Identity Checker on FPGA

Author: Junye Chen, Sheng-Hao Huang, Andy Shen, Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—**This paper presents a hardware implementation of facial detection, as part of a facial recognition system. We want to explore the advantages of running detection on an FPGA rather than running it purely in software. Our proposed solution is to use Vivado HLS to synthesize C code onto a Xilinx Kintex-7 FPGA.

***Index Terms*—**Facial Detection, Facial Recognition, FPGA

I. Introduction

Facial recognition has become very popular in the field of computer vision and machine learning because of its wide range of applications. It can be used to check attendance in classrooms or in the workplace, and can improve security measures in critical areas. Facial recognition can be divided into two steps: detection and identification. Two of the most common methods for performing the detection step are the Viola-Jones algorithm and neural networks. We have chosen Viola-Jones algorithm in the facial detection step because of its simplicity and parallelism, which offer significant speedup potential on FPGA. One common method for performing the identification step is the Eigenface algorithm, which uses statistical methods to approximate face features. Our goals in this project are: (1) reach a detection accuracy of 80% for frontal faces, (2) reach an identification accuracy of 80% for frontal faces, and (3) reach a 5x speedup in our FPGA facial detection system over a system completely implemented in software. We acknowledge that I/O may be a bottleneck in our system, so we will measure speedup without I/O.

This report has the following structure: Section II covers what requirements our design has to meet. Section III describes our system architecture. Section IV elaborates on trade-offs that we considered before arriving at our final design. Section V elaborates on the subsystems that make up the system architecture described in Section III. Section VI covers results from our project. Section VII covers project management. Section VIII lists related work.

II. Design Requirements

Our project needs to meet the following requirements in both accuracy and speedup.

A. 80% Accuracy on Frontal Face Detection

When taking frontal view pictures of people's faces, our system should be able to detect the face if there is one, or reject the image if there isn't any, at least 80% of the time. We will determine this visually.

B. 80% Accuracy on Frontal Face Identification

During testing, we will get people to take frontal view pictures in front of the camera and add their faces to a database of faces. Assuming a person's face is already in the database, our system should be able to recognize the face and display the correct name at least 80% of the time.

C. 5x Speedup in Hardware

We will measure the total time to recognize one face using our system in both software and hardware. In software, we will use the linux time command or the timing functions in standard C. In hardware, we will measure the number of clock cycles at our operating frequency. Our timing only measures the actual time of the algorithm. Our goal is to achieve at least 5x speedup on FPGA. We acknowledge that the I/O transfer to our FPGA is a bottleneck of our system, so we will measure the speedup without I/O.

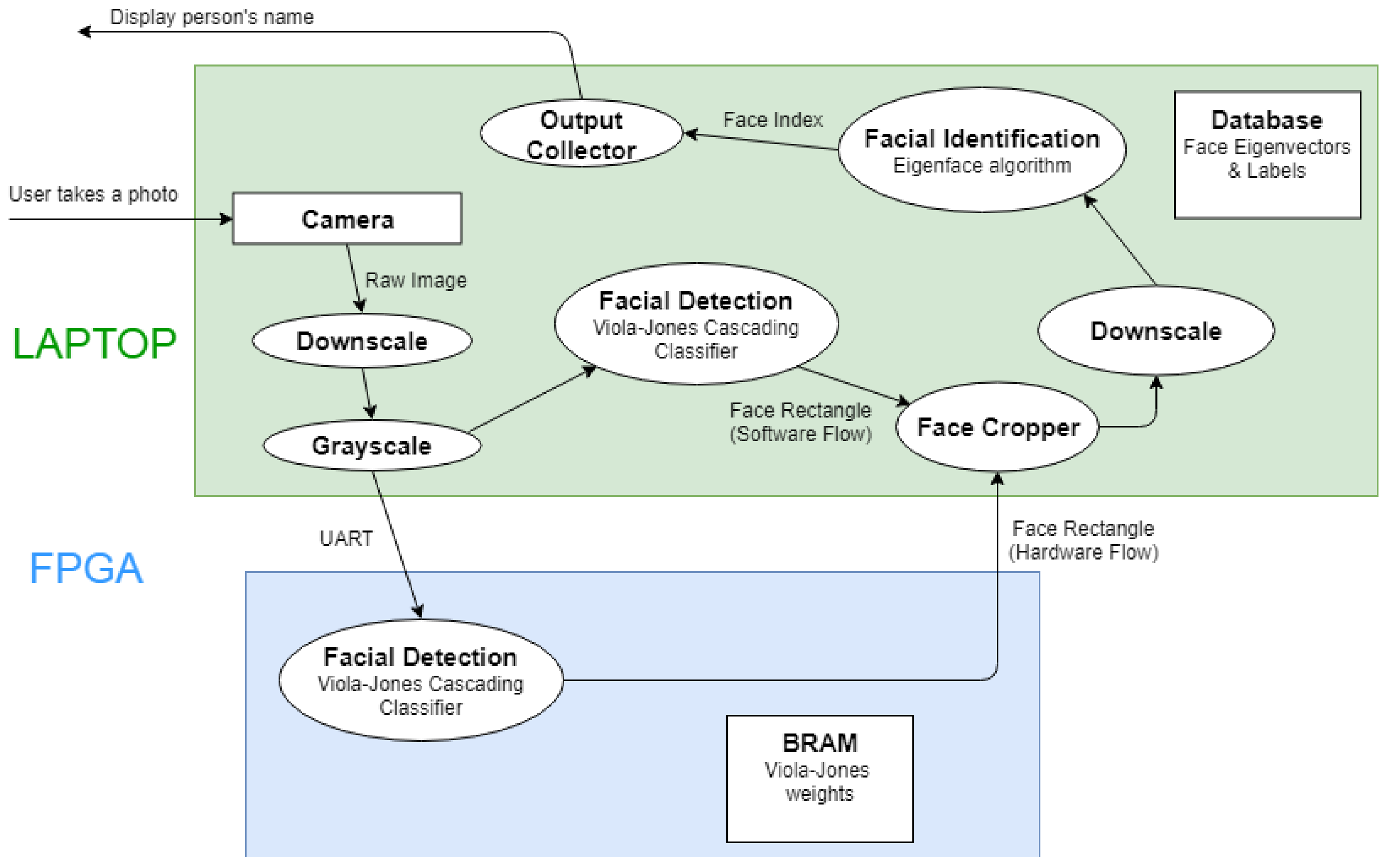

Figure 1: Block Diagram

III. Architecture

Our system is mainly divided into three components: a laptop (software), a FPGA (hardware), and a UART channel to transfer data between our laptop and our FPGA. Implementation details are discussed in Section V.

A. *Laptop*

The laptop captures camera input and downscales and grayscales the raw image. In the software flow, the downscaled and grayscaled image is passed into the software implementation of the Viola-Jones cascading classifier. The output of this classifier is a face rectangle. In the hardware flow, the downscaled and grayscaled image is sent via UART to the FPGA, to the hardware implementation of the Viola-Jones cascading classifier. Similar to the software implementation, this hardware implementation also outputs a face rectangle, which is sent to the laptop via UART. The two flows converge at the face cropper which takes the face rectangle in both flows and crops the face out of the image. The cropped face is then downscaled and sent to the facial identification module running the Eigenface algorithm. This module outputs the index of the face in the face database that is most similar to the face inputted to it. The index is used to grab the corresponding name in the face database.

B. *FPGA*

The FPGA receives the downscaled and grayscaled camera image from the laptop via UART. It contains a hardware implementation of the Viola-Jones cascading classifier, using BRAM to store the classifier weights.

C. *UART*

The UART channel transfers bits at a specified baud rate, 921600 bits/second in this case.

IV. Design Trade Studies

A. *Choosing an FPGA*

We mainly considered the Nexys Video Artix-7 FPGA Board before we had access to the course inventory. This was the board with the most logic cells and Configurable Logic Blocks that we could find within our budget. Once we were given access to the course inventory, we realized that this board couldn't compare to the Kintex-7 KC705 board or the Virtex-7 VC707 board in the inventory, as these two boards had specs that reflected costs of 3x-6x our budget of $600. We put in a request for either board and ended up getting the KC705 board, so we ended up using that for this project.

B. *Method of Data Transfer*

We briefly considered PCIe for transferring data quickly between our laptop and our FPGA. However, getting PCIe to adapt to the ports on a laptop is really tough, and in the adaptation process the PCIe transfer speed get bottlenecked to the speed of the port it's trying to adapt to. Thus, we decided to go with UART for transferring data, as it is simple to understand and since we can dial up the baud rate easily.

C. *Algorithm for Facial Detection*

We considered two algorithms for facial detection, the Viola-Jones algorithm we mentioned in the previous section, and a convolutional neural network. We summarize the tradeoffs between the two approaches in the following table.

Criteria	Viola-Jones	Neural Network
Familiarity	Good, a lot of documentation in OpenCV and related work on FPGA.	Not ideal because it is hard to find existing neural networks for facial detection on FPGA.
Computation Intensity	Not ideal, because of simple operations at each stage, but can pipeline stages.	Ideal on FPGA, heavy computations. More likely to see the speedup.
Training	No training, because we can use pre-trained weights.	Long training, need to train it ourselves or use weights from a pre-trained network.
Difficulty	Conceptually easy to understand, but hard to optimize	Complicated network, but each layer does similar things.

Table 1: Tradeoff between Viola-Jones and Neural Network

After considering the criteria in Table 1 and the timeline of our project, we decided to use Viola-Jones because it has good documentation to refer to, and because we don't have to worry about training.

D. *Software Storage Method for Face Database*

We considered two methods of storing the face database on our laptop: storing it locally and storing on a cloud storage, such as Amazon S3. For storing in the cloud, we would need to use additional protocols to access the images and vectors in our database. Additional latency would be introduced if we were to try and store our data in the cloud. The benefit would be that we would essentially have no limit in terms of how much storage we have to work with. However, once we computed how much storage we actually needed, we don't think there is any issue with storing the face database locally. Thus, we decided to store face database locally on our laptop.

E. *Using Vivado HLS or Handwritten HDL*

To implement facial detection in HDL, we considered using two options: Vivado High-Level Synthesis (HLS) and writing an implementation from scratch in Verilog. HLS easily converts C code to Verilog HDL and VHDL, but produces the HDL jumbled such that it's too hard to track all the signals and logic in the design. The only way to optimize the output HDL is to modify the HLS parameters, making the optimization very uncustomizable. Writing an HDL implementation from scratch in Verilog would provide much more customizable optimization. Thus, we chose to go with the handwritten HDL implementation. However, our implementation was too big to fit on the FPGA, even after we implemented multiple optimization approaches. Below are a few challenges we encountered when trying to handwrite our own HDL implementation:

1. Big Image Size
   Our input image size is 160 pixels by 120 pixels. If we used registers to store our image, our flip-flop and look-up table counts exceed number available on our FPGA. If we stored the image in block RAM, we would have had to design a finite state machine to load the image from block RAM one pixel at a time. This would entail complicated waiting/handshaking logic between every module needing the image.
2. Trade Off between Parallelism and Utilization
   Since we need to evaluate many feature classifiers on the image, we could either perform all of them at the same time, or divide them into stages. Performing all the classifiers at the same time would need more logic cells than available on the FPGA, but dividing the feature classifiers into stages would require us to store results from previous stages. This would in turn require a lot of registers, exceeding our flip-flop and look-up table limits.
3. Long Synthesis Time
   Vivado takes a long time to synthesize our design because our design is relatively big. Every time we finish

an optimization, it takes hours for synthesis to finish, further preventing us from quickly figuring out if the optimization is functionally or if it meets our expectations for a minimum viable product.

With all these challenges in mind, we decided to switch to using Vivado HLS to convert C code into Verilog.

F. *Real Time Face Detection*

We considered doing real-time facial detection at 15 fps, which was slower than the rate the software facial detection implementation was able to run at. However, since we have the I/O bottleneck from transferring data through UART, a setup running real-time facial detection through the FPGA wouldn't be able to run at 15 fps. If there was a camera and a HDMI monitor attached to the FPGA, this might have been possible, since the I/O would be much faster in this setup. Thus, we did not do real time face detection.

V. System Description

We break down this section into three parts: laptop, FPGA, and UART channel.

A. *Laptop*

The laptop captures camera input, then downscales and grayscales the image. It can perform the complete software pipelines for both facial detection and identification.

1) Downscale Module

The app downloads the camera-captured image as a 160 pixels by 120 pixels image, which is the input accepted by our facial detection module.

2) Grayscale Module

Since our detection and recognition algorithms work best on grayscale images, we grayscale the image taken on the laptop during preprocessing. This also minimizes I/O transfer delay, since we also need to transfer the image to the FPGA for facial detection. The laptop camera image is in RGBA scale, which takes 4 bytes per pixel. The grayscale image takes only 1 byte per pixel, reducing the amount of data to transfer by 75%.

3) Facial Detection Module

The facial detection module uses the Viola-Jones algorithm to detect the position of the face within the input image. It uses pre-trained weights from OpenCV and our own classifier implementation. We perform the following steps to detect any faces in the image:

1. Build an image pyramid for the input image. Each pyramid is the input image at a different scale, with each progressive pyramid level representing the image downscaled one more time. This makes our algorithm more robust against faces of variable sizes. In our solution, we use a downscaling factor of 1.2.
2. Compute the integral image for each image pyramid. Each pixel in the integral image is the sum of all pixels to its left and above. Integral image is useful later because we need to quickly compute the sum of pixels in an arbitrary area. In the image below, assume we want to compute the sum of pixels in area D. If we have the integral image, we can compute the area as II(4) + II(1) - II(2) - II(3), which are all constant-time operations.

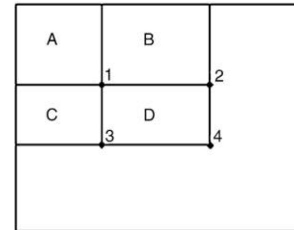

Figure 2: Integral Image Explained

3. A 24-by-24 scanning window is moved across each level of the image pyramid in order, eventually scanning all the levels. Each scanning window is considered a face candidate and is passed through a pipeline to determine if it is a face.
4. Each stage of the pipeline contains a number of feature classifiers that each generate a feature score based on the face candidate. In each stage, the sum of all feature scores needs to exceed a threshold for the face candidate to pass the stage. A face candidate is only considered a face if it passes all the pipeline stages. Some example feature classifiers are shown below. We take the difference between the sum of pixels in the black area and the sum of pixels in the white area. Notice that the sum of pixels can be computed using integral image. The feature in the middle image tries to detect the contrast between the eye area and cheek area. The feature in the rightmost image tries to detect the contrast between the eye area and the nose area. We can see both of them try to capture the characteristics on a typical human face.

Figure 3: Example Feature Classifiers

5. When multiple faces are detected, the face with the highest total feature scores is chosen. Its coordinates are outputted from the facial detection module.

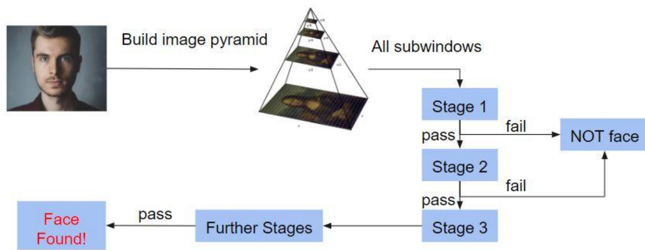

Figure 4: Facial Detection Process

4) Face Cropper and Downscaling Modules

Once the detection module outputs the face rectangle, the face cropper module crops the face out of the image, and the downscaling module downscales the cropped face into a 20 pixel by 20 pixel image. These dimensions are the input size to our identification module.

5) Facial Identification Module

We use the eigenface algorithm for our facial identification module. The algorithm is divided into a training phase and a testing phase.

Training:

We use the Yale faces database for training. There are 15 subjects and 166 images in the database. The following steps are performed for training:

1. **Preparation**. Crop all the faces out with our detection module and downscale all faces into 20 pixel by 20 pixel images. This process gives us 163 images, because there are no faces detected in 3 of the images. We use 8 images for each subject for training, so there are a total of 120 training images, each of which is 20 pixels by 20 pixels. If we put one face in a column, we have a matrix of size 400-by-N (height: 400, width: N), where N is the number of training subjects multiplied by 8.
2. **Normalization**. Compute the average face in the database, and subtract each face from the average face.
3. **Compute Eigenvectors**. Compute the eigenvectors and eigenvalues of the covariance of the training matrix. The eigenvectors represent how much each face differs from the average face, so they represent the variance of the face database. The covariance of the training matrix is a 400-by-400 matrix. After taking the eigenvectors and eigenvalues, we only use the first 36 eigenvalues because they represent more than 95% of the total eigenvalues. The resulting eigenvector matrix has size 36-by-400 (height: 36, width: 400).
4. **Projection**. Project each face in the training database onto the eigenvectors. This is done by directly multiplying the eigenvector matrix (36-by-400) by the training face matrix (400-by-N), which gives a 36-by-N matrix. This matrix is called the weight matrix. Notice that the dimension of each training face is essentially reduced from 400 to 36.

Figure 5: Yale Faces Database (15 subjects, 166 images)

Testing:

When doing identification, given an input 20-by-20 image, the following steps are performed:

1. **Normalization**. The input face is subtracted from the average face to achieve normalization as was done in training. The result is an image vector with length 400.
2. **Projection**. Project the result image onto the eigenvectors, so multiply the eigenvector matrix (36-by-400) by the input image vector (400-by-1), giving a resulting vector of length 36. This vector describes how the input image differs from the average face.
3. **Find Face**. Recall that the weight matrix has size 36-by-N. We compute the euclidean distance between the projection vector in the previous step and every column in the weight matrix. The column with the smallest distance corresponds to the face that has the closest match.

B. *FPGA*

The FPGA we are using is the Kintex-7 XC7K325T FPGA, on the Xilinx KC705 Development Board. It is responsible for performing facial detection because we want to visualize the speedup achieved by implementing our facial detection algorithm on an FPGA. We use the Vivado High-Level Synthesis (HLS) tool to synthesize our C code for detection into a Verilog module. We integrate that module with our SystemVerilog UART implementation, along with some overhead logic for queueing up face rectangles and some miscellaneous logic. We compile the final design along with a chip configuration file into a bitstream, which is programmed via JTAG onto the FPGA. We also generate a memory configuration file using the bitstream and program that file into the BPI Flash on the development board. Because of this, the design can be directly programmed from BPI Flash via the program button on the development board.

We perform the following steps in Vivado HLS to speed up our hardware facial detection implementation:

1. **Faster Clock**. We use the fastest clock possible, while still making sure all the operations can finish within a clock cycle. In other words, the critical path has to be less than the clock period.
2. **Loop Pipeline**. We use the loop pipelining optimization clause in HLS, which allows loop iterations to start as early as possible.
3. **Loop Unroll**. We use the loop unrolling optimization clause in HLS to tell FPGA the exact operations to perform. This allows Vivado to better optimize the code.

C. UART

The UART channel is responsible for transferring the downscaled and grayscaled input from the laptop camera to the FPGA. Once the FPGA finds the face rectangle, the UART channel transfers the coordinates of the face rectangle back to the laptop. The transfer of the input from the laptop camera takes the longest. We use a baud rate of 921600 bits per second. The 160 pixel by 120 pixel input image, with each pixel encoded in a byte and each byte requiring one start bit and one stop bit to transfer (192000 bytes total), takes around 0.208 seconds to transfer.

VI. RESULTS

We evaluate our project using two metrics: accuracy and speedup.

A. Accuracy

We evaluate the accuracy of facial detection and facial identification separately.

Facial Detection:

- Goal: 80%
- Standard: Detect the bounding box around the face if there is a face in the image, or reject the image if there is no face.
- Result: 98%, exceeding our expectations
- We use 166 images from the Yale faces database as test images. We are able to correctly detect the faces in 163 out of 166 images. That gives us an accuracy of 163/166=98%. This number is rather subjective, because the images in the database have good contrast between the face and the background.

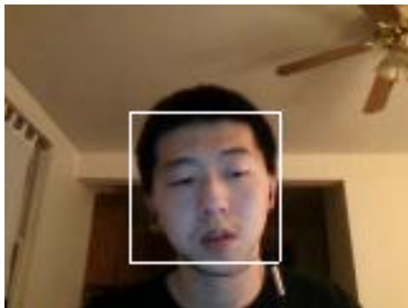

Figure 6: Successful Facial Detection Demo

Facial Identification:

- Goal: 80%
- Standard: Assuming the person in the input image is in the database, correctly identify the person.
- Result: 86%, exceeding our expectations
- We use the 163 correctly detected faces from the facial detection module. Since we use 120 images for training, we have 43 images left for testing. Among the 43 images, we correctly identify 37 of them. That gives us an accuracy of 37/43=86%.

Identity Checker

You must be:Junye Chen

Figure 7: Successful Facial Identification Demo

B. Speedup

We measure the facial detection implementation in C with the time command in Linux. We measure the FPGA implementation in number of clock cycles, and we multiply that number by the clock period.

- Goal: 5x speedup
- Result: 1.6x speedup, below our expectations
- Our baseline version on FPGA without any optimization takes 0.17s. The table below shows the improvement of each optimization technique.

Optimization Step	Improvement
Faster Clock: 50MHz => 200MHz	0.17s => 0.0826s
Loop Pipeline	0.0826s => 0.037s
Loop Unroll	0.037s => 0.031s

Table 2: Speedup Process

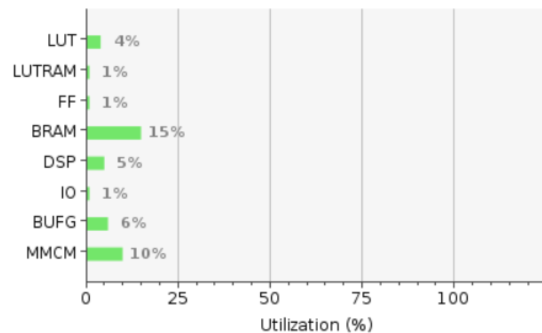

Figure 8: FPGA Utilization Graph

Our speedup result is less than ideal. The CPU facial detection is already very fast, so the room for optimization is very little. Some optimization techniques such as loop unrolling give us less ideal results than expected. We think loop unrolling should be a very effective technique, because it tells the FPGA the exact operations to perform. However, in our project, we were unable to unroll some big for loops either because the loop limit couldn't be determined at compile runtime, or because the loop limit was too large.

VII. Project Management

A. Schedule

Below is a rough description of the tasks we work on by week. We have attached the full gantt chart at the end of this document.

Week of	Task(s)
3/3	1. facial detection in C, 2. setup UART example project
3/7	1. facial detection testing, 2. design our own implementation of facial detection in Verilog
3/24	1. integrate software facial detection with app, 2. implement UART channel between software and FPGA, 3. implement facial detection in Verilog
3/31	1. optimize facial detection in Verilog
4/7	1. facial identification in Python, 2. shift to HLS for a baseline version on FPGA
4/14	1.facial identification improvement/testing, 2. integrate facial identification with app, 3. finish HLS implementation and connect with our UART channel
4/21	1. HLS optimizations
4/28	1. Integration test

Table 3: Tasks by Week

B. Team Member Responsibilities

Junye - Implement Viola-Jones cascading classifier in C, implement eigenface algorithm in Python, generate HLS output and optimizations

Sheng-Hao - Re-acquaint with Vivado and show Hans how to use it, work with Hans to convert algorithms to RTL and optimize them.

Andy - Design app interface, preprocess image (grayscale, downscale)

C. Budget

We have included a spreadsheet of our budget at the end of this report.

D. Risk Management

I/O - We identify that I/O is a major bottleneck for the speedup of our system. Assume our input is a 160 by 120 image. If we set baud rate to 921600 bits/second and use 1 start bit and 1 stop bit, it takes about 160x120x10 / 921600 = 0.208s to transfer the data from software and FPGA. The FPGA we have right now can support as much as 921600 bits/second. We will also make sure that we account for the speedup without I/O.

VIII. Related Work

This project is mainly inspired by a series of past research projects that eventually convened in implementing facial recognition fully on an FPGA. They reuse their HDL facial detection implementation, which uses the same algorithm as our facial detection implementation does. The series of research projects are listed below:

[1] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," in International Symposium on Field Programmable Gate Arrays, February 2009.
[2] J. Cho, B. Benson, and R. Kastner, "Hardware acceleration of multi-view face detection," in IEEE Symposium on Application Specific Processors, July 2009.
[3] J. Matai, A. Irturk, R. Kastner, "Design and Implementation of an FPGA-based Real-Time Face Recognition System," in IEEE

IX. Summary

A. Design Specifications

Our system was able to meet the accuracy requirements that we set for ourselves. Owing to time constraints, we were not able to meet the 5x speedup requirement for detection that we set. However, we are still content that we were able to get some speedup, especially with an FPGA running at only 200 MHz when our laptop was running at upwards of 2.8 GHz.

B. Future Work

In facial detection, we can try attaching a camera and a HDMI monitor to the FPGA to perform real-time facial detection, providing a feasible setup to try and achieve real-time speedup in.

In facial identification, we can try to detect if a face is not in the database by checking if the closest match meets a threshold.

In hardware, we can try inputting images with bigger sizes and more faces. We believe we will observe more speedup if we make the computation more intense.

In hardware, we can also implement facial recognition, and achieve speedup there.

C. Lessons Learned

1. Always have contingency plans.
   If we didn't have the contingency plan of switching to Vivado HLS, we would not have a demo-able project.
2. Identify dependencies between tasks, and follow your schedule closely.
   The schedule was the only way for us to tell if we were doing things on time. We had to follow it religiously, or risk running out of time to finish at the end.

3. Responsible delegation of work to team members is very important.
   While we had team members each doing their own separate parts of the project to begin with, we quickly realized we had underestimated the amount of work required for the FPGA portion. Our software/algorithms specialist had to step in to help our hardware specialist with the FPGA portion, since our app designer didn't know how to do hardware. Instead of only caring about separation of responsibilities, we should have considered the delegation of equal work to each member. In our case, we could have given the work for the app and the software/algorithms portions to one person, then had the remaining two completely focused on hardware.

REFERENCES

[1] KC705 User Guide, https://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf
[2] 7 Series FPGA Data Sheet, https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf
[3] Programming BPI Flash, https://scholar.princeton.edu/jbalkind/blog/programming-vc707-virtex-7-bpi-flash
[4] Pyserial Documentation, https://pythonhosted.org/pyserial/
[5] Vivado HLS User Guide, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf
[6] Clocking Wizard LogiCORE IP Product Guide, https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_3/pg065-clk-wiz.pdf
[7] Integrated Logic Analyzer LogiCore IP Product Guide, https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf
[8] Synthesizable Constructs, https://link.springer.com/content/pdf/bbm%3A978-81-322-2791-5%2F1.pdf
[9] What is a Constraints File, https://reference.digilentinc.com/learn/software/tutorials/vivado-xdc-file
[10] Viola-Jones Object Detection Framework, https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework
[11] Eigenface, https://en.wikipedia.org/wiki/Eigenface

2/3
2/10
2/17
2/24
3/3
3/10
3/17
3/24
3/31
4/7
4/14
4/21
4/28
S: Learn algorithms, Approximate FPGA memory needed
H: Learn algorithms, Approximate FPGA memory needed
A: Sketch out app skeleton, list out features
S: Select FPGA, Learn Vivado
H: Train facial detection classifier in Python
A: Learn image preprocessing algorithms
S: Help Hans, Figure out sending data over UART
H: Validate classifier, Tune stages and features
A: Design storage for face data bank
S: Create test projects using UART and BRAM
H: Write C code to use facial detection classifier
A: Implement the app UI
S: Help Hans
H: Convert detection code to RTL, Write testbenches to test RTL
A: Implement downscaling algorithms in Python
Spring break
S: Combine RTL from Hans with UART/BRAM RTL
H: Implement facial recognition algorithm in C
A: Finalize storage solution for face data bank
S: Port working facial detection design onto FPGA
H: Optimize detection code so converted RTL is optimized
A: Finish app
S: Finish app integration with FPGA
H: Help Sheng-Hao
A: Adjust app based on changes encountered
Final integration and testing
Update documents, Prepare for demo/final presentation

Part	Cost
Laptop (from us)	$1000 ($0)
KC705 Board Kit (from course inventory)	$1685 ($0)
Vivado License (from CMU) - BRAM modules	$3595 ($0)
Pre-trained Viola-Jones weights	$0
Total	**$0**