GPU Acceleration of k-Nearest Neighbor Search in Face Classifier based on Eigenfaces

by

Jennifer Dawn Rouan

(Under the direction of Thiab R. Taha)

Abstract

Face recognition is a specialized case of object recognition, and has broad applications in security, surveillance, identity management, law enforcement, human-computer interaction, and automatic photo and video indexing. Because human faces occupy a narrow portion of the total image space, specialized methods are required to identify faces based on subtle differences. One such method is the Eigenfaces classifier, a holistic statistical method which significantly reduces the dimensionality of the search space.

Despite the dimensionality reduction, the k-nearest neighbor (kNN) search remains a bottleneck in this application as well as most others that use it. The kNN search is both computationally complex and highly data-parallel, making it a candidate for acceleration on graphics hardware.

I present an implementation of an Eigenfaces classifier with a portion of the kNN search accelerated on an Nvidia Tesla K20X GPU with 2688 cores.

Index words: Face recognition, Eigenfaces, k-nearest neighbor search, General-purpose GPU programming

GPU Acceleration of k-Nearest Neighbor Search in Face Classifier based on Eigenfaces

by

Jennifer Dawn Rouan

B.S., University of Georgia, 2012

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2014

GPU Acceleration of k-Nearest Neighbor Search in Face Classifier based on Eigenfaces

by

Jennifer Dawn Rouan

Acknowledgments

I would like to acknowledge and thank my advisory committee, Professors Thiab Taha, Hamid Arabnia, and Rod Canfield, for their invaluable support and guidance. I would also like to thank Professors Khaled Rasheed and Suchi Bhandarkar for providing feedback and advice.

Portions of this research were made possible by a grant from Nvidia, in support of the UGA CUDA Teaching Center.

I owe a debt of gratitude to my husband and partner in life, Grady Smith, for his unwavering support and encouragement.

Contents

Chapter 1

Introduction

1.1 Motivation: Face Recognition

The tasks of face recognition, primarily verification (one-to-one) and identification (one-to-many), have broad applications in security, surveillance, identity management, law enforcement, human-computer interaction, and automatic photo and video indexing. Face recognition is a specialized case of object recognition. Because most human faces appear roughly alike, they occupy only a small, dense portion of the image space, making it nearly impossible for traditional object-recognition techniques to distinguish between faces based on subtle differences[1].

Face recognition methods vary depending on the application and image acquisition method. Surveillance at a crowded event may use live video capture, and must operate in near real time on low-resolution images with inconsistent lighting and facial orientation. Static intensity images such as passport photos, ID photos, and mug shots provide higher-resolution images with less variability in factors such as lighting angle and shooting distance. High security applications may require the subject to remain still for 3D or infrared image capture, using even more data to represent each face. A detailed survey of methods in all three classes is presented in "A Survey of Face Recognition Techniques"[1].

Recognition techniques for the second class, static intensity images, fall into two categories: feature-based, which reduce faces to graphs of extracted features (e.g., eyes, mouth) and distances between them, and holistic, which identify faces based upon global descriptions of the entire image. Feature-based methods have the potential to be less sensitive to variables such as image size or light intensity, but in practice suffer from the difficulty of extracting the features automatically or even knowing which features will provide the most value for identification[1].

Because holistic methods use the entire image to describe the face, they suffer from the curse of high dimensionality, especially as the image resolution increases. The dimensionality can be reduced using Principal Component Analysis (PCA) by reducing the image set to a small collection of eigenpictures and approximately reconstructing each image using projections which describe the differences between the eigenpictures and the original image[2]. Those projections can be used as feature vectors to recognize faces by subject or other classifications such as gender or ethnicity[3].

1.2 Eigenfaces

Introduced in 1991 by Turk and Pentland, the Eigenfaces concept facilitates dimensionality reduction in face recognition by decomposing face images into a small set of feature images called "Eigenfaces". An $N \times N$ face image may be described as a vector of dimension N^2, or a point in N^2-dimensional space. However, since most face images appear similar to each other, they will not be randomly distributed across this enormous image space. Using PCA, the faces can be described in a lower-dimensional subspace, called the face space[3].

Given a training set of m images of size $N \times N$, a classifier learns about the faces as follows:

1. Normalize (subtract from the mean image) and flatten each image onto a vector of length N^2

2. Create an $m \times N^2$ matrix M using the m image vectors as rows

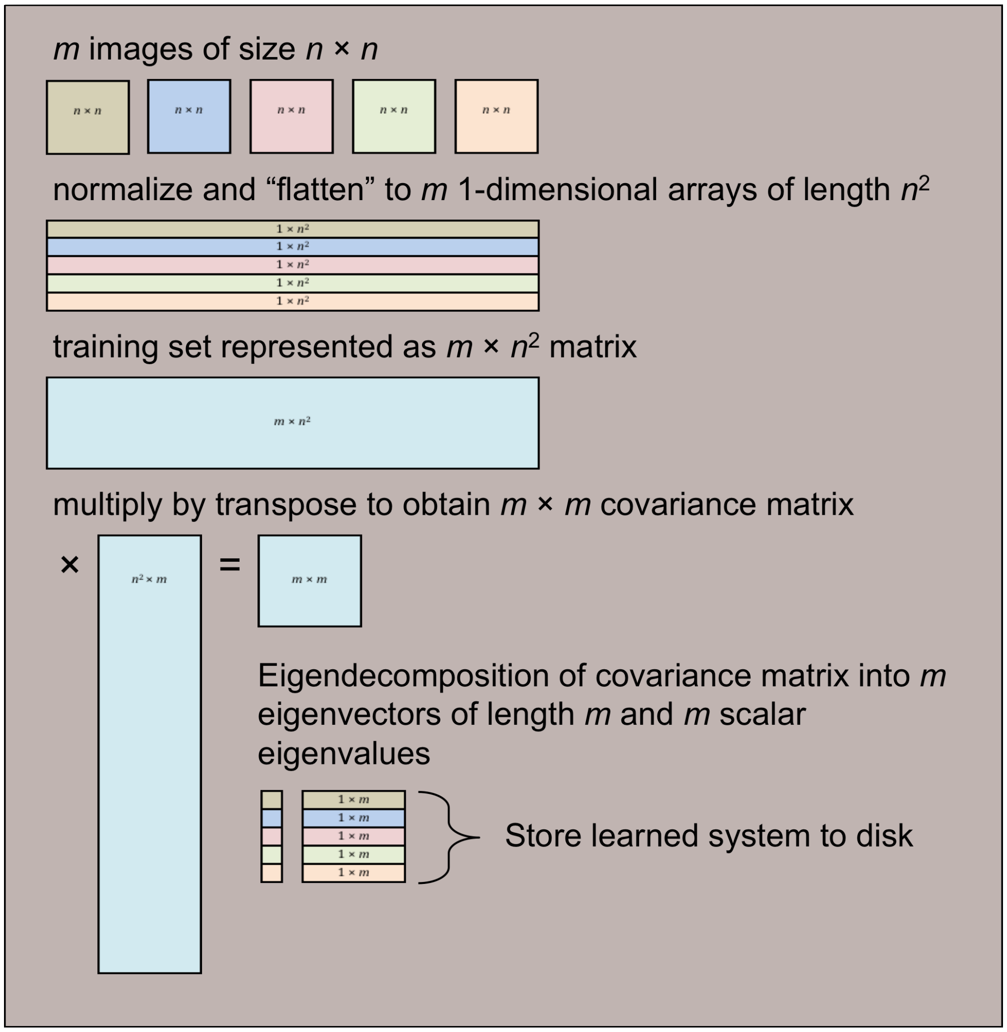
3. Build the covariance matrix $M \times M^T$

4. Decompose the covariance matrix into eigenvectors and eigenvalues

5. Normalize the eigenvectors and project them to the image space

6. Store the trained system for validation and classification

Figure 1.1 presents a visualization of the dimensionality reduction provided by these steps.



m images of size *n* × *n*
n × n
normalize and “flatten” to *m* 1-dimensional arrays of length *n*²
1 × n²
training set represented as *m* × *n*² matrix
m × n²
multiply by transpose to obtain *m* × *m* covariance matrix
×
n² × m
=
m × m
Eigendecomposition of covariance matrix into *m* eigenvectors of length *m* and *m* scalar eigenvalues
1 × m
Store learned system to disk

Figure 1.1: Visualization of Using Eigendecomposition to Reduce Dimension

The system may be tested and faces may be classified by computing the eigenweights of the query images based on the stored projections and then conducting a k-nearest neighbor (kNN) search against the learned images, as shown in Figure 1.2.

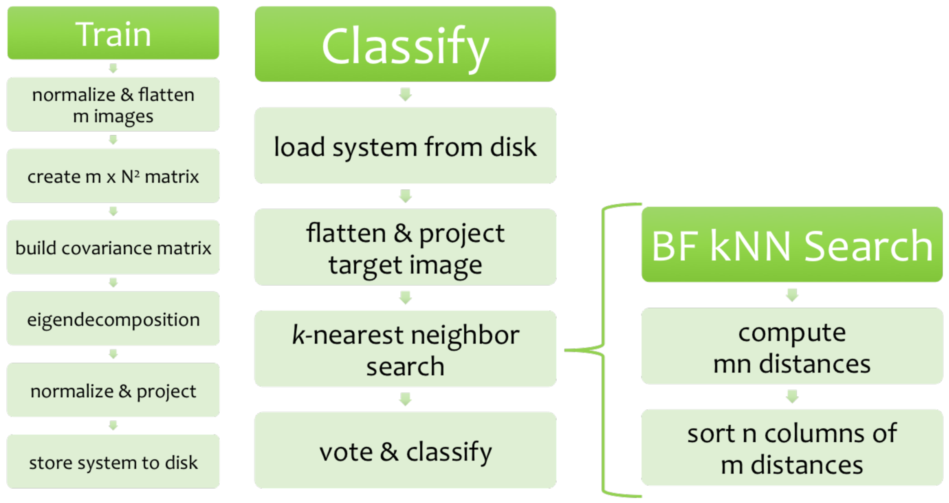


Figure 1.2: Summary of Eigenfaces Classification System

1.3 k-Nearest Neighbor Search

The kNN search is usually slow because it is a computationally intensive process. The computation of the distance between two points requires many basic operations, and the resolution of the kNN search grows polynomially with the size of the point sets. Frequently, this problem is the bottleneck of these applications and proposing a fast kNN search is crucial[4].

Problem statement

Let $R = \{r_1, r_2, \ldots, r_m\}$ be a set of m reference points in d-dimensional space, and let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n query points in the same space. The kNN search problem consists of identifying the k nearest neighbors of each query point $q_i \in Q$ in the reference set R, given a specific distance metric.

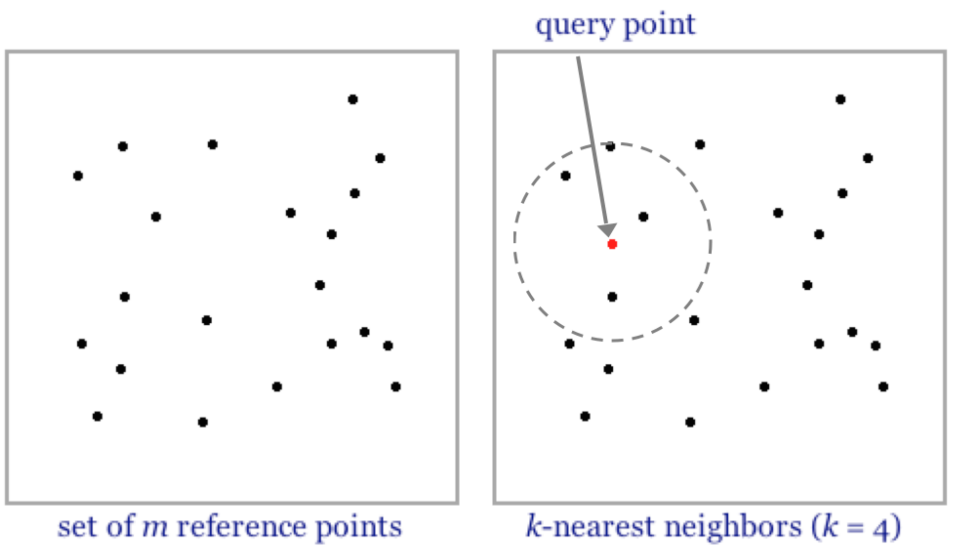


Figure 1.3: k-Nearest Neighbor Search

Brute force approach

The brute force method of kNN search is an exhaustive search that consists of the following algorithm[4]: For n query points q_i,

1. compute every pairwise distance between points q_i and r_j with $j \in [1, m]$
2. sort the computed distances
3. select the k reference points providing the smallest distances

This algorithm, while simple, is highly computationally complex. The time complexity for step 1 is $O(nmd)$ for the nm distances computed. For step 2, it is $O(nm \log m)$ for the n sorts performed[4]. It should be noted that step 3 is trivial and can be done in constant time.

Optimization with efficient data structures

There have been several approaches to accelerating kNN search using efficient data structures. A common method of optimization is to reduce the number of distance calculations by partitioning the space using a tree-like structure such as a k-d tree[4]. A k-d tree is a hierarchical spatial partitioning data structure used to organize objects in d-dimensional space. The k-d tree partitions points into axis-aligned cells called nodes. For each internal node of the tree, a hyperplane is formed by an axis and a split value, for example the median value. This hyperplane partitions all points at each parent node into left and right child nodes[5].

A k-d tree for a search set of nm d-dimensional points can be built in $O(dnm \log nm)$ time. The time complexity of the search is $O(d \log nm)$[6].

Another popular approach is to use probabilistically approximate algorithms when certain approximation is acceptable and beneficial. At the core of both of these approaches remains the kNN search over a smaller data set[7]. The computation time required by the kNN search continues to be the bottleneck of methods based on kNN[8]. Furthermore, in high-dimensional space, algorithms based on space partitions suffer from high overhead costs due to sparse and non-uniform data distributions[7].

1.4 General-purpose GPU Programming

A graphics processing unit (GPU) is a dedicated graphics rendering device for a personal computer, workstation, or game console[4]. GPUs are optimized for massively parallel computing using the Single Instruction Multiple Data (SIMD) model of parallelism. GPU architecture differs from that of a traditional central processing unit (CPU) because it is optimized for maximum throughput over minimum latency. The trade-off is that more of the chip's surface area is dedicated to processor cores at the expense of most of the on-board memory, as shown in Figure 1.4[9].

General-purpose GPU programming (GPGPU) is the technique of using a GPU to perform the computations usually handled by the CPU[8]. Many scientific applications, such as computer vision and machine learning, have an inherent data independence that maps

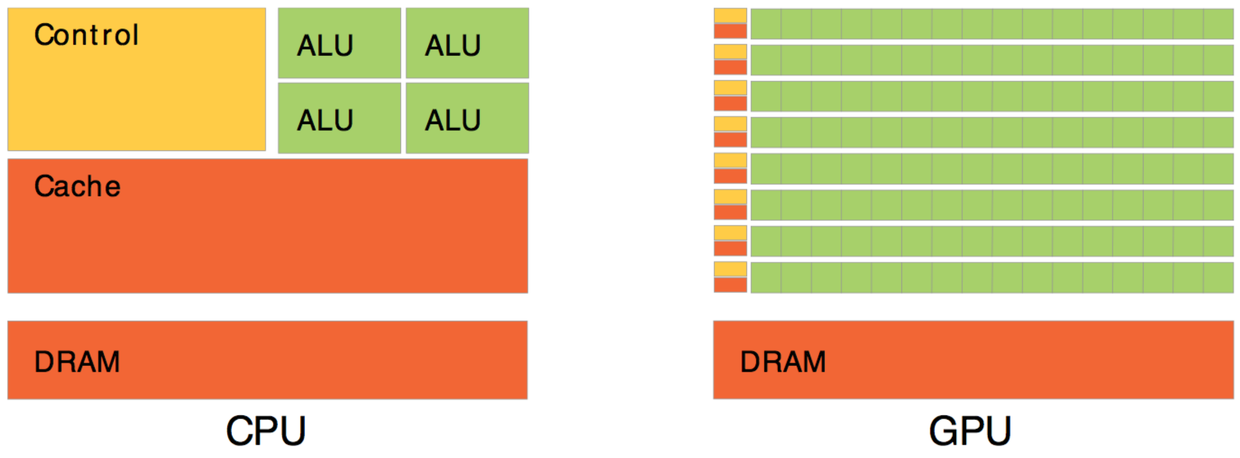


Figure 1.4: A Comparison of CPU and GPU Architectures[9]

well to the parallel architecture of the GPU. GPGPU allows those data to be operated on in parallel, usually providing a significant speed-up over a serial implementation on a CPU.

Until recently, the use of GPU for non-graphics applications was extremely difficult, owing to the necessity of mapping the algorithm to the graphics pipeline by way of an application programming interface (API). Nonetheless, the availability and affordability of GPUs provided by the massive economies of scale in the gaming industry has motivated researchers to exploit the processing power within.

Increasing programmability enables general purpose computation and yields a powerful massively parallel processing alternative to conventional multi-computer or multi-processor systems[10]. A GPU resides in nearly every desktop or laptop computer and NVIDIA provides a convenient software programming environment and API[7]. Moreover, costs of commodity graphic cards are lower when measured in cost per floating-point operations per second (FLOPS)[10].

In 2007, GPU manufacturers began to introduce architectural features and programming flexibility specifically for non-graphics applications. NVIDIA introduced Compute Unified Device Architecture (CUDA), which provided a means for developers to program the GPU using C, with no knowledge of the graphics pipeline required. With the introduction of CUDA, general-purpose GPU programming is no longer limited to image processing and other graphics-related applications, but to any application which has the degree of data parallelism necessary to leverage the massively parallel SIMD architecture of the GPU.

In utilizing architectural means, many efforts have used GPUs as many-core parallel processing units, available and affordable, with increased and increasing support of application interfaces[7]. Numerous recent publications use GPGPU to speed-up their methods[8].

Adaptability of kNN algorithms to GPU architecture

The brute force method of kNN search is highly-parallelizable by nature. All nm distances can be computed in parallel, and then the n sorts can be done in parallel. This property makes the brute force method perfectly suitable for a GPU implementation[4]. The search is easy to implement but still possesses the natural drawback of low efficiency compared with advanced data structures[10].

Advanced data structures, such as k-d trees, are efficient but difficult to implement on GPU[10]. They require recursion, support for which is limited on the GPU, as well as random data access, which requires complicated management of the GPU's limited memory capacity. Although the k-d tree is not a natural fit for GPU implementation, there have been a few creative initial attempts at very low dimension (four or less) which were somewhat effective[10][5].

Chapter 2

Related Work

2.1 Brute Force kNN Search on GPU

In 2008, Garcia, et al., implemented a naïve brute force kNN search on an NVIDIA GeForce 8800 GTX GPU using CUDA. They showed that their CUDA implementation executed 120 times faster than the equivalent serial C implementation and 40 times faster than a k-d tree based method on a Pentium 4 3.4 GHz CPU using the optimized Approximate Nearest Neighbor (ANN) library[4]. (Note: while the name of the library contains "Approximate", it also provides non-approximated methods, as in the k-d tree.)

Two years later, using the same hardware, Garcia, et al., improved on the previous implementation by optimizing Step 1 of the brute force algorithm, the computation of the distances. Rather than naïvely computing each of the distances in nm parallel threads, they accelerated that step by converting the distance computations into matrix operations and taking advantage of the optimized CUDA Basic Linear Algebra Subroutines (cuBLAS) library. With that optimization, testing on high-dimensional scale-invariant feature transform (SIFT) data, they observed speedups of 25 times over the naïve brute force CUDA implementation and 62 times over the optimized ANN k-d tree on the CPU[8].

In 2012, Sismanis, et al., optimized the brute force method by reducing the time in Step 2 of the algorithm, the sorting of the distances. Since only the first k distances are necessary, it is a waste of time and resources to sort the remainder of the list. They implemented a custom truncated bitotic sort algorithm in CUDA on an Nvidia Fermi GTX 480 GPU. By truncating the sort after the kth-shortest distance, they observed a 4 times speedup compared to sorting the entire list using the optimized CUDA Thrust library. They further optimized the algorithm by interleaving the sort steps with the distance calculation steps and observed an additional speedup of 2.4 times over the non-interleaved version[7].

All three brute force implementations were tested in high-dimensional space, with a maximum dimensionality of 96, 256, and 128, respectively.

2.2 kNN Search on GPU Using a k-d Tree

In 2009, Qiu, et al., implemented a more efficient kNN search than brute force by using a k-d tree. Due to the lack of recursion support at the time, their program had to build a left-balanced tree first on the CPU and then flatten it into an array which was then written to the GPU. In place of a recursive search algorithm on the GPU, they created a priority search queue using the GPU's registers. They dealt with memory management by employing a "structure of arrays" data access pattern, which is a common GPU programming method to reduce memory coalescing. Using an Nvidia GeForce GTX 280 GPU, they tested their CUDA implementation against a multi-threaded OpenMP-based version of their application on an Intel Core 2 Duo E6600 CPU, and observed a speedup of 88 times over a sequential implementation on the CPU, and 30 times over the multi-threaded OpenMP-based implementation[10].

Similarly, in 2010, Brown and Snoeyink took a similar approach, again building a left-balanced k-d tree on the CPU, flattening it to an array, and transferring it to the GPU. A stack-based depth-first search and memory management were both handled by creating minimal nodes which produced a small memory footprint while containing all the information necessary for tree traversal and back-tracking. They did not handle the issue of memory coalescing, opting instead to absorb it into the computational costs[5].

Chapter 3

Implementation Details

3.1 Equipment

The training phase and the serial portions of the validation phase were done on a dual 6-core **Intel Xeon CPU** with 96 GB of RAM. The parallelized portions of the validation phase were run an **Nvidia Tesla K20X GPU** with 2688 cores.

3.2 Image Corpus

The image corpus used to train and test the application is a subset of the **Extended Yale Face Database B**[11]. It consists of 4,082 photographs of ten subjects in nine poses, lit from a variety of angles, as shown in Figure 3.1.

3.3 Training and Validation Sets

Due to the nature of the dimensionality-reduction of the Eigenfaces algorithm, the dimension of the search space is constrained to the size of the training set. Four face classifiers were trained of dimension 535, 1070, 1427, and 2853. Each of those was validated using sets of size 2140, 2853, 3210, and 3785. The training sets and the validation sets were composed of images uniformly distributed across the range of subjects, poses, and lighting angles.

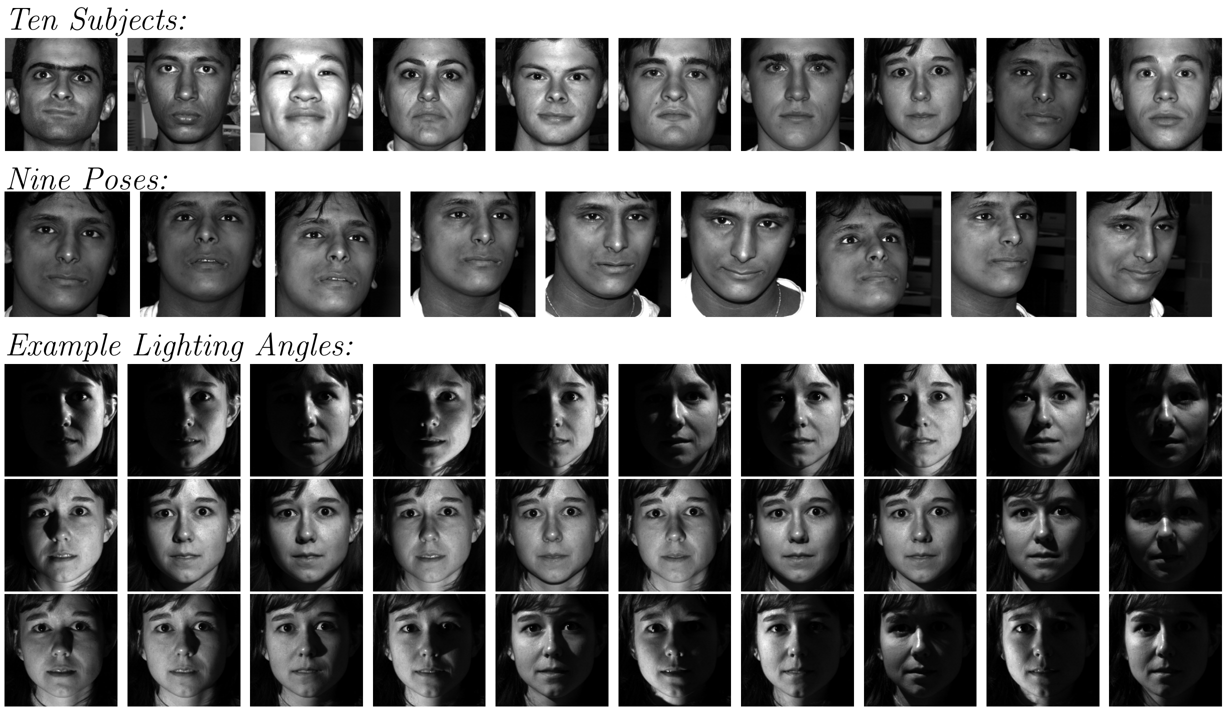


Figure 3.1: Image Corpus Examples

3.4 Parallelization on GPU

The most time-intensive portion of the kNN search algorithm, the compute distances step, is also the most inherently parallelizable. With each query image, it is possible to compute its m distances from the m images in parallel, reducing the time complexity of that step from $O(m^2)$ to nearly constant. In these tests, the serial and parallel applications are identical, with the exception of the compute distances step, which is performed on the GPU in the parallel version and on the CPU in the serial version.

Chapter 4

Discussion of Results

4.1 Sample Results

Figure 4.1 on page 15 shows eight target images and the five nearest neighbors of each, as determined by the program. As implemented, each of the five neighbors has an equal vote in the final classification and the behavior of the program is undefined in the event of a tie. The accuracy could be improved by weighting the votes inversely proportional to the distance from the target, as seen in the "miss" in the figure. The two nearest neighbors were images of the target subject, but the program classified it as the subject of the fourth- and fifth-closest images.

4.2 Accuracy

The results of the classification experiments were identical between the serial and parallel implementations, indicating that the accelerated application has the same accuracy as the original. They show that the accuracy of the program improves as the training set size increases, as shown in figure 4.2. The figure also shows that the accuracy remains consistent as the size of the testing set increases.

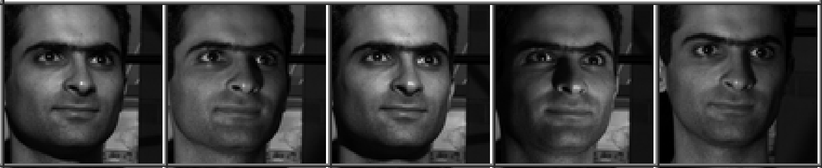
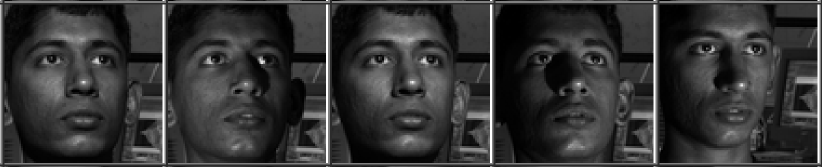
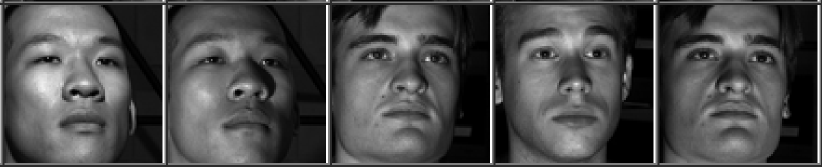
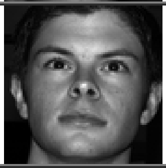
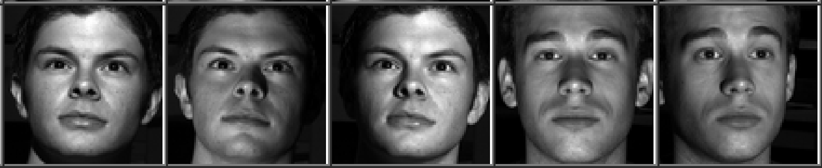
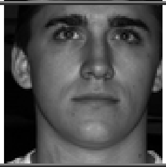
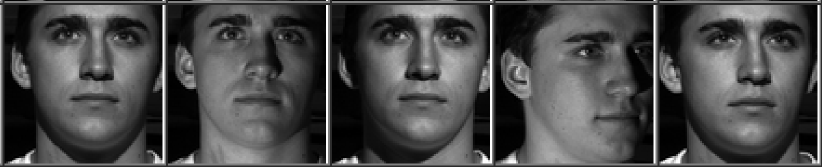
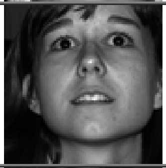
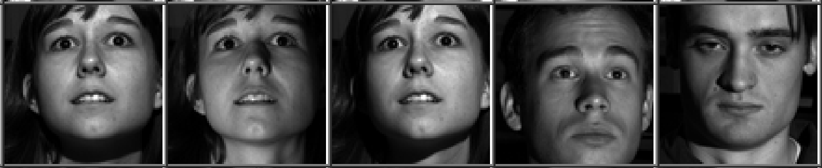
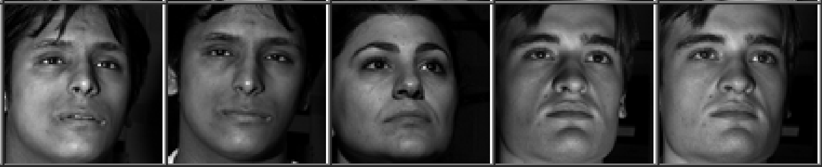
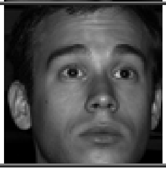
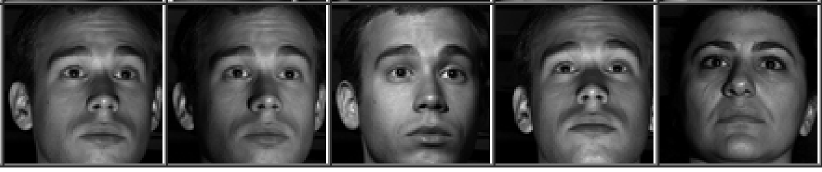
TARGET IMAGE	k-Nearest Neighbors (k = 5)	RESULT
		HIT
		HIT
		HIT
		HIT
		HIT
		HIT
		MISS
		HIT

Figure 4.1: A Sample of Test Results After Voting and Classification

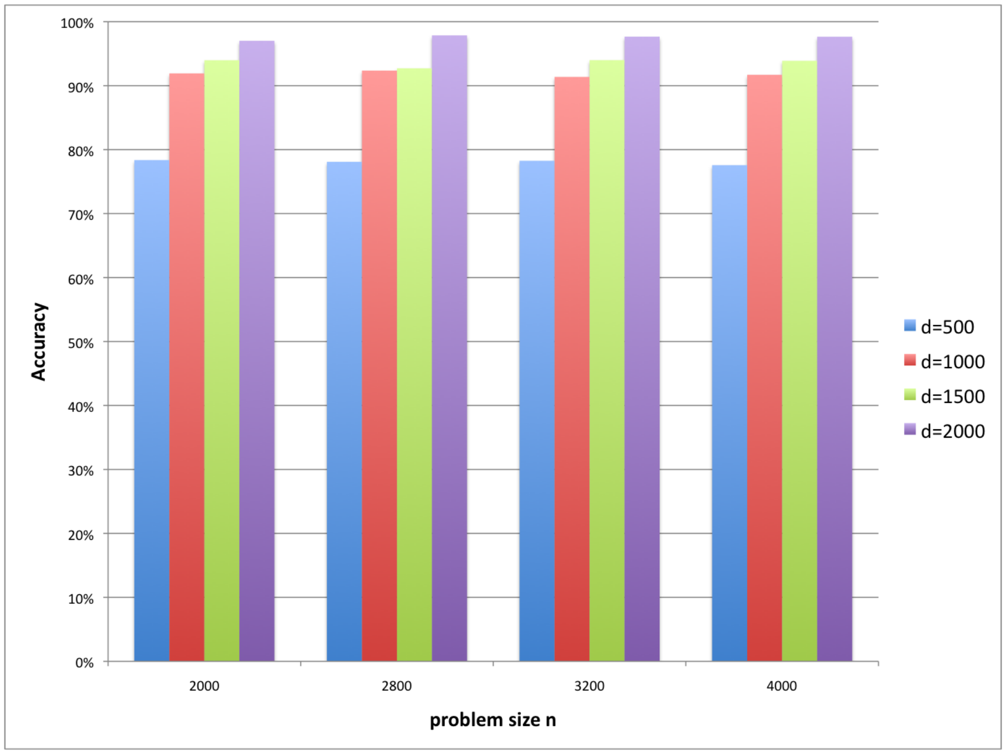


Figure 4.2: Accuracy by Testing Set Size

4.3 Timing and Speed-up

By parallelizing the distance calculations, the execution time complexity of the kNN search was reduced from quadratic with respect to n and d to nearly constant time, as shown in figure 4.3.

Figure 4.4 further shows that the speedup gained by the GPU implementation increases as the dimensionality of the search increases.

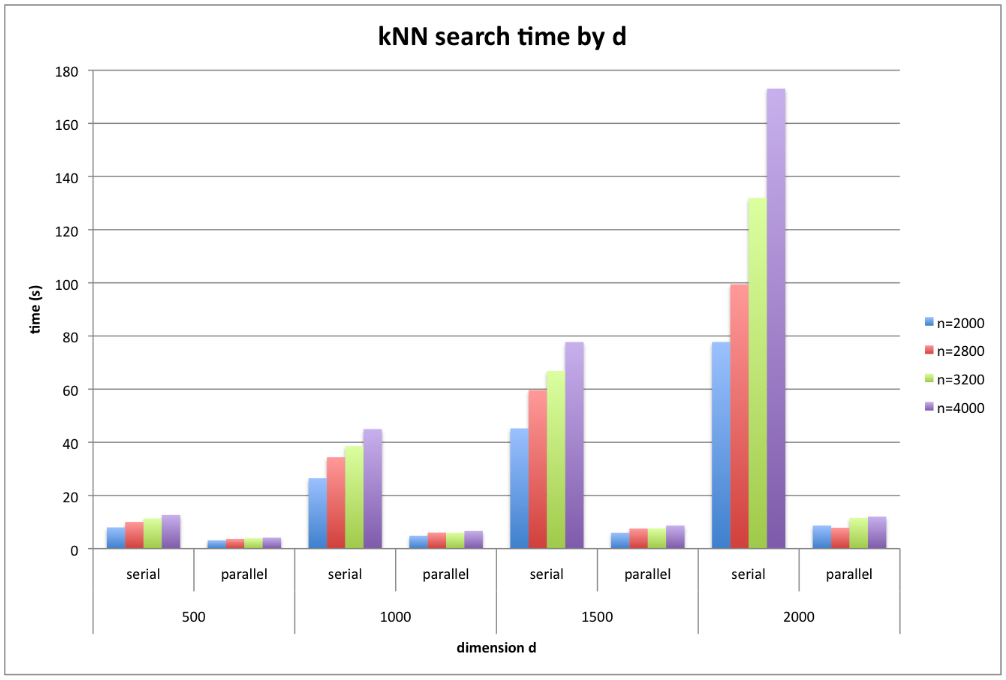


Figure 4.3: *k*NN Search Time by Dimension

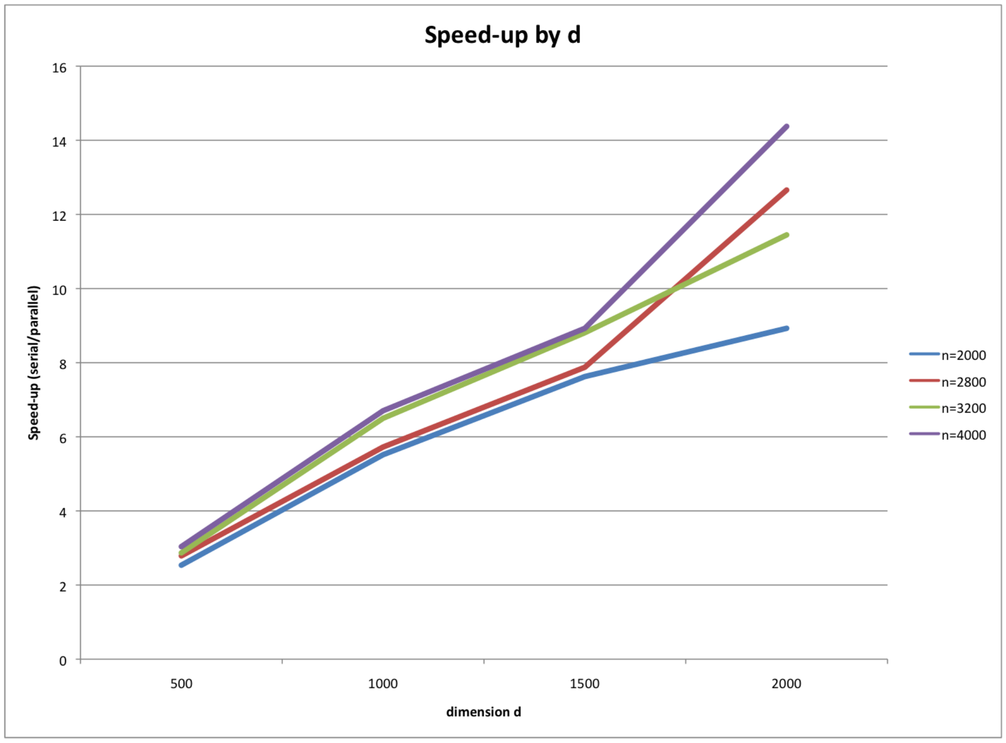


Figure 4.4: Speed-up by Dimension

Chapter 5

Conclusion and Future Work

The results observed by parallelizing the kNN search on the GPU were consistent with the expected performance, and indicate the potential for GPU-optimization of applications that use kNN search, including face recognition, computer vision, and many other fields. This implementation has potential for the same optimization techniques demonstrated by Garcia in [8] and Sismanis in [7].

The training phase of learning systems like the Eigenfaces classifier is typically not a target for acceleration, because training is usually done only once, while classification is done repeatedly. However, the training phase of this method is both extremely compute-intensive as well as highly parallelizable. The time complexity of the calculation of the covariance matrix, which serves initially to reduce the dimensionality of the search space, increases polynomially with the resolution of the input images. In other words, as the amount of face image information increases, the system takes significantly longer to train. It may be worth considering accelerating the training of learning systems of this type, especially in research situations where the systems may be need to be trained and re-trained repeatedly.

References

[1] R. Jafri and H. R. Arabnia, “A survey of face recognition techniques.,” *JIPS*, vol. 5, no. 2, pp. 41–68, 2009.

[2] L. Sirovich and M. Kirby, “Low-dimensional procedure for the characterization of human faces,” *JOSA A*, vol. 4, no. 3, pp. 519–524, 1987.

[3] M. Turk and A. Pentland, “Eigenfaces for recognition,” *Journal of cognitive neuroscience*, vol. 3, no. 1, pp. 71–86, 1991.

[4] V. Garcia, E. Debreuve, and M. Barlaud, “Fast k nearest neighbor search using gpu,” in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW’08. IEEE Computer Society Conference on*, pp. 1–6, IEEE, 2008.

[5] S. Brown and J. Snoeyink, “Gpu nearest neighbor searches using a minimal kd-tree,” *MASSIVE*, 2010.

[6] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[7] N. Sismanis, N. Pitsianis, and X. Sun, “Parallel search of k-nearest neighbors with synchronous operations,” in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pp. 1–6, IEEE, 2012.

[8] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, “K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching,” in *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pp. 3757–3760, IEEE, 2010.

[9] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[10] D. Qiu, S. May, and A. Nüchter, “Gpu-accelerated nearest neighbor search for 3d registration,” in *Computer Vision Systems*, pp. 194–203, Springer, 2009.

[11] A. Georghiades, P. Belhumeur, and D. Kriegman, “From few to many: Illumination cone models for face recognition under variable lighting and pose,” *IEEE Trans. Pattern Anal. Mach. Intelligence*, vol. 23, no. 6, pp. 643–660, 2001.

Appendix A

Source file: faceclassifier.cu

```
/* Filename: faceclassifier.cu
 * Author: J.D. Rouan, B.S., University of Georgia, 2012
 * A Thesis Submitted to the Graduate Faculty of The University of Georgia
 * in partial fulfillment of the requirements for the degree
 * Master of Science
 * May, 2014
 */

#include "knnsearch.h"

void print_usage(){
   printf("to train: faceclassifier <trainingset filename>\n");
   printf("to test: faceclassifier <testingset filename> <k> [parallel]\n");
}

int main( int argc, char **argv ){
   if( argc <= 1 ){
      print_usage();
      return -1;
   }
   char *filename = argv[1];
   FILE *file = fopen ( filename, "r" );
   if ( file == NULL ) {
      fprintf(stderr, "main: failed to open file %s\n", filename);
      print_usage();
      return -1;
   }

/* If 2 arguments, we want to train the system and then return. */

   if( argc == 2 ) {
      eigen_build_db( file );
      return 0;
   }

/* If we have gotten this far, we want to test the system */

   int knn = (int) strtol(argv[2], NULL, 10);
   if ( knn == 0 ) {
      fprintf(stderr, "main: k argument %s is not a number\n", argv[2]);
      print_usage();
      return -1;
   }

   bool parallel = ( argc > 3 && strcmp( argv[3], "parallel" ) == 0 );

/* Load mean face from disk and initialize variables */

   CImg<float> mean( "data/eigen.mean.bmp" ), face;
   vector< CImg<float> > faces;
   eigen_db_t db;
   float **projections;

   vector<image_info> trainingset_info;
   vector<image_info> testingset_info;
   int *kneighbors;
   image_info img;

/* Read in list of test image filenames and load faces to vector.
 * Filenames are formatted like "yaleB01_P01A+000E+00.png" for
 * subject #01, pose #01, azimuth +000, elevation +00.
 * Azimuth and elevation information is not used but could be useful.
 */

   char line [ 0x100 ];
   while ( fgets ( line, sizeof line, file ) != NULL ) {
      string filename = line;
      char *token = strtok(line, "\n");
      string path = "images/" + (string) token;
      try {
         face.load( path.c_str() );
         faces.push_back( face );
         img.subject = atoi(filename.substr(5,2).c_str());
         img.pose = atoi(filename.substr(9,2).c_str());
         img.azimuth = atoi(filename.substr(12,4).c_str());
         img.elevation = atoi(filename.substr(17,3).c_str());
         testingset_info.push_back( img );
      } catch (CImgException &e) {
         fprintf(stderr, "main: failed to load face: %s\n", e.what());
         continue;
      }
   }

/* Load learned system from disk and allocate memory for new structures. */

   eigen_load_db(&db);
   eigen_load_info(&trainingset_info, db);
   projections = eigen_load_vectors(db);

   float * weights = new float[db.faces * db.faces];
   eigen_load_weights(db, weights);

   float * iweights = new float[db.faces * faces.size()];

/* Project test images to learned eigen face space to obtain feature vectors. */

   eigen_build_iweights(faces, mean, projections, db.faces, iweights);

/* Conduct k-nearest neighbor search. */

   kneighbors = knn_search( iweights, weights, faces.size(), db.faces, knn,
                            parallel );

/* Conduct vote of k-nearest neighbors to determine likely match to target.
 * TODO: Improve accuracy by decreasing weight of votes by distance. */

   int hits = 0;
   for( int i = 0; i < faces.size(); i++ ) {
      int neighbors[knn];
      image_info target = testingset_info[i];

      for( int k = 0; k < knn; k++ ) {
         image_info ii = trainingset_info[kneighbors[i*knn+k]];
         neighbors[k] = ii.subject;
      }

      /* Hard-coded max number of subjects at 40. TODO: generalize. */
      int count[40] = {0};
      for( int i = 0; i < knn; i++ ){
         count[neighbors[i]]++;
      }
      int winnercount = -1;
      int winner = -1;
```

```
        for( int i = 0; i < 40; i++ ){
           if(count[i] > winnercount){
               winnercount = count[i];
               winner = i;
           }
        }
        if( winner == target.subject ) {
           hits++;
        }
     }
     printf("main: Accuracy = %f\n", (((float) hits) / faces.size()));
     return 0;
  }
```

Appendix B

Source file: knnsearch.h

```
/* Filename: knnsearch.h
 * Author: J.D. Rouan, B.S., University of Georgia, 2012
 * May, 2014
 */

#include <string>
#include <vector>

#include <cuda.h>
#include <cutil.h>
#include <dirent.h>
#include <math.h>
#include <stdio.h>
#include <unistd.h>

#include "eigenfaces.h"

#define ERROR_CHECK error = cudaGetLastError(); if(error != cudaSuccess) \\
        { printf("CUDA error: %s\n", cudaGetErrorString(error)); exit(-1); }

using namespace std;
typedef struct {
   int idx;
   float val;
} my_pair;

int compare_q( const void* a, const void* b ) {
     my_pair pr_a = * ((my_pair*) a);
     my_pair pr_b = * ((my_pair*) b);

     if ( pr_a.val == pr_b.val ) return 0;
     else if ( pr_a.val < pr_b.val ) return -1;
     else return 1;
```

```
}

/* kernel function which executes on GPU */

__global__ void compute_distances (float *iweights_d,
                                   float *weights_d,
                                   int size,
                                   my_pair *results_d,
                                   int k,
                                   int img){
   int i = blockIdx.x * blockDim.x + threadIdx.x;

   if( i < size ) {
      float mag = 0.0;
      for( int j = 0; j < size; j++ ){
         mag += ((iweights_d[img*size + j]-weights_d[i*size + j])
               * (iweights_d[img*size + j]-weights_d[i*size + j]));
      }
      results_d[i].val = sqrt(mag);
      results_d[i].idx = i;
   }
}

int *knn_search( float *iweights_h, float *weights_h,
                      int isize, int size, int k, bool parallel ){
   printf("knn: beginning knn search\n");
   unsigned int timer0;
   CUT_SAFE_CALL(cutCreateTimer(&timer0));

   int *n = new int [isize * k];

   if( parallel ) {     /* parallel implementation on GPU */
      CUT_SAFE_CALL(cutStartTimer(timer0));

      size_t iweights_size = isize * size * sizeof(float);
      size_t weights_size = size * size * sizeof(float);
      size_t results_size = size * sizeof(my_pair);

      float *iweights_d;
      float *weights_d;
      my_pair *results_d;
      int *n_d;

      CUDA_SAFE_CALL(cudaMalloc((void **) &iweights_d, iweights_size));
      CUDA_SAFE_CALL(cudaMalloc((void **) &weights_d, weights_size));
      CUDA_SAFE_CALL(cudaMalloc((void **) &results_d, results_size));

      CUDA_SAFE_CALL(cudaMemcpy(iweights_d, iweights_h, iweights_size,
                                cudaMemcpyHostToDevice));
      CUDA_SAFE_CALL(cudaMemcpy(weights_d, weights_h, weights_size,
                                cudaMemcpyHostToDevice));

      int b_size = 128, g_size;
      if( size < b_size ) b_size = size;
      g_size = size/b_size + (size%b_size == 0 ? 0 : 1 );

      my_pair results[size];
      cudaError_t error;

      /* TODO: parallelize this. Use m by n matrix for n > 1 */
      for( int img = 0; img < isize; img++ ){

      /* kernel function call for distance computation */
         compute_distances <<< g_size, b_size >>> (iweights_d, weights_d,
                                                   size, results_d, k, img);
         ERROR_CHECK
         cudaThreadSynchronize();

         CUDA_SAFE_CALL(cudaMemcpy(&results, results_d, results_size,
                                   cudaMemcpyDeviceToHost));

       /* serial sort on host. could move to GPU */
         qsort(results, size, sizeof(my_pair), compare_q);
         for( int i = 0; i < k; i++ ){
            n[img*k+i] = results[i].idx;
         }
      }
      CUT_SAFE_CALL(cutStopTimer(timer0));
      printf("knn: time-parallel %.16E\n", cutGetTimerValue(timer0));
      fflush(stdout);

   } else {        /* serial implementation on host CPU */
      CUT_SAFE_CALL(cutStartTimer(timer0));

      for( int img = 0; img < isize; img++ ){

         pair<float, int> *results = new pair<float, int> [size];

         for( int i = 0; i < size; i++ ){
            float mag = 0.0;
```

```
            for( int j = 0; j < size; j++ ){
               mag += ((iweights_h[img*size + j]-weights_h[i*size + j])
                     * (iweights_h[img*size + j]-weights_h[i*size + j]));
            }
            results[i].first = sqrt(mag);
            results[i].second = i;
         }

         sort(results, results + size);
         for( int i = 0; i < k; i++ ){
            n[img*k+i] = results[i].second;
         }
      }
      CUT_SAFE_CALL(cutStopTimer(timer0));
      printf("knn: time-serial %.16E\n", cutGetTimerValue(timer0));
      fflush(stdout);
   }
   CUT_SAFE_CALL(cutDeleteTimer(timer0));
   return n;
}
```

Appendix C

Source file: eigenfaces.h

Adapted from original algorithm presented by Matthew Turk and Alex Pentland in [3].

```
/* Filename: eigenfaces.h
 * Author: J.D. Rouan, B.S., University of Georgia, 2012
 * Adapted from original algorithm presented by Matthew Turk and Alex Pentland.
 * c++ code website: http://www.emoticode.net/c-plus-plus/
 *                   face-recognition-with-cimg-and-eigenfaces.html
 * May, 2014
 */

#include <fstream>
#include <string>
#include <vector>

#include <dirent.h>
#include <math.h>
#include <stdio.h>
#include <unistd.h>

#include <sys/stat.h>
#include <sys/types.h>

#include "CImg.h"

using namespace std;
using namespace cimg_library;

typedef struct {
   int faces;
   int width;
   int height;
} eigen_db_t;

typedef struct {
   int index;
   int subject;
   int pose;
   int azimuth;
   int elevation;
} image_info;

void process_image( CImg<float>& image,
                    CImg<float>& mean,
                    int trainingset_size ){
   int i, j;

   for( i = 0; i < image.width(); i++ ){
      for( j = 0; j < image.height(); j++ ){
         *mean.data( i, j ) += *image.data( i, j ) / (trainingset_size);
      }
   }
}

void normalize_image( CImg<float>& image,
                      int index,
                      CImg<float>& mean,
                      float **&normalized_m ){
   int i, j, k;

   for( i = 0, k = 0; i < image.width(); i++ ){
      for( j = 0; j < image.height(); j++, k++ ){
         normalized_m[index][k] = *image.data( i, j ) - *mean.data( i, j );
      }
   }
}

void eigen_covariance( float **& normalized_m,
                       float **& covariance_m,
                       int vector_size,
                       int trainingset_size ){
   float **transpose;
   int i, j, m;

   transpose = new float * [vector_size];
   for( i = 0; i < vector_size; i++ ){
      transpose[i] = new float [trainingset_size];
   }

   for( i = 0; i < vector_size; i++ ){
      for( j = 0; j < trainingset_size; j++ ){
         transpose[i][j] = normalized_m[j][i];
      }
   }

   for( m = 0; m < trainingset_size; m++ ){
      if( m % 64 == 0 ){
       printf( "step 4/6 (covariance): image %d of %d\n", m, trainingset_size );
       fflush(stdout);
      }
      for( i = 0; i < trainingset_size; i++ ){
         covariance_m[i][m] = 0;
         for( j = 0; j < vector_size; j++ ){
            covariance_m[i][m] += normalized_m[m][j] * transpose[j][i];
         }
      }
   }

   for( i = 0; i < vector_size; i++ ){
      delete[] transpose[i];
   }
   delete[] transpose;
}

int eigen_decomposition( float **& matrix,
                         int m_size,
                         float *& eigenvalues,
                         float **& eigenvectors ){
   float threshold, theta, tau, t, sm, s, h, g, c, p, *b, *z;
   int jiterations;

   /* max iterations in Jacobi decomposition algorithm */
   int jacobi_max_iterations = 500;

   b = new float[m_size * sizeof(float)];
   z = new float[m_size * sizeof(float)];

   /* initialize eigenvectors and eigen values */
   for( int ip = 0; ip < m_size; ip++ ){
      for ( int iq = 0; iq < m_size; iq++){
         eigenvectors[ip][iq] = 0.0;
      }
      eigenvectors[ip][ip] = 1.0;
   }
   for( int ip = 0; ip < m_size; ip++ ){
      b[ip] = eigenvalues[ip] = matrix[ip][ip];
      z[ip] = 0.0;
   }

   jiterations = 0;
   for( int m = 0; m < jacobi_max_iterations; m++ ){
      printf("jacobi iteration %d\n", m);
      fflush(stdout);
      sm = 0.0;
      for( int ip = 0; ip < m_size; ip++ ){
         for( int iq = ip + 1; iq < m_size; iq++ ){
            sm += fabs(matrix[ip][iq]);
         }
      }

      if( sm == 0.0 ){
      /* eigenvalues & eigenvectors sorting */
         for( int i = 0; i < m_size; i++ ){
            int k;
            p = eigenvalues[k = i];
            for( int j = i + 1; j < m_size; j++ ){
               if( eigenvalues[j] >= p ){
                  p = eigenvalues[k = j];
               }
            }
            if( k != i ){
               eigenvalues[k] = eigenvalues[i];
               eigenvalues[i] = p;
               for( int j = 0; j < m_size; j++ ){
                  p = eigenvectors[j][i];
                  eigenvectors[j][i] = eigenvectors[j][k];
                  eigenvectors[j][k] = p;
               }
            }
         }

         /* restore symmetric matrix's matrix */
         for( int i = 1; i < m_size; i++ ){
            for( int j = 0; j < i; j++ ){
               matrix[j][i] = matrix[i][j];
            }
         }

         delete z;
         delete b;
         return jiterations;
      }

      threshold = ( m < 4 ? 0.2 * sm / (m_size * m_size) : 0.0 );
      for( int ip = 0; ip < m_size; ip++ ){
         for( int iq = ip + 1; iq < m_size; iq++ ){
            g = 100.0 * fabs(matrix[ip][iq]);

            if( m > 4 &&
                fabs(eigenvalues[ip]) + g == fabs(eigenvalues[ip]) &&
                fabs(eigenvalues[iq]) + g == fabs(eigenvalues[iq]) ){
               matrix[ip][iq] = 0.0;
            } else if( fabs(matrix[ip][iq]) > threshold ){
               h = eigenvalues[iq] - eigenvalues[ip];
               if( fabs(h) + g == fabs(h) ){
                  t = matrix[ip][iq] / h;
               } else {
                  theta = 0.5 * h / matrix[ip][iq];
                  t = 1.0 / ( fabs(theta) + sqrt( 1.0 + theta * theta ) );
                  if( theta < 0.0 ){
                     t = -t;
                  }
               }

               c = 1.0 / sqrt(1 + t * t);
               s = t * c;
               tau = s / (1.0 + c);
               h = t * matrix[ip][iq];
               z[ip] -= h;
               z[iq] += h;
               eigenvalues[ip] -= h;
               eigenvalues[iq] += h;
               matrix[ip][iq] = 0.0;

#define M_ROTATE(M,i,j,k,l) g = M[i][j]; h = M[k][l]; \\
        M[i][j] = g - s * (h + g * tau); M[k][l] = h + s * (g - h * tau)

               for( int j = 0; j < ip; j++ ){
                  M_ROTATE( matrix, j, ip, j, iq );
               }
               for( int j = ip + 1; j < iq; j++ ){
                  M_ROTATE( matrix, ip, j, j, iq );
               }
               for( int j = iq + 1; j < m_size; j++ ){
                  M_ROTATE( matrix, ip, j, iq, j );
               }
               for( int j = 0; j < m_size; j++ ){
                  M_ROTATE( eigenvectors, j, ip, j, iq );
               }
               jiterations++;
            }
         }
      }

      for( int ip = 0; ip < m_size; ip++ ){
         b[ip] += z[ip];
         eigenvalues[ip] = b[ip];
         z[ip] = 0.0;
      }
   }

   delete z;
   delete b;

   return -1;
}

float ** eigen_project( float **& m_normalized,
                        float **& eigenvectors,
                        float *& eigenvalues,
                        int vector_size,
                        int trainingset_size ){
   float value = 0, mag,
         **projections,
         **transpose;

   projections = new float * [trainingset_size];
   for( int i = 0; i < trainingset_size; i++ ){
      projections[i] = new float[vector_size];
   }

   transpose = new float * [vector_size];
   for( int i = 0; i < vector_size; i++ ){
      transpose[i] = new float[trainingset_size];
   }
   for( int i = 0; i < vector_size; i++ ){
      for( int j = 0; j < trainingset_size; j++ ){
         transpose[i][j] = m_normalized[j][i];
      }
   }
   FILE *fp = fopen( "data/eigen.vectors.dat", "w+b" );
   for( int k = 0; k < trainingset_size; k++ ){
      if( k % 64 == 0 ){
       printf( "step 5/6 (projection): image %d of %d\n", k, trainingset_size );
       fflush(stdout);
      }
      for( int i = 0; i < vector_size; i++ ){
         for( int j = 0; j < trainingset_size; j++ ){
            value += transpose[i][j] * eigenvectors[j][k];
         }
         projections[k][i] = value;
         value = 0;
      }

      /* eigenfaces normalization */
      mag = 0;
      for( int l = 0; l < vector_size; l++ ){
         mag += projections[k][l] * projections[k][l];
      }
      mag = sqrt(mag);
      for( int l = 0; l < vector_size; l++ ){
         if( mag > 0) projections[k][l] /= mag;
      }

      /* save the projected eigenvector */
      for( int p = 0; p < vector_size; p++ ){
         fwrite( &projections[k][p], sizeof(float), 1, fp );
      }
   }
   fclose(fp);

   return projections;
}

float *eigen_weights( CImg<float>& face,
                      CImg<float>& mean,
                      float **& projections,
                      int trainingset_size ){
   CImg<float> normalized = face - mean;
   int m, n, i, index;
   float *weights, w;
   weights = new float[trainingset_size];

   for( index = 0; index < trainingset_size; index++ ){
      w = 0.0;
      for( m = 0, i = 0; m < normalized.width(); m++ ){
         for( n = 0; n < normalized.height(); n++, i++ ){
            w += projections[index][i] * *normalized.data(m,n);
         }
      }
      weights[index] = w;
   }
   return weights;
}

void eigen_build_db( FILE *file ){
   eigen_db_t db;

   vector< CImg<float> * > trainingset;
   vector< image_info > trainingset_info;

/* expects 256 x 256 image. TODO: generalize */
   CImg<float> mean( 256, 256, 1, 1 ); // mean face from training set

   float **normalized_m,     // normalized faces matrix A
         **covariance_m,     // covariance matrix (C = tA*A --> C = A*tA)
         **eigenvectors,     // eigenvectors of jacobi decomposition
         *eigenvalues,       // eigenvalues of jacobi decomposition
         **eigenprojections, // eigenvectors projected to face space
         **eigenweights;     // weights

   int  i, j;

   printf( "--- Building Eigen faces database ---\n" );

   printf( "\t@ Loading training set ...\n" );

   char line [ 0x100 ];
   while ( fgets ( line, sizeof line, file ) != NULL ) {
      string filename = line;
      char *token = strtok(line, "\n");
      string path = "images/" + (string) token;

      image_info img;
      img.subject = atoi(filename.substr(5,2).c_str());
      img.pose = atoi(filename.substr(9,2).c_str());
      img.azimuth = atoi(filename.substr(12,4).c_str());
      img.elevation = atoi(filename.substr(17,3).c_str());
      printf( "step 1/6 (loading): subject %d; pose %d\n",
               img.subject, img.pose );
      fflush(stdout);
      trainingset.push_back( new CImg<float>( path.c_str() ) );
      trainingset_info.push_back( img );
   }

   for( int k = 0; k < trainingset_info.size(); k++ ){
      trainingset_info[k].index = k;
   }
   printf( "\t@ Processing mean face ...\n" );
   for( i = 0; i < trainingset.size(); i++ ){
      printf( "step 2/6 (processing): image %d; subject %d; pose %d\n", i,
              trainingset_info[i].subject, trainingset_info[i].pose );
      fflush(stdout);
      process_image( *trainingset[i], mean, trainingset.size() );
   }
   mean.save("data/eigen.mean.bmp");

   printf( "\t@ Normalizing faces ...\n" );

   normalized_m = new float * [ trainingset.size() ];
   for( i = 0; i < trainingset.size(); i++ ){
      normalized_m[i] = new float[ mean.width() * mean.height() ];
   }
   for( i = 0; i < trainingset.size(); i++ ){
      normalize_image( *trainingset[i], i, mean, normalized_m );
         printf( "step 3/6 (normalizing): image %d; subject %d; pose %d\n",
                 i, trainingset_info[i].subject, trainingset_info[i].pose);
         fflush(stdout);
   }

   printf( "\t@ Computing covariance ...\n" );

   covariance_m = new float * [ trainingset.size() ];
   for( i = 0; i < trainingset.size(); i++ ){
      covariance_m[i] = new float[ trainingset.size() ];
   }

   eigen_covariance( normalized_m, covariance_m, mean.width() * mean.height(),
                     trainingset.size() );

   printf( "\t@ Computing Jacobi decomposition ... " );
   fflush(stdout);
   eigenvalues  = new float   [trainingset.size()];
```

```
eigenvectors = new float * [trainingset.size()];
for( i = 0; i < trainingset.size(); i++ ){
   eigenvectors[i] = new float [trainingset.size()];
}
int ret = eigen_decomposition( covariance_m, trainingset.size(), eigenvalues,
                               eigenvectors );
printf( "[%d]\n", ret );

printf( "\t@ Projecting eigenvectors ...\n" );
eigenprojections = eigen_project( normalized_m, eigenvectors, eigenvalues,
                                  mean.width() * mean.height(),
                                  trainingset.size() );

printf( "\t@ Saving eigen values ...\n" );
fflush(stdout);
FILE *fp = fopen( "data/eigen.values.dat", "w+b" );
for( i = 0; i < trainingset.size(); i++ ){
   fwrite( &eigenvalues[i], sizeof(float), 1, fp );
}
fclose(fp);

printf( "\t@ Computing eigen weights ...\n" );
fflush(stdout);
eigenweights = new float * [trainingset.size()];
for( i = 0; i < trainingset.size(); i++ ){
   if( i % 64 == 0 ){
   printf( "step 6/6 (eigen weights): image %d of %d; subject %d; pose %d\n",
           i, trainingset.size(), trainingset_info[i].subject,
           trainingset_info[i].pose );
    fflush(stdout);
   }
   eigenweights[i] = eigen_weights( *trainingset[i], mean, eigenprojections,
                                    trainingset.size() );
}
fp = fopen( "data/eigen.weights.dat", "w+b" );
for( i = 0; i < trainingset.size(); i++ ){
   for( j = 0; j < trainingset.size(); j++ ){
      fwrite( &eigenweights[i][j], sizeof(float), 1, fp );
   }
}
fclose(fp);

printf( "\t@ Saving db info ...\n" );
fflush(stdout);
db.faces  = trainingset.size();
```

```
   db.width  = mean.width();
   db.height = mean.height();
   fp = fopen( "data/eigen.db.dat", "w+b" );
   fwrite( &db, sizeof(eigen_db_t), 1, fp );
   fclose(fp);

   fp = fopen( "data/eigen.image_info.dat", "w+b" );
   for( int i = 0; i < db.faces; i++ ){
      image_info ii = trainingset_info.at(i);
      fwrite( &ii, sizeof(image_info), 1, fp );
   }
   fclose(fp);
}

void eigen_load_db( eigen_db_t *db ){
   printf("eigen: loading database\n");
   FILE *fp = fopen( "data/eigen.db.dat", "rb" );
   fread( db, sizeof(eigen_db_t), 1, fp );
   fclose(fp);
}

void eigen_load_info( vector<image_info> *info, eigen_db_t& db ){
   printf("eigen: loading image info\n");
   FILE *fp = fopen( "data/eigen.image_info.dat", "rb" );
   for( int i = 0; i < db.faces; i++ ) {
      image_info ii;
      fread( &ii, sizeof(image_info), 1, fp );
      info->push_back(ii);
   }
   fclose(fp);
}

float **eigen_load_vectors( eigen_db_t& db ){
   printf("eigen: loading vectors\n");
   float **projections;
   int i, j;

   projections = new float * [db.faces];
   for( i = 0; i < db.faces; i++ ){
      projections[i] = new float[db.width * db.height];
   }

   FILE *fp = fopen( "data/eigen.vectors.dat", "rb" );
   for( i = 0; i < db.faces; i++ ){
      for( j = 0; j < db.width * db.height; j++ ){
         fread( &projections[i][j], sizeof(float), 1, fp );
      }
   }
   fclose(fp);

   return projections;
}

void eigen_load_weights( eigen_db_t& db, float *weights ){
   printf("eigen: loading weights - size = %d\n", db.faces);
   fflush(stdout);
   int i, j;

   FILE *fp = fopen( "data/eigen.weights.dat", "rb" );
   for( i = 0; i < db.faces; i++ ){
      for( j = 0; j < db.faces; j++ ){
         fread( &weights[i*db.faces + j], sizeof(float), 1, fp );
      }
   }
   printf("finished loading weights\n");
   fflush(stdout);
   fclose(fp);
}

void eigen_build_iweights( vector< CImg<float> > faces, CImg<float> mean,
                           float **projections, int size, float *iweights ){
   printf("eigen: building iweights\n");

   for( int i = 0; i < faces.size(); i++ ) {
      CImg<float> normalized = faces[i] - mean;
      for( int j = 0; j < size; j++) {
         float w = 0.0;
         for( int m = 0, ii = 0; m < normalized.width(); m++ ){
            for( int n = 0; n < normalized.height(); n++, ii++ ){
               w += projections[j][ii] * *normalized.data(m,n);
            }
         }
         iweights[i*size + j] = w;
      }
   }

   printf(" >>> finished building iweights\n");
   fflush(stdout);
}
```

Appendix D

Compiling and Running

Files required

faceclassifier.cu Main source file

knnsearch.h Header file for kNN search functions, including GPU kernel calls.

eigenfaces.h Header file for Eigenfaces training system functions, including training and classifying.

Cimg.h Header file for The CImg Library, an open source C++ template image processing toolkit, available for download at `http://cimg.sourceforge.net/`.

data/ If training, an empty directory to store the trained system. If testing, the trained system files: `eigen.db.dat`, `eigen.mean.bmp`, `eigen.vectors.dat`, `eigen.image_info.dat`, `eigen.values.dat`, and `eigen.weights.dat`.

images/ A directory of images named like `yaleB15_P02A-070E-35.png` for subject 15, pose 2, azimuth -70, and elevation 35. On loading the images, the pose, azimuth, and elevation data are stored in a data structure with the subject number, but they are not used by this system. They could be useful for future research.

This system expects images of size 256×256, but that can easily be changed or generalized by editing line 320 of `eigenfaces.h`.

The image corpus for this system was extracted from the Extended Yale Face Database B[11], available for download at `http://vision.ucsd.edu/~iskwak/ExtYaleDatabase/ExtYaleB.html`.

training-set.txt or testing-set.txt A plain text file containing a list of image files to be read into the system. The filename is used as the first command line argument.

Compiling with nvcc

Compile the source files with the following command:

```
nvcc -w -I/{path to Nvidia SDK}/C/common/inc faceclassifier.cu
    -L/{path to Nvidia SDK}/C/lib -lm -lpthread -lX11 -lcutil_x86_64
    -o faceclassifier
```

Running the program

Run the compiled program with one of the following commands:

To train: `./faceclassifier training-set.txt`

To test: `./faceclassifier testing-set.txt <k> [parallel]`