ECE1373 Project Report - Classic CNN for Handwritten Digits

Richard Lin, Jianxiong Xu, Yifeng Zhang, Genwen Zhao

May 2nd, 2019

Github: https://github.com/richardlin23/ECE1373_CNN_Digits
Demo Video with Camera Input: https://youtu.be/ZbqtS1RtxsI
Demo Video with Computer Input: https://youtu.be/WsqfhlvcGVA

1. Introduction

In today's news, machine learning and artificial intelligence are inescapable, catchy topics; everything from revolutionizing healthcare [1], to self-driving cars [2], and "killer robots" [3]. But how much does the average person actually know about artificial intelligence, neural networks, and machine learning?

The motivation for this project is to create a system that can illustrate a classic convolutional neural network to distinguish handwritten digits. This project sets the foundation for applications such as license plate recognition and digitizing text. This report aims to provide an understanding of machine learning and digital system design. This project implements a hardware design on the Artix-7 XC7A200T FPGA of the Nexys Video board, complete with a Microblaze soft processor. This microprocessor core executes embedded C software to capture a handwritten number from an HDMI camera, and display the detected number on an HDMI computer screen. The below diagrams provide an overview of the hardware and software architecture.

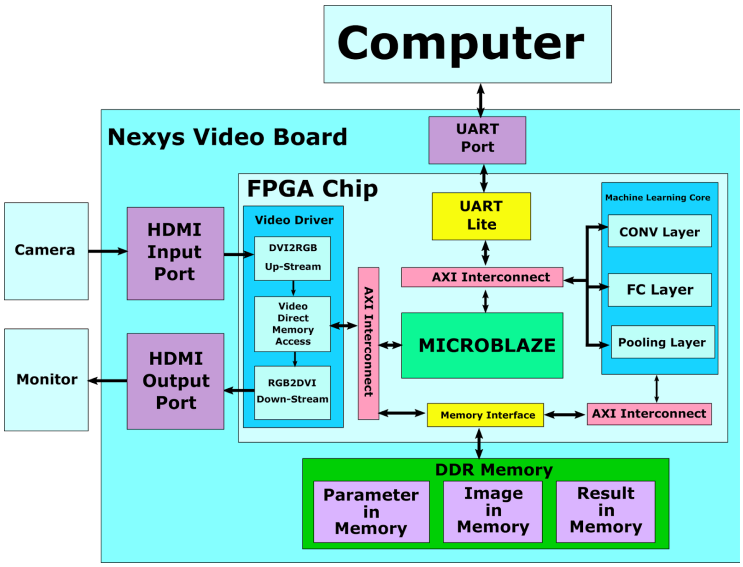


Figure 1: Final System Level Block Diagram

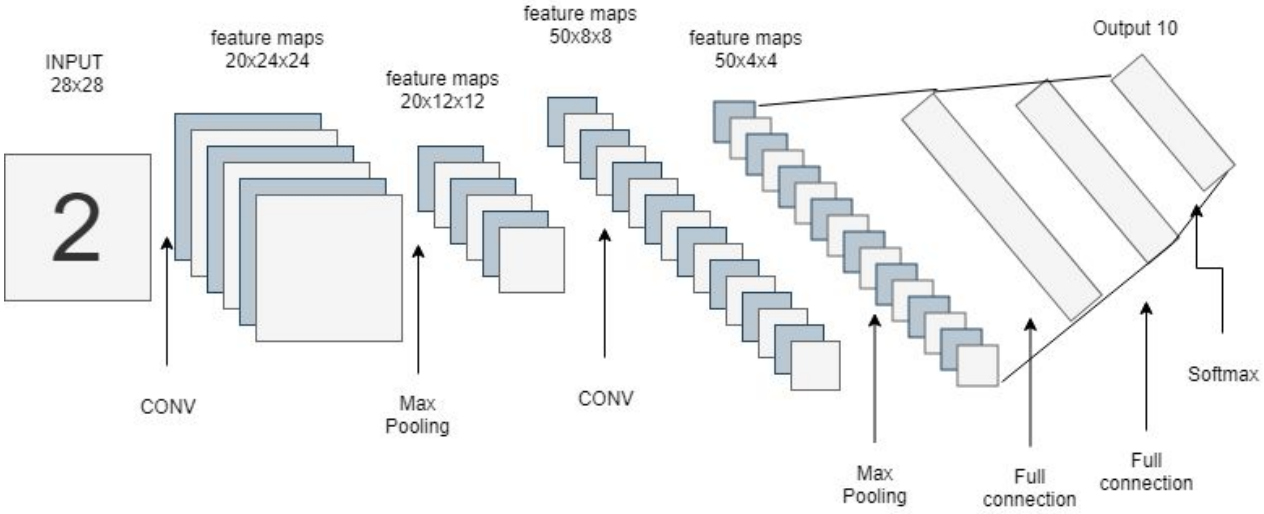


Figure 2: Final Block Diagram of CNN Layers

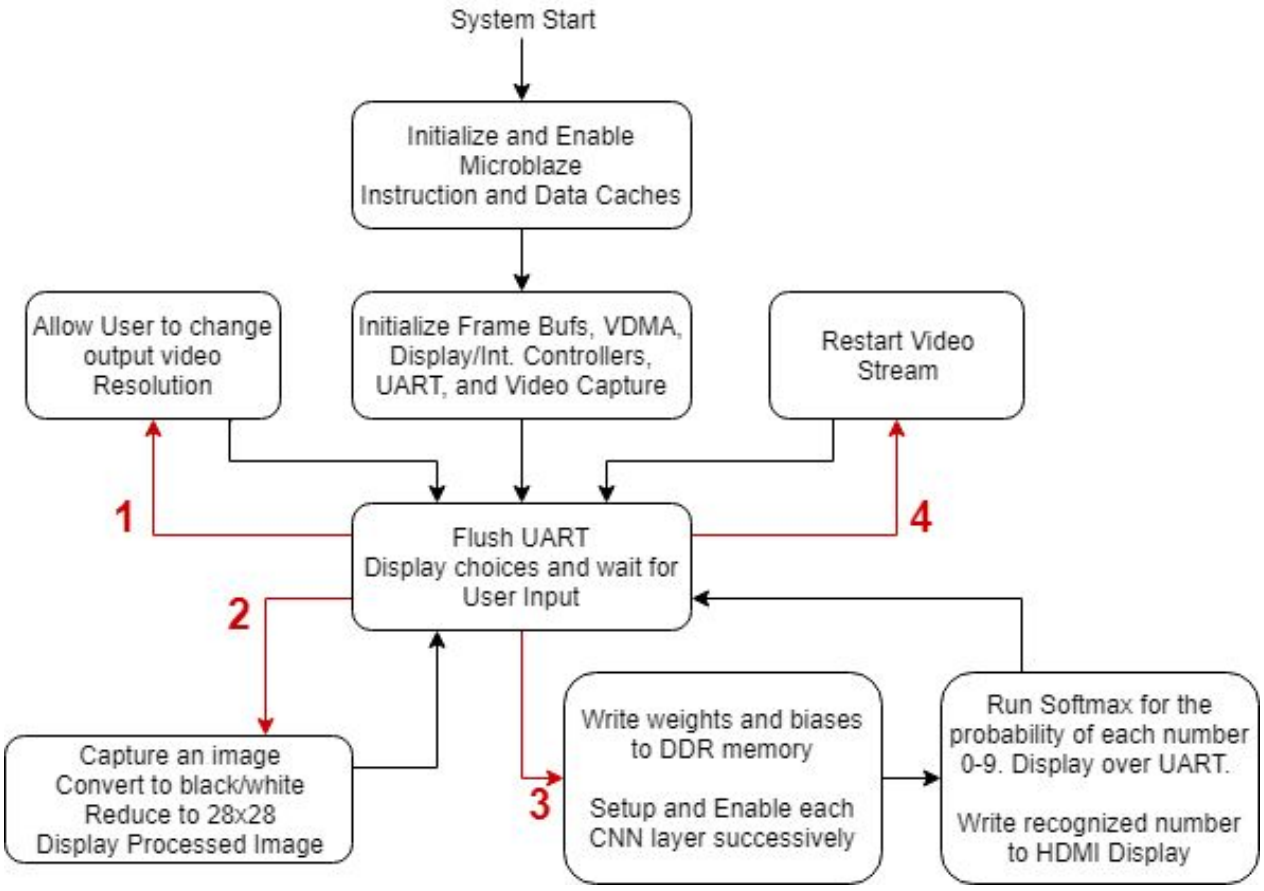


Figure 3: Top-Level Software Control Diagram

There are two major sections in this project, the first of which is the neural network core. Carrying forward from the ECE1373 assignments, this core utilized the Xilinx High-Level Synthesis engine to convert C code into hardware. The neural network core contains three different types of layers: fully-connected (FC), convolutional (CONV), and max-pool (POOL). The FC layer connects all inputs to all outputs and is the final decision making layer, providing ten outputs. The CONV layer applies the following operation on a K*K window with S stride: $output(x, y, fo) = \sigma[(\sum_{fi}^{ID}\sum_{i}^{K}\sum_{j}^{K} weight(i,j,fi,fo) * input(x*S+i, y*S+j, fi)) + bias]$. Finally, the POOL layer finds the maximum in a sliding window to reduce the number of parameters required. These layers are cascaded as CONV → POOL → CONV → POOL → FC → FC to increase the depth and accuracy of the neural network. Using the MNIST database of handwritten digits [4], the core was "trained", obtaining the weights and biases required by each CONV and FC layer. When initialized, these weights and biases are written to the corresponding DDR memory offsets for each layer via software. The core was provided a 28x28 black and white image and outputs ten numbers for softmax to calculate the probabilities of each digit.

The second major section of the project is the system components that interface with the neural network. As a starting point, the Digilent Nexys Video HDMI Demo [5] was used for the HDMI datapaths (Camera → DDR & Microblaze → Output screen). On the input datapath, the camera streamed 1080p video in Transition-Minimized Differential Signaling (TMDS) format, which is converted to an RGB AXI-4 Stream input using Xilinx IP blocks. This data is stored in DDR memory by the Video Direct Memory Access (VDMA). The Microblaze processor converts this data to black and white and also compresses the 1920x1080 video frame to 28x28 for the neural network core input. Finally the system displays the detected digit on the Nexys Video's HDMI output and communicates the probabilities of each number over UART. This process can be repeated by issuing new commands over the UART Serial console.

2. Current Status

As of May 2nd 2019, the project has been successfully implemented and a demo video was created. As described above, the completed work includes:

- Use HLS to create the neural network layers and combine multiple layers in one core.
- Train the neural network core with MNIST data. Write bias and weight data to DDR.
- Test the system by providing a Microsoft Paint image to the software as a 28x28 array (by-pass the HDMI datapaths).
- Update the Nexys Video HDMI Demo Project for additional pins and Vivado 2017.2
- Implemented video frame manipulation in software
- Integrated and tested the whole system.

This project fulfils the initial goals and the following list provides a brief overview of next steps:

- Encountered a bug while trying to write to DDR more than once via Microblaze.
- Adding HLS pragmas and optimizations resulted in routing resource errors.
- HDMI video data is converted from RGB to B/W and downsampled from 1920x1080 to 28x28 in software. Implementing this in hardware could be more efficient via the Xilinx Video Processing Subsystem IP.
- Upgrade to the Zynq processor for live video processing and more advanced HDMI output images (Microblaze is too slow for heavy video processing [5]).

These potential improvements are discussed in greater detail below in Sections 8 and 9.

3. Initial Architectural Design

From the above introduction, the project contained two very distinct sections - the HLS CNN core and the Microblaze/HDMI System. Early in the project, the team decided that Yifeng and Genwen would focus on the HLS CNN layers and IP exporting process. In the initial proposal presentation, the layers were selected to be CONV, ReLU, POOL, and FC [6]. Based on internet research, the team decided one layer of each type would provide a straight-forward starting point for understanding neural network layers, allowing the team to clearly understand the impact of each layer. At this early point in the project, it was not clear what input would be required for the core or how to train the layers, just that it would be done in python.

The remaining team members, Richard and Jianxiong, were tasked with understanding and building a system to showcase the CNN core; to provide a demo that would allow anyone to interact with and see machine learning in action. The team came across the Digilent Demos for the Nexys Video boards. The input was decided to be an HDMI camera, using the Digilent HDMI demo [5]. For the output, the team was leaning towards the HDMI output, but also heavily considered the OLED display. At this point in the project, the team had not discussed if the system would capture live video streams or if there would be a mechanism to capture a single video frame. Video noise reduction and downsampling were also not yet established and were to be researched. The following block diagram illustrates the initial system and the CNN layers.

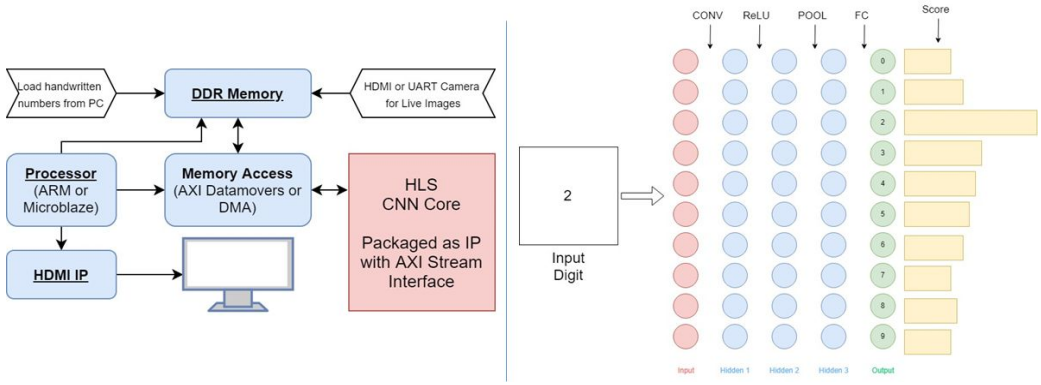


Figure 4: Initial Proposed Block Diagrams - System (Left), HLS CNN Core (Right)

4. Specification Evolution and Final Architectural Design

The figures provided in this section illustrate the final high-level block diagrams of the system architecture, the CNN layers, and the software control flow. Throughout the project, the team did not encounter any major setbacks that required architecture changes. Compared to the proposed block diagrams above, the team needed to create new block diagrams with significantly more detail as the project progressed.

Figure 5 below provides an overview of the final system, where the main SoC components are the Microblaze soft-core processor, the AXI interconnects, and UART interface. Additionally the below diagram illustrates the HDMI input and output path, as well as the CNN layers from the HLS engine and Vivado IP packager. From a very high-level perspective, the processor initializes the HDMI datapath and the parameter data for the CNN layers. The system waits for the user to issue a command over the UART interface, which triggers the HDMI input path to capture a video frame into DDR. This is processed into a black and white, 28x28 image for the HLS CNN core by software. Each layer of the core is successively enabled, resulting in a final detected number between 0-9 and the probabilities of each via softmax. Finally, the number is displayed on the HDMI output screen through a software function for drawing numbers. The following subsections provide greater details about these main hardware components.

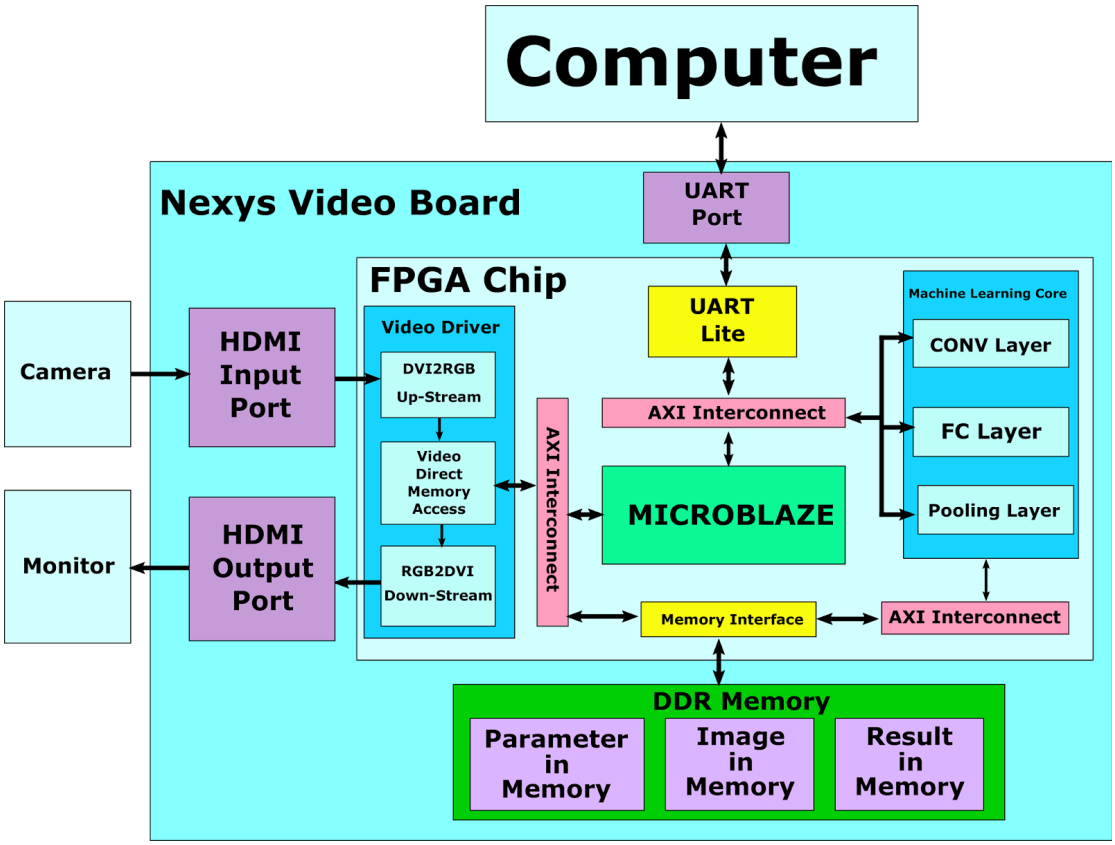


Figure 5: Final System Level Block Diagram

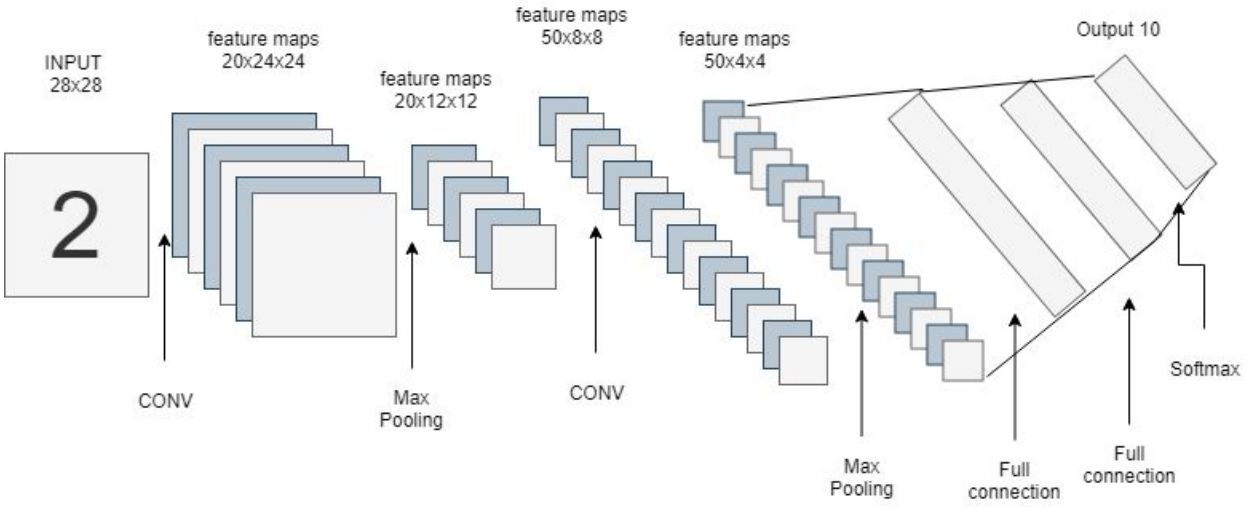


Figure 6: Final Block Diagram of CNN Layers

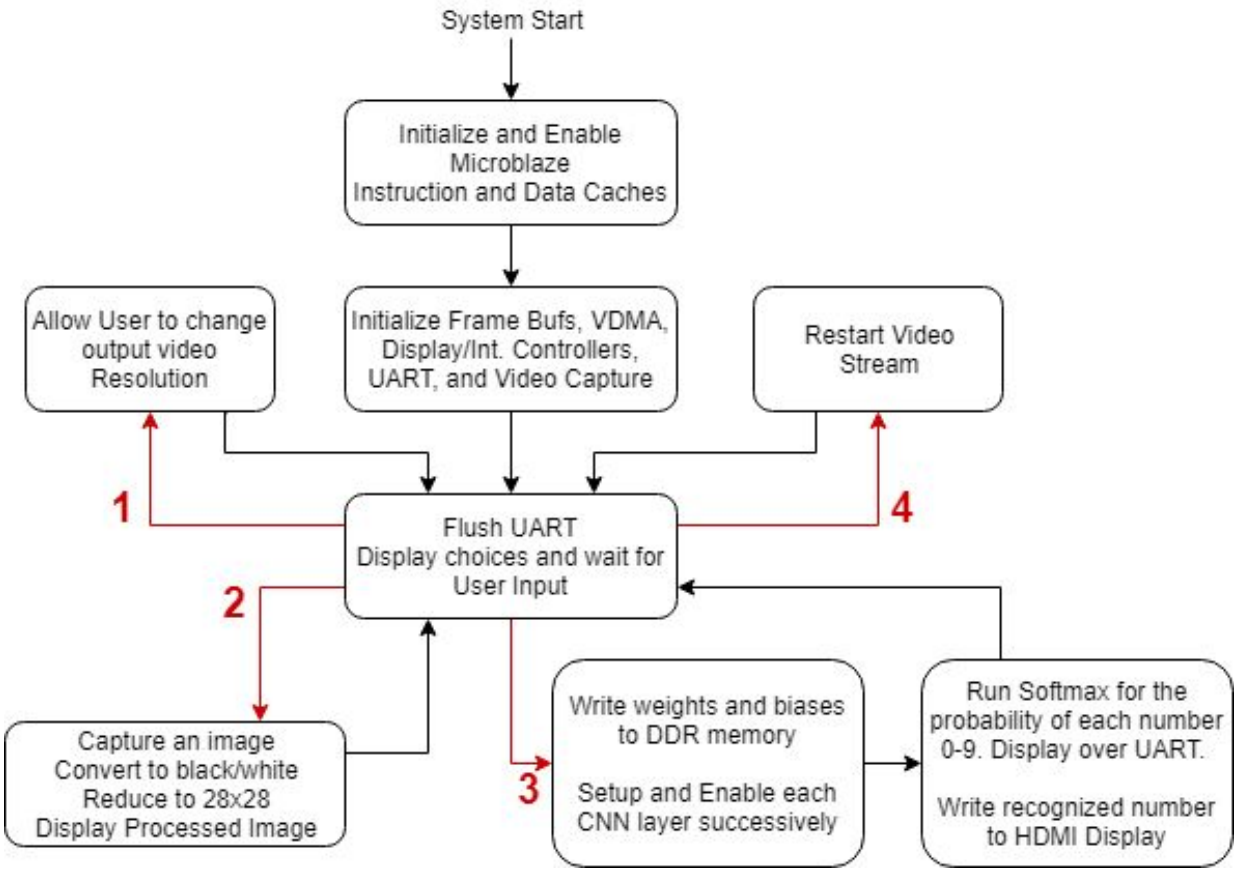


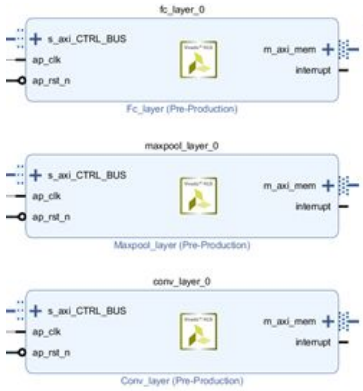
Figure 7: Top-Level Software Control Diagram

4.1 HLS CNN Layers

As shown in Figure 6 above, the complete convolutional neural network in this project consists of six layers, CONV → POOL → CONV → POOL → FC → FC. In the below diagrams, this core is implemented with only one layer of each (CONV, POOL, FC) in hardware and re-uses these layers via software. This was decided due to the image processing in software and overall system latency - the team decided to operate on single frames of size 28x28 rather than a live, high-definition video stream. In software, the cores are enabled in succession and are provided with input and output offsets - the first two arguments in the software function below. These values are decimal addresses in DDR memory and convert to the following in hexadecimal:

- Layer 1 - conv_1: Input_Offset 0x90000000, Output_Offset 0x91000000
- Layer 2 - pool_1: Input_Offset 0x91000000, Output_Offset 0x92018768
- Layer 3 - conv_2: Input_Offset 0x92000000, Output_Offset 0x93000000
- Layer 4 - pool_2: Input_Offset 0x93000000, Output_Offset 0x941871D0
- Layer 5 - fc_1: Input_Offset 0x94000000, Output_Offset 0x95004E48
- Layer 6 - fc_2: Input_Offset 0x95000000, Output_Offset 0x96000000

In the above mapping, the outputs from one layer feed the inputs of the next. The jumps in offsets are due to weights and biases being pre-loaded in memory during the SW initialization.




```
void hw_hls_core_processing ()
{
    printf("start hls core processing \n");
    //layer 1 - conv_1
       hw_conv_layer(2415919104, 2432696320,  1, 28, 28, 20, 24, 24, 1, 5);
    //layer 2 -pool_1
    hw_maxpool_layer(2432696320, 2449573736, 20, 24, 24, 20, 12, 12, 2, 2);
    //layer 3 - conv_2
       hw_conv_layer(2449473536, 2466250752, 20, 12, 12, 50,  8,  8, 1, 5);
    //layer 4 -pool_2
    hw_maxpool_layer(2466250752, 2484629968, 50,  8,  8, 50,  4,  4, 2, 2);
    //layer 5 -fc_1
         hw_fc_layer(2483027968, 2499825224, 800, 500, 1);
    //layer 6 -fc_2
         hw_fc_layer(2499805184, 2516582400, 500, 10, 0);
    printf("done hls core processing \n");
}
```

Figure 8: HLS CNN Layers in HW (Left), SW Function to Successively Enable Layers (Right)

The Caffe framework was used to train the LeNet model. Before we began to train the network, the MNIST database of handwritten digits were downloaded from the official website [4]. The database has a training set of 60,000 samples and a test set of 10,000 samples. In addition, the model definition (lenet_train_test.prototxt) and solver (lenet_solver.prototxt) files were created by following a Caffe tutorial [7]. The model definition file defines the layer type, connectivity, parameters and dimensions of the input and output. The solver file defines how Caffe should train the model. After the preparation was done, the training was carried out on a server CPU and took 18 minutes to complete. The output is the caffe model file with the weights and biases for each layer. Using the extractParams_imagenet.py script from the previous assignment, the weights, biases, input and output were extracted for the HLS CNN core development.

4.2 HDMI Input Datapath

The below figure highlights the HDMI input datapath in orange. Starting with a video stream from a camera or computer, this data is received by the Nexys Video board's HDMI_IN port, called TMDS_IN. This video data is in Transition-Minimized Differential Signaling (TMDS) format and must be converted before being processed or displayed. The input is a DC-balanced, twisted pair, digital video signal, that is converted to 8 bits of red, green, and blue (RGB) for each pixel by the dvi2rgb IP. This data, along with video timing information (blanking and sync from the v_tc IP) is converted to the AXI-4 Stream protocol and sent to the Video Direct Memory Access (VDMA) IP to be stored in DDR. This datapath is initialized and controlled by software functions to start streaming, stop streaming, or capture a single frame to one of three frame buffers. In software, a captured video frame is converted from 8-bit RGB to 1-bit black and white, as well as downsampled in resolution from 1920x1080 to 28x28. This image processing is required to interface with the HLS CNN core described in Section 4.1 above.

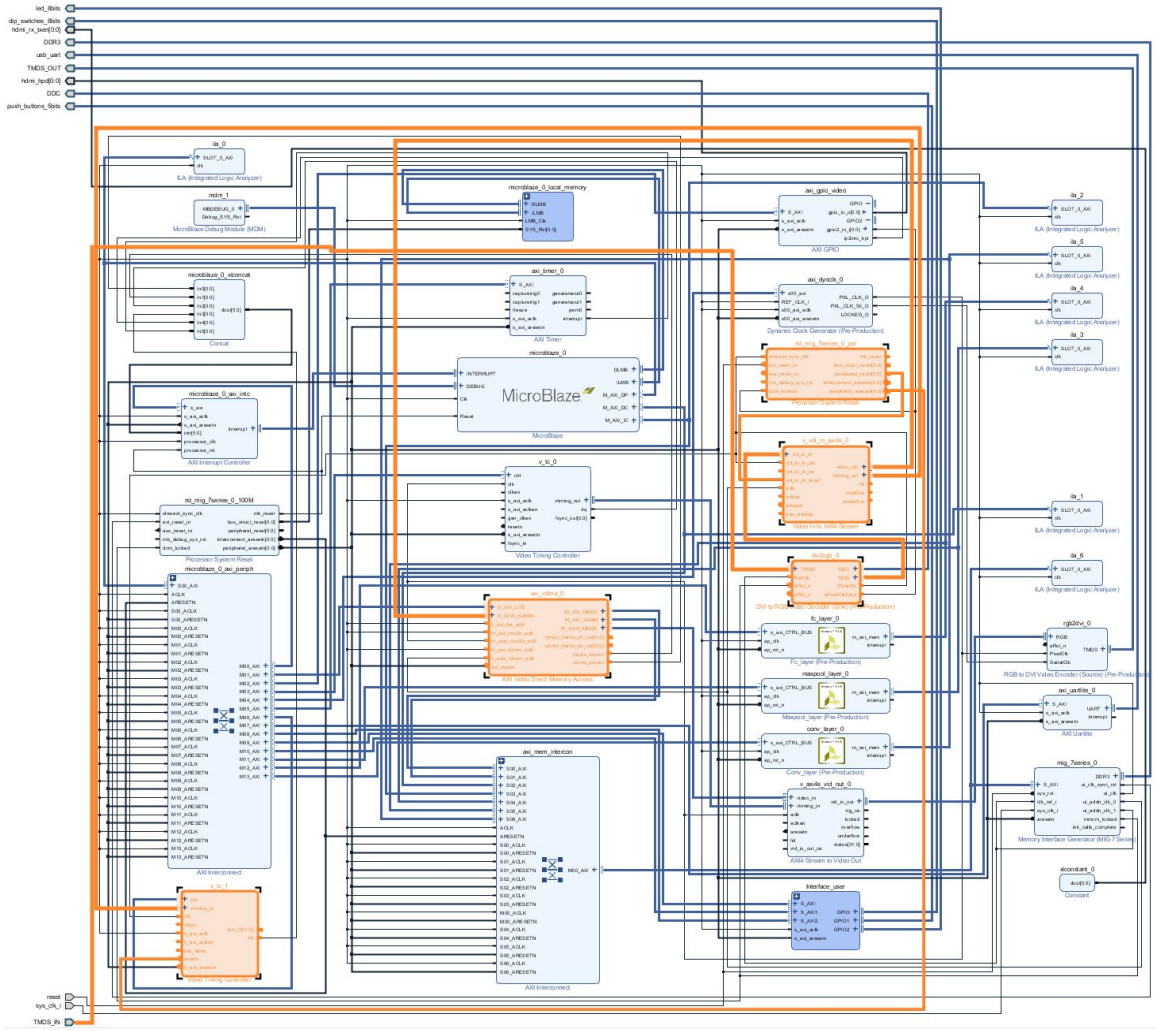


Figure 9: HDMI Input Datapath

4.3 HDMI Output Datapath

The HDMI output datapath is the opposite of the input datapath, with the affected blocks highlighted in orange below. After processing the input data and enabling the HLS CNN core, a software function draws the recognized number (0-9) to the frame buffer. When enabled, the hardware will take the frame via the same VDMA. This RGB AXI-4 Stream protocol data must be converted to a TMDS/DVI video stream. The reverse of the input datapath is required and uses the opposite IP blocks: “AXI-4 Stream to Video Out” and “RGB to DVI Video Encoder”, and another video timing controller. This converted signal can be sent to the TMDS_OUT pins and finally the HDMI output display. The datapath also controls the following FPGA pins:

- Display Data Channel (DDC) - pass through from camera input to monitor output - I2C bus to prevent jitter and transmission delay.
- hdmi_rx_txen - Tied to constant IP block to always enable receiver/transmitter.
- hdmi_hpd - Connected to AXI GPIO to allow Hot Plug Detection (HPD).

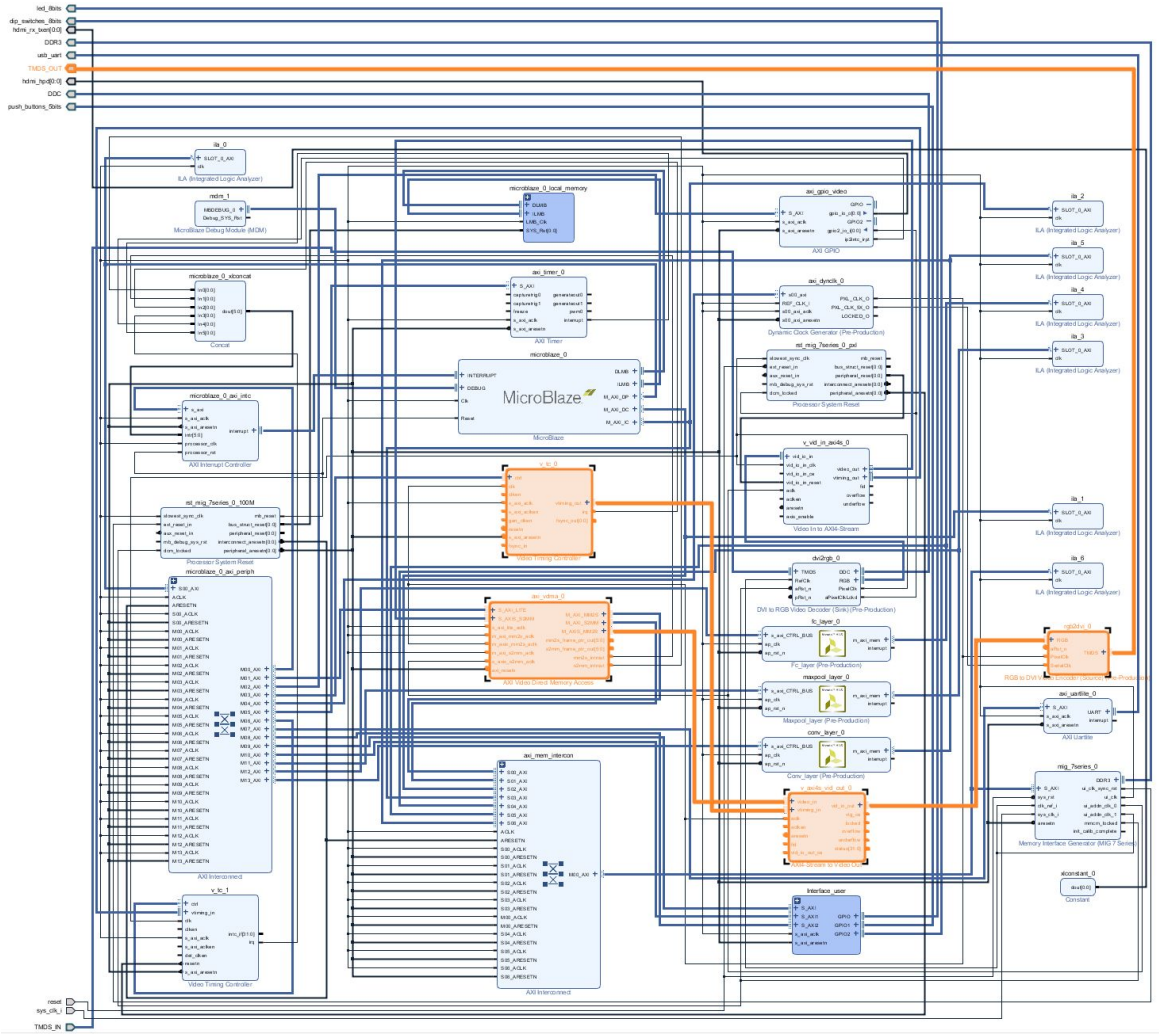


Figure 10: HDMI Output Datapath

4.4 System Glue Logic & Debugging - Microblaze, AXI Interconnects, MIG, UART, ILA

The remaining blocks have been highlighted in the following diagram and pertain to system level glue logic and debugging. The first and central block in this subsection is the Microblaze soft-core processor. This processor runs the software to initialize and control the hardware blocks. Supporting IPs for this processor are the debug module, local cache memory, interrupt handler, reset generation, and the MIG 7 DDR3 controller. This leads to the next major component, the AXI interfaces. This project contains two AXI interfaces, the first is to arbitrate DDR memory access between any blocks that need access. This includes the Microblaze's instruction and data caches, the VDMA, and the HLS CNN layers. The second AXI interface controls the Microblaze's AXI peripherals, which includes the HDMI datapath blocks, the HLS CNN layers, and the UART controller. For debugging purposes, several Integrated Logic Analyzers (ILAs) were added to the project.

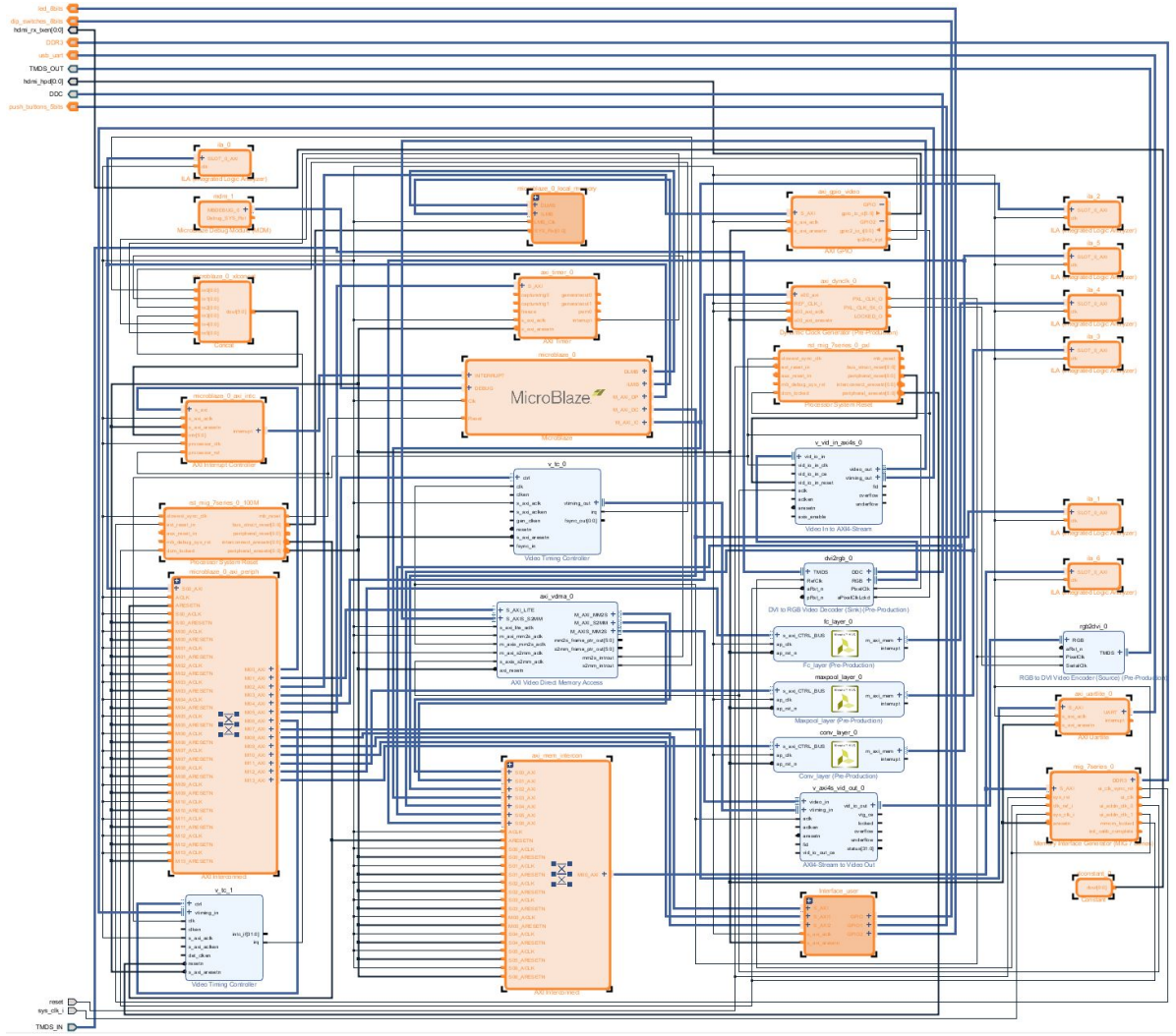


Figure 11: System Glue Logic & Debugging

5. Methodology

5.1 Design Environment

This project is developed on Nexys Video board with the Artix-7 XC7A200T-1SBG484C FPGA. The main software tools used were Vivado, Vivado-HLS, and Xilinx SDK 2017.2. Python 3.7.3 was also used for the MNIST training data and to bypass the input HDMI datapath during the integration and system bring-up phase. The final project is uploaded to Github and organized in the below structure:

Nexys_Video_HDMI

|--- hw_handoff/	- HDMI Wrapper handoff hardware description file (HDF)
|--- mlc/	- Contains Custom IP (HLS CNN Cores)
| |--- {conv/fc/maxpool}_test/	- Source files and a script to generate each layer
| |--- hls_proj/	- Synthesis/implementation solution1 for each layer
|--- proj/	- Main project folder for complete system
| |--- HDMI.sdk/	**- Exported HW Platform and Xilinx SDK Location**
| |--- HDMI.xpr	**- Vivado Project File**
|--- repo/	- Contains Vivado IPs from the Digilent HDMI Demo
|--- src/	- Constraints and Block Diagram related files
|--- Block Diagram.pdf	- High resolution Vivado block diagram
|--- Final Report.pdf	- This report file
|--- Final Presentation.pdf	- Presentation Slides
|--- README.md	- This readme contains the same file structure explanation

Early in the project, each team member worked on separate IP components. This allows all project files to be developed standalone, and avoid team members to access same design files simultaneously. With some upload limitations on the free Github version, the team decided to use a combination of Dropbox and Google Drive for sharing design files throughout the project. The team integrated and tested the final design through in-person meetings when all parts were ready.

5.2 Partitioning

This project involves both software and hardware portions. On the hardware side, the main divisions are the HLS CNN core, the HDMI datapaths, Microblaze and DDR. On software side, HDMI image processing, CNN drivers, and UART control are required. By partitioning the design based on these subsections, each can be independently developed and verified with minimal interference to others. The team also clearly defined all the interfaces between hardwares and software functions to avoid unnecessary integration issues. This allows the team

to work in parallel and improves work efficiency. This clear ownership also significantly reduces the source file conflicts and the time to merge design files.

5.3 Simulation, Verification and Testing

From the previous section, the design is partitioned into hardware and software parts such that each team member can complete their assigned parts independently. Through strong communication and the use of diagrams, the final integration occurred smoothly and quickly. Each subsection was individually developed and verified before being integrated into the system.

For the HLS CNN core, each layer was first verified in software, then tested in hardware. After exporting these layers as an IP onto the FPGA, an intermediate Python script was used to resize and pre-process an image from Microsoft Paint (bypassing the HDMI input path). This test ensured the HLS CNN core functioned correctly with only a Microblaze and DDR. Another major hardware task is the use of HDMI camera to capture image and write the data to DDR. The captured image was written to DDR and the data was read back from DDR to ensure that we have the same data. This captured image was also sent to the HDMI output display monitor to quickly verify the datapath visually.

For the software components, an image compression engine was developed for the Microblaze to downsize the 1920x1080 RGB image to a 28x28 black and white image. This function was tested similarly to above, by capturing numbers and viewing the processed output on the HDMI display. The team also used images from Microsoft Paint and compared the 28x28 output with the intermediate Python script to ensure the averaging algorithms were correct. Several other software functions were developed and tested individually, such as the softmax, DDR writes, and CNN core drivers.

After all the individual components were fully verified and tested on the FPGA, the team integrated the whole system into one Vivado project. The team used two input methods to test the system. The first used a computer to drive the HDMI input port, providing a full-screen handwritten number in Microsoft Paint. This provided noise-free images and allowed the team to accurately assess the accuracy and find any bugs in the system. The final testing method was to remove the computer and use an HDMI camera to drive the HDMI input on the FPGA. This introduces noise from shadows and light sources, but allows the team to rigorously test the design, leading to team adding a threshold in the B/W converter. For the following figure, the team created 40 handwritten digits to test the whole system and achieved a Top1 accuracy of 97.5%, as well as a Top5 accuracy of 100%.

label	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
recognized number	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
prob for 0	0.726	0.0612	0.0015	0	0	0	0.0012	0.0009	0.0001	0	0.9778	0.0011	0.0114	0	0	0	0.0012	0.0009	0.0009	0
prob for 1	0.0003	0.4501	0.0045	0	0	0	0.041	0.021	0	0	0	0.9776	0.1466	0	0	0	0.0396	0.0208	0.0208	0
prob for 2	0.2572	0.0512	0.9271	0	0	0	0.1628	0.0239	0.0731	0	0.001	0.0036	0.4239	0	0	0	0.1683	0.0236	0.0236	0
prob for 3	0.0005	0.0209	0.0476	1	0.0001	0.0209	0.037	0.1976	0.0107	0.0008	0	0.0009	0.0885	1	0.0001	0.0208	0.0376	0.2031	0.2031	0.0008
prob for 4	0	0.0613	0.0004	0	0.9982	0	0.3166	0.0002	0	0.1868	0	0.0053	0.0065	0	0.9982	0	0.317	0.0002	0.0002	0.1868
prob for 5	0.0028	0.0738	0.0079	0	0.0001	0.9754	0.0472	0.0108	0.0012	0	0	0.0006	0.0336	0	0.0001	0.9755	0.0485	0.0105	0.0105	0
prob for 6	0.0019	0.0462	0.0005	0	0	0	0.3857	0	0	0	0.0211	0.0005	0.0119	0	0	0	0.3795	0	0	0
prob for 7	0.0003	0.1893	0.0038	0	0	0	0.0012	0.4525	0.0009	0	0	0.0048	0.2454	0	0	0	0.0012	0.4492	0.4492	0
prob for 8	0.0103	0.0109	0.0026	0	0	0.0001	0.0056	0.0013	0.914	0.0011	0	0.0002	0.0081	0	0	0.0001	0.0055	0.0013	0.0013	0.0011
prob for 9	0.0007	0.0351	0.004	0	0.0015	0.0036	0.0017	0.2919	0	0.8113	0	0.0053	0.0241	0	0.0015	0.0037	0.0017	0.2905	0.2905	0.8113

label	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
recognized number	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	9	6	7	8	9
prob for 0	0.8833	0.009	0.0001	0	0	0	0	0.0006	0.0254	0	0.9994	0.0074	0	0	0	0.0003	0.0001	0.0006	0	0
prob for 1	0.0147	0.8887	0.0099	0	0	0.0006	0	0.0836	0	0	0	0.8618	0.0031	0	0	0.0009	0.0001	0.0836	0	0
prob for 2	0.0382	0.0283	0.9898	0.0001	0.0004	0.0013	0.0057	0.0158	0.0066	0	0	0.0687	0.9968	0.0008	0	0.0006	0.0029	0.0158	0	0
prob for 3	0.0004	0.0055	0	0.9995	0.0002	0.0747	0.168	0.3391	0.0009	0.0007	0	0.0137	0	0.9966	0	0.0103	0.0085	0.3391	0.0001	0.0105
prob for 4	0.0169	0.0026	0	0	0.9986	0.0182	0.0001	3	0.0002	0.0075	0	0.0154	0	0	0.9993	0.0137	0.0018	0.0005	0	0.0001
prob for 5	0.0171	0.0066	0	0.0003	0.0005	0.5671	0.0157	0.0015	0.0063	0.0002	0	0.0086	0	0.0004	0.0001	0.0308	0.0052	0.0015	0	0
prob for 6	0.0191	0.0037	0	0	0	0	0.8096	0	0.006	0	0	0.0119	0	0	0	0.0002	0.9778	0	0	0
prob for 7	0.0032	0.0341	0.0002	0	0.0001	0.0001	0	0.413	0.0001	0.0001	0.0002	0.006	0	0.0003	0	0.0001	0	0.413	0	0
prob for 8	0.0064	0.0094	0	0	0	0.001	0.0008	0.0038	0.9546	0.0007	0.0003	0.0013	0	0.0012	0	0.0005	0.0034	0.0038	0.9999	0.0005
prob for 9	0.0007	0.0122	0	0	0.0001	0.3369	0	0.1421	0.0001	0.9908	0	0.0052	0	0.0007	0.0005	0.9426	0	0.1421	0	0.9888

Figure 12: Results from 40 handwritten digits to measure the complete system

6. Contributions

6.1 Richard Lin

Richard worked with Jianxiong on the HDMI datapath and the software functions for image processing. At the beginning of the project, Richard worked on bringing up the Digilent HDMI Demo with an HDMI camera input. Richard focused on converting the RGB image to black and white, as well as ensuring the datapath and UART function correctly in Vivado 2017.2. Towards the project end, Richard also edited the demo videos and worked heavily on this report.

6.2 Jianxiong Xu

Jianxiong worked with Richard on the system level design - specifically the HDMI datapath and software control flow. At the beginning of the project, Jianxiong debugged the Digilent HDMI Demo project with another computer providing the HDMI input and used this input device throughout the project. Jianxiong developed the functions to compress a video frame from 1920x1080 to 28x28 and a function to draw the detected number from softmax on the HDMI output monitor. Jianxiong also worked on integrating the HLS CNN core and refining the overall software control flow.

6.3 Yifeng Zhang

Yifeng was responsible for training the LeNet model with the MNIST dataset. This process includes researching the training procedure using Caffe framework, preparing the required dataset and files, and performing the training on the LeNet model. After the training finished, the weights, biases, sample inputs and outputs for each layer were extracted and used for developing the initial HLS core in C++. Test benches were developed to verify the functionality of the HLS core by feeding it with drawn number and calculating the probabilities of the output numbers. Yifeng also helped on the HLS core bring up on FPGA, including the debug of the DDR write

issue and writing data to the DDR. After the successful bring up of the HLS core, Yifeng developed test bench to verify the functionality of the HLS core (on FPGA) by feeding it with images drawn in Paint (not from HDMI).

6.4 Genwen Zhao

Genwen was responsible for HLS CNN core performance optimization, IP package generation, integration and other CNN core related software development. Data Sparsity optimization has been added. Data type change has been experimented, but dropped due to performance issue. CNN core was successfully generated and integrated to system. Genwen heavily involved in HLS core bring up, debug and overall system software flow development. ILA debug blocks were added to system aiding for verification and debug purpose. Genwen debugged the DDR issue, and architected the DDR usage with the team. Genwen developed all the CNN core driver functions in C and developed the CNN IP level verification test bench. The CNN core was successfully brought up and verified functionality.

7. Design Characteristics

7.1 Resource Utilization

The following figures provide the post-implementation utilization of chip resources and the power summary.

Resource	Utilization	Available	Utilization...
LUT	56279	133800	42.06
LUTRAM	6680	46200	14.46
FF	77433	267600	28.94
BRAM	172.50	365	47.26
DSP	61	740	8.24
IO	93	285	32.63
BUFG	5	32	15.63
MMCM	3	10	30.00
PLL	1	10	10.00

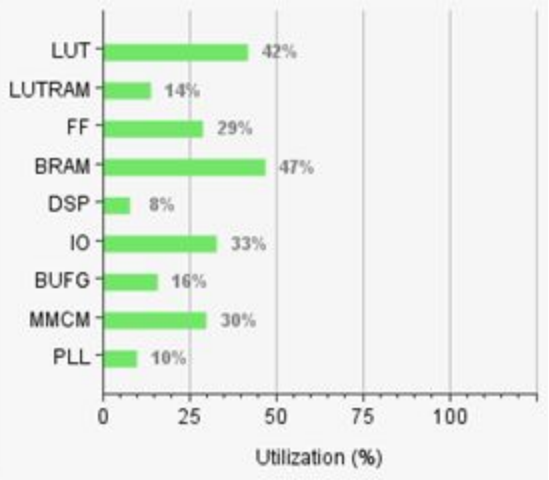


Figure 13: Resource Utilization Summary

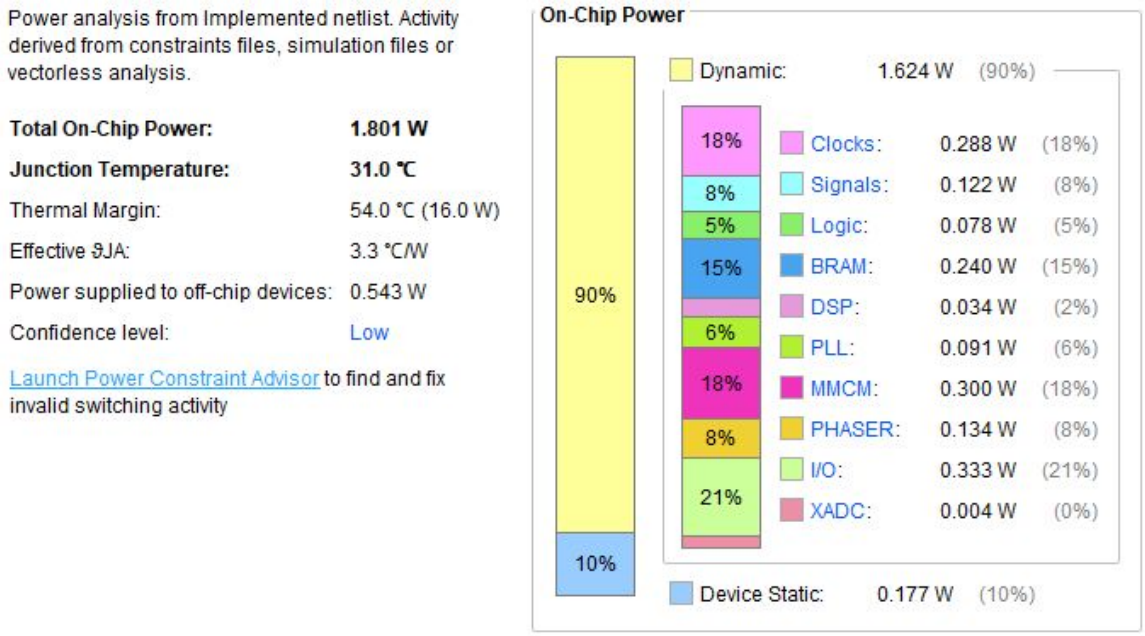


Figure 14: Power Summary

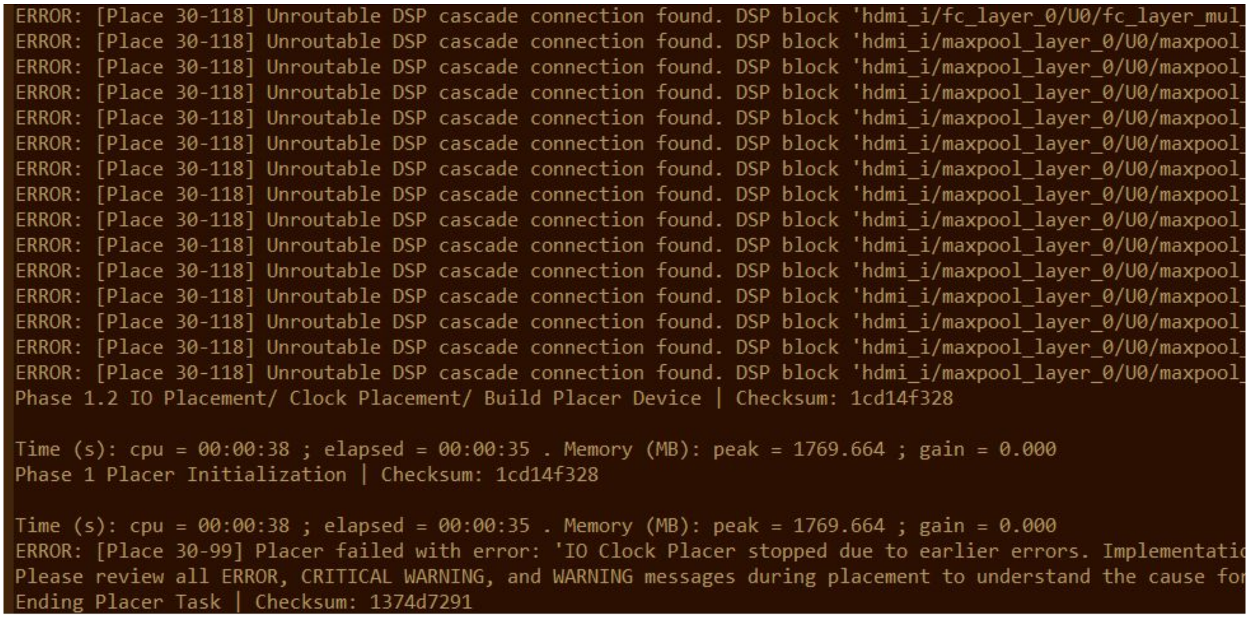
7.2 Where the Time Went

Initially it took some time (2-3 weeks) to research and learn how to train the LeNet model using the Caffe framework on a Linux server and later deploy that model in HLS. Setting up the Nexys FPGA board and understanding the interconnect requirements consumed some time too (2 weeks). Subsequently, the most time-consuming task was the bring up of the HLS core in FPGA (3 weeks). The team spent a lot of time to understand the interfaces between sub-blocks and customization of the sub-blocks. One of the major issues the team faced during the bring up was the access to the DDR memory. Integrated Logic Analyzers (ILA) were added to monitor the internal signal of the system. Eventually, the issue was found (initialization problem) and fixed. After the DDR issue was fixed, the team spent some time on the setup and configuration of the HDMI camera and the development of the image compression engine. Finally, the team worked on the system level test with handwritten numbers and resolved any issues encountered.

8. Problems

There were four main technical issues the team encountered during project. On the hardware integration side, the first issue was applying the PIPELINE PRAGMA in the HLS CNN core to improve design latency. This PRAGMA resulted in a hardware architecture with high data parallelism, but significantly higher hardware resource usage. After running HLS and exporting the IPs, the Vivado tool would fail placement with an “Unroutable DSP cascade error”, as seen in the below figure. The team had experimented with newer version Vivado, but did not resolve this issue. The team considers this error to be a limitation of the FPGA resources. The team

resolved this issue by removing PIPELINE PRAGMA, resulting in less hardware but higher latency.



```
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/fc_layer_0/U0/fc_layer_mul_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
ERROR: [Place 30-118] Unroutable DSP cascade connection found. DSP block 'hdmi_i/maxpool_layer_0/U0/maxpool_
Phase 1.2 IO Placement/ Clock Placement/ Build Placer Device | Checksum: 1cd14f328

Time (s): cpu = 00:00:38 ; elapsed = 00:00:35 . Memory (MB): peak = 1769.664 ; gain = 0.000
Phase 1 Placer Initialization | Checksum: 1cd14f328

Time (s): cpu = 00:00:38 ; elapsed = 00:00:35 . Memory (MB): peak = 1769.664 ; gain = 0.000
ERROR: [Place 30-99] Placer failed with error: 'IO Clock Placer stopped due to earlier errors. Implementatio
Please review all ERROR, CRITICAL WARNING, and WARNING messages during placement to understand the cause for
Ending Placer Task | Checksum: 1374d7291
```

Figure 15: Xilinx Vivado Placement Errors

A blocking issue the team resolved was debugging DDR memory accesses. During the initial HLS CNN core bring up, the output calculation was inaccurate. ILA debug blocks were added to monitor AXI bus activities. After debugging the address and data activities on ILA, the team was able to find the root cause on random data being read from DDR. The base address was assigned to be 0x80000000 in the hardware address mapping. When the hardware was first powered up, the Microblaze tried to initialize DDR with some built-in data cache and instruction cache functions. However, when the team tried to read data from an offset address, it read back random data and any writes returned unsuccessfully. The team further debugged and found the failure was in the SDK application with incorrect offset address pointers. During this process, the team also came up with a new offset mapping as shown above in Figure 8 to manage different CNN layers accessing data from DDR memory.

In the image processing software, the first issue related to converting a video frame to black and white, as well as compressing the 1920x1080 image down to 28x28; the team encountered some challenges with robustness in both operations. For converting RGB to black and white, the image can be noisy and disrupt the neural network's results based on the lighting and shadows. To address this issue, a threshold (based on trial-and-error) was added to help further distinguish between shadows and the actual number on the page. This solution improves results and accuracy of the neural network, but can vary depending on the lighting conditions and should be recalibrated manually.

The second issue encountered in the image processing software related to compressing the 1920x1080 image to 28x28 for the neural network input. The team approached this challenge like the maxpool and convolutional layers, by using a sliding window to find the averages. The stride was set with no overlap and a 68x38 window to produce a 28x28 result. This solution works for the 1920x1080 camera images, but is inflexible and can affect the accuracy with other image resolutions. This issue was discovered when using a computer to provide Microsoft Paint images at various resolutions (by mirroring the display output to the FPGA's HDMI input port).

9. Retrospective, Conclusions, Suggestions, Comments

9.1 Experience with HLS

High-level synthesis lowers the barrier for entry, allowing software developers to generate hardware accelerators with a higher level programing language and no RTL knowledge. In our project, we used HLS to develop a full convolutional neural network by writing the code at the algorithmic level. One interesting aspect of HLS is the usage of PRAGMAs, which are options to generate different hardware for particular design requirements. However, PRAGMA usage lead to an issue where Place and Route failed due to DSP resource overuse on this particular FPGA. The team resolved the problem by removing PRAGMA "PIPELINE", which reduced hardware resource usage, trading off performance and parallelism. If the team were to develop the full CNN in HDL, the team estimates it would take approximately 2-3 times longer in development, and whenever changes were required, significantly longer turnaround time.

Vivado's High-level synthesis engine proved to be a great asset for fast algorithm development and testing in hardware. However HLS created some uncertainty in the expected generated hardware. In some situations, the use of PRAGMAs did not generate what was expected and led to unexpected behaviours. For timing sensitive logic, the team recommends FSM control logic to be developed in HDL for more predictable hardware.

9.2 Other

What would you do differently next time?

- Improve revision control - A combination of Dropbox and Google Drive were used to share the project. Without a paid Github account, submit sizes and file limitations made it unusable. Issues were encountered in sharing files and importing Vivado projects.
- Try a different FPGA - As mentioned in Sections 8 and 9 above, the Vivado tool encountered errors during PnR of optimized HLS pragmas.

What do you think you got out of the course, if anything?

- This course offered an understanding of neural networks and digital systems design in the Xilinx environment. At the start, the team had limited exposure to both, but now have a better understanding.

What did you learn about systems design?

- The Xilinx Vivado IP Integrator made system design intuitive and straightforward. The tool makes it easy to add and auto-connect IP blocks for complex operations like video conversions and DDR access. The team also learned more about implementing soft-core microprocessors and using the AXI protocol.

Comments on the format of the course and suggestions for improvement?

- The course lectures cover important digital design concepts that are often not discussed in undergraduate courses, such as cross-domain clocking, SerDes, and high-level synthesis. However these discussions are very high-level and not structured enough.
- The assignments were great for learning the Vivado tools and the fundamentals of neural networks, but used many weeks of the course to debug the instructions and container.

10. References

[1] L. Williams, "Artificial Intelligence Boosts Healthcare Advancements", *Sensors Magazine*, 2019. [Online]. Available: https://www.sensorsmag.com/components/artificial-intelligence-boosts-healthcare-advancements. [Accessed: 27- Apr- 2019].

[2] A. Hawkins, "Watch a self-driving car handle hairpin turns like a race car", *The Verge*, 2019. [Online]. Available: https://www.theverge.com/2019/3/31/18285824/self-driving-car-race-car-stanford-research. [Accessed: 27- Apr- 2019].

[3] D. Bilefsky, "He Helped Create A.I. Now, He Worries About 'Killer Robots.'", *The New York Times*, 2019. [Online]. Available: https://www.nytimes.com/2019/03/29/world/canada/bengio-artificial-intelligence-ai-turing.html. [Accessed: 27- Apr- 2019].

[4] Y. LeCun, C. Cortes and C. Burges, "MNIST handwritten digit database", *yann.lecun.com*. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 27- Apr- 2019].

[5] "Nexys Video HDMI Demo", *Digilent*, 2016. [Online]. Available: https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-video-hdmi-demo/start. [Accessed: 27- Apr- 2019].

[6] "CS231n Convolutional Neural Networks for Visual Recognition", *Stanford Computer Science*, 2019. [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 27- Apr- 2019].

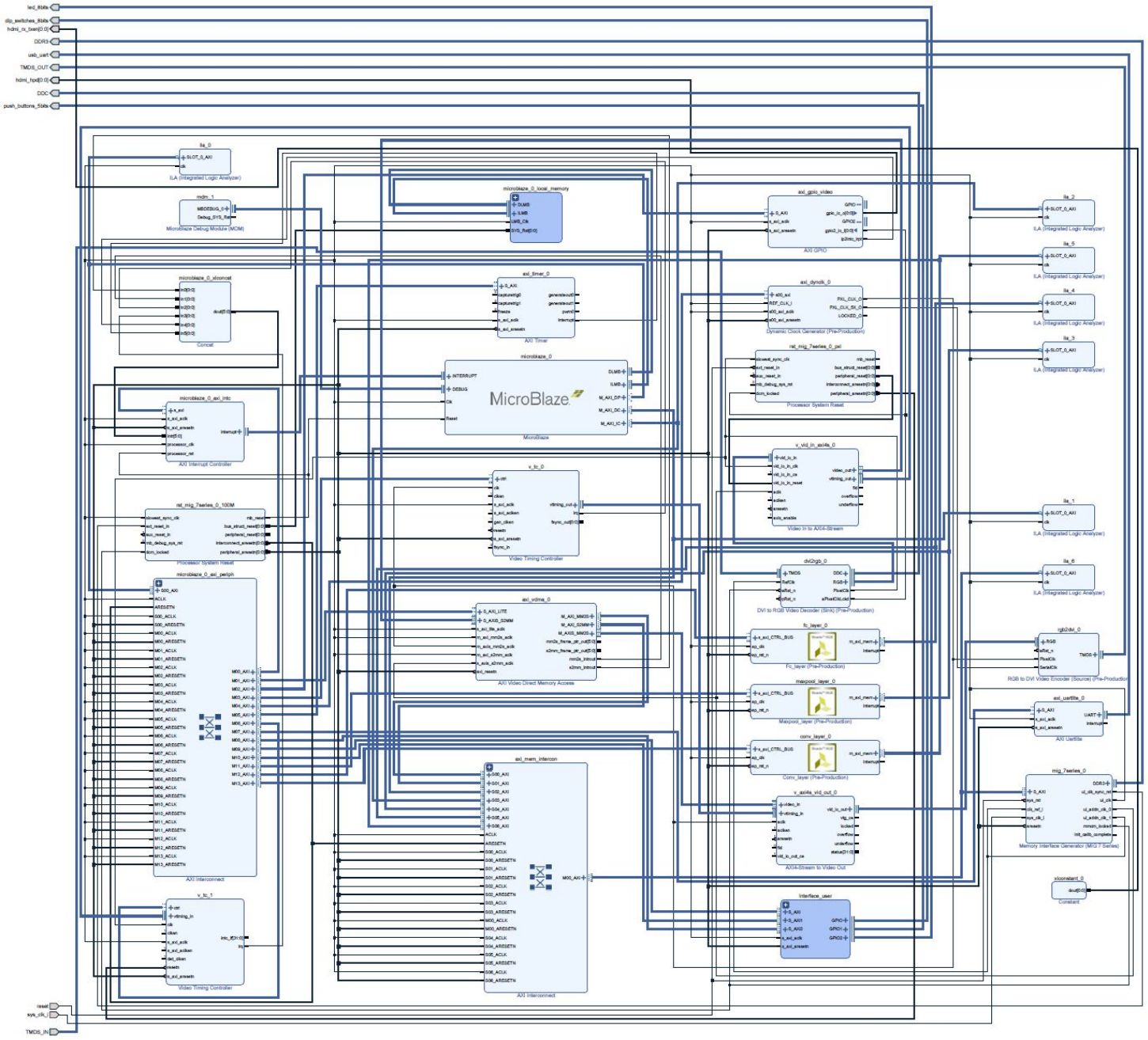
[7] Y. Jia and E. Shelhamer, "Caffe | LeNet MNIST Tutorial", *Caffe.berkeleyvision.org*, 2019. [Online]. Available: http://caffe.berkeleyvision.org/gathered/examples/mnist.html. [Accessed: 27- Apr- 2019].

Appendix: Hints for the Next Time

- Know the final date of submitting the project - this will help establish a timeline and give the team a good idea of how much has been completed versus what remains.
- Have a cutoff date and freeze the hardware design early in the project - setting up hardware handoffs and last-minute changes to hardware can lead to stressful and repeated synthesis/implementation runs.
- Be very careful when changing Vivado versions. This project was entirely developed in 2017.2. The team tried to avoid routing resource errors by trying the new 2018 version, but this resulted in new errors and issues upgrading IPs. Also be considerate when reading online documentation - Some features and IPs can be added or discontinued.
- On Windows, Vivado typically limits the performance to only 4 threads so multi-core computers can only help so much. Try using a more powerful computer with higher CPU clock speeds and more system RAM for faster turnarounds.

Appendix: Schematics and other Details

The following figure provides the block diagram generated by Vivado for the project, complete with Microblaze, HDMI Datapaths, and the CNN layers. A higher resolution PDF is available in the top level of the github directory as *Block Diagram.pdf*.



The following figure provides the AXI addresses and versions for the above IP blocks, captured from the hardware handoff file in the SDK.

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
Interface_user_axi_gpio_LED	0x40010000	0x4001ffff	S_AXI	REGISTER
maxpool_layer_0	0x44a60000	0x44a6ffff	s_axi_CTRL_BUS	REGISTER
mig_7series_0	0x80000000	0x9fffffff	S_AXI	MEMORY
axi_gpio_video	0x40000000	0x4000ffff	S_AXI	REGISTER
v_tc_0	0x44a10000	0x44a1ffff	ctrl	REGISTER
axi_dynclk_0	0x44a20000	0x44a2ffff	s00_axi	REGISTER
Interface_user_axi_gpio_SW	0x40020000	0x4002ffff	S_AXI	REGISTER
fc_layer_0	0x44a50000	0x44a5ffff	s_axi_CTRL_BUS	REGISTER
conv_layer_0	0x44a40000	0x44a4ffff	s_axi_CTRL_BUS	REGISTER
microblaze_0_axi_intc	0x41200000	0x4120ffff	s_axi	REGISTER
axi_vdma_0	0x44a00000	0x44a0ffff	S_AXI_LITE	REGISTER
v_tc_1	0x44a30000	0x44a3ffff	ctrl	REGISTER
axi_uartlite_0	0x40600000	0x4060ffff	S_AXI	REGISTER
microblaze_0_local_memory_dlm...	0x00000000	0x00007fff	SLMB	MEMORY
axi_timer_0	0x41c00000	0x41c0ffff	S_AXI	REGISTER
Interface_user_axi_gpio_Push	0x40030000	0x4003ffff	S_AXI	REGISTER

IP blocks present in the design

axi_vdma_0	axi_vdma	6.3	Registers
microblaze_0_local_memory_dlmb_bram_if_cntlr	lmb_bram_if_cntlr	4.0	
v_tc_0	v_tc	6.1	
microblaze_0_local_memory_ilmb_bram_if_cntlr	lmb_bram_if_cntlr	4.0	
v_tc_1	v_tc	6.1	
mdm_1	mdm	3.2	
microblaze_0_local_memory_lmb_bram	blk_mem_gen	8.3	
v_axi4s_vid_out_0	v_axi4s_vid_out	4.0	
fc_layer_0	fc_layer	1.0	Registers
maxpool_layer_0	maxpool_layer	1.0	Registers
microblaze_0_axi_periph	axi_interconnect	2.1	
rgb2dvi_0	rgb2dvi	1.3	
dvi2rgb_0	dvi2rgb	1.7	
microblaze_0	microblaze	10.0	
microblaze_0_local_memory_dlmb_v10	lmb_v10	3.0	
microblaze_0_local_memory_ilmb_v10	lmb_v10	3.0	
axi_mem_intercon	axi_interconnect	2.1	
v_vid_in_axi4s_0	v_vid_in_axi4s	4.0	
Interface_user_axi_gpio_LED	axi_gpio	2.0	Registers
rst_mig_7series_0_100M	proc_sys_reset	5.0	
Interface_user_axi_gpio_SW	axi_gpio	2.0	Registers
ila_3	ila	6.2	
ila_4	ila	6.2	
microblaze_0_xlconcat	xlconcat	2.1	
axi_uartlite_0	axi_uartlite	2.0	Registers
ila_1	ila	6.2	
microblaze_0_axi_intc	axi_intc	4.1	Registers
ila_2	ila	6.2	
axi_timer_0	axi_timer	2.0	Registers
axi_dynclk_0	axi_dynclk	1.0	
ila_5	ila	6.2	
rst_mig_7series_0_pxl	proc_sys_reset	5.0	
ila_6	ila	6.2	
axi_gpio_video	axi_gpio	2.0	Registers
mig_7series_0	mig_7series	4.0	
conv_layer_0	conv_layer	1.0	Registers
Interface_user_axi_gpio_Push	axi_gpio	2.0	Registers
ila_0	ila	6.2	
xlconstant_0	xlconstant	1.1	