be complex, the design tree was slightly modified to make it easier to read and navigate. This design tree does not contain all of the files uploaded to GitHub. Please refer to src/ directory in the GitHub repo for a list of all design files.

```
G3_FPGA_Object_Classification_Accelerator
├── README.txt
├── docs                                    # Documentation
|   ├── final_report.pdf                    # Final group report
|   └── final_presentation.pptx             # Final presentation
├── ml_training
|   └── pytorch_edlenet.ipynb               # PyTorch training code for Lenet-5 model
├── server                                  # Python server and GUI that interacts with both FPGAs
|   ├── __init__.py
|   ├── fpga.py                            # Main server file
|   ├── fpga_gui.ui                        # Python GUI
|   ├── sign_label_images.py
|   └── signnames.csv
└── src                                     # Hardware and Software components of the design
    ├── lenet5                              # Video board component containing the Neural Core
    |   ├── lenet
    |   |   ├── lenet.srcs
    |   |   |   ├── lenet.bd                # Top-level Vivado block diagram
    |   |   |   └── lenet_wrapper.v         # Top-level Verilog wrapper generated from the block diagram
    |   |   └── sw                          # SDK software component
    |   |       ├── lscript.ld              # Linker script
    |   |       └── main.c                  # Main code containing LWIP client, DMA controller, interrupt handler
    |   ├── neural_core                     # Custom Neural Core IP
    |   |   ├── component.xml               # Custom IP-XACT file generated by Vivado IP integrator
    |   |   └── neural_core.srcs            # Most important Verilog source files (not everything)
    |   |       ├── cnn_core.v              # Top-level module for Convolutional Layer
    |   |       ├── fully_connected_network.sv  # Top-level module for Fully Connected Layer
    |   |       ├── neural_core.v           # Top-level module of the Neural Core subsystem
    |   |       └── weight_bias             # Memory files containing weights and biases
    |   └── scripts                         # Useful automation scripts
    |       ├── install_opencv.bat          # Install OpenCV, pyqt5, PyTorch, Pandas
    |       ├── sdk.bat                     # Open vivado in batch mode and execute sdk.tcl
    |       └── sdk.tcl                     # Launch SDK with correct hardware definition file
    └── vga                                 # DDR board component containing VGA controller code
        ├── sw                              # SDK software component
        |   ├── classes.h                   # Contains arrays of classification labels, used to draw on VGA
        |   ├── lscript.ld                  # Linker script
        |   └── main.c                      # Main code containing LWIP client, VGA controller, image drawing logic
        └── tft_vga_ctrl.srcs
            ├── constrs_1
            |   └── constrs.xdc             # constraint file for RGB pins of TFT controller and reset signals
            └── sources_1
                ├── tft_vga_ctrl.bd         # Top-level Vivado block diagram
                └── tft_vga_ctrl_wrapper.v  # Top-level Verilog wrapper generated from the block diagram
```

Figure 17: Design tree structure

6. Tips and Tricks

The use of automated scripts greatly accelerated our work process and made it simple to start working on the DESL machines as files were deleted when we logged out. Examples of some scripts we used include one for installing python dependencies and one for opening the SDK without launching the Vivado GUI.

Vivado has the capability of handling multiple simulation sets in one project. We used this feature to easily simulate submodules of our design from the same Vivado project. This improved our efficiency since there was no need to manage multiple projects in order to simulate different components. Figure 19 in Appendix B shows this feature. To run simulation for a particular submodule, we can right-click on that simulation set and click "Make Active".

Naming convention is also important while working in a team. For example, naming signals based on the direction they're going (input/output) and which block they come out from or go into is beneficial. This allows other teammates to easily understand what the signals are meant for.

Planning ahead by creating block diagrams with simple interfaces was also very beneficial for our team. By having an overview of how interfaces should interact with each other, everyone can be on the same page on how they should implement their features. It also saves a lot of headache at the end of the project as integration becomes simpler.

The second tip from section 6 of this GitHub repo is also very useful (report location: G1_FPGA_Surveillance_System/docs/G1_532_final_report.pdf).

7. Video

A demo of our project can be found through this link.

8. References

[1] E. Forson, “Recognising Traffic Signs With 98% Accuracy Using Deep Learning,” *Medium*, 28-Aug-2017. [Online]. Available: https://towardsdatascience.com/recognizing-traffic-signs-with-over-98-accuracy-using-deep-learning-86737aedc2ab.

[2] A. Ahmad, M. A. Pasha and G. J. Raza, "Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design," 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Seville, Spain, 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180843. https://ieeexplore.ieee.org/document/9180843

9. Appendices

Appendix A - Software ML Model Training

The final software model of LeNet-5 that we used achieved 93% validation accuracy.

```
17:58:14 --- Epoch: 0   Train loss: 2.6582      Valid loss: 1.5990      Train accuracy: 64.76   Valid accuracy: 59.14
17:58:16 --- Epoch: 1   Train loss: 0.8357      Valid loss: 0.7876      Train accuracy: 88.97   Valid accuracy: 80.34
17:58:17 --- Epoch: 2   Train loss: 0.3525      Valid loss: 0.5311      Train accuracy: 94.87   Valid accuracy: 86.85
17:58:19 --- Epoch: 3   Train loss: 0.2029      Valid loss: 0.5299      Train accuracy: 96.89   Valid accuracy: 88.53
17:58:21 --- Epoch: 4   Train loss: 0.1294      Valid loss: 0.4090      Train accuracy: 98.00   Valid accuracy: 90.86
17:58:22 --- Epoch: 5   Train loss: 0.0997      Valid loss: 0.5156      Train accuracy: 98.42   Valid accuracy: 90.88
17:58:24 --- Epoch: 6   Train loss: 0.0744      Valid loss: 0.4177      Train accuracy: 98.82   Valid accuracy: 91.75
17:58:25 --- Epoch: 7   Train loss: 0.0538      Valid loss: 0.4445      Train accuracy: 99.28   Valid accuracy: 91.86
17:58:27 --- Epoch: 8   Train loss: 0.0495      Valid loss: 0.3965      Train accuracy: 99.33   Valid accuracy: 92.81
17:58:28 --- Epoch: 9   Train loss: 0.0468      Valid loss: 0.3839      Train accuracy: 99.50   Valid accuracy: 93.02
17:58:30 --- Epoch: 10  Train loss: 0.0329      Valid loss: 0.5028      Train accuracy: 99.66   Valid accuracy: 92.90
17:58:32 --- Epoch: 11  Train loss: 0.0272      Valid loss: 0.4168      Train accuracy: 99.65   Valid accuracy: 94.08
17:58:33 --- Epoch: 12  Train loss: 0.0237      Valid loss: 0.4857      Train accuracy: 99.78   Valid accuracy: 93.24
17:58:35 --- Epoch: 13  Train loss: 0.0140      Valid loss: 0.4114      Train accuracy: 99.83   Valid accuracy: 94.24
17:58:36 --- Epoch: 14  Train loss: 0.0215      Valid loss: 0.4349      Train accuracy: 99.61   Valid accuracy: 93.47
```

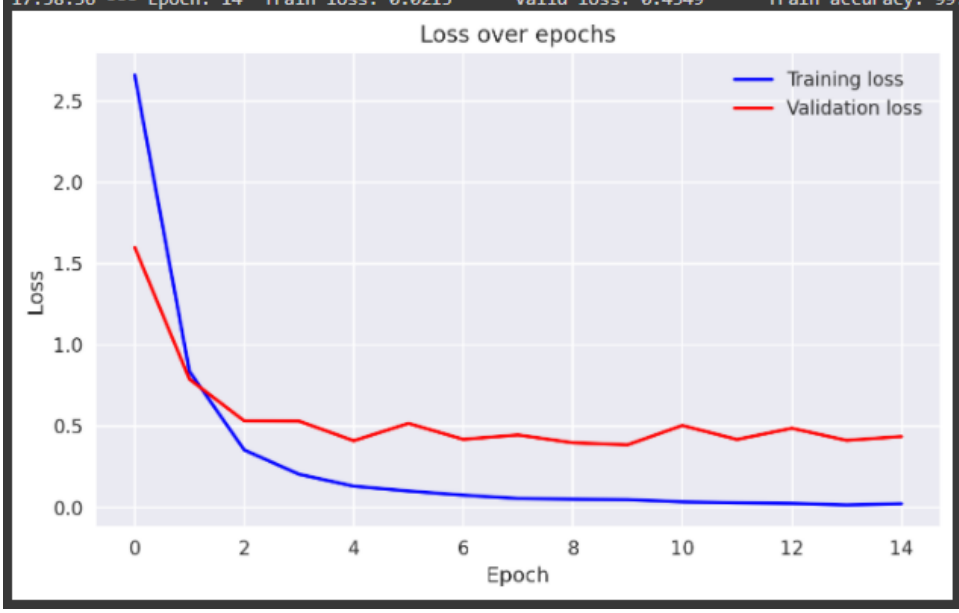


Figure 18: Snippet of PyTorch training progress for LeNet-5 model.

Appendix B - Testbench Structure

The testbench structure used to simulate submodules in our design is shown below. This structure takes advantage of the feature in Vivado that multiple simulation sets can be present in the same project.

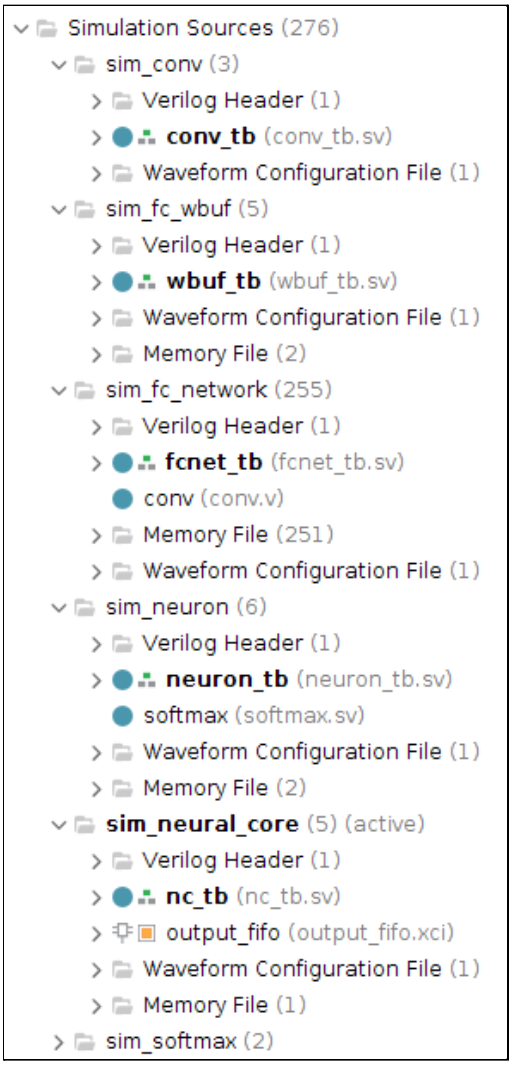


Figure 19: Testbench structure for the Neural Core