Enabling Real–Time Mobile Cloud Computing through Emerging Technologies

Tolga Soyata
University of Rochester, USA

A volume in the Advances in Wireless Technologies and Telecommunication (AWTT) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Enabling real-time mobile cloud computing through emerging technologies / Tolga Soyata, editor.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4666-8662-5 (hc) -- ISBN 978-1-4666-8663-2 (eISBN) 1. Cloud computing. 2. Mobile computing. I. Soyata, Tolga, 1967-
QA76.585.E55 2015
004.67'82--dc23
2015015533

This book is published in the IGI Global book series Advances in Wireless Technologies and Telecommunication (AWTT) (ISSN: 2327-3305; eISSN: 2327-3313)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 11
Theoretical Foundation and GPU Implementation of Face Recognition

William Dixon
University of Rochester, USA

Yang Song
University of Rochester, USA

Nathaniel Powers
University of Rochester, USA

Tolga Soyata
University of Rochester, USA

ABSTRACT

Enabling a machine to detect and recognize faces requires significant computational power. This particular system of face recognition makes use of OpenCV (Computer Vision) libraries while leveraging Graphics Processing Units (GPUs) to accelerate the process towards real-time. The processing and recognition algorithms are best sorted into three distinct steps: detection, projection, and search. Each of these steps has unique computational characteristics and requirements driving performance. In particular, the detection and projection processes can be accelerated significantly with GPU usage due to the data types and arithmetic types associated with the algorithms, such as matrix manipulation. This chapter provides a survey of the three main processes and how they contribute to the overarching recognition process.

INTRODUCTION

Humans have the innate skill to continuously gather complex information from another person's face. While it comes naturally to us, it is really quite the feat to be able to do something as simple as recognizing individuals based on a quick gaze over a face. What may differentiate individual faces are inherently subtle variations in features such as skin, skeletal structure, hair and lips. Humans can do this with ease at any angle, most lighting, at a massive scale with a high degree of accuracy.

Computers have a comparatively difficult time with face recognition, even today. At a basic level, in order to recognize faces, a computer must first be able to determine which objects from an image are actual faces. It must then gather enough information by analyzing multiple image fragments to develop patterns associated with a face which may be presented in a variety of aspects and lighting conditions.

DOI: 10.4018/978-1-4666-8662-5.ch011

The computer can then accept new images, detect and analyze faces, then attempt to match these analyzed faces to established representations stored in a database. If a comparison surpasses a particular confidence threshold, the computer will state that it has established a match. If the threshold is not met, the computer can throw the image out or use it to start a new face index, depending on what the developer has programmed the computer to do.

Each of these processes is, by nature of the computational implementation, reducible to a series of steps. Like many computing problems, not every solution utilizes a single method or algorithm. Computer scientists, programmers, mathematicians and others have been making progress on simplifying and solving the problems of computer vision since the 1960's. At first, facial recognition was done partially manually, with people marking locations of features such as the nose and eyes. This information was fed into a computer which sifted through its database of other marked images and returned which match was closest. Over the next two decades, this system was made autonomous and finely-tuned. The parent of many systems used today, including ours, was developed in 1988 by Lawrence Sirovich and Michael Kirby. This system makes use of a broader statistical method called Principal Component Analysis (PCA). PCA essentially parses and manipulates a dataset into uncorrelated sets of correlated data (Kim, Jung, & Kim, 2002). These sets are called principal components, which fall under the more mathematically general term of eigenvectors.

FACE RECOGNITION ALGORITHMIC STRUCTURE

This particular facial recognition algorithm includes three main processes: face detection, projection, and search. Face detection identifies potential faces, projection conducts the bulk of the image processing and obtains quantified data from the faces, and the search compares the face data to the face database, finding the closest match, if one exists. Face detection uses a Viola-Jones algorithm, a lighting-based detector, to find key facial features in an image in rectangular sections compared against one another. The next step in face recognition is projection. Projection reduces each of the detected faces into "Eigenfaces" by removing data of lesser importance from each facial image. PCA is the fundamental tool used in projection and will be given its own section here; however a deep understanding of it is not required to understand most of facial recognition. The database is constructed using the same set of routines employed during real-time operation of projection. This not only simplifies the searching process by reducing it to a comparison of identical data structures, it streamlines the program implementation by relying on the same set of functions for database generation and real-time operation. Searching will then find the most closely matching database face to the newly projected face. If the face is matched with sufficient confidence, the algorithm completes by outputting correlated identification data to the user.

Face Detection (FD)

As humans have eyes, machines have a camera. *Images,* to which we constantly refer, are two-dimensional data constructs consisting of picture elements or *pixels*. In the case of real-time perception, images come and go and persist only for a small interval of time, which we refer to as *frames*. We begin the detection process assuming that an external camera has supplied a frame containing an image that may or may not contain a face or faces (Goldstein, Harmon, & Lesk, 1987). This image resides in the memory of the machine and is accessible to subsequent processes. The Viola-Jones detection algorithm (Viola &

Jones, 2001) uses the brightness of each pixel in the grayscale-converted image to determine what areas of an image are faces by locating general features, and then progressively becoming more specific. The first step in this is representing the relative brightness efficiently, since this algorithm uses this data in a specific way. Each pixel will be represented in terms of the summed pixel intensities above it and to its left, a construct termed *integral image*. Also referred to as *summed area table*, it is a representation that efficiently allows a description of a neighborhood of pixels (Crow, 1984).

With this representation, the total brightness of a rectangular section of the image can be determined by adding and subtracting the summed intensities of the rectangle's corner pixels. For example, the shaded area in Fig. 1 can be evaluated by adding the values at points *A* and *C*, and subtracting the values at *B* and *D* (Yang, Kriegman, & Ahuja, 2002). Additionally, the total intensity of adjacent rectangles can be determined with fewer values, since some of those values are included in the other rectangle representations. More specifically, a single rectangle's intensity sum can be represented with four values, two adjacent rectangles can have each of their sums represented with six values, three rectangles with eight values, and four rectangles with nine values.

Figure 1. Determining the intensity of an image area using only four corner values.

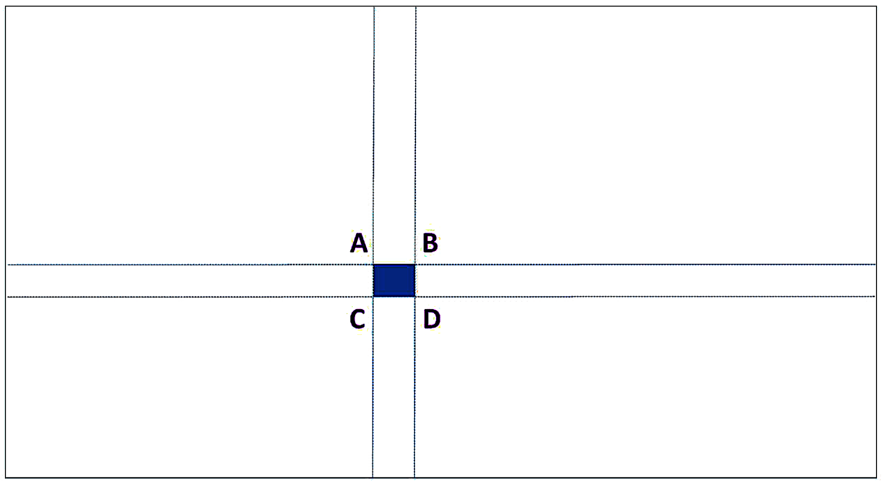
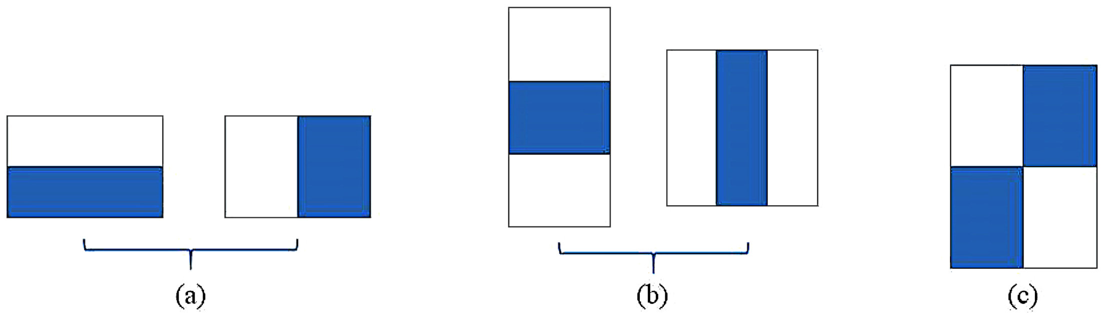


Figure 2. (a) two-rectangle features. (b) three-rectangle features. (c) four-rectangle features.



The locating is then accomplished by comparing the total intensities of adjacent rectangular sections of an image, so it is clear why this summed intensity representation of the pixels is useful. There are three main patterns which classify each feature found: two-rectangle features, three-rectangle features, and four-rectangle features, each potentially oriented horizontally, vertically, or diagonally, shown on Fig. 2.

Certain facial features can be recognized as objects in the image based on the way these simple rectangular features are patterned. For example, eyes and the nose are easily recognized with this method because the lighting value of eyes is typically significantly darker than the surrounding areas while the lighting value of the nose is significantly lighter than the surrounding areas. If these recognized features are positioned in a certain way, then that area of the image could potentially include a face. All faces will have those areas, so the algorithm can then throw out the non-face areas and continue processing on the face areas. One might wonder how the algorithm knows how to do this, and it turns out that the algorithm itself must be *trained* to detect faces in this way.

AdaBoost, or *Adaptive Boosting*, was initially introduced by Freund and Schapire in 1995 (Freund, Schapire, & Abe, 1999), and is widely used to solve pattern recognition problems. It is an iterative machine learning algorithm employed in order to form an optimal set of constructs which guide the detection process. The result is to determine features which will most quickly begin to weed out non-face areas. Those features, like the nose and eyes, will be identified first, while other features are progressively added to the detection process. This cascade of identifications maintains the high accuracy of the detection and reduces false positives as much as possible while minimizing fruitless or redundant processing.

Its core idea is to respond to training by different classifiers for the same set of object types, so called *weak classifiers*, and then generate a strong classifier out of those weak classifiers. The algorithm itself is implemented by data reallocation. At each iteration, a new weak classifier is added, for which we assign a weight that indicates the possibility for this classifier being selected for every sample in the training set. Therefore, those well-classified samples would have less possibility to be selected in the next set, and vice versa. In this way, the algorithm tends to concentrate on the information-rich samples that are also harder to classify.

A general description of Adaboost:

1. Begin in sample set $\{x^1, y_1, x^2, y_2, \ldots, x^n, y_n\}$, maximum number of iterations $T_{\max}$, and weight for the samples in each iteration $W_{t,i}$ where t denotes the iteration and i denotes the sample numbering $1,2,\ldots,n$
2. For t=1 to $T_{\max}$,
    a. Using $W_{t,i}$ to train the weak classifier for the sample set
    b. Calculate the training error ε_t for this weak classifier
    c. Regarding ε_t, set $\alpha_t = \frac{1}{2}\log\frac{1-\varepsilon_t}{\varepsilon_t}$
    d. Applying a normalization constant, update the weight for next iteration
3. Return weak classifiers and α_t for every tth iteration.

The final adjudication, which we call the strong classifier, can be obtained as a weighted summation of the weak classifiers.

The AdaBoost algorithm offers many advantages. Primarily, the final strong classifier has a relatively high accuracy comparing to other similar method like Bootstrapping and Bagging. As AdaBoost provides a framework instead of particulars, various schemas can be applied to build the weak classifiers, which could be very simple and easy to understand . Finally, AdaBoost doesn't need feature selection, thus developers wouldn't need to worry about over fitting. This customization of feature selection can also be used to train classifiers to detect virtually any application-specific object or set of objects such as vehicles, text patterns or flora/fauna. This face recognition application particularly employs a *frontal face* cascade classifier. The quality of detection with this algorithm is high, such that positive detection rates approach 100% quickly (approximately 10 times faster than a similar non-cascaded algorithm) and false-positive rates stay on the order of 0.1% (Soyata, Muraleedharan, Funai, Kwon, & Heinzelman, 2012).

Once a region in the source frame has been evaluated to contain a face or faces, the OpenCV library function cascade_gpu.detectMultiScale forms an object buffer containing the coordinates of rectangles bounding each face. These 'bounding boxes' are used to define sub-images (containing only a face) which are separated from the source frame and returned to the caller as a list, or vector of image fragments. OpenCV employs the *Mat* datatype, which stands for Matrix. Since we are working so extensively with two dimensional image data, this type is well suited for our application. Each sub-image *Mat* contains a face whose identity remains unknown at this point. In order to complete the recognition process, the machine must have a basis of reference. This requires a training set of images and corresponding identification data. The training set must be used to initialize the Face Recognizer, a process which must be completed before beginning a recognition routine. We bring it up now because computationally, it is nearly identical to the process following detection; projection.

Projection (PJ)

Projection is wholly a data conversion process. Beginning with a two dimensional pixel array (the cropped sub-image of a face), the image ultimately becomes represented as a set of orthogonal coordinates in a multi-dimensional Eigenspace; a single point. The dimensionality of this space can be selected by the developer before runtime and has a large impact on the number of operations (thus time) it takes to perform a complete recognition routine, including projection and searching. The obvious tradeoff is representation accuracy; an external requirement based on the degree of variation between subjects. In typical cases projection significantly reduces the amount of data required to uniquely represent a face. While many details are lost (in comparison with the actual image of the face), the idea is to retain just enough information necessary for a positive identification.

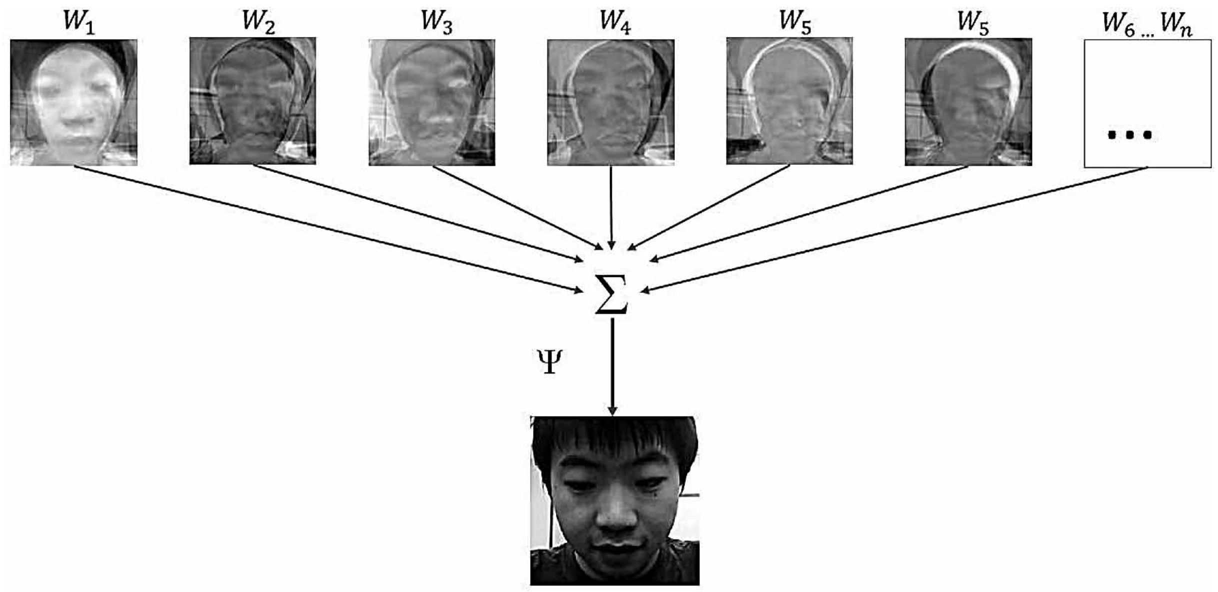
Each dimension of this Eigenspace is represented by something called an "Eigenface". What is being said here, is that a small set of Eigenfaces are sufficient to describe a large data set of unique faces. Using the linear combination of each Eigenface, with each dimension weighted by the test (detected) face projection coefficients (p_k), the expectation is that the result is unique and sufficiently differentiable from any other potential results. This means that the result from any weighted linear combination of Eigenfaces must have a sufficient region of emptiness about it ("headroom") in the Eigenspace in order to preserve the integrity of prediction accuracy. This is achieved by selecting an Eigenspace dimensionality appropriate to the number of objects in the database, as well as a quasi-quantitative degree of variation in their appearances.

The maximum possible number of Eigenfaces for a data set is the number of unique elements, and for Face Recognition this is the number of unique training images. This level of representation can quickly become intractable, as the amount of raw data required can exceed the memory capacity of typical machines. Furthermore, the computational impact of such a large data structure might preclude real-time performance. The following method does allow for an acceptable degree of accuracy while using a relatively small selection of Eigenfaces by assigning an eigenvalue to each Eigenface (Sirovich & Kirby, 1987). This eigenvalue quantifies the corresponding Eigenface's degree of similarity to the *Mean* face; a per-pixel arithmetic average of every training image used to create the initial database. For a database consisting of *n* images, the percentage of information *P*(*k*) covered by the first *k* Eigenfaces can be calculated as:

$$P\left(k\right)=\frac{\sum_{i=1}^{k}\left(\sum_{j=1}^{n}\varphi_{i,j}\right)^{2}}{\sum_{i=1}^{n}\left(\sum_{j=1}^{n}\varphi_{i,j}\right)^{2}}$$

where the eigenvalue $\varphi_{i,j}$ is the i^{th} component of the eigenvector for the j^{th} image in the database. The precision of the algorithm will increase by retaining a larger number of vectors, which presents an obvious trade-off between accuracy and performance. Resultantly, in order to represent the same amount of information, the necessary ratio of Eigenfaces to training images varies inversely with the size of the database. In Fig. 3, an example is shown where the bottom image is calculated as a linear weighted average of the Eigenfaces on top . Counteracting this assertion is the need for a greater diversity of possible descriptors, i.e., eigenvectors in order to make differentiable representations of unique facial

Figure 3. Specifying a face in terms of weighted averages of Eigenfaces

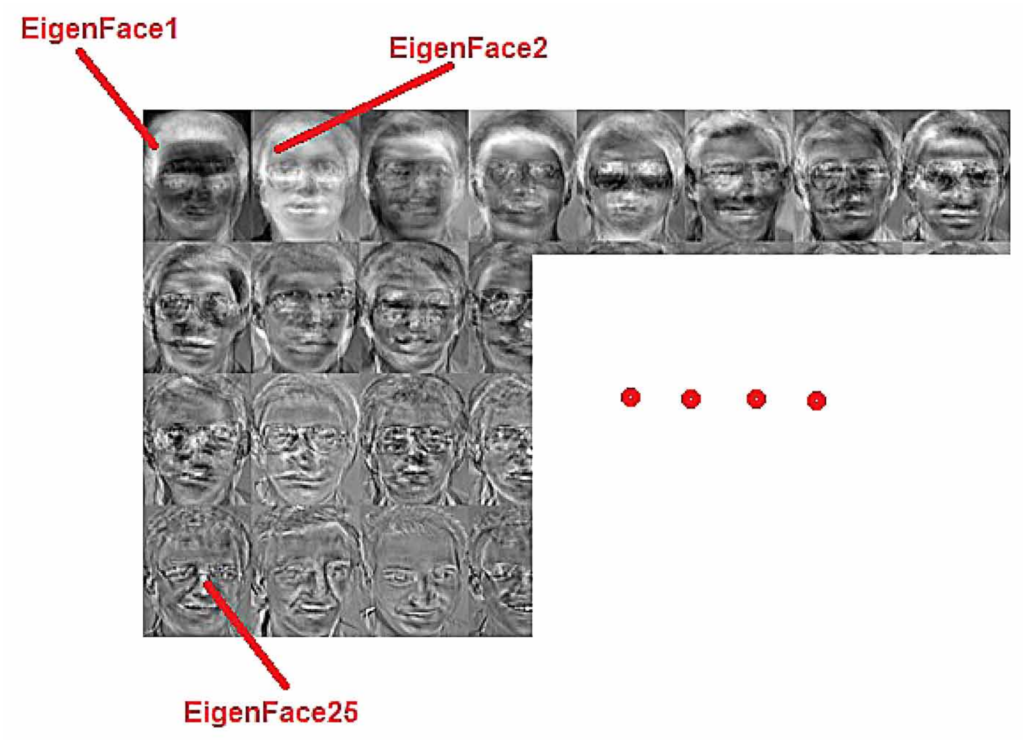


features. Thus the optimal number of eigenvectors used in any application is a point of balance which depends on the number of elements in the database (images) and the degree of variation between them. This variation can be altogether qualitative; optimization might require the application of heuristics, largely by trial and error.

It can be easily inferred that the face can only be perfectly recreated from coordinates in theory, particularly when a face was not part of the initial database. However, complete accuracy is by no means required and is also not practical. Similar to how the application of cascade classifiers makes decisions on what features to use during detection, the projection step uses the statistical method of principal component analysis (PCA) to figure out which information can be discarded with minimal impact on the quality of the representation.

As with eigenvectors in mathematics, Eigenfaces each have an associated eigenvalue. These eigenvalues are used to quantify the quality of data. In terms of each face, the eigenvalues are a measure of how significant the Eigenface information is to the database, and more specifically are a calculation of how different an Eigenface is from the average of the projected faces; the Mean face (a direct result of using PCA). A sum using Eigenface eigenvalues can be used to determine what portion of data that Eigenface covers. From the nature of PCA, the dimensions formed earlier on will contain the largest portion of relevant information. Subsequent dimensions will contribute information in a decreasing order of relevance. If one dimension's contribution is sufficiently small, which is true for many potential Eigenfaces, it should not be included in the set of Eigenfaces. To give an example for how much reduction is typically done while still maintaining accuracy, a set of 500 faces has been reasonably represented with 29 Eigenfaces, shown in Fig. 4.

Figure 4. A set of 29 Eigenfaces is sufficient for a database of 500 training images. Adapted from (Soyata, Muraleedharan, Funai, Kwon, & Heinzelman, 2012).



Creating the Eigenspace and database isn't real-time level processing, so both are precompiled and the Eigenfaces are loaded into memory for the actual facial recognition. In order to match a new face with the database, its image must be processed to have the same pixel dimensions as the database images. This process is intermediate and executed immediately after detection on each sub-image that is to be projected. Additionally, the sub-image is made grayscale and histogram equalization is applied in order to enhance the contrast between neighboring pixels and dominant regions. This allows for notably different features to be recognized, since the new face image is then subtracted from the Mean face. This is then projected onto the Eigenspace where the new face is represented as a point in the theoretical multi dimensional space, and within the computer's memory as a linear vector of floating-point numbers. NOTE: implementation with GPUs may require a sensitivity to the floating point precision specified in the source code. The loss of performance when utilizing double precision may not be justified by the level of prediction accuracy gained as a result of a finer-grained Eigenspace. As stated before, the location of all possible projections in the Eigenspace must be given appropriate "headroom" by a prudently configured database compilation. If two neighboring projections in the Eigenspace require the use of double precision floating point variables in order to make a definitive distinction, it is more likely a consequence of a database compilation that used too few dimensions/Eigenfaces. Optimization thus begins with a clear understanding and use of PCA.

Reducing Dimensionality using Principal Component Analysis (PCA)

Principal component analysis is a relatively old but highly successful mathematical technique with many applications. Its primary strength is being able to take related data and parse it into each data point's most significant or unique components when compared with the dataset as a whole. This, in effect, reduces the raw amount of information required in representing each data point, starting with the least impactful removals. Part of this removal results in the reduction of dimensionality of the dataset, or in other words reducing the number of variables that mathematically represent each point. In terms of our facial recognition algorithm, this means that PCA allows the efficient and quantitative representation of a large number of faces in a way that automatically parses similarities and differences between them.

PCA for facial recognition starts off by reframing the two dimensional pre-processed sub-image as a linear vector. Next, an element-wise Mean of all these images is created. Instead of each face image being processed as itself during projection, it is instead processed as its difference from the Mean image. This "normalization" is a step similar to many basic statistical operations, such as the calculation of variance. Similarities such as this are prevalent throughout the details of PCA and should give some intuitive idea as to the nature of the analysis.

Currently each data point that we have represents an image of dimensionality N times M where N and M are the width and height of each normalized image. The value of each of the data point's components is the intensity of the corresponding pixel compared to the intensity of the mean image's corresponding pixel. Therefore, we have a space with N times M axes. Say each image were 50 by 50 pixels – that means we have 2500 axes, even if we have just one image! Compare this to the end result, which is, as in a previous example, a set of 500 images represented on 29 axes.

In order to accomplish such a reduction, PCA creates each axis iteratively with the highest impact axis first. Let us work in a 3 dimensional space for simplicity in explanation. This is a drastic simplification, since that would mean each image would be 3 by 1 or 1 by 3 pixels, however this process is easily generalized to any number of dimensions. The first axis is a line going through the origin (the centroid) of the data and through the line which is as close as possible to every data point in the set. In other words, if you were to sum the Euclidian distances from each data point to the nearest point on the line, this line would be such that every other line would have a greater or equal sum. Yet another way of saying this is that this line goes through the highest variation of the dataset. The next axis is determined the same way, except that it must be orthogonal to the first axis. In this case, we can easily understand the orthogonal requirement as projecting the data points to the plane such that the first line runs directly into or out of it.

We could completely specify the dataset by adding a third axis, however the point of PCA is to reduce the dimensionality of the dataset, so the third axis will be omitted. As stated previously, each new axis is less impactful to the data than the one before. We can quantify how impactful axes are by how small their respective eigenvalues are, and in most datasets, especially those with an initially large dimensionality, omitting some of the later developed axes is a highly reasonable optimization.

Search (S)

The compiled database consists of an ordered set of coordinates. These coordinates are of the dimensionality of the Eigenspace and represent the projections of each image used in its construction. Ideally, the images used are as similar to the manner in which images will be presented to the computer, i.e., faces oriented to the front with a particular aspect ratio and relative zoom level.

Searching essentially calculates the standard Euclidian distance between the test face projection and the projections in the database. This, unfortunately, requires at least a partial comparison with every single element of the database set. The cost of this operation grows super-linearly with the size of the database, so the current method described definitely can stand some optimizations. The test face is finally identified by the correlated information pertaining to the closest projection in the database. If the minimum distance is not within that threshold, the search algorithm may mark this input face as a new face (one that is not already in the database). This face can then potentially be added to the database if desired, or just left as "not a match" and ignored/discarded. Additionally, if the distance is particularly large, the search process may be configured to tag the input image as not a face. This would mean that there was a false positive in the detection algorithm. Particular feedback methods are entirely possible, although retraining the cascade classifiers for detection have not been tried during runtime.

INSTALLATION OF THE FACE RECOGNITION SOFTWARE

This particular implementation of face recognition is a system of a number of pre-existing technologies and algorithms working together. The installation of this face recognition software is therefore somewhat involved and specific, and so this section explains in detail the steps for proper installation.

Required Base Libraries and SDKs

Firstly, a Windows-based PC platform is required for this face recognition software, since much of the required contributing software is Windows-only. Additionally, this software is configured to use an Nvidia GPU, providing a significant performance increase over strict CPU usage. It follows that a compatible Nvidia GPU is necessary for running this software. A list of such GPUs can be found at https://

developer.nvidia.com/cuda-gpus. Additionally, the user should make sure that GPU drivers and other related software are updated to the most recent versions. What follows is a list of prerequisite software technologies used in this face recognition implementation.

- Nvidia CUDA (version 6.5)
- Nvidia Nsight
- Microsoft Visual Studio 2010
- Open CV (version 2.4.9) and the altered source files (OpenCV-WEB; Bradski & Kaehler, 2008).
- CMake (version 3.0.2)

The most recent versions tested at the time this chapter was written are noted, however users are encouraged to attempt to use the most recent versions of the available technologies. One exception to this is Visual Studio. At the time of writing, CMake does not yet support integration of the most recent release of Visual Studio, but does support integration with the 2010 and 2012 releases.

Purpose of Base Libraries and SDKs

OpenCV is the most integral part of this face recognition system. OpenCV provides reliable and recent programs and algorithms directly involved with face recognition, including the detection, projection, and search processes. Additionally, using OpenCV ensures that this package can be used in a widespread and standardized way. The following snippet from the OpenCV about page gives a concise overview of its value and purpose. "OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products."

Nvidia CUDA is the primary software which makes GPU acceleration viable (Nvidia CUBLAS; Nvidia CUFFT). CUDA is a framework built to allow users to easily take advantage of the processing power of modern GPU devices in an optimized way. For many applications, especially those regarding data derived from visual sources, computation with GPUs is much faster than computation with CPUs. Performance increases in this particular application of CUDA will be discussed later. Nvidia also provides toolkit software with CUDA, which contains tools for easy integration with C and C++ development, among other things. Nsight is a software bundled in the toolkit that integrates CUDA development into the Visual Studio environment.

CMake provides an easy method for integrating OpenCV libraries with Visual Studio 2010. CMake will create the proper binaries (ie. executables and dynamic-link libraries) which can then be easily used in the C and C++ code created to implement the algorithms and programs that OpenCV provides. CMake is built with goals in mind which are similar to those cited by OpenCV, and is hence also open source. Therefore, CMake also provides advantages such as availability and standardization for end-users.

Visual Studio is the software which unifies all the other technologies. CMake and the CUDA toolkit provides ways for developers to directly implement face recognition and GPU optimizations in the programs coded with Visual Studio. Additionally, Visual Studio is useful for its primary intended purpose as an IDE (Integrated Development Environment), making coding more user-friendly with built-in debugging, code building and compilation, and other features.

Installation of the Software Components

- **Visual Studio:** Start the installation process off by installing Visual Studio if it is not already installed. Visual Studio can be purchased at http://www.visualstudio.com/downloads/download-visual-studio-vs, but can also be often downloaded for free for academic uses by students at https://www.dreamspark.com with a verified DreamSpark account. Installation for Visual Studio is mostly automated and may depend on your method of acquisition. Microsoft provides general instructions at this webpage http://msdn.microsoft.com/en-us/library/e2h7fzkw.aspx.
- **CUDA:** The proper downloads for CUDA can be found at https://developer.nvidia.com/cuda-downloads. Run the CUDA installation exe and follow the on-screen instructions. For most cases, choose the express installation option. Again, installation here is mostly automated and the on-screen instructions should be sufficient. When installation finishes, verify that some Nsight tools have been installed depending on your installed versions of Visual Studio.
- **OpenCV:** Download OpenCV v2.4.9 from http://sourceforge.net/projects/opencvlibrary/files/opencv-win/. General installation instructions can be found at http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html#windows-installation, however relevant and more specific material will also be presented here. To begin installation, run the opencv-2.4.9 executable as administrator and extract it to your root hard drive folder (e.g. C:\). Now, open up command prompt (cmd.exe) as administrator and enter the following code, exchanging the directory for where you installed OpenCV. Exclude the –m call if you do not want this installed for all user accounts on the computer.

```
setx -m OPENCV_DIR C:\OpenCV\Build\x86\vc10 (suggested for Visual Studio 2010 - 32 bit Windows)

setx -m OPENCV_DIR C:\OpenCV\Build\x64\vc10 (suggested for Visual Studio 2010 - 64 bit Windows)
```

Command prompt should return "SUCCESS: Specified value was saved." after entering the proper code. Next, find the provided modified OpenCV source files (they should include a build and a sources directory). In a separate file explorer window, find the opencv directory. Inside this should also be a build directory and a sources directory. Before copying over the new folders, consider creating backups of the relevant old folders and files, which can be determined by browsing the modified source folder. When ready, copy over the build and sources folders from the modified source files and into the OpenCV directory.

- **CMake:** Go to http://www.cmake.org/download/ and download the CMake Win32 Installer file. Run this installer. Mark the option that says "Add CMake to the system PATH for all users" or "current user" depending on your preference, and click next. The rest of the installation options are user preference.
- **Building with CMake:** Run cmake-gui.exe in the bin folder of the CMake directory. For the "Where is the source code" field, browse and select the "sources" folder found in the opencv directory. For the "Where to build the binaries" field, browse and select the "build" folder found in the opencv directory. Click the Configure button at the bottom of the CMake window and select the proper version of Visual Studio (e.g., Visual Studio 10 2010 Win64). Press finish and wait for the generator to complete. It is okay when red items pop up in the two UI fields. Now go

to Options and check off “Suppress dev warnings.” Press configure again. Now there should not be any red text in the lower field. If this is true, continue to press the Generate button. Boot up Visual Studio. Browse to the opencv\build directory and open up opencv.sln in Visual Studio. Wait until Visual Studio has finished processing the initial startup, indicated by saying “Ready” in the bottom left. Next, build the Release and Debug binaries. In Visual Studio 2010, do this by selecting “Release” in the leftmost drop down menu in the top toolbar and press F7. When this completes, repeat the same thing for the Debug selection in the same drop down menu. This may take a while. Each build will be finished when the drop down menu becomes selectable again.

OPENCV-BASED IMPLEMENTATION OF FACE RECOGNITION

OpenCV has built-in implementations for the three main face recognition processes (detection, projection and search). These processes are accomplished sequentially in the order they were presented in this chapter. The specific APIs and algorithms which contribute to Open CV’s model of face recognition will be explained in a more code-level scale in the following sections after a brief overview of how Open CV’s implementations relate to the general face recognition processes.

OpenCV face detection uses algorithms derived from the Viola-Jones method of face detection and also the machine learning “boosting” system AdaBoost for choosing classifiers used in detection. This set-up requires some input data so that the boosting system can “train” to recognize the probabilistically most useful classifiers, which in turn allows for classifier usage to be effectively cascaded, increasing detection efficiency. This training is done by feeding OpenCV a set of database face images. These images must first be histogram equalized and equally sized before they are loaded into OpenCV.

The rest of the face detection process is essentially identical to the general processes described earlier. OpenCV will apply the feature detection algorithm in a cascade determined by the training algorithm, sorting out faces from non-faces in the input images. The Viola-Jones rectangular features are slightly different in the OpenCV implementation when compared to the general implementation described previously. First, the three types of features are 2-rectangle features, 3-rectangle features, and center-surround features (a rectangle within a rectangle), plus the ability to identify these features diagonally, rather than the 2-, 3-, and 4-rectangle vertical and horizontal features. Furthermore, particularly in the code of OpenCV, these features are referred to as “Haar-like” features or “Haar classifiers” due to their similarity to the mathematics term “Haar wavelet.”

Face Detection Algorithm Implementation

OpenCV provides two main classes for object detection: cv::gpu::HOGDescriptor, where HOG stands for Histogram of Gradients and cv::gpu::CascadeClassifier_GPU. Nevertheless, the former one doesn’t provide pre-trained module for face features and is thus mainly used in body detection. What we use is the latter one cv::gpu::CascadeClassifier_GPU, which is derived from its CPU version cv::CascadeClassifier. Below are some class APIs we may use.

```
1. gpu::CascadeClassifier_GPU::CascadeClassifier_GPU
Class constructor. Both haar classifier and NVIDIA's nvbin are supported.
2. gpu::CascadeClassifier_GPU::empty
Return a Boolean value to check if the classifier is successfully loaded.
3. gpu::CascadeClassifier_GPU::load
Destroy the old loaded classifier and load a new classifier.
4. gpu::CascadeClassifier_GPU::release
Destroy the current loaded classifier.
5. gpu::CascadeClassifier_GPU:: detectMultiScale
```

This is the core function of the class. The function would return the number of detected objects, and those detected objects are returned as a list of cv::rectangles.

This class is used for detection and can only be used with an existing pre-trained set. OpenCV provides several pre-trained feature sets include eyes, faces and so on. OpenCV also provides a separate class CvCascadeClassifier for classifier training. This class has only one public API:

```
CvCascadeClassifier::train (const std::string _cascadeDirName,
const std::string _posFilename,
const std::string _negFilename,
int _numPos, int _numNeg,
int _precalcValBufSize,
int _precalcIdxBufSize,
int _numStages,
const CvCascadeParams& _cascadeParams,
const CvFeatureParams& _featureParams,
const CvCascadeBoostParams& _stageParams,
bool baseFormatSave = false ;
```

The function needs two types of samples sets: positive and negative, where positive samples includes the objects while negative samples doesn't. An executable program opencv_createsamples can be used to prepare the samples. After the function finished, the trained cascade would be saved in an .xml file.

Eigenvector Projection Algorithm Implementation

In OpenCV, face recognition is performed by a synthetical class cv::FaceRecognizer. cv::FaceRecognizer contains a series of virtual member functions that are implemented by different algorithms. What we are going to use is one of its subclass cv::Eigenfaces corresponding to the Eigenface algorithm.

Considering a set of sample vectors, $S=\{S_1, S_2, \ldots, S_n\}$

1. Firstly, calculate the mean face vector, $\mu = \frac{1}{n}\sum S_i$
2. Let M be the set of mean-subtracted sample vectors. Using the mean face vector to compute Covariance Matrix $\Gamma = MM^T$
3. Covariance Matrix gives the eigenvectors v_i and λ_t as

$$\Gamma v_i = MM^T v_i = \lambda_i$$

OpenCV doesn't provide GPU-accelerated Eigenface algorithm, thus we make some modification of the source code by adding a GPU version projection function. In that way, our cv::Eigenfaces will have the following core member functions.

```
1. Eigenfaces::train
```

Trains the face recognizer with a vector of images and corresponding labels in C++ STL format. This forms the matrix of basis vectors used in the conversion of image data to Eigenspace coordinates (the projection).

```
2. Eigenfaces::update
```

Update the face recognizer with an additional vector of images and corresponding labels.

```
3. Eigenfaces::search
```

Take an input image and return the prediction label as well as the distance. Return -1 and DBL_MAX if no matching found.

```
4. Eigenfaces:: project_GPU
```

Applying a new added function subspaceProject_GPU to compute the projection in PCA subspace. In the function, we use gpu::gemm instead of gemm to leverage GPU acceleration.

```
5. Eigenfaces::save
```

Save the current Eigenface model for later use.

```
6. Eigenfaces::load
```

Load a saved Eigenface model instead of constructing the database from scratch.

Database Search Algorithm Implementation

After the projection is calculated, the Euclidean distance between the test face projection and all database projections are calculated. The nearest point in the Eigenspace is termed the *prediction*, and whether this distance is below a configurable threshold determines its validity. Euclidean distance is simply the line segment between points *p* and *q*, for an *n*-dimensional Eigenspace,

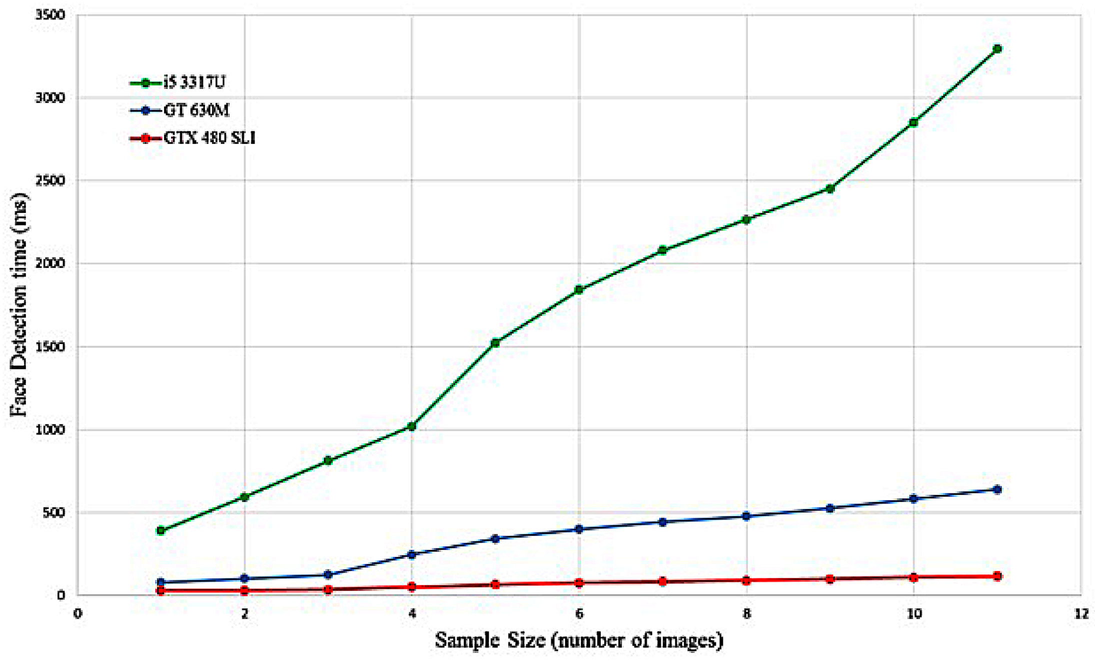
$$D = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2 + \cdots + (p_n - q_n)^2}$$

Database search algorithm is implemented by the API Eigenfaces::search in which it uses cv::norm to calculate the Euclidean distance. Eigenfaces::search is a recursive procedure for traversing the samples and return the minimum distance as well as the corresponding *label*, or the stored data correlating to the image whose projection was nearest to the projection of the test face. This implementation simply uses a string containing the name of the individual, although it would be possible to return data structures containing more detailed information as well. If all distances are larger than threshold, the function returns -1 and DBL_MAX. Note that it is not necessary to calculate the square root of the sum of differences squared in order to find the nearest neighbor. In the spirit of optimization, reducing the number of square root operations by the number of database images for *each frame* might become noticeable for large databases.

EXPERIMENTAL EVALUATION

Figure 5 shows the experimental result of Face Detection on different platforms over several sample sizes. Eleven source frame samples were used containing one to eleven faces, respectively. We use three platforms, a purely CPU (i5 3317U), an entry-level mobile GPU (GT 630M) as well as a powerful desktop GPU (GTX480 SLI). Detection via the GPU shows a remarkable performance increase over pure CPU implementation (Kirk & Hwu, 2010). OpenCV libraries effectively leverage the parallel processing capability of the GPU in order to perform its cascade evaluations and image parsing. Numerous tests over a range of devices showed that even a commercial grade NVIDIA GPU can perform detection of as many as one hundred faces in a single frame, within tens of milliseconds. The number of faces detected per frame, in fact, has a measurable, although negligible effect on the frame-detection rate when using a GPU. The largest impacting factor on the computation rate of each frame would be the pixel density.

Figure 5. Processing time for Face Detection for different processors over several image samples that contain different number of faces.



Applications requiring real-time performance ought to consider the expected aspects and scaling of the images being presented to the algorithm and adjust system resolutions accordingly, since detected frames are resized to accommodate the operations of linear algebra in the projection method. For example, applications involving single-object frame detection, such as biometric security, will expect the input data to consist of a detectable object nearly filling the frame. This would not require a high frame resolution upon implementation. Whereas applications involving multiple-object frames such as street-view security would have to operate at higher resolutions to process smaller areas of the frame in order to generate results with an acceptable degree of accuracy. These tradeoffs are entirely testable, and fairly easy to alter in the source code.

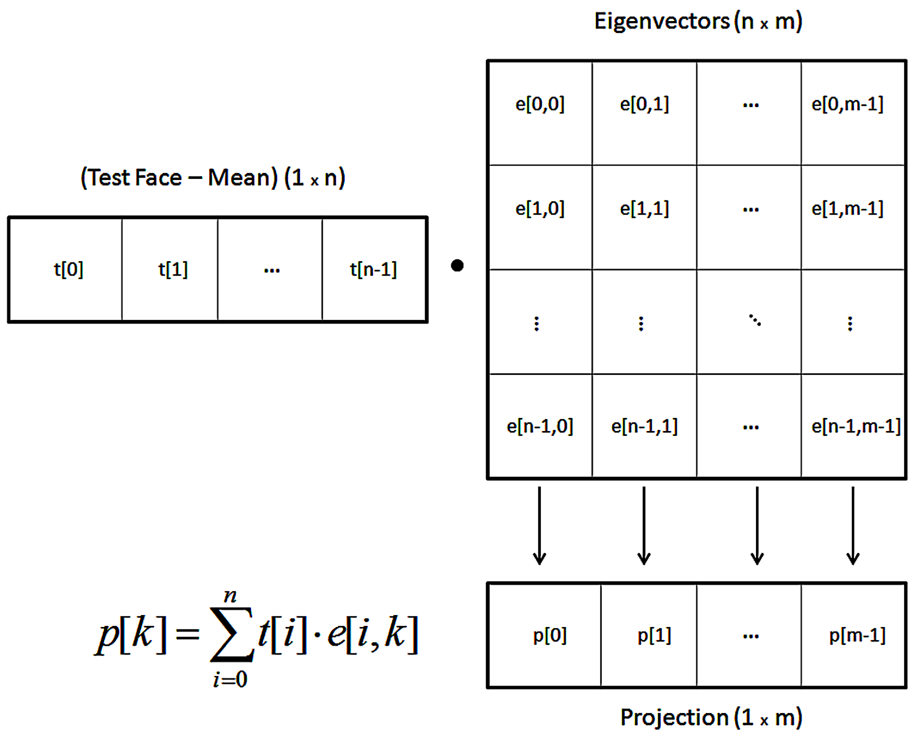
Once an indexed list of face-containing rectangle boundaries are generated by the detection process, each sub-image defined by its rectangle is pre-processed to facilitate its conversion to a point in the higher dimensional Eigenspace. The sub-image (or 'test face') is resized to a standard pixel height H and width W as defined in the training and database generation phase. This is to ensure that the mathematical operations carried out during the projection are valid. The test face, while captured as a color image, is converted to grayscale and histogram equalization is applied in order to accentuate differentiable features. At this point, the two-dimensional $H \times W$ Mat structure of the image is reformed as a $1 \times HW$ linear array, preserving the pixel count, but referencing them sequentially as a scan of the test face image: from top-left to bottom-right.

Within the database file residing in Host memory, there are two structures called upon for operations on every test face. The Mean image is a reformed $1 \times HW$ linear array whose entries are corresponding averages of each image used to train the database. That is, the n^{th} pixel in the Mean is the sum of the n^{th} pixel in every training image, divided by the number of training images used. This structure is persistent, either in a file on disk, or maintained in local memory during application runtime. Its values do not change unless the collection of training images changes and a new database compilation occurs. This Mean image is subtracted from the test face. The Eigenvectors' data structure is a two-dimensional $HW \times m$ Mat structure whose columns consist of the linearized Eigenfaces generated by database training. There are exactly HW rows, since there are HW pixels in each training image. There are exactly m columns, where m is the selected dimensionality of the Eigenspace into which faces are projected for comparison.

GPU implementation of projection requires the Mean-adjusted test face array and the Eigenvectors Mat structures to be uploaded to Device (GPU) memory before executing the core operation of projection: matrix multiplication. This operation is handled by gpu::gemm, a member function of the NVIDIA CUDA library CUBLAS. Recall that the eigenvector consists of (HWD) elements, each consisting of 4 Bytes when using single precision floating point. A standard 640×480 image size with a 29-dimensional Eigenspace would result in an Eigenvector sized approximately 35.6MB. 200GB/s memory transfer speed would ideally transfer this particular Eigenvector between Host and Device in 0.18ms, which for real-time applications may not be an acceptable memory transfer overhead. Increasing resolution images and Eigenspace dimensionalities would obviously have a deleterious effect on that overhead, and this becomes an application-dependent concern. One note to developers; if this overhead becomes an issue, it is always possible to upload the Eigenvector structure to Device (GPU) memory at the initiation of the application and store it there permanently during runtime, effectively removing its contribution to memory overhead.

Aside from the issue of memory overhead, there is the issue of computation. The test face projection is finally calculated as the product of the Mean-adjusted test face array and the Eigenvector matrix, as illustrated in Fig. 6. GEMM forces the GPU kernel to undergo a series of multiply-accumulate operations

Figure 6. Core projection operation: the dot-product of mean-adjusted test face and eigenvector matrix, where n is the per-image pixel density and m is the Eigenspace dimensionality.



(along with indexing and other memory access operations). OpenCV methods streamline this process and the use of CUBLAS allows the problem to be parallelized in the GPU leading to dramatic acceleration. Parallelization enables all *n* elements of the projection p_k to be computed concurrently:

$$p_k = \sum_{i=0}^{n} t_i e_{i,k}$$

Hierarchal granularity could permit the concurrent processing of multiple test faces as well, while pipelining the resultant projections back to host memory for Search operations. Mathematically, the most significant impacting factors of the process time are the image pixel density, Eigenspace dimensionality, the source-defined floating-point precision, the core clock speed of the GPU and the manner in which the GEMM kernel is configured to parallelize the process. Our application using gpu::gemm relied on the compiler to make its own decisions about configuring the CUDA device assembly code, thus the register usage and ALU pipelining was completely out of our scope. This did lead to some unexpected results, such as slower GEMM times on theoretically superior devices. Optimizations will most certainly require a greater level of developer control over the lower level workings of the GPU kernel and the manner in which projection computations are to be carried out.

Once there is a 1×*m* projection array, it must be transferred back to Host memory where it can be compared to the database. Note how much smaller the projection is than the test face. Using our previous example, a 640×480 image of size 1.2MB versus a 1×29 projection of size 116 Bytes implies an approximate 10593:1 data ratio, or 99.99% data reduction. This substantiates the earlier claim that the Eigenfaces method requires far less information to represent uniquely identifiable objects.

CONCLUSION AND FUTURE WORK

This chapter has presented the most significant aspects of a modern implementation of a Face recognition application. First, the theoretical aspects of Face recognition are introduced by describing the three-step implementation of the Face recognition. These steps are Face Detection (FD), Projection (P), and Database Search (S). Each of these steps is described in detail and the underlying Principal Component Analysis (PCA) methodology is elaborated on.

To provide a tutorial for installing and implementing Face Recognition, detailed steps are documented which include the installation of the Open Source Computer Vision library (Open CV) and Nvidia CUDA GPU software components. Additionally, a detailed documentation is provided about which functions in the Open CV library are responsible for which one of the functions in the FD, P, and S steps. Experimental results are provided on certain CPU and GPU architectures to allow the readers to compare their results against our experimental results.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation grant CNS-1239423 and a gift from Nvidia Corporation.

REFERENCES

Bradski, G., & Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc.

Crow, F. C. (1984). Summed-area tables for texture mapping. *Computer Graphics*, *18*(3), 207–212. doi:10.1145/964965.808600

Freund, Y., Schapire, R., & Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society for Artificial Intelligence, 14*(771-780), 1612.

Goldstein, A. J., Harmon, L. D., & Lesk, A. B. (1971). Identification of human faces. *Proceedings of the IEEE*, *59*(5), 748–760. doi:10.1109/PROC.1971.8254

Kim, K. I., Jung, K., & Kim, H. J. (2002). Face recognition using kernel principal component analysis. *Signal Processing Letters, IEEE*, *9*(2), 40–42. doi:10.1109/97.991133

Kirk, D. B., & Hwu, W. M. W. (2010). *Programming Massively Parallel Processors. Hands-on Approach*. Burlington, MA, USA: Morgan Kaufmann Publishers.

Nvidia, *CUBLAS*. (n.d.). Retrieved August 18, 2014, from https://developer.nvidia.com/cublas

Nvidia, *CUFFT*. (n.d.). Retrieved August 18, 2014, from https://developer.nvidia.com/cufft

Open, CV-WEB. (n.d.). *Open CV (Open Source Computer Vision)*. Retrieved August 18, 2014, from http://opencv.org/

Sirovich, L., & Kirby, M. (1987). Low-dimensional procedure for the characterization of human faces. *JOSA A*, *4*(3), 519–524. doi:10.1364/JOSAA.4.000519 PMID:3572578

Soyata, T., Muraleedharan, R., Funai, C., Kwon, M., & Heinzelman, W. (2012, July). Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Computers and Communications (ISCC), 2012 IEEE Symposium on* (pp. 59-66). IEEE.

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on* (*Vol. 1*, pp. 511-518). IEEE. doi:10.1109/CVPR.2001.990517

Yang, M. H., Kriegman, D., & Ahuja, N. (2002). Detecting faces in images: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(1), 34–58.

KEY TERMS AND DEFINITIONS

Database Search (S): The final phase of the face recognition algorithm, in which the re-represented face is compared against the database elements' Eigenface values by calculating their Euclidean distances. The distances that are below a specified threshold are candidates for recognized faces.

Eigenface: Used as the backbone of the Face recognition algorithm. For a given set of database images, a set of Eigenfaces are initially calculated, which simply change the coordinate space. Once this step is performed, every image in the database is re-represented in terms of this new coordinate space (e.g., in terms of Eigenfaces).

Face Detection (FR): The first phase of the Face recognition algorithm. In the FR phase, a face (or multiple faces) are cropped from the image that they are in. This "detected" face is then fed into the rest of the algorithm.

Face Recognition: The process of mapping a newly-captured face to one of the faces in a given database. For example, if a database has 500 images, F_1... F_{500}, for a given face F, the Face Recognition algorithm can map it to one of these 500 faces, i.e., recognize it. Alternatively, the algorithm might simply report that, this image is not in the database, or not even a human face.

NVidia CUDA (Computer-Unified Device Architecture): Nvidia's programming language and the hardware architecture. This language was introduced a decade ago to allow programmability for GPUs, with the intent to turn GPUs into general purpose scientific computation devices.

Principal Component Analysis (PCA): A technique for reducing the dimensionality of multi-dimensional data. For example, a 500 image database can be re-represented in terms of 29 Eigenfaces as described in the chapter. This allows a substantial reduction in the database size, while minimally reducing the accuracy of the new representation.

Projection (P): Second phase of the Face recognition algorithm, in which the detected face is re-represented in terms of the Eigenfaces.