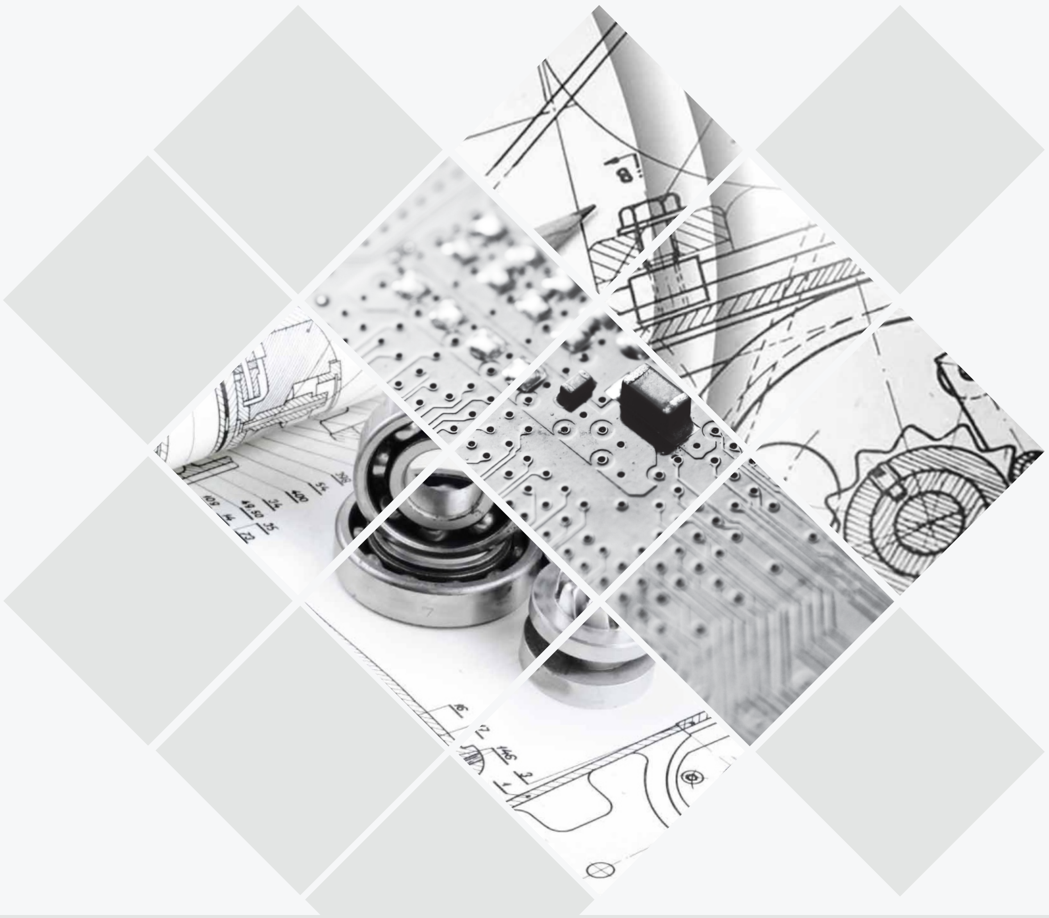
CUDA accelerated face recognition

This paper presents a study of the efficiency and performance speedup achieved by applying Graphics Processing Units for Face Recognition Solutions.



Sibi A

QuEST Global

contents

0.1 Abstract 01

0.2 Introduction 01

0.3 PCA Theory 01

0.4 CPU Implementation 02

0.5 GPU Implementation 03

0.6 Performance Statistics 05

0.7 Conclusion 08

0.8 Future Works 08

CUDA Accelerated Face Recognition

Abstract

This paper presents a study of the efficiency and performance speedup achieved by applying Graphics Processing Units for Face Recognition Solutions. We explore one of the possibilities of parallelizing and optimizing a well-known Face Recognition algorithm, Principal Component Analysis (PCA) with Eigenfaces.

I. INTRODUCTION

In recent years, the Graphics Processing Units (GPU) has been the subject of extensive research and the computation speed of GPUs has been rapidly increasing. The computational power of the latest generation of GPUs, measured in Flops[1], is several times that of a high end CPU and for this reason, they are being increasingly used for non-graphics applications or general-purpose computing (GPGPU). Traditionally, this power of the GPUs could only be harnessed through graphics APIs and was primarily used only by professionals familiar with these APIs.

CUDA (Compute Unified Device Architecture) developed by NVIDIA, tries to address this issue by introducing a familiar C like development environment to GPGPU programming and allows programmers to launch hundreds of concurrent threads to run on the "massively" parallel NVIDIA GPUs, with very little software overhead. This paper portrays our efforts to use this power to tame a computationally intensive, yet highly parallelizable PCA based algorithm used in face recognition solutions. We developed both CPU serial code and GPU parallel code to compare the execution time in each case and measure the speed up achieved by the GPU over the CPU.

2 PCA Theory

2.1 Introduction

Principal Component Analysis (PCA) is one of the early and most successful techniques that have been used for face recognition. PCA aims to reduce the dimensionality of data so that it can be economically represented and processed. Information contained in a human face is highly redundant, with each pixel highly correlated to its neighboring pixels and the main idea of using PCA for face recognition is to remove this redundancy, and extract the features required for comparison of faces. To increase accuracy of the algorithm, we are using a slightly modified method called Improved PCA, in which the images used for training are grouped into different classes and each class contains multiple images of a single person with different facial expressions. The mathematics behind PCA is described in the following subsections.

2.2 Mathematics of PCA

Firstly, the 2-D facial images are resized using bilinear interpolation to reduce the dimension of data and increase the speed of computation. The resized image is represented as a 1-D vector by concatenating each row into one long vector . Let's suppose we have M training samples per class, each of size *N* (=total pixels in the resized image). Let the training vectors be represented as x_i. p_j's represent pixel values.

$$x_i = [p_1 \ldots p_N]^T, i = 1, \ldots, M$$

The training vectors are then normalized by subtracting the class mean image *m* from them.

$$m = \frac{1}{M}\sum_{i=1}^{M} x_i$$

Let w_i be the normalized images.

$$w_i = x_i - m$$

The matrix *W* is composed by placing the normalized image vectors w_i side by side. The eigenvectors and eigenvalues of the covariance matrix *C* is computed.

$$C = WW^T$$

The size of C is *N x N* which could be enormous. For example, images of size 16 16 give a covariance of size 256 *x* 256. It is not practical to solve for eigenvectors of C directly. Hence the eigenvectors of the surrogate matrix $W^T W$ of size M *x* M are computed and the first M - 1 eigenvectors and eigenvalues of C are given by Wd_i and μ_i, where di and μ_i are eigenvectors and eigenvalues of the surrogate matrix, respectively.

The eigenvectors corresponding to non-zero eigenvalues of the covariance matrix make an orthonormal basis for the subspace within which most of the image data can be represented with minimum error. The eigenvector associated with the highest eigenvalue reflects the greatest variance in the image. These eigenvectors are known as *eigenimages* or *eigenfaces* and when normalized look like faces. The eigenvectors of all the classes are computed similarly and all these eigenvectors are placed side by side to make up the *eigenspace S*.

The mean image of the entire training set, m′ is computed and each training vector x_i is normalized. The normalized training vectors w'_i are projected onto the eigenspace S, and the projected feature vectors y_i of the training samples are obtained.

$$w'_i = x_i - m'$$

$$y_i = S^T w'_i$$

The simplest method for determining which class the test face falls under is to find the class k, that minimizes the Euclidean distance. The test image is projected onto the eigenspace and the Euclidean distance between the projected test image and each of the projected training samples are computed. If the minimum Euclidean distance falls under a predefined threshold θ, the face is classified as belonging to the class to which contained the feature vector that yielded the minimum Euclidean distance.

3 CPU Implementation

3.1 Database

For our implementation, we are using one of the most common databases used for testing face recognition solutions, called the ORL Face Database (formerly known as the AT&T Database). The ORL Database contains 400 grayscale images of resolution 112 x 92 of 40 subjects with 10 images per subject. The images are taken under various situations, such as different time, different angles, different expressions (happy, angry, surprise, etc.) and different face details (with/without spectacles, with/without beard, different hair styles etc). To truly show the power of a GPU, the image database has to be large. So in our implementation we scaled the ORL Database multiple times by copying and concatenating, to create bigger databases. This allowed us to measure the GPU performance for very large databases, as high as 15000 images.

3.2 Training phase

The most time consuming operation in the training phase is the extraction of feature vector from the training samples by projecting each one of them to the eigenspace. The computation of eigenfaces and eigenspace is relatively less intensive since we have used the surrogate matrix workaround to decrease the size of the matrix and compute the eigenvectors with ease. Hence we have decided to parallelize only projection step of the training process. The steps till the projection of training samples are done in MATLAB and the projection is written in C.

The MATLAB routine acquires the images from files, resizes them to a standard 16 x 16 resolution and computes the eigenfaces and eigenspace. The data required for the projection step, namely, the resized training samples, the database mean image and the eigenvectors, are then dumped into a binary file with is later read by the C routine to complete the training process. The C routine reads the data dumped by the MATLAB routine and extracts the feature vectors by normalizing the resized training samples and projecting each of them onto the eigenspace. With this the training process is complete and the feature vectors are dumped onto a binary file to be used in the testing process.

3.3 Testing phase

The entire testing process is written in C. OpenCV is used to acquire the testing images from file, and resize them to the standard 16 x 16 resolution using bilinear interpolation. The resized image is normalized with the database mean image and projected onto the eigenspace computed in the training phase. The euclidean distance between the test image feature vector and the training sample feature vectors are computed and the index of the feature vector yielding the minimum euclidean distance is found. The face that yielded this feature vector is the most probable match for the input test face.

4 GPU Implementation

4.1 Training phase

4.1.1 Introduction

As mentioned in Section 3.2, only the final feature extraction process of the training phase is parallelized. Before the training samples can be projected, all the data required for the projection process is copied to the device's global memory and the time taken for copying is noted. As all the data are of a read-only nature, they are bound as texture to take advantage of the cached texture memory.

4.1.2 Kernel

The projection process is highly parallelizable and can be parallelized in two ways. The threads can be launched to parallelize the computation of a particular feature vector, wherein, each thread computes a single element of the feature vector. Or, the threads can be launched to parallelize projection of multiple training samples, wherein each thread projects and computes the feature vector of a particular training sample. Since the number of training samples is large, the latter is adopted for the projection operation in training phase. We have adopted the former in the testing phase, where only one image has to be projected, details of which are explained in Section 4.2.

Before the projection kernel is called, the execution configuration is set. The number of threads per block, T_1, is set to a standard of 256 and the total number of blocks is B_1, where B_1 = (ceil) (N_1/T_1), N_1 = total number of training samples.

Each thread projects and computes the feature vector of a particular training sample by serially computing each element of the feature vector one by one. Each element of the feature vector is obtained by taking inner product of the training image vector and the corresponding eigenvector in the eigenspace. The training sample is normalized with the database mean image element by element as it is fetched from texture memory and the intermediate sum of the inner product with eigenvector is stored in the shared memory. After each element of the feature vector is computed, the data is written back into the global memory and the next element is computed. All the data is aligned in a columnar fashion to avoid uncoalesced memory accesses and shared memory bank conflicts.

CUDA Accelerated Face Recognition

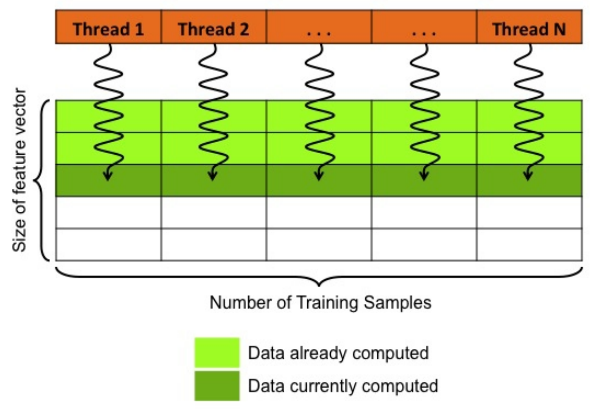


Figure 1: Threads in Projection Kernel (Training)

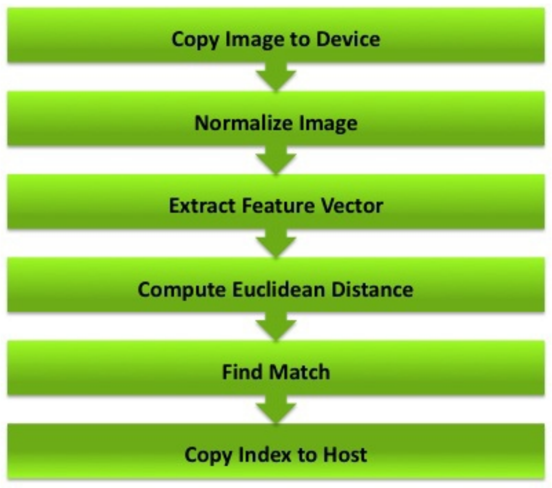


Figure 2: Recognition pipeline

After the kernel has finished running on the device the entire data is copied back to the host memory and dumped as a binary file to be used in the testing phase.

4.2 Testing Phase

4.2.1 Introduction

The testing process is completely run on the GPU and is handled by three kernels. The first kernel normalizes and projects it onto the eigenspace and extracts the feature vector. The second kernel parallely computes the euclidean distance between the feature vector of the test image and that of the training images. The final kernel, finds the minimum of the euclidean distance and index of the training sample which yielded that minimum. The resized test image, database mean image, eigenvectors and the projected training samples are first copied to the device memory and the test image, mean image and eigenvectors are bound as texture to take advantage of the cached texture memory. Due to the relatively larger size of the projected training samples data and the limitation on maximum texture memory, the projected training samples are not bound as texture.

4.2.2 Projection Kernel

As mentioned in section 4.1.2, the projection kernel in testing process is parallelized to concurrently compute each element of the feature vector. The number of threads per block, T_2, is set to a standard of 256 and the total number of blocks is B_2, where B_2 = (ceil) (N_2/T_2), N_2 = size of feature vector.

Each thread computes each element of the feature vector, which is obtained by taking inner product of the test image vector and the corresponding eigenvector in the eigenspace. The test image is normalized with the database mean image, element by element as it is fetched from texture memory and the intermediate sum of the inner product with eigenvector is stored in the shared memory.

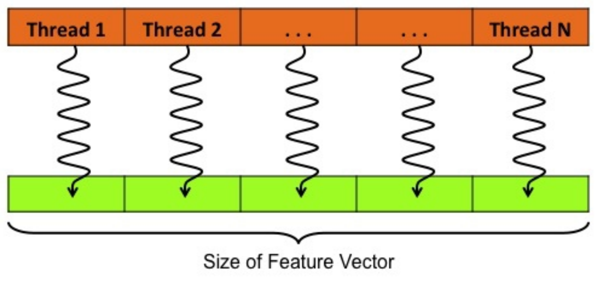


Figure 3: Threads in Projection Kernel (Testing)

After the entire feature vector is computed, the data is written back into global memory. The columnar alignment of all the eigenvectors avoids uncoalesced memory accesses and shared memory bank conflicts.

4.2.3 Euclidean Distance Kernel

The kernel for computing the euclidean distance is very similar to the projection kernel used in training phase. Threads are launched to concurrently compute the euclidean distance between the test image feature vector and the training sample feature vectors. The number of threads per block, T_3, is set to a standard of 256 and the total number of blocks is B_3, where B_3 = (ceil) (N_3=T_3), N_3 = total number of training samples.

Each thread computes a particular euclidean distance serially. The difference of each element of the vectors are computed, squared and summed. The intermediate sum is stored in shared memory. After all the euclidean distances are computed, the data from the shared memory is written to the global memory. The columnar alignment of training sample feature vectors avoids uncoalesced memory accesses and shared memory bank conflicts.

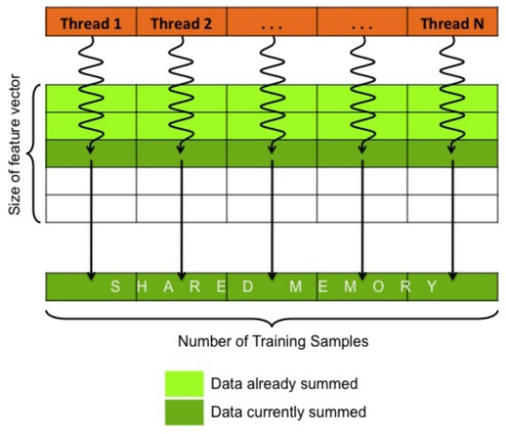


Figure 4: Threads in Euclidean Distance Kernel

4.2.4 Minimum Kernel

The minimum kernel computes the minimum value of euclidean distance and its distance. The vector containing euclidean distances is divided into smaller blocks and each thread serially finds the value and index of the minimum in a particular block. The kernel is called iteratively with fewer and fewer threads till only one block is left. After execution of the kernel the minimum value and its index is copid back to host memory. The training sample at the index computed by the kernel is the most probable match for the test image.

5 Performance Statistics

To test the performance of CPU and GPU, 5 images per subject of the ORL Database was selected as the training set. This set of 200 images was then replicated and concatenated to create databases of size ranging from 1000 to 15000 images. The eigenvectors corresponding to the 4 highest eigenvalues per class, were selected for forming the eigenspace. This led to feature vectors which grew in size as the database grew. This replicated database was trained with CPU and GPU and the execution time was noted and the GPU speedup for the training process was calculated. For testing, one image per subject from the ORL Database were selected and the total time taken by the CPU and GPU to test all 40 test images was noted and was used to calculate the GPU speedup for testing process.

To get accurate performance statistics, the training and testing processes were run multiple times on different CPUs and GPUs. The following graphs were plotted with the data obtained from the performance tests. *All the CPU times are based on single-core performances.*

CUDA Accelerated Face Recognition

Fig. 5 shows the time taken three different CPUs to execute the projection process during training.

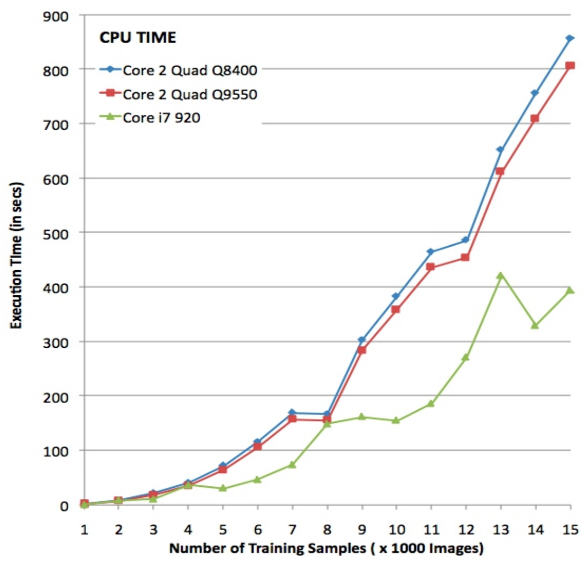


Figure 5: Training time for different CPUs

Fig. 6 shows the time taken by different NVIDIA GPUs to execute the projection process during training. It includes the time taken for data transfers to and from the device.

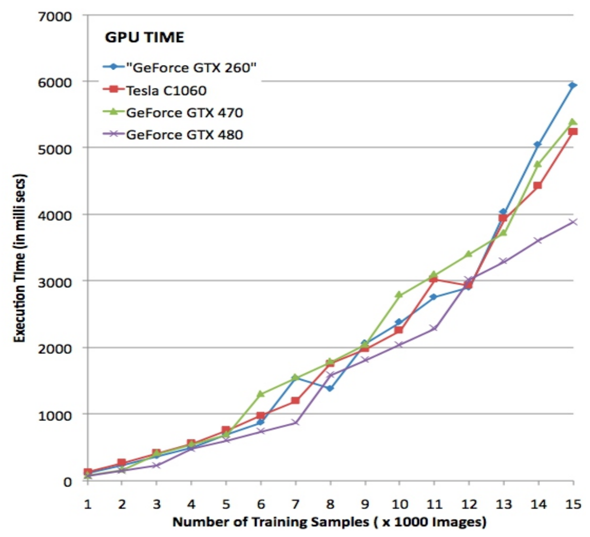


Figure 6: Training time for different GPUs

Fig. 7 shows the performance speedup of different GPUs over Intel Core 2 Quad Q9550 CPU during training databases of varying sizes.

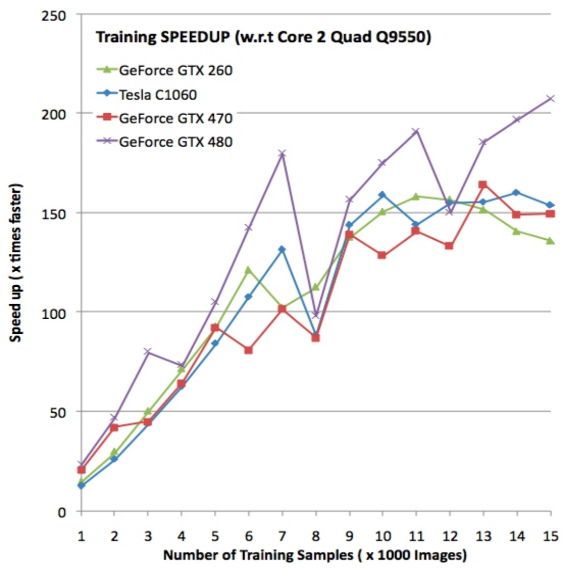


Figure 7: Training Speedup

Fig. 8 shows total time taken by different CPUs for testing 40 images.

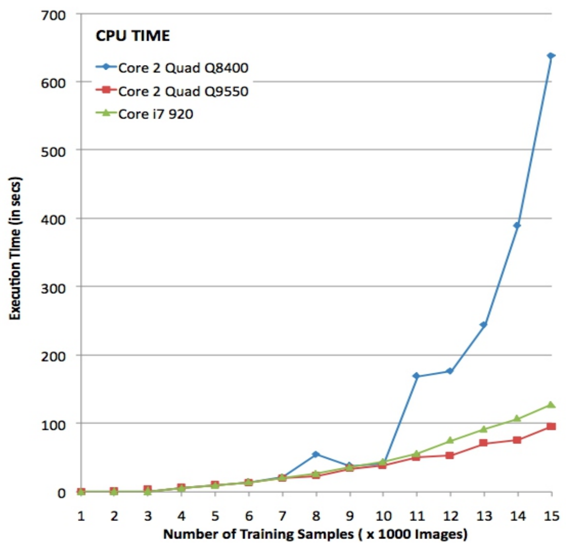


Figure 8: Testing using GPU

CUDA Accelerated Face Recognition

Fig. 9 shows total time taken by GPUs to test 40 images. For this test, the trained database was copied to device memory once and 40 images were tested one by one. It includes time taken for transferring test image to device and getting match index from device.

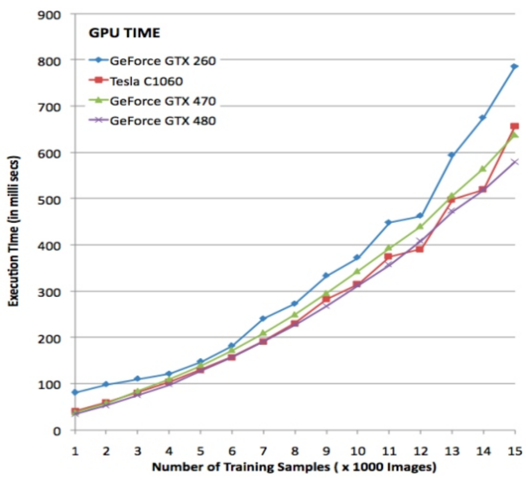


Figure 9: Testing using GPU

Fig. 10 shows the performance speedup of different GPUs over Intel Core 2 Quad Q9550 CPU when testing 40 images.

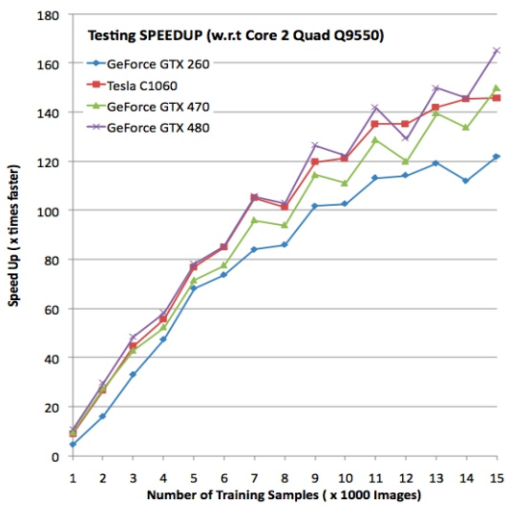


Figure 10: Testing Speedup

Fig 11 shows the execution time of the recognition pipeline on the GPU for varying database sizes.

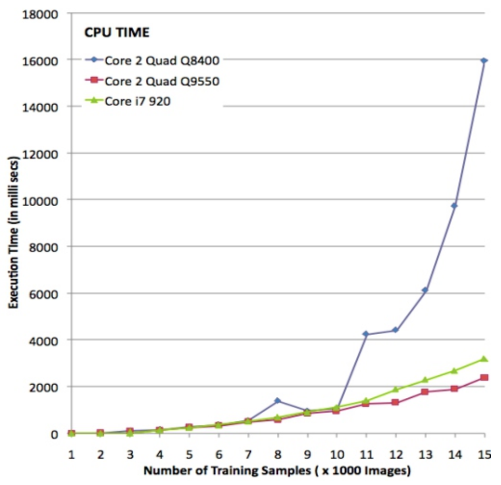


Figure 11: Recognition pipeline on CPU

Fig. 12 shows the execution time of the recognition pipeline on the GPU for varying database sizes. It is the time taken to transfer test image to device, find the match and transfer its index back to host.

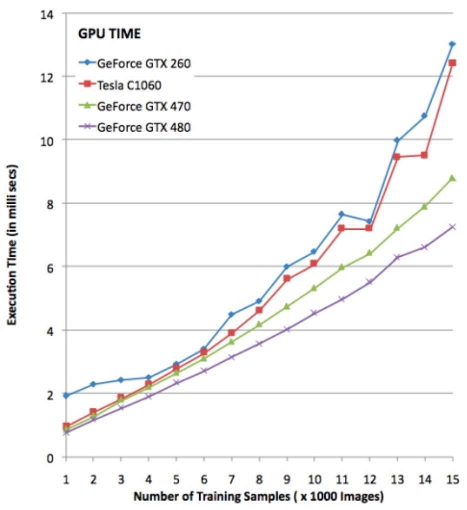


Figure 12: Recognition Pipeline on GPU

Fig. 13 shows the performance speedup of different GPUs over Intel Core 2 Quad Q9550 CPU when executing the recognition pipeline.

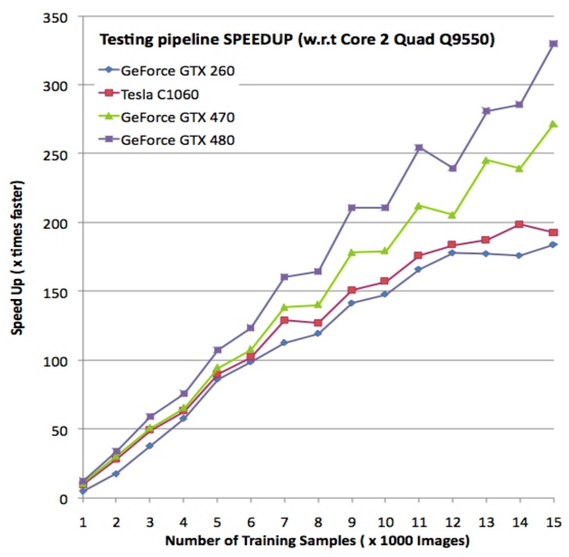


Figure 13: Recognition Pipeline Speedup

6 Conclusion

The recognition rate of a PCA based face recognition solution depends heavily on the exhaustiveness of the training samples. Higher the number of training samples, higher the recognition rate. But as the number of training samples increases, CPUs get highly strained and the training process will take several minutes to complete (refer Fig. 5). But the same process, when run on a GPU, will be completed in a manner of seconds (refer Fig. 6). The highest speedup achieved was 207x for training process, 330x for the recognition pipeline and 165x for overall testing process on the latest GeForce GTX 480 GPU, for a database size of 15,000 images.

The execution time of the recognition pipeline on the GPU is in the order of a few milli seconds even for very large databases and and this allows the GPU based testing to be integrated with real time video and used for other applications involving large volumes of test images. Our primary purpose in writing this paper is to make clear, the high performance boosts that can be obtained by developing GPU based face recognition solutions.

7 Future Works

Our future plans on this field include the parallelization of other face recognition algorithms like LDA (Linear Discriminant Analysis) and to replace the euclidean distance based matching process with a neural network based one. We feel that algorithms with a high degree of parallelism in them, like neural networks, will benefit immensely, if implemented on the GPU. We are also working on integrating the GPU recognition pipeline with real time video.

BORN TO ENGINEER

www.quest-global.com