IMPLEMENTATION OF THE EIGENFACE
FACE RECOGNITION ALGORITHM IN
FPGA AND MICROPROCESSOR

A project submitted in partial fulfillment of the requirements
For the degree of Masters of Science
in Electrical Engineering

By

Edward Carl Lin

May, 2013
The graduate project of Edward Lin is approved by:

________________________________________
Dr. Ronald W. Mehler
Date

________________________________________
Professor Ahmad Bekir
Date

________________________________________
Dr. Xiyi Hang, Chair
Date

California State University, Northridge
# Table of Contents

SIGNATURE PAGE ........................................................................................................... ii

ABSTRACT......................................................................................................................... v

INTRODUCTION .................................................................................................................. 1

THEORY OF EIGENFACE FACE RECOGNITION ALGORITHM ..................................... 2
  Eigenface Training Algorithm......................................................................................... 4
  Eigenface Classification Algorithm.............................................................................. 6
  Test Results................................................................................................................... 7

DESIGN IMPLEMENTATION ............................................................................................ 9
  Development Environment .......................................................................................... 9
    Laptop Computer....................................................................................................... 10
    FPGA Development Board....................................................................................... 10
    Software Tools.......................................................................................................... 12
  Design Flow.................................................................................................................. 12
  Face Image Database.................................................................................................... 14
  FPGA microBlaze Setup............................................................................................... 16
  Microprocessor Top Level C Code............................................................................... 19
  Eigenface Algorithm C Code....................................................................................... 21
    Mean Face Function................................................................................................. 22
    Shifted Images Function............................................................................................ 24
    Covariance Function................................................................................................. 25
    Jacobi Eigen Function............................................................................................... 26
    Sort Matrix Function ............................................................................................... 30
    Matrix Multiply Function......................................................................................... 31
    Features Function....................................................................................................... 32
ABSTRACT

IMPLEMENTATION OF THE EIGENFACE

FACE RECOGNITION ALGORITHM IN

FPGA AND MICROPROCESSOR

By

Edward Carl Lin

Master of Science
in Electrical Engineering

This project investigates the theories of the eigenface face recognition algorithm by Pentland and Turk and demonstrates its effectiveness in hardware. It uses MATLAB to simulate the mathematical theories and implements the algorithm in FPGA and microprocessor. The same test images are used with MATLAB and FPGA implementations and the results are compared for accuracy. Many design trade-offs are made in order to operate within the resource constraints of the low-end FPGA board. Analysis shows these sacrifices lead to an error rate of up to 24% compared to the original publication. Based on what is learned from this project, future improvements are explored.
INTRODUCTION

Face recognition technology is a field that has become increasingly important as well as controversial. Today it can be found in many places such as Facebook’s tag suggestion to help identify friends in your photos, Google’s Android smartphone face unlock, London’s security camera systems and face recognition systems at various airports and government organizations. Even though the field has progressed leaps and bounds since the early 1990’s when the first practical face recognition systems were developed, in many ways it is still in its infancy. The challenges of recognizing facial features and the sheer number of variations due to the large human population as well as environmental conditions when images are taken means these systems still do not produce reliable results in non-optimal conditions. The increasing ubiquity of recognition systems along with doubts about their effectiveness conjures up fear over invasion of privacy and being misidentified as a wanted individual.

The primary goal of this project is to gain further insights into this cutting edge area of research by understanding one of the first successful face recognition techniques, the eigenface algorithm, and implement it on a small portable hardware system. The project will use an inexpensive FPGA development board as an educational tool to apply algorithms to MATLAB simulation and actual hardware system implementation, as well as to teach the tradeoffs necessary to make a system work despite its limitations.
THEORY OF EIGENFACE FACE RECOGNITION ALGORITHM

The face recognition algorithm investigated in this project is based on the seminal work “Eigenfaces for Recognition” published in 1991 by Matthew Turk and Alex Pentland. This approach is considered to be the first implementation of face recognition technology with a high degree of accuracy.

Prior to the eigenface approach, various methods focused on the individual facial features rather than the sum of the whole face in face recognition. These methods included measuring the size and ratios between the various facial features, characterizing the face by geometric parameters and using local feature template matching to find and measure facial features. Some of these methods were not fully automated, requiring a person to determine notable features to focus on or enter fiducial markers by hand. Another criticism was that while these algorithms may achieve good results within a subset of faces, they were not easily extendable to recognition of other faces.

The eigenface algorithm applies an information theory approach to the challenges of face recognition. Rather than focusing on features that may seem more intuitive for identification such as eyes, nose and mouth, the eigenface approach applies the mathematical concept of principal component analysis (PCA) to a database of face images.

Principal component analysis is an eigenvector based method of finding patterns in large data sets and determining the points of similarities and differences across the data space by weight values. This can be graphically illustrated three dimensionally with the data set as the X and Y axis and the weights in the Z axis. Another advantage of this method is
that after the principal components have been calculated, the data can be compressed with minimal loss of information. This is essential to face recognition since it allows for larger sets of data to be stored and lower processing power to analyze images.

The eigenvectors that results from the PCA are called eigenfaces since they appear as rather ghostly images of faces and represent the characteristic variations between the face images. The concept of eigenface was original developed by L. Sirovich and M. Kirby in 1987 and 1990 for image compression. Their work shows that given an ensemble of face images, a coordinate system can be determined which they termed eigenpicture. Using a collection of weights and projecting on the eigenpicture coordinates, the original images can be roughly reconstructed. Turk and Pentland realized that this efficient representation of face images can be applied to face recognition by comparing these eigenpicture weights of a new face with those of previously identified persons.

The eigenface algorithm comprises of two phases, training and classification. The general steps of the training phase are summarized as follows:

1. Calculate the mean of a set of face images corresponding to the same person.
2. Compute eigenvector and eigenvalues on the deviation of each face image from the mean.
3. Order the eigenvectors in decreasing order by the eigenvalues of the face images, keeping only the top ranked eigenvectors.
4. Project the mean shifted images onto the eigenvector space. This is the eigenface.

Once all the face images have been mapped into the eigenface space coordinates, classification of new previously unknown images can begin. A new test image is
evaluated by projecting itself onto the eigenface of each face image in the database and finding the Euclidean distance between the corresponding points. A threshold is set that classifies the Euclidean distance values of the new face image as either a recognized face, a face that is not in the database or not a human face at all. The next section will detail the eigenface algorithm in mathematical terms.

**Eigenface Training Algorithm**

The face images used for the eigenface algorithm are 256 x 256 pixels of 8-bit intensity values (0 to 255 in grayscale). The images are then represented as 65,536 x 1 dimensional vectors in order to group several images together to form the database and compute eigenvectors from it. Let the set of training face images be \((\Gamma_1, \Gamma_2, \ldots, \Gamma_n)\). The mean of images is given by:

\[
\Psi = \frac{1}{M} \sum_{n=1}^{M} \Gamma_n
\]

This mean is the average face of the training set. The difference of each face image in the database from this average face is \(\Phi_i = \Gamma_i - \Psi\).

The followings steps spell out the principal component analysis which involves taking the eigenvector of the covariance matrix of \(\Phi_n\) and reordering it according to the eigenvalues from largest to smallest. First the covariance matrix \(C\) is derived from the difference images \(\Phi\):
\[ C = \frac{1}{M} \sum_{n=1}^{M} \Phi_n \Phi_n^T = AA^T \]

where the matrix \( A = [\Phi_1, \Phi_2, ..., \Phi_n] \). One major problem that arises is that the covariance matrix \( C \) becomes a huge 65,536 x 65,536 matrix which will be very difficult to work with. Instead use \( C = A^T A \) which produces an \( M \times M \) matrix (i.e. 20 x 20), a significantly more manageable size. Compute the eigenvectors:

\[ A^T A \nu_i = \mu_i \nu_i \]

The eigenvalues of \( A^T A \) and \( AA^T \) are identical and the eigenvectors follow the equation:

\[ u_i = A \nu_i \]

Therefore, the same eigenvalues and eigenvectors can be arrived at while performing operations that are orders of magnitude less computationally intensive. The resulting eigenvectors are sorted according to its eigenvalues and normalized. Finally only the eigenvectors corresponding to the \( N \) largest eigenvalues are kept. The number \( N \) is experimentally determined based on data variance and computational limitations.

Each face \( \Gamma \) in the database can now be projected on the eigenvector space by multiplying with the eigenvectors \( u_i \) to create the eigenface \( \omega_i \) as follows:

\[ \omega_i = u_i^T (\Gamma - \Psi) \]

where \( i = 1, 2, ..., K \). The vector \( \Omega^T = [\omega_1, \omega_2, ..., \omega_K] \) is used to represent the contributions of all the eigenfaces to define an individual. This vector can now be used
for several different types of pattern recognition algorithms to see how an input test image can be classified into several predefined face classes.

**Eigenface Classification Algorithm**

The paper “Eigenface for Recognition” presents one simple method of pattern recognition based on Euclidean distance. Given an unknown face image \( \Gamma \), the mean adjusted face image will be \( \Phi = \Gamma - \Psi \) and the projection of the unknown face into the eigenface space will be:

\[
\Phi_f = \sum_{i=1}^{K} \omega_i u_i
\]

This is the projection of the unknown face image onto the eigenface space. The Euclidean distance \( \epsilon \) of the unknown face onto the eigenface space can now be calculated as:

\[
\epsilon^2 = \| \Phi - \Phi_f \|^2
\]

From the calculation, the results can be classified into four different possibilities based on the defined thresholds:

1. Near a known face and in the face space. The unknown input image has been matched with a known face in the database so there is a positive match.
2. Near a face space but not a known face. The unknown input image is identified as a face but does not match any known face images in the database.
3. Far from a known face and face space. The unknown input image is not a face image.
4. Near a known face but far from the face space. This result indicates a false positive and should be discarded. Since it signifies that there is a positive identification of a known face in the database but the image is not in the face space.

Given a large enough sample of new test input images, it is possible to construct new known face profiles in the database by analyzing the proximity of these images to each other in the eigenface space.

Test Results

Two major experiments were conducted in “Eigenface for Recognition” to test the effectiveness of this algorithm. In the first experiment, over 2500 images of 16 individuals were used as the database with face pictures taken under different light conditions, face orientations (straight, tilted to left or right side) and distance from subject (different face sizes). The results were the algorithm achieved 96% accuracy with variation in light conditions, 84% accuracy with variation in head orientation and only 64% accuracy with variation in image face size. This is clearly a weakness in the algorithm since it relies on the correlation between neighborhood pixels.

The second experiment used the same database of images but tuned the classification thresholds to achieve different goals. When the thresholds are set to achieve 100% recognition accuracy, the percentage of images that gets classified as unknowns becomes 19% for variation in light conditions, 39% for variation in head orientation and 60% for
variation in image face size. Therefore there is a trade off in recognition accuracy and ability to classify images when selecting appropriate thresholds.
DESIGN IMPLEMENTATION

Once the theories and mathematics behind the eigenface face recognition algorithm are understood, implementation can now be carried out in a real-life design. First a proper development environment is assembled consisting of all required hardware and software for this task. Then the design implementation is built up from simulation to hardware realization. Lastly the tradeoffs necessary to carry out this design within the limitations of available resources are explored.

Development Environment

The development environment of this project consists of three categories: laptop computer, FPGA development board and design software. The below picture shows the FPGA development board connected to the laptop computer via the USB programming cable (black) and RS-232 data cable (silver):
**Laptop Computer**

An Acer Aspire S3 laptop computer with an Intel Core i5 1.6 GHz processor and 4 GB of DDR3 SDRAM running Windows 7 Ultimate 64-bit operating system is used to run the design and simulation software and provide the interface to the FPGA development board. An USB cable provides the interface for FPGA programming and debugging while an USB to RS232 adaptor cable is used to send face image data to and receive processed data from the FPGA.

**FPGA Development Board**

The FPGA development board used is a Xilinx Spartan 3A Starter Kit manufactured by Digilent first released in 2007. It is a general purpose development board that centers on a Xilinx Spartan 3A FPGA along with various interfaces for memory, user I/O and peripherals. Memory interfaces into DDR2 SDRAM, Flash and PROM. User I/O interfaces include push buttons, switches, rotary knob and 2-line LCD screen. Peripheral interfaces include ports for USB, Ethernet, RS-232, VGA Monitor and general purpose serial and parallel connectors. These board elements and others that are not used for this project are labeled below:
The Xilinx Spartan 3A device on the development board is an XC3S700A. It is a low cost, full featured field programmable gate array (FPGA) with around 700,000 system gates and 372 user I/O pins. With the use of the proprietary Xilinx ISE design suite, a wide variety of digital projects can be pursued including user designed logic modules, the Xilinx microBlaze soft processor core, Xilinx or third party intellectual property (IP) modules or a combination of all three as is accomplished in this project.
Software Tools

Two main design software suites are required for this project. MATLAB R2010b with the image processing and statistics toolboxes is used for proof of concept, design simulation and FPGA interface. Xilinx ISE Design Suite 13.4 is used to program, debug and interface the FPGA to the laptop computer. Within the ISE Design Suite, several programs are used for specific tasks: The Project Navigator is used to design and compile VHDL logic and top level design modules; XPS is used to configure the microBlaze soft processor core IP; EDK is used for coding and debugging the C program that resides in the microBlaze; Chipscope is a customizable logic analyzer IP that resides in the FPGA used for debugging. Another essential tool is GIMP (GNU Image Manipulation Program), an open source image editing program that is used to modify and optimize images to build the face database and test images.

Design Flow

With the development system in place, the first goal of the design implementation is to reproduce the algorithm step by step using MATLAB to prove that it can be realized. This is accomplished using MATLAB functions such as princomp, norm, and sort knowing that these functions will have to be written later on. The same database of controlled face images is used for both the MATLAB and FPGA implementation to ensure the inputs are identical. Once the MATLAB design is verified to be able to classify input images, each of the higher level MATLAB function is broken down into simple functions for implementation in FPGA. The most important function is princomp used to find the principal component analysis of the input images. This function is
rewritten using the Jacobi method to determine eigenvectors and eigenvalues and a sorting algorithm is used to order the eigenvectors based on the eigenvalues from highest to lowest values. Now the eigenface algorithm is laid out in a format that can be implemented in the FPGA. See Appendix A for the MATLAB code of the eigenface algorithm. This approach is an expanded version of the MATLAB eigenface tutorial from Cordiner (2009) with additional information from Vinther (2002).

The Spartan 3A FPGA is set up with the microBlaze configurable microprocessor as its core for implementation of the algorithm using the software programming language C. Due of the sequential nature of the eigenface algorithm, every stage of the C implementation can be verified by sending the debug data back to the PC via UART and comparing with MATLAB results. This ensured that the C code matches MATLAB at every step of the algorithm and makes debugging a complex design possible.

Eventually the MATLAB eigenface algorithm functions and results are reproduced in the FPGA in a straightforward but inefficient way. Several optimization techniques are applied including microBlaze IP setups, C function algorithm improvements and inclusion of FPGA logic hardware modules for faster processing over the microprocessor. This allowed the algorithm processing time to be reduced to a more manageable level.

Once the FPGA implementation has been optimized and results are verified across both MATLAB and FPGA designs using the same input images, the algorithm is tested against several more images of recognized faces, unknown faces and non-face images to ensure it is robust. This increased number of data points also improves the setting of the classification thresholds to decrease the chances of false positive results.
**Face Image Database**

There is a plethora of available databases for face images including ones from Yale, AT&T, MIT and many others. Most of these databases contain hundreds of images taken in controlled environments with several images per individual. They typically aim to present different challenges for face recognition algorithms to test against. These include different facial expressions, light conditions, face orientations, facial hairs, accessories, etc. Knowing that the eigenface algorithm was discovered in the infancy of face recognition technology, it is decided that a database has to be chosen that will not be overly rigorous given eigenface limitations. The database selected for this project is the Caltech Faces database created in 1999 consisting of 450 images of size 896 x 592 pixels. This database consists of photos of 27 individuals taken with different expressions, background and lighting conditions. The main reason for choosing this database is that the majority of the pictures are taken with the head facing straight at the camera and at roughly the same distance, which addresses two known weaknesses of the eigenface algorithm.

The face images of the Caltech Faces database are further processed using GIMP, a free open source image editing program with similar features to Adobe Photoshop. The images are cropped to focus only on the face to prevent background and hair from distracting the algorithm. The cropping also aligns the face so that the eyes, nose and mouth are in relatively the same positions across the entire set of face images used. After an image is cropped, the face image size is scaled to a more manageable 60 x 48 pixels of 8-bit grayscale. With all these variables controlled, the eigenface algorithm will be allowed to focus on facial features, skin tone and lighting condition for face recognition.
which are within its capabilities. Note that by using GIMP and focusing on front facing faces, it is possible to create your own face database by using personal photo collections or pictures of famous people from internet image search.

The below tables shows examples of some of the original images and their grayscale cropped images used for the face database that were created using GIMP:

<table>
<thead>
<tr>
<th>Original Image</th>
<th>Grayscale Cropped Image</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Original Image" /></td>
<td><img src="image2" alt="Grayscale Cropped Image" /></td>
</tr>
<tr>
<td><img src="image3" alt="Original Image" /></td>
<td><img src="image4" alt="Grayscale Cropped Image" /></td>
</tr>
<tr>
<td><img src="image5" alt="Original Image" /></td>
<td><img src="image6" alt="Grayscale Cropped Image" /></td>
</tr>
<tr>
<td><img src="image7" alt="Original Image" /></td>
<td><img src="image8" alt="Grayscale Cropped Image" /></td>
</tr>
</tbody>
</table>
Several sets of 10-20 face images of different individuals along with around 30 test images were created using GIMP to test the classification capability of the algorithm. The same images are fed to both the MATLAB design and the FPGA design and compared at the end for consistency. In the final version of the design, ten test images are selected to test against a database of three individuals with ten face images each and the results are presented toward the end of this report.

**FPGA microBlaze Setup**

The microBlaze microprocessor inside the Spartan 3A FPGA is the centerpiece of the FPGA design implementation. MicroBlaze is a configurable soft processor IP core designed specifically to work with Xilinx FPGAs. It is a 32-bit RISC (reduced instruction set computing) Harvard architecture processor that uses an PLB (processor local bus) interface from which several Xilinx or third party IP modules can connect to.

To handle the computational needs of the eigenface algorithm, the microBlaze is configured to include a floating point unit (FPU) to handle 32-bit floating point numbers, instruction / data caches to aid performance, and a debug module to enable use of software debugging tools. The processor runs at 100 MHz which is derived from the 50 MHz crystal oscillator on the FPGA development board using the Xilinx Digital Clock Manager (DCM) module.

Several memory bus interface IP modules are instantiated to communicate with hardware on the development board including UART, SDRAM and BRAM. For communication with the laptop computer, a UART module is instantiated with a baud rate of 115,200
bps. This UART uses a RS-232 to USB cable to connect to the laptop where MATLAB opens a serial port for communication. This simplifies the data interface since it allows MATLAB to directly talk to the FPGA instead of relying on a separate hyper terminal program or custom GUI as an intermediate step. Once a serial port is active, MATLAB can send raw image data to the FPGA board and receive back the processed data and debug information. A DDR2 SDRAM memory controller running at twice the processor speed (200 MHz) is also instantiated to interface a Micron 512-Mbit DDR2 SDRAM with 16-bit wide data bus. This external memory is necessary since the FPGA itself does not have enough memory to store all the image data and processed data necessary. However the penalty of using SDRAM is that memory access is slower and the memory control much more complex compared with SRAM. Another memory used is a small dual-port 16-kbit block RAM (BRAM) module that resides inside the FPGA itself. This memory is used for local and intermediate data storage since memory access is fast and straightforward. The intent is to instantiate the largest BRAM possible to aid performance over having to use SDRAM. Unfortunately 16-kbit is the limit of this particular FPGA. The dual port allows both the microBlaze and FPGA logic modules to access the memory which is necessary for FPGA logic co-processing.

Three General Purpose I/O (GPIO) modules are instantiated for this design. A FPGA control GPIO module is used for the purpose of passing control data between the microBlaze and FPGA logic. An 8-bit LED module and a 4-bit switch GPIO module are created for user interface. They are used in coordination with the UART module to display the status of data processing and start/stop data transfers.
The below screen capture from the Xilinx Platform Studio (XPS) design software shows the instantiation of the microBlaze soft processor and all the interfaces that are created for this project, including their connections to the microBlaze through the processor local bus (PLB). Also shown are the data local memory bus (DLMB) and instruction local memory bus (ILMB) and how they are linked to the microBlaze through their respective controllers.

Appendix C shows the compilation report utilization summary from Xilinx ISE Navigator of the design implementation. It shows that the microBlaze processor and other FPGA logic takes up 40% of available flip flops and 50% of input look up tables (LUTs), which gives the design good margin for routing. However the number of block RAMs is near capacity (RAMB16WE 95%) since this design tries to maximize the amount of block RAM available for local caching. Not stated in the summary is that in an effort to push the performance of the design, the compiler actually reports three timing violations in the set up times of the clock generator. The violations are minor and do not prevent the
FPGA from returning correct results every time but it is still an issue that needs to be remembered.

Microprocessor Top Level C Code

Once the microBlaze processor is set up with the appropriate features and interfaces, the development environment is ready for the C code implementation of the eigenface algorithm. The main section of the C code defines the address offsets and global variable used in the program. It then receives the 40 raw 8-bit grayscale images of dimensions 64 x 48 (10 images each of three people and 10 test images) from MATLAB via the UART interface. A total of 115,200 bytes (40 x 64 x 48) are transferred and stored in the DDR2 SDRAM as 32-bit data to match the microBlaze data bus width. As such four 8-bit pixels are stored at each SDRAM write. Once all the raw image data is storage in memory, the code runs the chosen test image number (0-9) against the eigenface algorithm three times, one for each individual in the database. The processed results are stored back in SDRAM and the UART interface is used once more to send the data to MATLAB for analysis.

The C code function can be seen below:

```c
int main()
{
    Xuint8 led_data[4], led_out, database_num;
    Xuint32 memdata, memdata_in;
    Xuint32 total, c, ptr;
    Xint32 uart_data;
    Xuint8 sdata_out[4];
    init_platform();

    led_out = 170;
    XGpio_WriteReg(XPAR_LEDS_8BIT_BASEADDR, XGPIO_IR_CH1_MASK, led_out);

    total = 0;
```
ptr = IMAGES_DDR2_BASEADDR;

// Code to read data serially from MATLAB to DDR2
while (total < 28800)
{
    c = 0;
    while (c < 4)
    {
        led_data[c] = XUartLite_RecvByte(XPAR_UARTLITE_1_BASEADDR);
        c++;
    }

    memdata = ((led_data[0]) | (led_data[1] << 8) | (led_data[2] << 16)
               | (led_data[3] << 24));
    XIo_Out32(ptr, memdata);
    ptr = ptr + 4;
    total++;
}

memdata_in = XIo_In32(IMAGES_DDR2_BASEADDR + 20*4); // Test read 84th byte
led_out = memdata_in >> 24;

// Output to LEDs
XGpio_WriteReg(XPAR_LEDS_8BIT_BASEADDR, XGPIO_IR_CH1_MASK, led_out);

// Eigenface Algorithm
for (database_num = 0; database_num < 3; database_num++) {
    mean_face_func(database_num);
    shifted_images_func(database_num, INPUT_IMAGE_NUM);
    covariance_func();
    jacobi_eigen_func();
    sort_matrix_func();
    matrix_mult_func2 (SHIFTED_IMAGES_DDR2_BASEADDR, BRAM_BASEADDR + 400,
                       EVEC_DDR2_BASEADDR, 2880, 10, 0);
    features_func();

    // CLASSIFICATION
    feature_vec_func();
    similarity_score_func(database_num);
}

led_out = 255;
XGpio_WriteReg(XPAR_LEDS_8BIT_BASEADDR, XGPIO_IR_CH1_MASK, led_out);

// Try out this wait counter if above LED write causes read problems.
int tick;
for(tick = 0; tick < 1000; tick++) {} 

// Wait for DIP SW #1 to start reading out data
while (XGpio_ReadReg(XPAR_DIPS_4BIT_BASEADDR, 1) == 0) {} 

// Code to read out to MATLAB
while (total < 36)
{
    memdata_in = XIo_In32(ptr);
    sdata_out[0] = memdata_in;
    sdata_out[1] = memdata_in >> 8;
    sdata_out[2] = memdata_in >> 16;
    sdata_out[3] = memdata_in >> 24;

    c = 0;
    while (c < 4)
    {
        XUartLite_SendByte(XPAR_UARTLITE_1_BASEADDR, sdata_out[c]);
        c++;
    }
    ptr = ptr + 4;
    total ++;
}
The 512-Mbit DDR2 SDRAM external memory is used to store the original raw images, the intermediate data generated by the eigenface algorithm steps and the result data. The below memory map shows the locations of these data storage along with their starting addresses and data type. Note the first block of memory is allocated for system use.

**DDR2 SDRAM (512-Mbit) Memory Map**

- 0xC000000: System Use
- 0xC100000: Raw Images (8-bit int)
- 0xC200000: Mean Face Images (8-bit int)
- 0xC300000: Shifted Face Images (32-bit int)
- 0xC900000: Eigenface Images (32-bit int)
- 0xD000000: Unused

**Eigenface Algorithm C Code**

The eigenface algorithm C code implementation can be broken down into nine sequential steps, seven for image training and two for image classification. These steps are as follows:
Image Training:

1. Find the mean face of all the face images in the database set of an individual
2. Find the deviation from the mean of each face image (called shifted images)
3. Find the covariance of the shifted images
4. Find the eigenvector and eigenvalue
5. Sort the eigenvector according to its eigenvalue
6. Find the eigenvector of the original image set
7. Project eigenvector into the eigenface space

Image Classification:

1. Project the test input image into the eigenface space
2. Determine the Euclidean distance of the test input image from each image in the database set and classify based on the threshold

Each of these steps is handled by a different function and they can be seen in the top level C code from the previous section under the comment heading “// Eigenface Algorithm”. The following sections describe each function in detail.

Mean Face Function

The function mean_face_func reads the raw face images data back from the SDRAM four 8-bit pixels at a time and calculate the mean by first adding up the same pixel location across each image in the data set and dividing by the number of images at the
end, in this case 10. The resulting mean face is stored back into the SDRAM four 8-bit pixels at a time. The C code function can be seen below:

```c
// Find the mean face by averaging all images of the person
void mean_face_func(Xuint8 database_num) {
    Xuint32 memdata_in, memdata_out, ddr2_addr;
    Xuint16 pixel_num, image_num, image_add[4];
    Xuint8 pixel_avg[4], image_byte[4];

    ddr2_addr = MEAN_FACE_DDR2_BASEADDR;

    // Averages 4 bytes at a time across each image and store averaged byte in DDR2
    for (pixel_num = 0; pixel_num < 720; pixel_num++) {
        image_add[0] = 0;
        image_add[1] = 0;
        image_add[2] = 0;
        image_add[3] = 0;

        for (image_num = 0; image_num < SIZE; image_num++) {
            memdata_in = XIo_In32(IMAGES_DDR2_BASEADDR + database_num*28800 + image_num*2880 + pixel_num*4);
            image_byte[0] = memdata_in;
            image_byte[1] = memdata_in >> 8;
            image_byte[2] = memdata_in >> 16;
            image_byte[3] = memdata_in >> 24;
            image_add[0] = image_add[0] + image_byte[0];
        }

        pixel_avg[0] = image_add[0]/SIZE;
        pixel_avg[1] = image_add[1]/SIZE;
        pixel_avg[3] = image_add[3]/SIZE;

        XIo_Out32(ddr2_addr, memdata_out);
        ddr2_addr = ddr2_addr + 4;
    }
}
```

The below image is an output of the mean face function for one of the subjects in the database after FPGA processing. It is an averaged face of ten face images that are fairly well aligned. It can be seen that the image is recognizable as a face but every facial
feature appears blurry. Mean faces of poorly aligned faces will be more difficult to recognize and take on a somewhat scary appearance.

**Shifted Images Function**

The function `shifted_images_func` reads back the mean face value for each pixel and subtracts it from each of the ten face images in the data set. The outcome is a value that ranges from -255 to +255 signifying the deviation from the mean face of each pixel of the face images. The resulting shifted images are stored back into the SDRAM as signed 32-bit integers since its value range would overflow an 8-bit integer and the subsequent functions require 32-bit values. Along with calculation for the face images in the data set, the function also finds the shifted image for the test input image since this value will be needed later on during classification. The C code function can be seen below:

```c
// Find the shifted image of each person by subtracting the mean_face
// Change data storage from uint8 to int32
void shifted_images_func (Xuint8 database_num, Xuint8 input_image_num)
{
    Xuint32 memdata_in, ddr2_addr;
    Xuint16 pixel_num, image_num;
    Xuint8 image_byte[4], mean_face_byte[4];
    Xint32 shifted_image[4];
    ddr2_addr = SHIFTED_IMAGES_DDR2_BASEADDR;

    for (image_num = 0; image_num < SIZE+1; image_num++) { // +1 for input_image
        for (pixel_num = 0; pixel_num < 720; pixel_num++) {
            memdata_in = XIo_In32(MEAN_FACE_DDR2_BASEADDR + pixel_num*4);
            mean_face_byte[0] = memdata_in;
```
mean_face_byte[1] = memdata_in >> 8;
mean_face_byte[2] = memdata_in >> 16;
mean_face_byte[3] = memdata_in >> 24;

if (image_num == SIZE)  // input_image = mean_face
    memdata_in = XIo_In32(IMAGES_DDR2_BASEADDR + 3*28800
                          + input_image_num*2880 + pixel_num*4);
else
    memdata_in = XIo_In32(IMAGES_DDR2_BASEADDR + database_num*28800
                          + image_num*2880 + pixel_num*4);
image_byte[0] = memdata_in;
image_byte[1] = memdata_in >> 8;
image_byte[2] = memdata_in >> 16;
image_byte[3] = memdata_in >> 24;

shifted_image[0] = image_byte[0] - mean_face_byte[0];

XIo_Out32(ddr2_addr, shifted_image[0]);
XIo_Out32(ddr2_addr + 4, shifted_image[1]);
XIo_Out32(ddr2_addr + 8, shifted_image[2]);
XIo_Out32(ddr2_addr + 12, shifted_image[3]);
ddr2_addr = ddr2_addr + 16;
}
}

---

**Covariance Function**

The 2880 x 10 shifted images data set matrix will be multiplied by the transpose matrix of itself to calculate the covariance matrix in the function `covariance_func`. As explained in the Eigenface Face Recognition Algorithm section, the equation $C = A^TA$ is used as opposed to $C = AA^T$ to create a covariance matrix of 10 x 10 instead of 2880 x 2880, which would have been prohibitively time consuming for the FPGA to process.

In preparation for the next step, the results are stored as 32-bit floating point number. Because of C’s inability to reinterpret a floating point as an integer as required for data storage into memory, the C declaration `union` is used to properly store and read floating point numbers. The C code function can be seen below:
void covariance_func (void)
{
    Xuint32 wr_addr;
    Xuint16 height, width, pixel_num;
    Xint32 mult_row, mult_col, mult_result;

    union u_type{
        Xint32 i;
        Xfloat32 f;
    };
    union u_type result_union;

    wr_addr = BRAM_BASEADDR;
    for (height = 0; height < SIZE; height++) {
        for (width = 0; width < SIZE; width++) {
            mult_result = 0.0;
            for (pixel_num = 0; pixel_num < 2880; pixel_num++) {
                mult_row = XIo_In32(SHIFTED_IMAGES_DDR2_BASEADDR + height*2880*4 + pixel_num*4);
                mult_col = XIo_In32(SHIFTED_IMAGES_DDR2_BASEADDR + width*2880*4 + pixel_num*4);
                mult_result = mult_result + mult_row * mult_col;
            }
            result_union.f = mult_result;
            XIo_Out32(wr_addr, result_union.i);
            wr_addr = wr_addr + 4;
        }
    }
}

Jacobi Eigen Function

The key operation of the eigenface algorithm is the principal component analysis which relies on deriving the eigenvectors and eigenvalues at its core. The standard mathematical approach to finding the eigenvectors and eigenvalues of a matrix is impractical in a computer system. The function jacobi_eigen_func implements the Jacobi method which is an iterative algorithm for this task that only works on real symmetrical matrices. In this case, the input to the Jacobi method is a covariance matrix which fits the requirements.

The algorithm involves finding the maximum absolute value of the matrix and performing a Givens rotation based on the location of that value on the matrix. The Givens rotation is the matrix:

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\]
with a rotational angle $\theta$ which when multiplied with another matrix performs a counterclockwise matrix rotation of $\theta$ radians. To build the rotational matrix after the sine, cosine and $\theta$ have been determined, an identity matrix needs to be built which is handled by the function `create_id_func`. The trigonometric values are inserted in this matrix to create the Givens rotational matrix. This matrix is multiplied with the input covariance matrix to create the first iteration of the eigenvector and it is also multiplied with the unit matrix to create the first iteration of the eigenvalue. Each rotation decreases the maximum absolute value of elements in the matrix. The iteration will continue until the maximum value drops below a set threshold at which time the algorithm as converged on the closest approximation of eigenvectors and eigenvalues. See Appendix A for the MATLAB implementation of the Jacobi method. In application to this eigenface algorithm implementation, the procedure goes through 145-170 rotations to arrive at the final answer. A detailed discussion of another software implementation of the Jacobi method can be found in Press (1992).

The mathematic operations in this function uses 32-bit floating point since the many iterative calculations of sine, cosine and $\theta$ involves division of smaller and smaller values so an implementation using integer would lose too much precision or involve complex precision tracking overhead. The microBlaze processor instantiates a floating point unit (FPU) specifically to meet the requirements of this function. The C code function can be seen below:
void jacobi_eigen_func (void)
{
    Xuint32 wr_addr;
    Xuint16 rotations, r,c,p,q, v_cnt, rot_cnt;
    Xint32 theta_int, t_int, c_a_int, s_a_int, test_int, test_max,
            max, theta, abs_theta, t, c_a, s_a, test_f;

    // Necessary to read and write floating point to BRAM
    union u_type{
        Xint32 i;
        Xfloat32 f;
    };

    union u_type value, num1, num2, dem, v_value, c_a_union, s_a_union,
            s_a_neg_union,
    test_union;

    create_id_matrix_func (BRAM_BASEADDR + 400); // Create eye identity matrix (E)
    create_id_matrix_func (BRAM_BASEADDR + 1200); // Create initial eigenvector matrix (V)

    wr_addr = BRAM_BASEADDR;
    rot_cnt = 0;

    for (rotations = 0; rotations < 2000; rotations++) {
        max = 0.0;
        for (r = 0; r < SIZE - 1; r++) {
            for (c = r + 1; c < SIZE; c++) {
                value.i = XIo_In32(BRAM_BASEADDR + r*4 + c*SIZE*4);
                if (fabs(value.f) > max) {
                    p = r;
                    q = c;
                    max = fabs(value.f);
                }
            }
        }
        if (max < 1e-5) break;

        num1.i = XIo_In32(BRAM_BASEADDR + q*4 + q*SIZE*4);
        num2.i = XIo_In32(BRAM_BASEADDR + p*4 + p*SIZE*4);
        dem.i = XIo_In32(BRAM_BASEADDR + p*4 + q*SIZE*4);

        theta = (num1.f - num2.f)/(2.0*dem.f);
        // Find absolute value. abs only works on double
        if (theta < 0)
            abs_theta = -theta;

        t = 1 / (fabs(theta) + sqrt(theta*theta + 1));
        if (theta < 0)
            t = -t;

        c_a = 1/sqrt(t*t+1);
        s_a = t*c_a;
        c_a_union.f = c_a;
        s_a_union.f = s_a;
        s_a_neg_union.f = -s_a; // Needed to set s_a negative

        // Build rotation matrix P
        create_id_matrix_func (BRAM_BASEADDR + 800); // Create base matrix P = E
        XIo_Out32(BRAM_BASEADDR + 800 + p*4 + p*SIZE*4, c_a_union.i);
        XIo_Out32(BRAM_BASEADDR + 800 + q*4 + q*SIZE*4, c_a_union.i);
        XIo_Out32(BRAM_BASEADDR + 800 + p*4 + q*SIZE*4, s_a_union.i);
        XIo_Out32(BRAM_BASEADDR + 800 + q*4 + p*SIZE*4, s_a_neg_union.i);

        // P'*A, temp store in D
        matrix_mult_func(BRAM_BASEADDR + 800, BRAM_BASEADDR, BRAM_BASEADDR + 400, 1);
        // (P'*A)*P using temp storage of A in T1, store back in A
        matrix_mult_func(BRAM_BASEADDR + 400, BRAM_BASEADDR + 800, BRAM_BASEADDR, 0);
        // V*F store in temp T2
    }
}
matrix_mult_func(BRAM_BASEADDR + 1200, BRAM_BASEADDR + 800, BRAM_BASEADDR + 1600, 0);

// Copy back V result from temp to V
for (v_cnt = 0; v_cnt < 100; v_cnt++) {
    v_value.i = XIo_In32(BRAM_BASEADDR + 1600 + v_cnt*4);
    XIo_Out32(BRAM_BASEADDR + 1200 + v_cnt*4, v_value.i);
}
rot_cnt++;
}
theta_int = theta*1000000;
t_int = t * 1000000;
c_a_int = c_a * 1000000;
s_a_int = s_a * 1000000;
test_union.i = XIo_In32(BRAM_BASEADDR + 800 + q*4 + p*SIZE*4);
test_int = test_union.f * 1000000;
test_max = max * 1000000;
}

void create_id_matrix_func (Xuint32 BASEADDR)
{
    Xuint16 i,j;

    union u_type{
        Xint32 i;
        Xfloat32 f;
    };
    union u_type one, zero;
    one.f = 1.0;
    zero.f = 0.0;

    // Set matrix to all zeros
    for (i=0; i < SIZE; i++) {
        for (j=0; j < SIZE; j++) {
            if (i == j)
                XIo_Out32(BASEADDR + i*4 + j*SIZE*4, one.i);
            else
                XIo_Out32(BASEADDR + i*4 + j*SIZE*4, zero.i);
        }
    }
}

This function is also utilizes the Xilinx Block RAM (BRAM) on-chip memory for fast data access. Since the input is a 10 x 10 matrix and several of the intermediate data are also 10 x 10 matrices, the 16-kbit BRAM is partitioned into five sections of one hundred 32-bit words each to store these sets of values as needed. The remaining few bytes are
used for debugging during development. This partition along with their address ranges can be seen in the following memory map diagram:

![Memory Map Diagram]

**Sort Matrix Function**

Once the eigenvectors and eigenvalues of the face image data set are determined, the function `sort_matrix_func` reorders the eigenvectors based on the eigenvalues from largest to smallest value. The function uses a simple sorting algorithm since the eigenvalue list is only 10 items long. Once the eigenvalues have been sorted, the eigenvectors are rearranged entire column at a time based on the eigenvalue order with the first column corresponding to the highest eigenvalue. The resulting eigenvectors are multiplied by 100 and stored back into the BRAM as 32-bit signed integers, converting the implementation back to integer operations. This step is necessary since the follow on functions require a large number of multiply accumulate (MAC) operations and using floating point would increase processing time dramatically. The multiply by 100 is
chosen to gain a higher degree of precision while avoiding overflow conditions on the 32-bit signed integer type. The C code function can be seen below:

```c
void sort_matrix_func(void) {
    Xuint8 i, j, evalue_order[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, order_temp;
    Xuint16 width, pixel_num;
    Xint32 evalue_test, evec_test_int;
    Xfloat32 evalue[10], sort_temp;

    union u_type{
        Xint32 i;
        Xfloat32 f;
    };
    union u_type diag, evec;

    // Extract diagonal of eigenvalues
    for (i=0; i < SIZE; i++) {
        diag.i = XIo_In32(BRAM_BASEADDR + i*4 + i*SIZE*4);
        evalue[i] = diag.f;
    }

    // Sort Eigenvalues and save original order
    for (i = 1; i < SIZE; i++) {
        for (j = 0; j < SIZE - i; j++) {
            if(evalue[j] > evalue[j+1]) {
                sort_temp = evalue[j];
                evalue[j] = evalue[j+1];
                evalue[j+1] = sort_temp;
                order_temp = evalue_order[j];
                evalue_order[j] = evalue_order[j+1];
                evalue_order[j+1] = order_temp;
            }
        }
    }

    // Sort Eigenvector (V) according to eigenvalue order and store NV in BRAM T2 location
    for (width = 0; width < SIZE; width++) {
        for (pixel_num = 0; pixel_num < SIZE; pixel_num++) {
            evec.i = XIo_In32(BRAM_BASEADDR + 1200 + evalue_order[SIZE-width-1]*SIZE*4 + pixel_num*4);
            XIo_Out32(BRAM_BASEADDR + 1600 + width*SIZE*4 + pixel_num*4, evec.i);
        }
    }

    // Read back test
    for (i = 0; i < 100; i++) {
        evec.i = XIo_In32(BRAM_BASEADDR + 1600 + i*4);
        evec_test_int = evec.f * 100;
        XIo_Out32(BRAM_BASEADDR + 400 + i*4, evec_test_int);
    }
}
```

Matrix Multiply Function

The eigenvectors calculated by the Jacobi method is from the input covariance matrix $C = A^T A$ instead of $C = AA^T$. The function `matrix_mult_func` reconstructs the eigenvectors.
of the original matrix by multiplying with the shifted image data set to create an eigenvector matrix of 2880 x 10. The resulting 32-bit signed integers are then stored into SDRAM. The C code function can be seen below:

```c
// Matrix multiplication. Can handle transpose
void matrix_mult_func (Xuint32 A_ADDR, Xuint32 B_ADDR, Xuint32 RESULT_ADDR,
                      Xuint16 HEIGHT_A, Xuint16 WIDTH_B, int transpose)
{
    Xuint32 wr_addr;
    Xuint16 height, width, pixel_num;
    Xint32 mult_row_int, mult_col_int, mult_result_int, test_int;

    // Necessary to read and write floating point to BRAM
    union u_type{
        Xint32 i;
        Xfloat32 f;
    };
    union u_type mult_row, mult_col, mult_result;
    wr_addr = RESULT_ADDR;

    for (width = 0; width < WIDTH_B; width++) {
        for (height = 0; height < HEIGHT_A; height++) {
            mult_result_int = 0;
            for (pixel_num = 0; pixel_num < SIZE; pixel_num++) {
                if (transpose == 1) {
                    mult_row_int = XIo_In32(A_ADDR + height*SIZE*4 + pixel_num*4);
                }
                else {
                    mult_row_int = XIo_In32(A_ADDR + height*4 + pixel_num*HEIGHT_A*4);
                }
                mult_col_int = XIo_In32(B_ADDR + width*SIZE*4 + pixel_num*4);
                mult_result_int = mult_result_int + mult_row_int * mult_col_int;
            }
            XIo_Out32(wr_addr, mult_result_int);
            wr_addr = wr_addr + 4;
        }
    }
}
```

**Features Function**

The function `features_func` projects the eigenvectors onto the eigenface space by multiplying the eigenvectors with the shifted image data set again. The resulting 32-bit signed integers are then stored into SDRAM. These values represent the eigenfaces of each of the ten face images that make up a data set of an individual and will be used to
test the effectiveness of the classification algorithms. This brings an end to the training phase of the eigenface algorithm. The C code function can be seen below:

```c
void features_func (void)
{
    Xuint32 wr_addr;
    Xuint16 height, width, pixel_num;
    Xint32 mult_row, mult_col, mult_result;

    union u_type{
        Xint32 i;
        Xfloat32 f;
    };
    union u_type result_union;

    wr_addr = BRAM_BASEADDR;
    for (height = 0; height < SIZE; height++) {
        for (width = 0; width < SIZE; width++) {
            mult_result = 0;
            for (pixel_num = 0; pixel_num < 2880; pixel_num++) {
                mult_row = XIo_In32(SHIFTED_IMAGES_DDR2_BASEADDR + height*2880*4 + pixel_num*4);
                mult_col = XIo_In32(EVEC_DDR2_BASEADDR + width*2880*4 + pixel_num*4);
                mult_result = mult_result + mult_row * mult_col;
            }
            result_union.f = mult_result;
            XIo_Out32(wr_addr, mult_result);
            wr_addr = wr_addr + 4;
        }
    }
}
```

The below set of images shows the eigenface projections of five of the face images in the database that were extracted from the FPGA:
Feature Vector Function

The feature_vec_func function projects the test input image onto the eigenface space. It uses the shifted image which was calculated in an earlier function to determine the deviation from the mean face and multiplies that with the eigenface matrix. The end result is ten 32-bit signed integers representing the projection of the test image onto the ten face images in the data set which will be used for image classification. The C code function can be seen below:

```c
void feature_vec_func (void) {
    Xuint32 wr_addr;
    Xuint16 height, width, pixel_num;
    Xint32 mult_row, mult_col, mult_result;
    wr_addr = BRAM_BASEADDR+400;
    for (height = 0; height < SIZE; height++) {
        mult_result = 0;
        for (pixel_num = 0; pixel_num < 2880; pixel_num++) {
            mult_row = XIo_In32(EVEC_DDR2_BASEADDR + height*2880*4 + pixel_num*4);
            mult_col = XIo_In32(SHIFTED_IMAGES_DDR2_BASEADDR+2880*4*SIZE + pixel_num*4);
            mult_result = mult_result + mult_row * mult_col;
        }
        XIo_Out32(wr_addr, mult_result);
        wr_addr = wr_addr + 4;
    }
}
```

Similarity Score Function

The similarity_score_func function is the final function in the eigenface algorithm implementation. It subtracts the feature vector found in the previous function from the each column of the eigenface and determines the norm from the result. This is followed by a division by 100 to remove the multiplication factor that was introduced earlier to improve precision of integer division. This data represents the Euclidean distance of the test input image from each of the face images in the data set. The smaller the value means
the closer the test image is to that face image. The Euclidean distances from the ten face images are then averaged together to find the mean Euclidean distance. This number is then compared with a threshold number to determine if the test image matches the individual in the data set, is an unknown face or a non-face image all together. The discussion of the threshold setting and results achieved will be discussed in a later section. The C code function can be seen below:

```c
void similarity_score_func (Xuint8 database_num)
{
    Xuint8 classification;
    Xuint32 wr_addr, wr_addr2;
    Xuint16 height, width;
    Xint32 features, feature_vec, subtract_result;
    Xuint32 sim_score_int, sim_score_add, sim_score_avg;
    Xfloat32 add_squared, sim_score;
    wr_addr = BRAM_BASEADDR + 800;
    wr_addr2 = SIM_SCORE_DDR2_BASEADDR + database_num*(SIZE+2)*4;
    sim_score_add = 0;
    for (width = 0; width < SIZE; width++) {
        add_squared = 0.0;
        for (height = 0; height < SIZE; height++) {
            features = XIo_In32(BRAM_BASEADDR + width*SIZE*4 + height*4);
            feature_vec = XIo_In32(BRAM_BASEADDR + 400 + height*4);
            subtract_result = features - feature_vec;
            XIo_Out32(wr_addr, subtract_result); // test
            subtract_result = subtract_result/100;
            add_squared = add_squared + (float)subtract_result*(float)subtract_result;
            XIo_Out32(wr_addr+400, add_squared); // test
            wr_addr = wr_addr + 4; // test
        }
        sim_score = sqrt(add_squared);
        sim_score_int = sim_score;
        sim_score_add = sim_score_add + sim_score_int;
        XIo_Out32(wr_addr2, sim_score_int);
        wr_addr2 = wr_addr2 + 4;
    }
    sim_score_avg = sim_score_add/SIZE;
    if (sim_score_avg < 1350000)  
        classification = 1;       // face in database
    else if (sim_score_avg < 2800000)  
        classification = 2;     // human not in database
    else
        classification = 3;     // non-human
    XIo_Out32(wr_addr2, classification);
    wr_addr2 = wr_addr2 + 4;
    XIo_Out32(wr_addr2, sim_score_avg);
    wr_addr2 = wr_addr2 + 4;
}
```
Design Tradeoffs

Implementing the eigenface algorithm as presented in the paper “Eigenface for Recognition” requires a large set of data and processing power that is beyond the scope of the budget FPGA development board used for this project. Several design trade-offs had to be made in order to stay within the limited data storage and processing power of the FPGA. However each of these tradeoffs means the accuracy of the algorithm suffers so every step is carefully analyzed using MATLAB to make sure the implementation will still be reasonably accurate.

First the amount of data to be processed has to be reduced significantly. In their paper, Turk and Pentland worked with image size up to 256 x 256 and used a database of 2500 images with up to 30 images per individual for thorough analysis. This is unachievable with the FPGA board as just a few of these images will fill up the available memory storage and even if all the images are accessible by the FPGA, the total processing time will take several hours if not days and weeks. The image size for the project is therefore reduced to 60 x 48, more than 22 times smaller than original image size. A MATLAB simulation was run to compare the results of these two different image sizes and the differences was shown to be up to 13% degradation using the smaller image. This is a fairly big penalty but even with a simple test, MATLAB took several minutes to complete the test with 256 x 256 images compared to a few seconds with 60 x 48 images. This result will only worsen as the processing power of the Spartan 3A FPGA is only a tiny fraction of the Intel Core i5 1.6 GHz processor. The number of data sets and images per individual is also reduced to three sets and 10 images each mainly to reduce processing time. This move also eliminates another step in the eigenface algorithm in that
it is no longer feasible to take only the top $N$ eigenvectors for analysis since every data point is now relevant. A MATLAB comparison was performed between data sets of 20 images versus 10 and the result variation was up to +/- 8%. However since several of the operations involved matrix multiplication between data sets, the reduction in processing is actually over four times smaller (ie 10x10 compared to 20x20 matrices). This averted problem is compounded by the fact that the Jacobi method intermediate data will then be larger than the small BRAM can store and the implementation will have to take the performance hit of using SDRAM for data storage.

Another area of performance tradeoff is the comparison of integer vs. floating point. Even though the FPGA microBlaze has a floating point unit which increases computation speed by up to 400%, floating point operations are still inherently slower than integer operations. Therefore the majority of the design is converted to integer operation. Typically this means operations involving division end up having the fractional portion removed, losing a slight level of precision at each step. For the calculation of eigenvector and eigenvalue, it was decided that converting the Jacobi method to entirely integer would involve too much redesign of the algorithm and increased overhead. Given the fact the data being processed is quite small, the extra time floating point operation takes is tolerable. The reduction in the number of images per data set to 10 ultimately enabled the Jacobi method to be processed using floating point without too much sacrifice. After the eigenvectors have been calculated and sorted, the design is reverted back to integer operation. However a multiply by 100 was necessary because the design would lose significant precision without it. This number is chosen since it was the largest base-10 multiplier possible without overflowing the integer in subsequent operations. The results
of the classification functions are then divided by 100 to arrive at the actual values. MATLAB simulation running all floating point is compared to the result of the FPGA implementation using this mixed integer/floating point design shows that the results are within 3% of each other.

When all the penalties are added together the variation of the FPGA implementation from the Turk and Pentland algorithm can be up to 24%, a sizeable performance degradation. Every effort is made to stay faithful to the original eigenface algorithm within the confines of the Spartan 3A FPGA development board. These sacrifices allowed the original algorithm to be implemented while still maintaining all its core functions.

It is possible to improve performance by a few more percentage by optimizing the design but significant improvements can only be realized by moving to a more capable FPGA development board with increased memory capacity and a more powerful FPGA such as a Xilinx Virtex-5. A quick check of the Xilinx website shows a suitable system for this task would cost $1200, which is considerably more expensive than the $150 board this project uses.
FPGA IMPLEMENTATION RESULTS

The eigenface algorithm implementation is tested using a database of three individuals, each with 10 face images. The individuals are two male and one female subject as shown in Appendix C. Ten test input images are used to test the effectiveness of the algorithm: One new face image each of the three individuals in the database, two unknown males, two unknown females, one cartoon face image and two completely non-face images. The complete test results are shown in Appendix D. These images are carefully selected to be a representative of the different scenarios a face recognition algorithm may face. The test result data shows the Euclidean distance of each test input image from the three data sets. The identity heading states how the test input is classified against the database: matched face, unknown face or non-face object. The closest match and best score show which face image is the closest match and its value. The average score shows the mean of the test input image distances from the ten face images in one data set. As can be seen in the results tables, it is possible to get a recognized face match against a face images in the data set, however the average shows that in fact a false positive identification has not occurred. Therefore it is important to have multiple images of each person to build a more accurate database. Appendix D also displays how the results from MATLAB simulation and FPGA implementation compare to each other. Due to the loss of fractional precision from using integer over floating point, the FPGA results are always lower than MATLAB. However the average difference is around 2% and it does not skew the results by much.
From the test results 100% accurate classification can be accomplished by setting the thresholds for classification to be less than 1.35 billion for a recognized face, between 1.35 billion and 2.8 billion for an unknown human face and greater than 2.8 billion for a non-face image. These threshold numbers are obtained experimentally by running the ten input test images against the three data sets of individuals along with other unpublished images. By looking at the results it clear that the algorithm works very well at distinguishing non-face images from human faces but some of the face classifications are a little too close to false positive result. Also as these threshold numbers are experimentally determined, using different data sets of individuals and different test input images may yield somewhat different thresholds. Testing with different images also show that the algorithm is sensitive to light variation and face orientation even though efforts are made to try to control these variables as much as possible. The below table summarizes the classification thresholds:

<table>
<thead>
<tr>
<th>Classification of input x</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recognized face</td>
<td>$x &lt; 1.35$ billion</td>
</tr>
<tr>
<td>Unknown face (not in database)</td>
<td>$1.35$ billion $&lt; x &lt; 2.8$ billion</td>
</tr>
<tr>
<td>Non-face image</td>
<td>$2.8$ billion $&lt; x$</td>
</tr>
</tbody>
</table>

As noted in the Design Tradeoff section, the many design changes necessary to operate in the FPGA introduced a fairly large degree of uncertainty. This shows that while the design implementation proves the concept of eigenface algorithm can be carried out on an FPGA, the results are not as solid as presented by Turk and Pentland.

The other area of performance is the processing time required to complete the algorithm. In the current form, to compare one test input image against one data sets of 10 face
images takes ~13.6 seconds. Of this time, the longest process is the Jacobi method calculation which takes 5.3 seconds to perform. Another 7.7 seconds are taken by the three large size matrix multiplication functions in the design. Eigenface classification takes less than 0.2 seconds to complete. The slow UART interface takes 8.6 seconds to load all 30 face images of the database and 10 test input images to the SDRAM. This combined with running one test input image against the three data sets of 10 face images each means the entire procedure from start to end will take 49.5 seconds. The below table summarizes the duration of time the FPGA implementation takes to perform the algorithm on three data sets:

<table>
<thead>
<tr>
<th>Implementation Process</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load images through UART</td>
<td>8.7 s</td>
</tr>
<tr>
<td>Mean and shifted images functions</td>
<td>1.2 s</td>
</tr>
<tr>
<td>Covariance matrix function</td>
<td>7.5 s</td>
</tr>
<tr>
<td>Jacobi method and sort functions</td>
<td>15.9 s</td>
</tr>
<tr>
<td>Matrix multiplication function</td>
<td>8.4 s</td>
</tr>
<tr>
<td>Features function</td>
<td>7.2 s</td>
</tr>
<tr>
<td>Classification</td>
<td>0.6 s</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>49.5 s</strong></td>
</tr>
</tbody>
</table>

This process time represents the limit of what can be accomplished with the current design implementation. The UART has been increased to the limit of what the cable can support at 115,200 bps. The microBlaze system clock of 100 MHz and SDRAM clock of 200 MHz are also the maximum of what the FPGA can support without introducing
errors. As a matter of fact they are being run at higher clock speed than Xilinx recommendations.

The current processing time already represents a sizeable improvement over the first working version of the design which took over 6 minutes to complete. Design and architectural optimization as well as boosting the system clock have improved the performance up to this stage. Further improvements in processing time can probably be achieved by optimizing some of the function calls, however without significant design architectural changes the improvements will likely be less than 5 seconds total. The next section shows some of the steps that may be taken to improve performance as part of a follow on project.

**Future Improvements**

The ideal scenario would be to build a face recognition system that has a large database which can detect the presence of a face and perform face recognition in real time and match a person with high degree of accuracy. An example of how such a system may be used is at an airport security line which checks for known persons of interest while the passengers are going about their regular check in. Although this would still be beyond the scope of a continuation research project, many steps can be taken to work toward this ideal goal.

As stated in the Design Tradeoffs section, many sacrifices are made implementing the eigenface algorithm in the FPGA that performance may degrade by up to 24%. One obvious improvement will be to acquire a more advanced FPGA development board with
greater memory capacity (including SRAM) and perhaps an on-board digital signal processor (DSP). Such a system will allow larger images, more data sets of individuals with more faces per set. Each of these will improve the accuracy of the algorithm. Having a SRAM present will also improve processing time by allowing for greater local caching as opposed to always relying on reading and writing to the SDRAM directly using inefficient single word access. SDRAM by nature have large latency and work best in burst mode. The presence of a DSP will enable the system to process floating point numbers much faster than the FPU of microBlaze can accomplish. The slow data transfer rate of the UART can also be addressed by using USB or Ethernet for communication with the FPGA. All these improvements will allow the FPGA design to approach the same performance as that achieved by Turk and Pentland while at the same time reducing the processing time compared to the current implementation.

The improvements mentioned so far do not fundamentally change the design implementation of the eigenface algorithm. Other design changes can be done to improve processing capability even while using the same budget FPGA development board. For example the Jacobi eigenvector function is the single most time consuming process in the design. The reason is because it is performed using floating point and several hundreds of iterations are necessary to converge on the closest approximation values. One solution is to redesign the Jacobi method function from scratch with an FPGA logic module which uses a parallel CORDIC architecture. CORDIC stands for coordinate rotation digital computer which enables fast calculations of trigonometry as required by the Jacobi method algorithm. This approach is fully detailed in Bravo, Jimenez, Mazo, Lazaro, and Gardel (2006).
Another design improvement will be to replace the simplistic Euclidean distance based threshold for classification. Once the training phase of the eigenface algorithm is complete, the classification phase is a completely independent process and can be easily replaced with a more effective algorithm. Even Turk and Pentland described an approach for classification based on neural network. More research will be required to identify and implement a more modern and sophisticated algorithm for image classification.

By implementing all the improvements mentioned in this section, it is hoped that a system can be designed which can approach real time processing (less than 2s per individual data set) and at the same time improve the accuracy of the algorithm. However, it can also be argued that the eigenface algorithm first published 22 years ago is archaic by modern standard and it should only serve as an educational tool to understand the problems and tradeoffs in face recognition. This line of thinking would then require even more research to identify an algorithm to implement (or a combination of algorithms) as well as re-evaluate what a good platform would be to carry out the design.

**Design Challenges**

Several design challenges were encountered during this project that at times brings up questions over the feasibility of implementing the eigenface face recognition algorithm in an inexpensive FPGA platform. Several of these challenges ended up having simple solutions, which others involved significant research and experiments before reaching a
more intricate resolution. Here are some of the more notable challenges that were confronted.

First major problem encountered was the inability to reproduce the eigenface face recognition results in MATLAB. This was in spite of the fact that the algorithm appeared to work and it behaves exactly as Turk and Pentland described. In the end, the focus was placed on the images used and the realization that their variability had to be controlled more due to the limitations of the eigenface algorithm. Therefore the proper creation of the image database became a very important task.

The memory interface also presented some problems which can be mainly attributed to the lack of experience in microprocessor coding. The numerous data access across different addresses, data size changes (8-bit to 32-bit) and data type changes (integer to floating and back) led to several instances of incorrect data reads and writes. These problems persisted from the start of the project to the end since every stage of the design presented a different data setup which led to a lot of confusions. Though highly annoying and time wasting, this was never an issue that stopped development for significant period of time.

Once the entire algorithm was implemented in FPGA, it was discovered that it look over 6 minutes to complete, which was unacceptable result. Several steps were taken to reduce this time down to around 50 seconds. These including maximizing the hardware clock speed, introducing a BRAM for caching of temporary data and changing some SDRAM reads and writes to be burst mode when possible.
Fortunately the many challenges faced were not detrimental enough to prevent the successful implementation of the eigenface algorithm. However it does show that persistence and allocating plenty of time for each task will go a long way toward completing a project.
CONCLUSION

This project has achieved its goal of understanding of the fundamentals of face recognition technology by studying one of the first practical and effective algorithms and applying that knowledge to MATLAB simulation and FPGA implementation. It demonstrates that a calculation intensive face recognition algorithm can be implemented on a budget FPGA development board given enough design tradeoffs and that it is capable of performing this task in an acceptable time frame for educational purposes.

This result was not a given at the start of the project and there were many doubts as to whether the FPGA had enough processing power and resources to complete this job and do it in a timely manner. Along with achieving a working implementation in the FPGA, many of the strengths and weaknesses of the eigenface algorithm are verified.

This experience has built a foundation for further research into face recognition technology and identified key areas to improve on the design implementation that can be carried out as part of a continuation research project in this field.
REFERENCES


%% Training Algorithm

num_images = size(images,2);

% Compute the average vector
% row vector containing the mean of each column of images
mean_face = mean(images, 2);

% Compute difference with average for each vector
for i = 1:num_images
    shifted_images(:,i) = images(:,i) - mean_face;
end

% Get the patternwise (num_images x num_images) covariance matrix
L = shifted_images' * shifted_images;

%---------------------------------------
% Jacobi Method for Eigenvectors and Eigenvalues

A = single(L);

Size = size(A,1);
E = eye(Size);
V = E; % Start with unit matrix

rot_count = 0;
for Rotations = [1:500] % Limit number of rotations
    % Find maximum off-diagonal element
    Max = 0;
    ptr = 1;
    for r = 1:Size-1
        for c = r+1:Size
            if abs(A(r,c)) > Max % New Max found
                p = r;
                q = c;
                Max = abs(A(r,c));
            end
            test(ptr) = Max;
            ptr = ptr +1;
        end
    end
    % Compare Max with working precision
    if Max < 1e-5
        break % A is diagonalized, stop now
    end
end
% Find sin and cos of rotation angle
theta = single( (A(q,q)-A(p,p))/(2*A(p,q)) );
t = single( 1/(abs(theta)+sqrt(theta^2+1)) );

if theta < 0
    t = -t;
end

c_a = single( 1/sqrt(t^2+1) );
s_a = single(t*c_a);

% Build rotation matrix
P = E;
P(p,p) = c_a;
P(q,q) = c_a;
P(p,q) = s_a;
P(q,p) = -s_a;

% Do rotation
A = single(P'*A*P);
V = single( V*P );

rot_count = rot_count +1;
end

D = single( diag(spdiags(A,0)) );  % Return diagonal
evectors = V;
evalues = D;

%-----------------------------------------
% Sort the vectors/values according to size of eigenvalue
V1 = evectors;
D1 = evalues;
dvec2 = diag(D1);
NV = zeros(size(V1));
[dvec,index_dv2] = sort(dvec2);
index_dv = flipud(index_dv2);
for i = 1:size(D1,1)
    ND(i,i) = D1(index_dv(i),index_dv(i));
    NV(:,i) = V1(:,index_dv(i));
end;
evectors = NV;
evalues = ND;

% Convert the eigenvectors of A'*A into eigenvectors of A*A'
evectors = shifted_images * evectors;
features = evectors' * shift;
num_eigenfaces = num_images;
%%% Classification Algorithm

% calculate the similarity of the input to each training image
feature_vec = evectors' * (input_image(:) - mean_face);
similarity_score(:, dataset) = arrayfun(@(n) norm(features(:,n) - feature_vec), 1:num_images);

% find the image with the highest similarity
[match_score, match_ix] = min(similarity_score(:, dataset));
average_score(dataset) = mean(similarity_score(:, dataset));
### Appendix B: FPGA Utilization

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Flip Flops</td>
<td>4,793</td>
<td>11,776</td>
<td>40%</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>5,931</td>
<td>11,776</td>
<td>50%</td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>4,818</td>
<td>5,888</td>
<td>81%</td>
</tr>
<tr>
<td>Number of Slices containing only related logic</td>
<td>4,818</td>
<td>4,818</td>
<td>100%</td>
</tr>
<tr>
<td>Number of Slices containing unrelated logic</td>
<td>0</td>
<td>4,818</td>
<td>0%</td>
</tr>
<tr>
<td>Total Number of 4 input LUTs</td>
<td>6,161</td>
<td>11,776</td>
<td>52%</td>
</tr>
<tr>
<td>Number used as logic</td>
<td>5,396</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as a route-thru</td>
<td>230</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as 16x1 RAMs</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used for Dual Port RAMs</td>
<td>342</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as Shift registers</td>
<td>189</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>90</td>
<td>372</td>
<td>24%</td>
</tr>
<tr>
<td>IOB Flip Flops</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOB Master Pads</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOB Slave Pads</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of ODDR2s used</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of BUFGMUXs</td>
<td>6</td>
<td>24</td>
<td>25%</td>
</tr>
<tr>
<td>Number of DCMs</td>
<td>2</td>
<td>8</td>
<td>25%</td>
</tr>
<tr>
<td>Number of BSCANs</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Number of BSCAN_SPARTAN3As</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Number of MULT18X18SIOs</td>
<td>7</td>
<td>20</td>
<td>35%</td>
</tr>
<tr>
<td>Number of RAMB16BWEs</td>
<td>19</td>
<td>20</td>
<td>95%</td>
</tr>
<tr>
<td>Number of RPM macros</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Fanout of Non-Clock Nets</td>
<td>3.25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C: Face Image Database

**Database #1**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**Database #2**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**Database #3**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
# Appendix D: FPGA Test Results

## Database #1 Results (Colored row shows input match to database)

<table>
<thead>
<tr>
<th>Test Image</th>
<th>MATLAB</th>
<th>FPGA</th>
<th>Difference from MATLAB</th>
</tr>
</thead>
</table>
| ![Image 1](image1.jpg) | Identity: Matched Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 238 million  
Avg Score: 1049 million | Identity: Matched Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 231 m  
Avg Score: 1033 m | Identity: Same  
Closest Match: Same Image  
Best Score: -2.9%  
Avg Score: -1.3% |
| ![Image 2](image2.jpg) | Identity: Unknown Face  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 1149 m  
Avg Score: 2041 m | Identity: Unknown Face  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 1121 m  
Avg Score: 2014 m | Identity: Same  
Closest Match: Same Image  
Best Score: -2.4%  
Avg Score: -1.3% |
| ![Image 3](image3.jpg) | Identity: Unknown Face  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 1433 m  
Avg Score: 2365 m | Identity: Unknown Face  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 1399 m  
Avg Score: 2299 m | Identity: Same  
Closest Match: Same Image  
Best Score: -2.4%  
Avg Score: -2.8% |
| ![Image 4](image4.jpg) | Identity: Unknown Face  
Closest Match: 6\textsuperscript{th} Image  
Best Score: 1028 m  
Avg Score: 1405 m | Identity: Unknown Face  
Closest Match: 6\textsuperscript{th} Image  
Best Score: 1016 m  
Avg Score: 1377 m | Identity: Same  
Closest Match: Same Image  
Best Score: -1.2%  
Avg Score: -2.0% |
| ![Image 5](image5.jpg) | Identity: Unknown Face  
Closest Match: 5\textsuperscript{th} Image  
Best Score: 1093 m  
Avg Score: 1411 m | Identity: Unknown Face  
Closest Match: 5\textsuperscript{th} Image  
Best Score: 1086 m  
Avg Score: 1383 m | Identity: Same  
Closest Match: Same Image  
Best Score: -0.6%  
Avg Score: -2.0% |
| ![Image 6](image6.jpg) | Identity: Unknown Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 910 m  
Avg Score: 1403 m | Identity: Unknown Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 887 m  
Avg Score: 1374 m | Identity: Same  
Closest Match: Same Image  
Best Score: -2.5%  
Avg Score: -2.1% |
| ![Image 7](image7.jpg) | Identity: Unknown Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 1012 m  
Avg Score: 1457 m | Identity: Unknown Face  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 997 m  
Avg Score: 1423 m | Identity: Same  
Closest Match: Same Image  
Best Score: -1.5%  
Avg Score: -2.3% |
| ![Image 8](image8.jpg) | Identity: Non-Face Object  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 2621 m  
Avg Score: 3107 m | Identity: Non-Face Object  
Closest Match: 4\textsuperscript{th} Image  
Best Score: 2595 m  
Avg Score: 3041 m | Identity: Same  
Closest Match: Same Image  
Best Score: -1.0%  
Avg Score: -2.1% |
| ![Image 9](image9.jpg) | Identity: Non-Face Object  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 2358 m  
Avg Score: 3044 m | Identity: Non-Face Object  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 2325 m  
Avg Score: 2955 m | Identity: Same  
Closest Match: Same Image  
Best Score: -1.4%  
Avg Score: -2.9% |
| ![Image 10](image10.jpg) | Identity: Non-Face Object  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 2844 m  
Avg Score: 3714 m | Identity: Non-Face Object  
Closest Match: 1\textsuperscript{st} Image  
Best Score: 2801 m  
Avg Score: 3629 m | Identity: Same  
Closest Match: Same Image  
Best Score: -1.5%  
Avg Score: -2.3% |
<table>
<thead>
<tr>
<th>Test Image</th>
<th>MATLAB</th>
<th>FPGA</th>
<th>Difference from MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1375 million</td>
<td>Best Score: 1348 m</td>
<td>Best Score: -0.0%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 2166 million</td>
<td>Avg Score: 2136 m</td>
<td>Avg Score: -1.4%</td>
</tr>
<tr>
<td>2</td>
<td>Identity: Matched Face</td>
<td>Identity: Matched Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 2(^{nd}) Image</td>
<td>Closest Match: 2(^{nd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 502 m</td>
<td>Best Score: 490 m</td>
<td>Best Score: -2.3%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 1200 m</td>
<td>Avg Score: 1174 m</td>
<td>Avg Score: -2.2%</td>
</tr>
<tr>
<td>3</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 2(^{nd}) Image</td>
<td>Closest Match: 2(^{nd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 954 m</td>
<td>Best Score: 935 m</td>
<td>Best Score: -2.0%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 1445 m</td>
<td>Avg Score: 1414 m</td>
<td>Avg Score: -2.1%</td>
</tr>
<tr>
<td>4</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1861 m</td>
<td>Best Score: 1807 m</td>
<td>Best Score: -2.9%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 2588 m</td>
<td>Avg Score: 2549 m</td>
<td>Avg Score: -1.5%</td>
</tr>
<tr>
<td>5</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1278 m</td>
<td>Best Score: 1269 m</td>
<td>Best Score: -0.7%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 2163 m</td>
<td>Avg Score: 2124 m</td>
<td>Avg Score: -1.8%</td>
</tr>
<tr>
<td>6</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1861 m</td>
<td>Best Score: 1846 m</td>
<td>Best Score: -0.8%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 2648 m</td>
<td>Avg Score: 2587 m</td>
<td>Avg Score: -2.3%</td>
</tr>
<tr>
<td>7</td>
<td>Identity: Unknown Face</td>
<td>Identity: Unknown Face</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1936 m</td>
<td>Best Score: 1895 m</td>
<td>Best Score: -2.1%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 2678 m</td>
<td>Avg Score: 2635 m</td>
<td>Avg Score: -1.6%</td>
</tr>
<tr>
<td>8</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: 3(^{rd}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 4172 m</td>
<td>Best Score: 4051 m</td>
<td>Best Score: -2.9%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 4950 m</td>
<td>Avg Score: 4816 m</td>
<td>Avg Score: -2.7%</td>
</tr>
<tr>
<td>9</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 6(^{th}) Image</td>
<td>Closest Match: 6(^{th}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1748 m</td>
<td>Best Score: 1739 m</td>
<td>Best Score: -0.5</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 3791 m</td>
<td>Avg Score: 3727 m</td>
<td>Avg Score: -1.7%</td>
</tr>
<tr>
<td>10</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Non-Face Object</td>
<td>Identity: Same</td>
</tr>
<tr>
<td></td>
<td>Closest Match: 6(^{th}) Image</td>
<td>Closest Match: 6(^{th}) Image</td>
<td>Closest Match: Same</td>
</tr>
<tr>
<td></td>
<td>Best Score: 1594 m</td>
<td>Best Score: 1581 m</td>
<td>Best Score: -0.8%</td>
</tr>
<tr>
<td></td>
<td>Avg Score: 3769 m</td>
<td>Avg Score: 3671 m</td>
<td>Avg Score: -2.6%</td>
</tr>
</tbody>
</table>
Database #3 Results (Colored row shows input match to database)

<table>
<thead>
<tr>
<th>Test Image</th>
<th>MATLAB</th>
<th>FPGA</th>
<th>Difference from MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 844 m Average Score: 1712 million</td>
<td><img src="image2" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 837 m Average Score: 1669 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -0.8% Average Score: -2.5%</td>
</tr>
<tr>
<td>2</td>
<td><img src="image3" alt="Image" /> Identity: Unknown Face Closest Match: 4&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 882 m Average Score: 1468 m</td>
<td><img src="image4" alt="Image" /> Identity: Unknown Face Closest Match: 4&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 872 m Average Score: 1426 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -1.1% Average Score: -2.8%</td>
</tr>
<tr>
<td>3</td>
<td><img src="image5" alt="Image" /> Identity: Matched Face Closest Match: 6&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1019 m Average Score: 1334 m</td>
<td><img src="image6" alt="Image" /> Identity: Matched Face Closest Match: 6&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1003 m Average Score: 1308 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -1.5% Average Score: -1.9%</td>
</tr>
<tr>
<td>4</td>
<td><img src="image7" alt="Image" /> Identity: Unknown Face Closest Match: 9&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1249 m Average Score: 2104 m</td>
<td><img src="image8" alt="Image" /> Identity: Unknown Face Closest Match: 9&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1218 m Average Score: 2080 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -2.5% Average Score: -1.1%</td>
</tr>
<tr>
<td>5</td>
<td><img src="image9" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 972 m Average Score: 2029 m</td>
<td><img src="image10" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 946 m Average Score: 1995 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -2.7% Average Score: -1.7%</td>
</tr>
<tr>
<td>6</td>
<td><img src="image11" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 1220 m Average Score: 2309 m</td>
<td><img src="image12" alt="Image" /> Identity: Unknown Face Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 1191 m Average Score: 2256 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -2.3% Average Score: -2.3%</td>
</tr>
<tr>
<td>7</td>
<td><img src="image13" alt="Image" /> Identity: Unknown Face Closest Match: 9&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1431 m Average Score: 2302 m</td>
<td><img src="image14" alt="Image" /> Identity: Unknown Face Closest Match: 9&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 1405 m Average Score: 2274 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -1.8% Average Score: -1.2%</td>
</tr>
<tr>
<td>8</td>
<td><img src="image15" alt="Image" /> Identity: Non-Face Object Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 3149 m Average Score: 4136 m</td>
<td><img src="image16" alt="Image" /> Identity: Non-Face Object Closest Match: 1&lt;sup&gt;st&lt;/sup&gt; Image Best Score: 3073 m Average Score: 4070 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -2.4% Average Score: -1.6%</td>
</tr>
<tr>
<td>9</td>
<td><img src="image17" alt="Image" /> Identity: Non-Face Object Closest Match: 3&lt;sup&gt;rd&lt;/sup&gt; Image Best Score: 3150 m Average Score: 3775 m</td>
<td><img src="image18" alt="Image" /> Identity: Non-Face Object Closest Match: 3&lt;sup&gt;rd&lt;/sup&gt; Image Best Score: 3093 m Average Score: 3666 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -1.8% Average Score: -2.9%</td>
</tr>
<tr>
<td>10</td>
<td><img src="image19" alt="Image" /> Identity: Non-Face Object Closest Match: 4&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 2165 m Average Score: 2901 m</td>
<td><img src="image20" alt="Image" /> Identity: Non-Face Object Closest Match: 4&lt;sup&gt;th&lt;/sup&gt; Image Best Score: 2154 m Average Score: 2857 m</td>
<td>Identity: Same Closest Match: Same Image Best Score: -0.5% Average Score: -1.5%</td>
</tr>
</tbody>
</table>