CAPTCHA Beater

1st Mauris Winata
Department of Electrical and Computer Engineering
University of Toronto
mauris.winata@mail.utoronto.ca

2nd Nikhil Deshpande
Department of Electrical and Computer Engineering
University of Toronto
n.deshpande@mail.utoronto.ca

3rd Randy Ewing Chow
Department of Electrical and Computer Engineering
University of Toronto
randy.ewingchow@mail.utoronto.ca

4th Srivatsan Srinivasan
Department of Electrical and Computer Engineering
University of Toronto
srivatsan.srinivasan@mail.utoronto.ca

***Abstract*—Field Programmable Gate Arrays (FPGAs) are reprogrammable devices that allow a user to develop a wide range of circuits all on a single chip. They have a variety of uses and are very useful when trying to prototype application-specific integrated circuits (ASICs) or processors. One emerging application for FPGAs is in the field of machine learning and specifically image classification. Image classification makes use of the convolution operation, which is a series of multiply accumulates. This operation can be easily performed in parallel and made more efficient with deep pipelines, something FPGAs specialize in. To take advantage of this property, a method to solve text-based CAPTCHAs using FPGAs is designed as a term project for ECE1373 dubbed the CAPTCHA Beater. In this paper, we present the method to design and test such a project as well as outline some of the considerations and tradeoffs that are required to complete the project in four months. Overall, the project is a success as it can achieve a prediction accuracy of approximately 88% and can make predictions on the fly in several seconds. Additionally, a graphical user interface is created such that the user can input the CAPTCHA that they want to be predicted by the CAPTCHA Beater.**

I. Introduction

A. Background and Motivation

As a method to protect against automated spam from bots, websites have been implementing the use of CAPTCHAs to verify that a user is human. These tests range from clicking a checkbox to identifying images with a particular object. One particular CAPTCHA test of interest is to accurately interpret an image of a distorted string of text as shown in Figure 1. With the rise in popularity of machine learning, it would be interesting to see if this type of CAPTCHA can be solved using a computer rather than a human. This project aims to devise a system that is capable of cracking such a CAPTCHA through the use of an FPGA programmed specifically using HLS.

B. Applications

At first glance, the potential applications of such a project may only seem to include those that are malicious such as CAPTCHA farms. However, there are legitimate applications such as:

- Helping individuals with vision impairment
- Finding vulnerabilities in CAPTCHA generators
- CAPTCHA avoidance for automated testing purposes

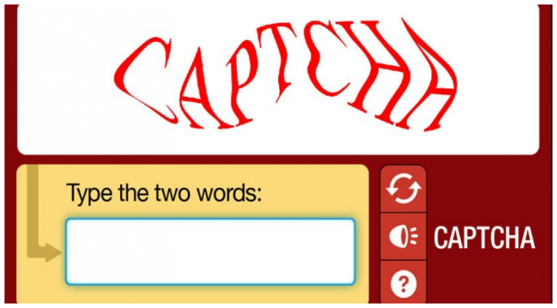

Fig. 1. Simple CAPTCHA

II. Current Status

Through the course of this project, a neural network was designed which had a prediction accuracy of 80% for string-based CAPTCHAS. Using the previous neural network, a hardware processing kernel was developed and implemented within an FPGA. The hardware kernel was able to process an input image (a string-based CAPTCHA) and output a set of characters that are found within the CAPTCHA. Additionally, an external python-based GUI program was developed which allowed users to supply a CAPTCHA image, then this program would write the image to memory and initiate the previously mentioned hardware kernel; the results from the hardware kernel were then read from memory and displayed to the user. To summarize, currently, a CAPTCHA recognition system has been fully developed. This system contains a software component to interact with a user and communicate to the hardware processing kernel, which performs the CAPTCHA recognition.

With the current system, the CAPTCHA image goes through a host PC before being transferred to the FPGA (this was done through the PCIe to the FPGA memory interface). Although this form of data transfer is acceptable, it was a severe bottleneck with the current system flow. The first issue was that the data must be pre-processed before being transferred to the FPGA (storing the image to a bin file) and the second issue was that the image is written to memory and then read from memory before being processed. Reading and writing from memory is expensive in terms of latency and was a contribution to the system bottleneck. Instead of having central storage to communicate between the host PC and the FPGA, it would be significantly faster to stream the data to the FPGA and process them immediately (no memory reads or writes).

To eliminate the previous bottleneck, one future improvement that was suggested was to stream the image data to the FPGA directly using a network interface. An additional IP would need to be added to communicate with the network interface and output the image data. Also, another block would be needed to accept new image data and initiate CAPTCHA processing. By making the previous improvements, the entire data flow would always be streamed, and this would help improve overall performance.

III. Initial Architectural Design

A. Neural Network Architecture

A custom dataset was preferred for the group to be in control of which characters are included in the CAPTCHA and the length of the CAPTCHA. By limiting the length and the different characters that can be found in the CAPTCHA, the scope of the project could be lessened to simplify the required neural network. The design features the ability to distinguish between 19 different characters in a 5 letter CAPTCHA. By limiting the number of characters, the size of the model can be constrained to a more manageable size. This is done to avoid possible future resource issues when implementing the hardware. Constraining the number of possible characters in a CAPTCHA to 5 significantly reduces the complexity of the neural network. This avoids the issue of having to detect the number of characters that are present in a CAPTCHA. An example of one of the custom CAPTCHAs is depicted in Figure 2.

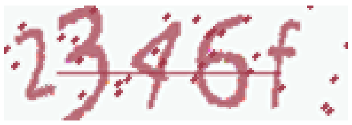

Fig. 2. Example CAPTCHA

Additionally, the neural network was designed to process only grayscale images. The colours featured in text-based CAPTCHAs are usually never relevant to the solution of the CAPTCHA, thus processing in grayscale allows for only a third of the image data to be used, which saves on resources. The initial neural network architecture is depicted in Figure 4.

The initial neural network features a convolution layer, batch normalization layer and max-pooling layer repeated three times, as well as a uniquely dense and output layer for each of the character positions. The purpose of the convolution layers is to extract features from the input image. In theory, each convolution layer will extract a different set of features from

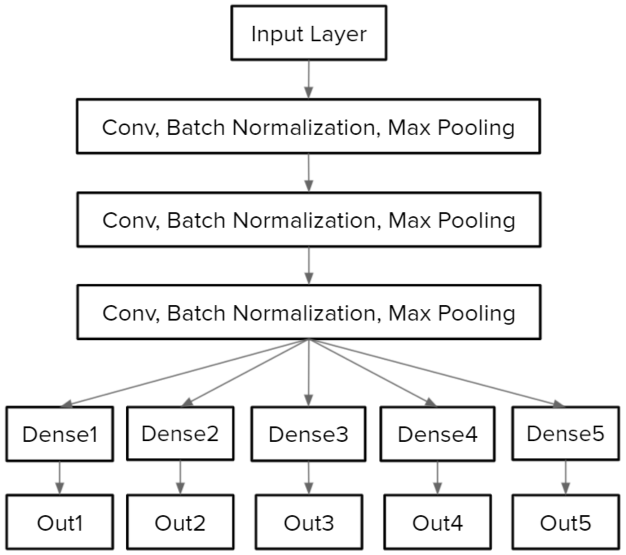
Input Layer
Conv, Batch Normalization, Max Pooling
Conv, Batch Normalization, Max Pooling
Conv, Batch Normalization, Max Pooling
Dense1
Dense2
Dense3
Dense4
Dense5
Out1
Out2
Out3
Out4
Out5

Fig. 3. Initial Software Model

a given input. Batch normalization layers are implemented to standardize the inputs to a network, which should accelerate training [1]. Max pooling layers are used for downsampling the image. The effect of max-pooling layers is that the most prominent features are highlighted and are more effective than average pooling in computer vision applications [2]. Each dense layer is a fully connected layer and is tasked with identifying which character is at a particular position. In each dense layer, there are 19 different neurons, with a neuron corresponding to each character. The dense layer uses a "softmax" activation which calculates the probability of each character being the one shown in the image. The character with the highest probability is taken by the output layer as the character prediction.

The model is trained using the following parameters:

- Optimizer: Adam
- Learning rate: 1e-5
- Batch size: 64
- Number of epochs: 500

The model was also trained using an early stopping callback and a checkpoint callback to stop when the validation loss plateaus and continuously save the best performing version of the model after every epoch. The best-performing model was defined to be the model that obtains the lowest validation loss. The model was trained and tested using a dataset of 75,000 unique samples. The results of this test did not meet the target accuracy of 80%, which led to changes to the model architecture and an increase in the size of the dataset.

B. User Interface

The initial user interface was planned to be a program that would automatically identify the CAPTCHA on the user's screen, process the CAPTCHA, make a prediction and automatically type the prediction into the text box. This would allow for a fully autonomous and seamless process from the user's perspective and would be very convenient to use. This would have been tested with a custom mock website that would generate random CAPTCHAs for the user to solve.

C. Hardware Architecture

The initial hardware design intended was to implement the neural network software model for our own created CAPTCHAs. The original intention was to have each layer of our neural network be its respective hardware with as much optimization and balance between resource utilization of the FPGA and the throughput of each of the layers. For the rest of the components of our hardware design, it was intended to use existing hardware blocks found in Vivado, this included a MicroBlaze softcore processor to control and debug our hardware, a DMA subsystem for PCI Express communication, an AXI DMA core to communicate with our neural network hardware block and the DDR4 MIG to communicate with the DDR4 off-chip memory.

D. System Architecture

The initial system architecture can be seen in Figure 4 below, which comprises all previous sections mentioned in one final design. As per Figure 4, a Python program captures the CAPTCHA from the requesting web page and sends this CAPTCHA to the FPGA through PCIe. This data will be stored on the DDR4 off-chip memory and will be sent to our neural network core through AXI DMA by the MicroBlaze processor using bare-metal coding. The MicroBlaze will receive the result from the neural network core and send this data back to the PC through PCIe in which the Python program will input the CAPTCHA result in the corresponding text box.

IV. Specification Evolution and Final Architectural Design

A. Specification Evolution

The initial project specifications are listed below:

- Achieve a CAPTCHA prediction accuracy of at least 80%
- Perform CAPTCHA prediction within 30 seconds
- Users should be able to send CAPTCHAS to a central server which would have an FPGA connected to it (Users do not need to have an FPGA connected to their local systems)
- The hardware design must be implemented on the ADM-PCIE-8V3 FPGA board
  - ∗ The design must be within the resource limits (BRAM, LUTs, FFs) of the Xilinx Virtex Ultrascale FPGA (XCVU095-2-FFVC1517)

Most of the project specifications above were quite lenient and as a result, most of the specifications were achieved. During the final stages of the project a client and server program was developed to allow users working on different PCs to send prediction requests to the central machine connected to the FPGA. Initial testing of the software indicated that the FPGA machine was within its private network. To properly verify the client/server programs, the additional infrastructure needed to be set up (Mininet) to properly imitate that standard use case. Due to limited time, the client/server programs were not tested, so the specification which said that the user did not need to have an FPGA device connected to their PC was modified to the user needing an FPGA connected to their PC.

The final project specifications are listed below:

- Achieve a CAPTCHA prediction accuracy of at least 80%
- Perform CAPTCHA prediction within 30 seconds
- Users should interface with a GUI application which would send CAPTCHA images to the FPGA that is connected to the user machine and then display the prediction results to the user
- The hardware design must be implemented on the ADM-PCIE-8V3 FPGA board
  - ∗ The design must be within the resource limits (BRAM, LUTs, FFs) of the Xilinx Virtex Ultrascale FPGA (XCVU095-2-FFVC1517)

B. Neural Network Architecture

Two main changes were made to the initial software model architecture - removal of the softmax activation and the addition of a second dense layer. Adding a second dense layer allows for a much better performing neural network model. After this addition and training on the same dataset as the initial model, the accuracy improves significantly from approximately 40% to 55%. Removing the softmax layer allows for simplification of the hardware. Softmax activation involves the use of the exponential function which can be resource-intensive when not designed as a lookup table. After investigating the use of the softmax activation function further, it was found that this function is used to determine the likelihood (in percentage) of each of the possible classifications. As the goal of the neural network model is to make a prediction rather than calculate the probabilities, it was decided that the softmax activation function is redundant as in general, the largest input to the softmax function corresponds to the character with the largest probability. The final neural network model is depicted in Figure 5.

C. User Interface

The main difference between the final user interface and the initially planned user interface is that the final version requires the user to identify where the CAPTCHA is on the screen and manually type in the prediction. The user is responsible for snipping an image or inserting an image for a file that contains the CAPTCHA of interest. Then the user adjusts the sizing and positioning of the image such that the CAPTCHA is encompassed by a dotted rectangle. Then the user clicks predict and the prediction is shown at the bottom of the GUI. This adjustment was made to simplify the design for the user interface and allow for greater time allocation into the hardware design. The final user interface can be seen in Figure 6.

D. Hardware Architecture

Figure 7 below shows the final block diagram of the CAPTCHA prediction neural network hardware kernel. The

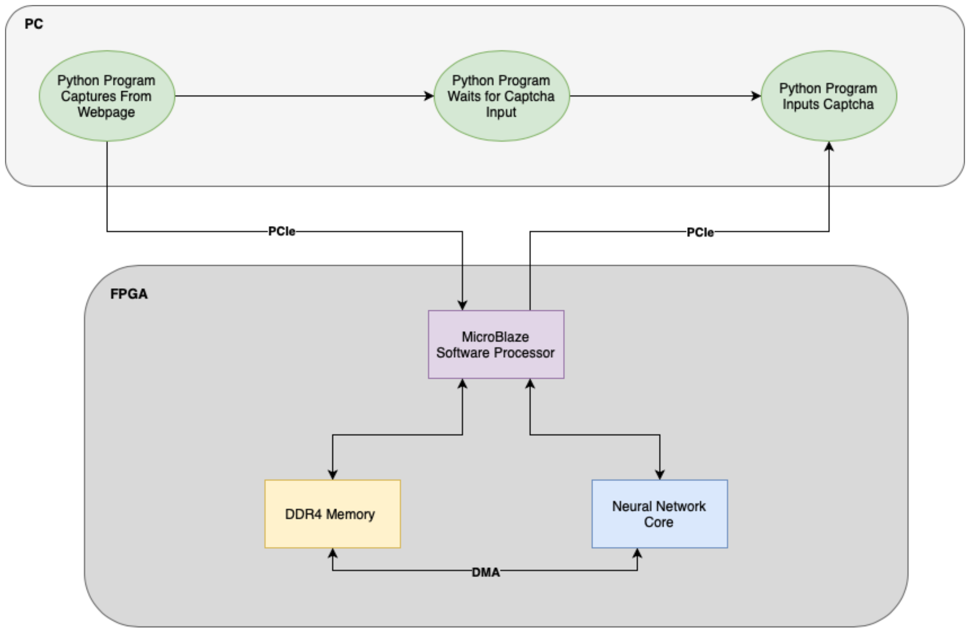
PC
Python Program Captures From Webpage
Python Program Waits for Captcha Input
Python Program Inputs Captcha
PCIe
PCIe
FPGA
MicroBlaze Software Processor
DDR4 Memory
Neural Network Core
DMA

Fig. 4. Initial System Architecture

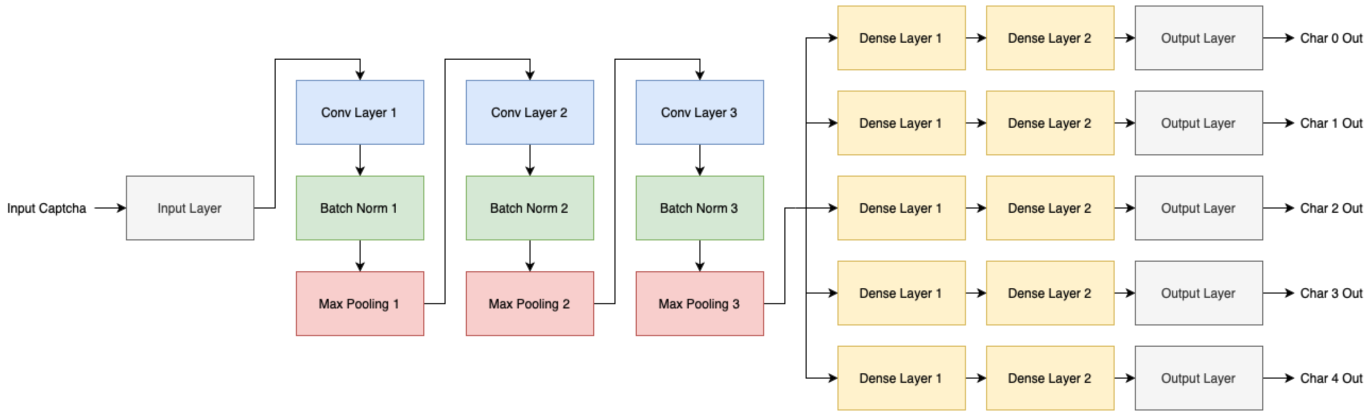

Fig. 5. Final Neural Network Architecture

first block in the design was the input layer. Once the CAPTCHA image was written to memory, the input layer read the image data from memory and converted the data from floating-point to fixed-point precision. The input layer then sent the converted image data to a convolution layer block, which convolved the image data with several filters. The convolution layer outputs were then fed to the batch normalization layer block, and this block normalized the input data through mathematical transformations such that the overall mean was 0 and the standard deviation was 1. The batch normalization output was then sent to the max pool layer block and this block downscaled the input data by dividing the input into groups of “x” elements and replacing them with the maximum element within that group. So, the max pool layer block reduced the input data size by a factor of “x”. The max pool layer was then followed by another two sets of convolution layers followed by a batch normalization layer which was followed by a max pool layer.

The final max pool layer output was then sent to a broadcast block, which fed the max pool layer outputs to 5 dense layer blocks. The dense layer performed a dot product on the input data with a 2-D matrix of weights. The dense layer outputs were sent to another set of dense layer blocks. The final dense layer outputs were sent to an output layer block, which found the largest element from the previous dense layer and determined what the character representation of that was. The 5 output layers then sent their single character results to an output memory block, which arranged the characters in order and wrote the results to the memory.

The input layer and output memory block interfaced to the memory using the AXI interface and all the blocks within the hardware kernel were connected using the AXI-stream interface. Finally, the host machine communicated to the FPGA memory using a PCIe to DMA interface.

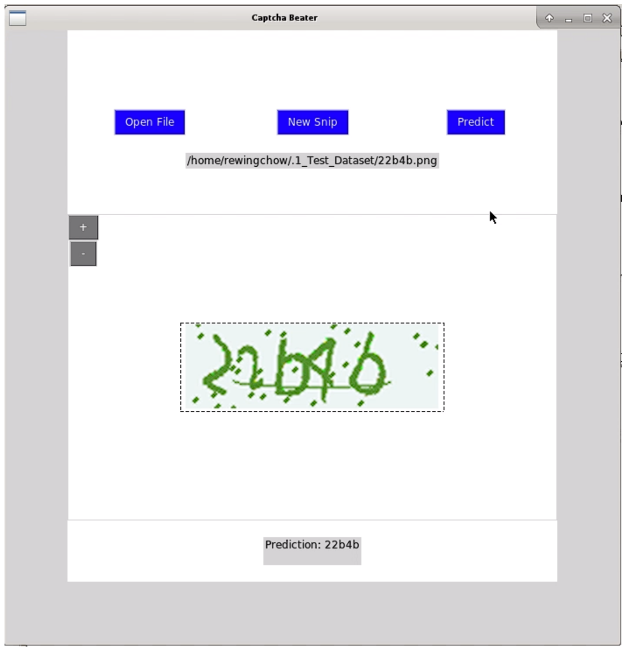

Fig. 6. Final User Interface

E. System Architecture

The final system architecture can be found in Figure 8 below and the full system block diagram can be found in Figure 9. The main difference between the initial and final system architecture (Figures 4 and 9) was the method of communication between the host PC and the FPGA. Instead of using a MicroBlaze, a design decision was made to use the PCIe to DMA interface, shown as the xDMA block in Figure 9. There were three main reasons why the original system architecture was modified. The first reason was that the infrastructure for the PCIe to DMA interface was already set up in assignment 2, whereas additional work would have been needed to set up the MicroBlaze. The second reason was due to the size of the CAPTCHA processing kernel, which took a large portion of the FPGA resources and adding a MicroBlaze would have exceeded the resource constraints. Finally, the third reason was due to the speed limitations of the MicroBlaze, which was a slower form of data transfer than going directly from PCIe to memory.

V. Methodology

A. Designing the Neural Network Model

The first step in creating a software model is to develop a basic model using already developed machine learning libraries in Python that can accurately predict CAPTCHAs that include a small subset of characters. This simplified the design process as the focus can now be placed on the overall architecture rather than understanding how the layers work and interface with each other. Next, a custom dataset that is consistent and contains many samples to avoid underfitting is constructed. Again, Python libraries are leveraged to make this process much simpler and quicker.

Training the machine learning model is performed using the custom dataset. Initially, the model is trained using 75,000 images, but accuracy is increased by including more training images. The final model was trained using more than 200,000 images. In general, the greater the number of training images, the greater the accuracy. However, there is a point of diminishing returns with the number of training samples where it is not time-efficient to train the model further.

After the training is completed and the model achieves the desired accuracy, the model is then re-written from scratch in C for HLS. This is performed by researching how the individual layers (convolution, batch normalization, max pooling and dense) function internally. The logic is then mapped to C-code. Additionally, the weights are extracted from the Python model and saved as floats for use in the C software model. To more easily collaborate between multiple people when writing the C code for the neural network, the model is split up into different C files corresponding to each layer rather than having one large C file to represent the whole model.

The C model layers are written such that they read/write their inputs/outputs from/to text files. This allows each layer to operate more independently and allows the group to test a specific layer without needing the previous one by using a ”golden output” extracted from the Python model. As a result, the duties of writing the neural net layers could be split between three people - one writing the IO layers (input/output), one writing the first half (convolution, batch normalization and max pooling) and one writing the latter half (dense layers). As a method to verify the correctness of the C software model, a test bench in C is written to test each of the layers as well as the model as a whole. Initially, each layer's output is inspected manually for correctness using a single input image and the layers are corrected if necessary. Once confident regarding the model's performance using floating-point numbers, the C-code is then modified to use fixed-point numbers, which should allow for more efficient hardware. Again, the layers are tested manually for correctness. Once each layer is verified, a more rigorous testing method is implemented which tests the whole system on 1000 unseen test images. This test involves providing the model a text file input that represents a grayscale version of the image, reading the predicted output and comparing it to the correct output.

B. Designing the Hardware

Once the neural network model had been completed we had partitioned the initial hardware design in which each layer need to be implemented in Vivado HLS with CSIM, Synthesis, COSIM and RTL export all passing without any issues. Three out of the four of us were allocated 2-3 layers of the neural network model to optimize while ensuring all stages of the HLS process pass with the correct AXI interfaces for each layer. In doing so, we had created a Google spreadsheet to track our resource usage and throughput at varying clock periods, this allowed for us to determine where the bottleneck was and what was using up the most resources.

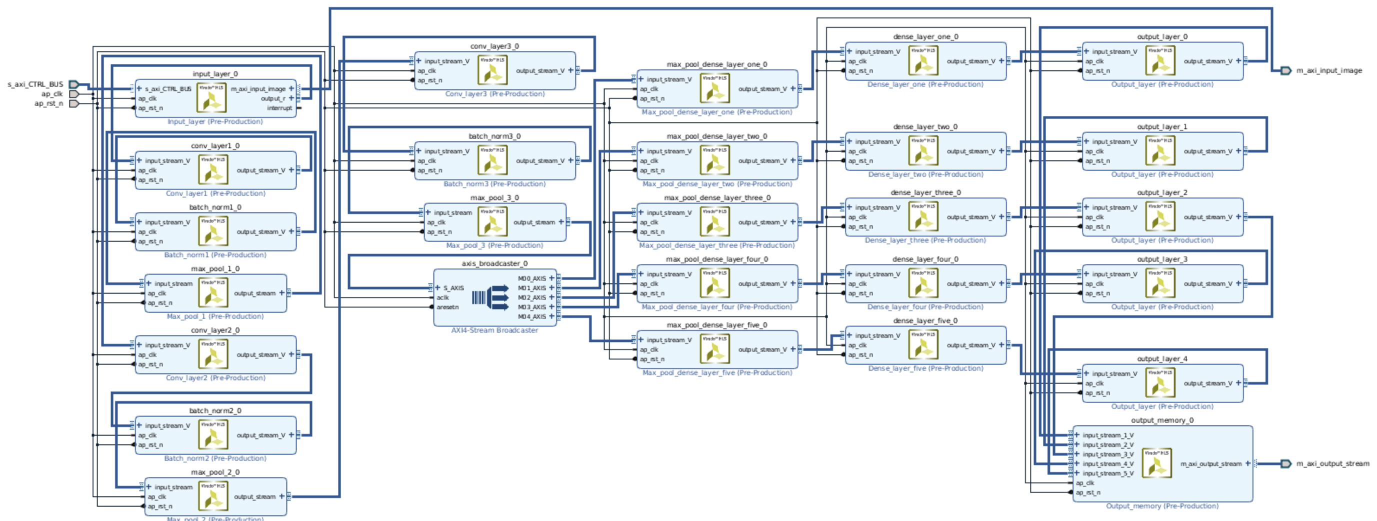
Fig. 7. Final Block diagram of the CAPTCHA Prediction Neural Network Hardware Kernel

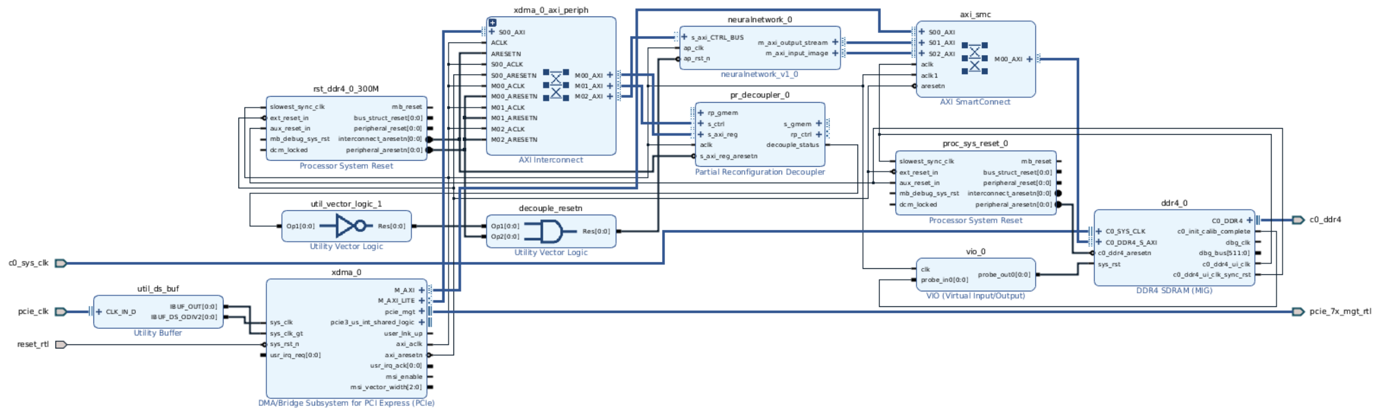
Fig. 8. Final Full System Block Diagram of the CAPTCHA Prediction Hardware

Once all layers were optimized and fully tested in Vivado HLS using very similar test benches from section A, all layers were exported and added to a Vivado project. This Vivado project was worked on by one individual on the Agent8 system with all other team members assisting. The Vivado Project was leveraged from Assignment 2, in which the partial region was replaced by our neural network layers all utilizing the AXI Stream interface. With our design consisting of many layers, we used a predetermined 7500-pixel image in a 32-bit floating-point to test that all the layers were working correctly together. Using initially VIOs and an ILA, we tested our design layer by layer in which we check the output from the ILA with our predetermined values. Once the full neural network design had been working, we moved onto the next stage of integration which was to have our input layer read from memory and output layer write to memory. This was tested and validated through the creation of two new kernels, one which wrote to memory, which the input layer read from and one which read from memory, which the output layer wrote to. Once this was confirmed to be working, we moved onto the next stage of the full system integration with our user interface program designed in Python and xDMA communication through PCIe.

C. Designing the User Interface

The user interface can be thought of as having two parts: the image capture GUI written in Python and the memory interface which is written in C. These two parts are designed such that the interface only through text files, which means that they are fully modular and can be tested and written completely separately. The Python GUI is created through the use of Tkinter, which is a Python GUI package. The use of this package allows the group to easily create a functional program that allows a user to input an image either through a file upload or a screen snip. The user can then adjust the image as necessary to place the CAPTCHA in the middle of a dotted rectangle. The memory interface is designed such that it takes an input grayscale image from a text file, saves it to a binary file and calls a shell script to write the values of the binary file to memory. The C code has a preset delay before

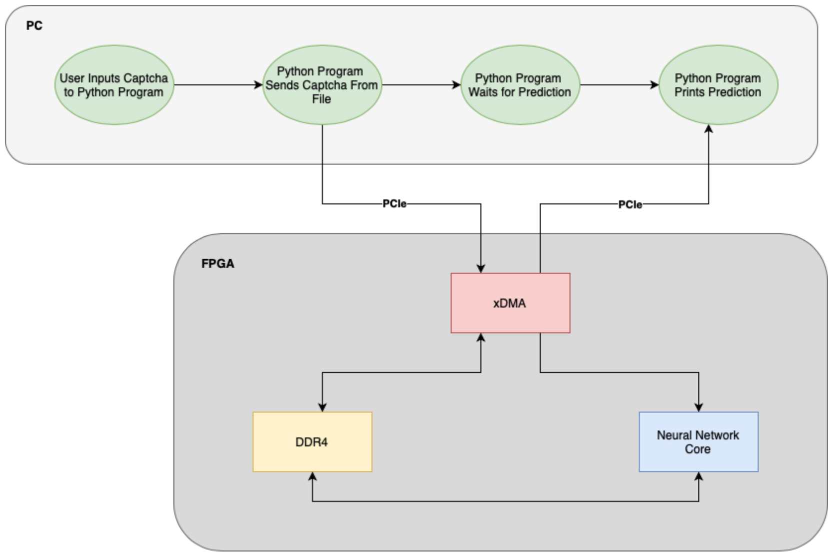
PC
User Inputs Captcha to Python Program
Python Program Sends Captcha From File
Python Program Waits for Prediction
Python Program Prints Prediction
PCIe
PCIe
FPGA
xDMA
DDR4
Neural Network Core

Fig. 9. Final System Architecture

calling a secondary shell script which reads the prediction from memory and saves it to a binary file. This binary file is then read by the memory interface C code and saved into a text file and read by the Python GUI which prints it on the bottom of the user interface.

D. Finalizing the Design

With both our hardware ready and Python program ready, our next steps were to integrate the two. First using the xDMA driver for PCIe communication and example scripts, we were able to successfully write and read from the off-chip DDR4 memory. With a small example test, we were able to get our binary image written to the same location on memory from which our input layer is reading. Using a VIO and ILA confirmed that we are successfully writing our image binary to the memory correctly and our input layer is reading correctly and our output layers have the correct values. With some troubleshooting, we did the same for the output layers but needed to create a kernel to merge all five outputs and to output all 5 characters to an offset address of 0x0. With these changes made, we were able to successfully read and write to memory with our most of our design working. The last step was to call these scripts and manipulate the data for our python program to spit out the predicted CAPTCHA.

E. Organization

As a method to organize the files required for all parts of the project, GitHub and Google Drive are utilized as version control methods. GitHub is used primarily for the code required for the software model, software test bench, golden outputs, weights and input data, while Google Drive is used for storing the presentations, resources and other miscellaneous items.

VI. Contributions

A. Mauris

Implemented Python-based software model. Wrote and tested convolution, batch normalization and max-pooling layers in C. Contributed to the creation of the software model test bench. Wrote user interface in Python/C. Provided advice regarding hardware/HLS design. Equal contribution in presentations and report write-up.

B. Nikhil

Assisted with preliminary software model coding. Modifying and optimizing input, max pool and output layers in HLS design. Assisted with debugging PCIe problems. Equal contribution in presentations and report write-up.

C. Randy

Assisted with the initial system architecture model. Optimized and exported convolution and batch normalization layers as hardware blocks within Vivado HLS. Worked on system integration and final testing of our full design within Vivado using VIOs and an ILA alongside our software user interface. Equal contribution in presentations and report write-up.

D. Srivatsan

Developed python script to generate the CAPTCHAS for training the neural network model. Wrote and tested the two types of dense layers in C. Helped develop the test-bench framework to verify all layers of the neural network. Converted the two C-based dense layer models to HLS friendly code and also optimized them. Assisted with full system integration. Equal contribution in presentations and report write-up.

TABLE I
FPGA Resource Utilization

Resource	Utilization
LUT	14%
LUTRAM	11%
FF	8%
BRAM	57%
DSP	9%
IO	23%
GT	20%
BUFG	1%
MMCM	6%
PLL	9%
PCIe	25%

TABLE II
Neural Network Resource Utilization Breakdown by Component

Neural Network Components	LUT	FF	BRAM	DSP
Input Layer	816	872	1.5	3
Convolution Layer 1	431	598	8	2
Convolution Layer 2	439	613	37	3
Convolution Layer 3	393	576	25.5	2
Batch Normalization Layer 1	865	1149	1	8
Batch Normalization Layer 2	853	1059	1	3
Batch Normalization Layer 3	860	1061	1	3
Max Pooling Layer 1	1091	1228	2	0
Max Pooling Layer 2	1031	1219	2	0
Max Pooling Layer 3	1027	1219	2	0
Dense Layer 1 - 1	490	540	161	5
Dense Layer 1 - 2	490	540	161	5
Dense Layer 1 - 3	490	540	161	5
Dense Layer 1 - 4	488	537	153	5
Dense Layer 1 - 5	490	540	161	5
Dense Layer 2 - 1	269	470	2.5	4
Dense Layer 2 - 2	265	418	2.5	4
Dense Layer 2 - 3	265	418	2.5	4
Dense Layer 2 - 4	265	418	2.5	4
Dense Layer 2 - 5	265	418	2.5	4
Output Layer 1	693	767	0	0
Output Layer 2	693	767	0	0
Output Layer 3	693	767	0	0
Output Layer 4	693	767	0	0
Output Layer 5	693	767	0	0
Output Memory	732	799	0.5	0

VII. Design Characteristics

A. Resource Utilization

Table 1 above shows the resource utilization for the whole design post-implementation. Due to the storing of weights and using some caching techniques, a lot of the FGPA's on-chip BRAM have been used. From Table 2 below, a post-synthesis breakdown estimation of how each component uses the resources of the FPGA. It can be seen that the first dense layer which has 242,649 32-bit weights uses up the most BRAM in the neural network hardware.

B. Time Allocation

Throughout the project, the majority of the time spent was on generating the bitstream as it took around 30-50 minutes to generate one to test. Along with this, a lot of time was spent on getting PCIe communication between the host PC and FPGA to work, also getting the output layer to write to memory correctly. Table 3 above gives a breakdown of each section of development in our project starting on February 1st and ending on May 28th. Though it took 6 weeks to convert our Python model to a C model, a lot of our team was bottle-necked by other means where very little focus was on the project during this time.

TABLE III
Project Timeline

Project Phase	Estimated Start	Estimated End	Estimated Duration
Planning/Proposal	Feb 1st	Feb 26th	4 Weeks
Python Model	Feb 22nd	Mar 5th	2 Weeks
C Model	Mar 8th	April 16th	6 Weeks
HLS Implementation	April 19th	April 23rd	1 Week
System Integration	April 26th	May 14th	3 Weeks
Report & Github	May 17th	May 28th	2 Weeks

VIII. Problems

A. Neural Network Model

The main problem faced when designing the software model was understanding exactly how each layer functions. This involved a lot of research as well as trial and error to construct functionally correct layers in C. Additionally, none of the group members have a wealth of experience when it comes to neural networks, so devising an efficient architecture to tackle the issue was a challenge that required lots of experimentation and research. Finally, locating an appropriate dataset to train the neural network was a challenge as there were none available with the number of samples that would have been required to train the model. As such, the group decided that the best course of action would be to create our own for this project.

B. HLS

When the C code was exported into HLS, it was observed that certain layers took up a lot of BRAM. For example, the design has five dense layers that, at first, required the storage of all the incoming input to do the dot product. This put a huge burden on the available BRAM as this would require the storage of multiple copies of the input. To resolve this issue, the dense layer was restructured to make it a streaming output such that whenever an input was received, it would do the operation for that specific input and write it to an output. This pipelined dense layer helped save a lot of BRAM. Similar fixes were applied to the batch normalization and max pool layers as well further reducing BRAM usage.

C. Hardware Integration

After synthesizing the layers in HLS, the next step was to test the neural network (NN) on the FPGA. At first, the decision was to use the setup from assignment 2 and add the NN layers in the partial reconfiguration (PR) region. Two problems were encountered when doing so; firstly, the design was too big to fit in the PR region and secondly, the PCIe interface was not functional. To resolve the first problem, a new clean project for our NN was created so, we have enough space to implement our design on the FPGA. To get around the second issue, the input was hard-coded in BRAM and start signal triggering was achieved using a VIO.

During hardware testing, there was another issue with meeting timing for some of the layers. The layers were synthesized in HLS for the correct clock period but, when implemented on the FPGA, they failed to meet timing due to their placement and routing. Since the RTL was not written manually, it was not obvious which path was causing the timing error. This issue was resolved by synthesizing the design in HLS for a lower clock period than what was needed to ensure that timing is met.

The last remaining problem left, before project completion, was interfacing with memory through PCIe. At first, the code from assignment 2 was modified, but PCIe was not functional. So, the decision was made to reinstall the drivers for PCIe and to use the supplied testing shell script files to check if read/write to memory was working as intended. Through a lot of troubleshooting, the PCIe was successfully able to read and write through software. Next, the input layer was configured to read the correct data from the correct memory location. Once this was accomplished, the write operation for our output layer was resolved and, thus, the memory had successfully integrated with the NN. Having a system set up with proper memory interfacing for the projects would have been very useful and would save a lot of time for future students.

One issue encountered while testing, that could not be resolved, was the length of time required for bitstream generation. Throughout the project, over 30 bitstreams were generated and each took at least 30 minutes. The main issue was that the team did not have much to do during the bitstream generation as these were the final stages of the project. Consequently, a lot of time was being wasted which resulted in postponement of the completion date. One possible solution might have been to investigate Partial Reconfiguration, where testing is done in the PR region and approved changes were committed to the fixed region after ensuring the correct functionality. This would save time for placing and routing IPs that are known to work and have not changed in between iterations.

Another issue encountered was that after changing the connections on the Smart Connect module in Vivado, sometimes, the bitstream generation would just hang. It would be stuck in, most commonly, the "route design" phase for very long. The only way around this was to restart the bitstream generation which would eventually generate the bitstream. There were no errors or any alerts that would inform on what was going wrong. This bug also resulted in a lot of time being wasted. Perhaps newer versions of Vivado have patched this, but, in Vivado 2017.2, this issue is persistent.

IX. Thoughts, Conclusions and Recommendations

A. Interesting Aspects

During the design, HLS was used in unique ways to boost the speed of the design process. One interesting thing in the NN layers was the optimization of the dense layers. When our C code was imported into HLS, it was found that the synthesis used a lot of BRAM because it would store all the incoming data and do single matrix multiplication and addition after it had received all the data (dense operation). The layer was modified such that the input stream would get processed immediately as it is received; this reduced our latency and resource usage significantly. A similar modification was applied to the max pool layers. The expectation was that HLS would have synthesized RTL such that it would wait for the appropriate input and immediately do the operation when it is received. Instead, HLS just took whatever was in the FIFO and used that as its data. To fix and optimize this, two buffer arrays were created such that when the first is filled, the max pool operation is executed on all data in that row while the incoming data is buffered into the second buffer. The incoming data would alternate between the two buffers which allowed for a high level of parallelism and boosted the performance of the max pool layer as well as save a lot of resources.

B. Future Work

There are two key approaches to extend this project: more characters and more styles. This project used a total of 19 characters so, it is possible to add all alpha-numeric characters from the ASCII table to the NN model through increasing the training dataset. The second way is to add more styles of CAPTCHA; we created our own dataset as we did not have access to a good dataset for the purposes of this project. If possible, many websites' CAPTCHA generation algorithms can be used to create varying styles of CAPTCHAs to make the NN more generalized and increasing it application range.

C. Thoughts

Throughout the project, HLS was used frequently, and its advantages and limitations were discovered. HLS allowed the testing of multiple optimizations of the NN design by simply changing some pragmas which would have been unrealistic in the case of writing RTL. Additionally, these pragmas handled all the AXI interfaces, and the ability to target multiple clock periods via the synthesis settings. HLS also synthesizes optimal RTL directly from C code which had two major implications. The first is that a few lines of C code could be changed easily and have a completely different design in seconds, due to its higher level of abstraction, which increased our design space. The second is the advantage of skipping an entire step in the typical design process; there would have been a need to write C code to get an understanding of how the NN layers work but, fortunately HLS could directly convert this C code into RTL. Testing was also made easier as the C testbenches were able to seamlessly integrate into HLS for RTL Co-simulation. This saved a lot of time as traditionally, the generation of multiple bitstreams to test the RTL would be required. The main disadvantage of all this was that the RTL became a "black box", and changing the RTL was not easy. For example, it was not easy to just add more registers in the critical path, for deeper pipelining, because it could affect other paths in the design that are unknown. Overall, HLS was a fantastic tool because of its ability to boost productivity and provide a wide range of designs with minimal coding. If the project were done in Verilog, it would have taken, at least, another 4 months to reach completion.

The course was very interesting, but there are some things that could further improve the experience for future students. Looking back, for the project, the creation of sample input and output blocks in Vivado and having them communicate with memory via PCIe using hard-coded data would have saved some time. A great deal was learned about HLS and how to optimize processes using HLS and integrate them with Vivado. In terms of system design, the experience on how to take an idea and implement it on an FPGA along with an application to create a full system was invaluable. It was found that testing takes a lot longer than expected and the different types of challenges that are faced when testing the different components of a system. Lectures and assignments were good, and they were delivered very well. To further improve the course, it would be very useful if the lectures had a stronger focus on HLS, like in the first lecture, rather than FPGA architecture. Also, a couple of lab sessions would be extremely valuable, where the TAs guides the students on how to set up the system, especially for assignment 2 as many students had difficulty in getting it to work. The assignments were predominantly optimization focused; it would be nice to integrate other aspects such as, coding, Vivado integration and memory interfacing into them as extensions. More support from the TAs for the final project would have been a big help, like a 1-2 hours meeting every week where troubleshooting and progress updates could be shared. Overall, the course was enjoyable, and the team would like to thank professor Chow and the teaching staff for providing us with this fantastic learning experience.

References

[1] J. Brownlee, "A Gentle Introduction to Batch Normalization for Deep Neural Networks," Machine Learning Mastery, 03-Dec-2019. [Online]. Available: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/. [Accessed: 26-May-2021].

[2] J. Brownlee, "A Gentle Introduction to Pooling Layers for Convolutional Neural Networks," Machine Learning Mastery, 05-Jul-2019. [Online]. Available: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/: :text=Maximum%20pooling%2C%20or%20max%20pooling,the%20case%20of%20average%20pooling. [Accessed: 26-May-2021].

Appendix

D. Setting up PCIe to DMA Interface

- Hardware
  - ∗ Used Vivado 2017.1 to generate the DMA/Bridge Subsystems for PCI Express IP core
  - ∗ Essentially re-used assignment 2 project (2017.1 version) and added the CAPTCHA processing IP
- Software
  - ∗ First cloned the latest DMA drivers from Xilinx: https://github.com/Xilinx/dma_ip_drivers.git
  - ∗ The driver used for the project could be found under the "XDMA" folder
  - ∗ The driver code was built and installed
  - ∗ After restarting the machine, a script file within the "XDMA" folder called "load_driver.sh" was used to load the driver
  - ∗ As mentioned within the XDMA driver instructions, the program under the "tools" directory was also built
  - ∗ Data was sent to the memory by modifying and using the script file named "dma_memory_mapped.sh"
  - ∗ Please note that the data which was being written to memory and read from memory needed to be stored in binary files (.bin) when using the "dma_memory_mapped.sh" script file