FAST PCA-BASED FACE RECOGNITION ON GPUS

Youngsang Woo, Cheongyong Yi, Youngmin Yi

{ys.woo, opwen10, ymyi}@uos.ac.kr
School of Electrical and Computer Engineering, University of Seoul, Korea

ABSTRACT

Face recognition is very important in many applications including surveillance, biometrics, and other domains. Fast face recognition is required if she wants to train or test more images or to increase the resolution of an input image for better accuracy in the recognition. Meanwhile, Graphics Processing Units (GPUs) have become widely available, offering the opportunity for real-time face recognition even for larger set of images with a high resolution. In this paper, we explore the design space of parallelizing a PCA (Principal Component Analysis) based face recognition algorithm and propose a fast face recognizer on GPUs by exploiting the fine-grained data-parallelism found in the face recognition algorithm. We successfully accelerated the major three tasks by 120-folds, 70-folds, and 110-folds, compared to a sequential C implementation. For the end-to-end comparison, our CUDA face recognizer achieved a 30-fold speedup.

Index Terms— Face recognition, PCA, CUDA, GPU

1. INTRODUCTION

The face is one of the most important objects that people deal with in a daily life. Making a computer to recognize faces and learn new faces has been a very interesting topic and attracted many researchers. One of the most successful approaches that have been used for face recognition is based on Principal Component Analysis (PCA) [1]. This technique, usually referred to as eigenface approach, has been proposed by Turk and Pentland [2]. It enables efficient face recognition through capturing only a set of characteristics of training face images, and through comparing only this information with that of a given test image.

However, training face images is still time-consuming for reasonable amount of images with typical pixel size. For example, it takes almost 5 minutes to train only 600 images with 90K pixels, on a latest computer machine (e.g., Intel Core i7 with 3.4GHz with 4GB of memory). Testing face image is less complex and can be done in a shorter time compared to training. However, testing time increases as the number of images to compare in the DB increases. Therefore, accelerating the training as well as the testing of face recognition is the primary concern of this paper.

On the other hand, NVIDIA has recently introduced CUDA (Compute Unified Device Architecture) [5] parallel programming framework so that Graphics Processing Units (GPUs) [6] can be used as a general purpose computing platform. GPU aims at increasing throughput rather than decreasing latency of individual computation. With that purpose, it has massive number of processing elements that can execute hundreds of threads in parallel. It gained much popularity as it successfully accelerated a wide range of applications in different domains.

Face recognition has tremendous data parallelism. In general, accuracy of recognition will increase as the number of images increases and the number of pixels in a facial image increases. Also, the larger the number of principal components, which is the dimension of the eigenvectors, the higher the accuracy of the recognition will be. All these contribute to huge data-parallelism in face recognition. In this paper, we present a fast face recognizer on GPUs that efficiently maps the fine-grained data-parallelism found in the face recognition algorithm onto the massive number of processors in GPUs. We explore the different parallelization strategies for the PCA based face recognition on GPUs. We successfully accelerated the major three tasks by 120-folds, 70-folds, and 110-folds, compared to a sequential C implementation. For the end-to-end comparison, our CUDA face recognizer achieved a 30-fold speedup.

2. PCA BASED FACE RECOGNITION

Eigenface approach is based on PCA. PCA is a technique used to find out the significant information or principal components in the data set. With the principal components, one can reduce the dimensionality of the vector space where the data are originally represented. Then, we can identify or represent the data with only this set of significant information (i.e, principal components). Such significant features, in the context of face recognition, are called eigenfaces as it is obtained from eigenvectors.

Figure 1 shows the computation flow of Eigenface approach for training face images. The blue boxes depict the main tasks and white boxes their inputs and outputs. The

first task computes the covariance of training images to reflect the redundancy. It computes N by N covariance matrix C, where N is the number of training face images in the face DB. If we denote the kth training image as Γ_k , the average face as Ψ , the pixel size as P, then, the element in ith row and jth column in a C, C_{ij} , is described as follows.

$$C_{ij} = \frac{1}{P}(\Gamma_i - \Psi) \cdot (\Gamma_j - \Psi)^T$$

The second task finds out the principal components from the covariance matrix C. Since principal components with larger variances represent the interesting structure while those with lower variances represent noise, C is diagonalized such that the variances are maximized and the redundancies are minimized. This corresponds to computing the eigenvectors of C, and these eigenvectors are the principal components of the training images [1]. We use Jacobi method for this computation.

The third task computes eigenfaces u_m as below using the eigenvectors v_m obtained in the second task.

$$u_m = \sum_{n=1}^{N} v_{m,n}(\Gamma_n - \Psi)$$

As can be seen in the equation, each eigenface is computed as a linear combination of the training images.

Finally, the fourth task projects each training image to the subspace defined by eigenvectors, and obtains weights for each eigenface. It performs dot product of eigenvectors and the normalized image as shown in the following equation.

$$w_{n,m} = u_m^T \cdot (\Gamma_n - \Psi)$$

An image is reconstructed using this weight of each eigenface. When we test a face image, it is compared only with these weights, not with pixel by pixel. Testing face images consists of this task and a task that calculates the distance in the subspace.

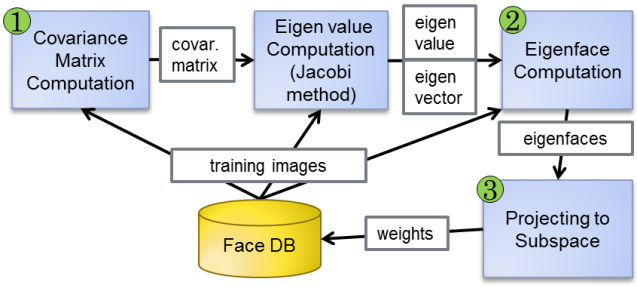


Figure 1. The computation flow of training face images in the PCA based face recognition. (The three tasks denoted with numbers are the tasks that we parallelized in this paper)

3. GPU AND CUDA

Graphics Processing Units (GPUs) were designed originally for processing graphics applications, where millions of operations can be executed in parallel. In order to increase the efficiency by exploiting this parallelism, typical GPUs have hundreds of processing cores in a hierarchical organization. For example, the NVIDIA GTX580 GPU has 512 processing cores called *streaming processors* (SP). The processing cores are organized hierarchically: A group of SPs makes up a *streaming multiprocessor* (SM). A number of SMs form a single graphics device. The GTX480, for example, contains 16 SMs, with 32 SPs in each SM, resulting in the total of 512 SPs.

Recently, NVIDIA introduced the Compute Unified Device Architecture (CUDA). It allows programmers to utilize GPUs to accelerate applications in domains other than graphics. CUDA is essentially the C programming language with extensions for thread execution and GPU-specific memory access and control. A CUDA thread is executed on an SP and a group of threads (called a *thread block*) is executed on an SM. CUDA enables the acceleration of a wide range of applications in various domains by executing a number of threads and thread blocks in parallel. This data-parallel function executed on GPUs is called a *kernel*. In order to utilize the massive parallelism in the GPU better, it is typical to have hundreds of threads in a thread block, and have hundreds or thousands of thread blocks launched for a single kernel.

Threads can synchronize with one another by using atomic operations APIs in CUDA, or by using shared memory and synchronization APIs such as *__syncthreads()* [7].

4. PARALLELIZING EIGENFACE

In this section, we explain how the concurrency explained in section 2 can be mapped to threads and thread blocks in CUDA. Recall that we denote the number of pixel size in an image as P, the number of input image as N, the number of eigenvectors as M. M can be at most N-1.

Table 1. The execution time of the C implementation [4] (FERET DB [8], *P*= 300x300, *N*= 560, *M*=559)

	Time(ms)	*Portion(%)*
Covar. matrix computation	55,728	20
Eigenface computation	108,383	38
Projecting to subspace	112,223	40
Jacobi method	6,086	2
Total	282,420	100

Table 1 shows the training execution time profiling of our reference implementation in C when P is 300x300 and N is 560. Since all the three tasks except eigenvalue computation using Jacobi method take substantial amount of portions, we decided to implement these three tasks in CUDA.

Note that, in testing, Projecting to subspace dominates the overall execution time and the task which calculates the distance in the subspace takes negligible time.

Now, we will describe the design choices of each task when it is implemented in CUDA.

4.1. Covariance Matrix Computation Task

Since this task computes covariance between all images, the complexity is in $O(N^2P)$. If we map each image to a thread, more images are executed in parallel but the covariance computation of an image itself is done sequentially. Thus, we map each image to a thread block, and the pixels in the image to the threads in the block. This enables parallel execution of the covariance of an image. In this mapping, there would be N^2 thread blocks (NxN two dimensional blocks). Note that the maximum number of threads in a thread block is either 512 or 1024 depending on the type of GPUs. If the pixel size is larger than this limit, each thread has to carry on the computations for multiple pixels sequentially.

To obtain the covariance of the image that the thread block is mapped to, the threads in the block must perform sum reduction. After each thread completes the computation for one or more pixels that it is assigned, they must synchronize for the reduction. Such synchronization can be done through atomic operation or using by parallel reduction on the shared memory

4.2. Eigenface Computation Task

The complexity of this task is in $O(NMP)$. We can execute both M eigenvectors and N training images in parallel with P pixels. In this mapping, there would be NxM thread blocks and each block is assigned P pixels. Alternatively, if we parallelize only N images, along with P pixels, the kernel will have N thread blocks and it will be invoked M times sequentially from the CPU, as shown in Figure 2(a).

However, these mappings are inefficient since the sum reduction is required for each of the pixel in the same location in N images. Since this reduction requires synchronization among threads across *different* thread blocks, we must use atomic operations on global memory, which is very costly. Alternatively, we can store the intermediate values in global memory first, and perform synchronization in a separate kernel. Still, this incurs significant overhead.

Thus, we parallelize M eigenvectors, along with P pixels, and invoke the kernel N times sequentially from the CPU, as shown in Figure 2(b). If we parallelize over M eigenvectors, the same image, $image_n$, is mapped to a thread block, each with the mth eigenvector. To obtain the mth eigenface, sum reduction is required. In this mapping, we only need to accumulate the result from the current invocation to the partial results already obtained from the previous invocations of the kernel. Another advantage of this mapping is that all thread blocks read the same image, which would results in better locality for the cache.

4.3. Projecting to Subspace Task

The complexity of this task is also in $O(NMP)$. This can be parallelized both in N and M, along with P. Unlike Eigenface computation task, this mapping needs to perform sum reduction only for the same image, not for different images. It generates the single reduced value, w_{nm}, for each of NM thread blocks.

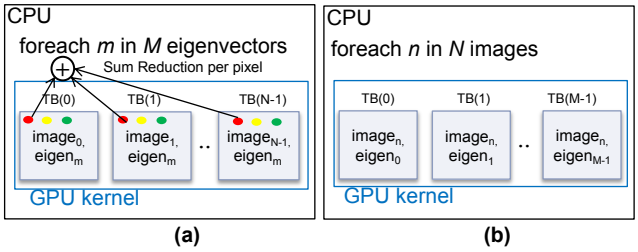


Figure 2. (a) *NP* mapping where there are *N* thread blocks and (b) *MP* mapping where there are *M* thread blocks

4.4. Increasing Block Level Parallelism

As we produce more thread blocks, it increases the potential parallelism, providing enough thread blocks that an SM will execute simultaneously. We can change the number of thread blocks by changing the number of pixels, T, which is assigned to a thread: if P is larger than the number of threads in a block, a thread must compute for multiple pixels sequentially. If we assign smaller T to a thread, we will have more than one thread blocks for each image, resulting in a larger number of thread blocks in the system. This could in turn increase the throughput and reduce the overall execution time, if it suffered from insufficient parallelism.

However, as the number of thread blocks increases, it may worsen the cache locality: too many concurrent thread blocks could pollute the cache and, when a thread block is resumed, it would find that its contents had been replaced in the cache. Moreover, additional reduction must be performed across multiple thread blocks (for the same image), which incurs overhead.

When smaller T is set, the gain that could be obtained with the sufficient parallelism with the increased number of thread blocks, and the loss that could come from the increased cache misses and the additional reduction, will be different for each kernel.

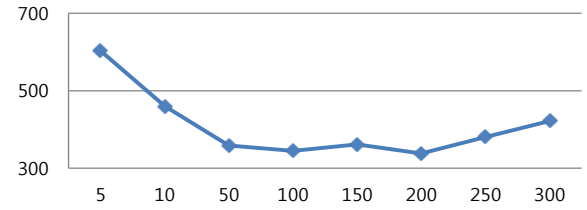
5. EXPREIMENTAL RESULTS

We used the C/OpenCV [3] implementation obtained from [4] as our reference baseline and executed this single threaded C implementation on Core i7 3.40 GHz machine. We implemented the three tasks in CUDA as discussed in the previous section and executed on a NVIDIA GTX580. We used images from FERET DB [8], whose pixel size is 300x300.

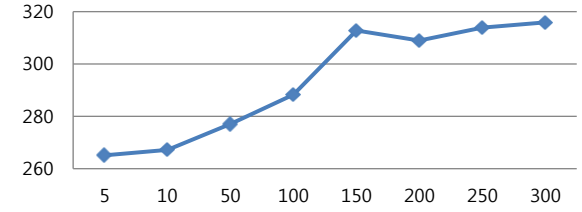
5.1. Effects of Thread Block Number

As discussed in section 4.4, varying the number of pixels, *T*, that are assigned to a thread changes the total number of thread blocks executed on a GPU. Figure 3 shows the execution time of three tasks when *T* varies. As shown in Figure 3(a), Covariance Matrix Computation (CMC) task takes shortest time when *T* is 200. Since CMC task already has sufficient parallelism of at least N^2 thread blocks, having small T results in too many concurrent thread blocks, which also increases synchronization overhead and worsen the cache locality. Thus, the execution time decreases as *T* increases. When *T* is larger than 200, it starts to increase as the parallelism becomes insufficient.

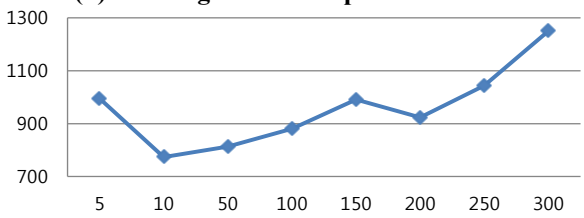
As discussed in section 4, Eigenface Computation (EC) task has only *M* thread blocks. Thus, the execution time of EC decreases as *T* decreases, as shown in Figure 3(b). Projecting to Subspace (PS) task has at least *NM* thread blocks and shows a similar trend in CMC task. In this case, the best execution time is obtained when *T* is 10.



(a) Covar. Matrix Computation task



(b) Eigenface Computation task



(c) Projection to Subspace task

Figure 3. The execution time (msec) of each CUDA task when the number of pixels per thread (*T*) changes

5.2. Execution Time Comparison

We selected the best *T* value for each task in CUDA, and compared the execution time with the C implementation. The execution time of the CUDA task includes the memory transfer time between CPU and GPU memory, and memory allocation time as well as the kernel execution time. Figure 4 shows the speedup of the three tasks as the number of *N* increases. CMC task is accelerated best among the three tasks and about 120-fold speedup is achieved when *N* is 560. EC task achieves about 70-fold speedup, and PS task achieves about 110-fold speedup. These correspond to 0.5 second from 56 seconds, 1.6 second from 108 seconds, and 1 second from 112 seconds. The end-to-end comparison including Jacobi method results in over 30-fold speedup.

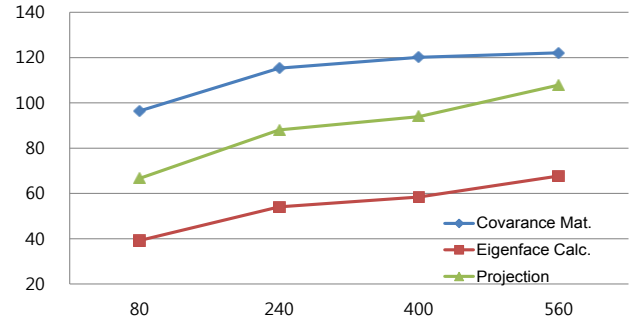


Figure 4. Speedup of three tasks on GPU when *M=N-1*

6. RELATED WORK

Although there is plenty of work for face detection on GPUs, there is not much work for the acceleration of face recognition on GPUs.

Local Binary Pattern (LBP) based face recognizer on GPUs using CUDA is presented in [9]. They parallelized three tasks: the one that computes LBP value for an input image, the one that computes local histogram from LBP values, and the one that computes k-Nearest Neighboring. It reports a 29-fold speedup was achieved.

Acceleration of PCA based face recognition is presented in [10]. It only parallelized Projecting to subspace task. For testing of face images, it additionally parallelized the task that calculates the distance, in the subspace, between the test image and the one in the DB. It reports a 200-fold speedup on GTX480. However, the pixel size was very small as 16x16, and the whole image is mapped to a thread. This approach is hard to achieve efficient acceleration when the pixel size is larger. By contrast, in our work, we parallelized all three tasks. Moreover, we investigated the impact of different mappings.

7. CONCLUSION

In this paper, we have discussed different possible mappings of PCA based face recognition onto GPUs. We have successfully accelerated the overall PCA based face recognition by 30 folds when the training image was 560. With larger number of images, we expect further higher speedup. As we have accelerated the OpenCV APIs for general PCA, we envision the proposed CUDA implementations can also be used for other PCA based applications.

8. REFERENCES

[1] L. I. Smith, "A Tutorial on Principal Components Analysis", http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf, 2009

[2] M. Turk, A. Pentland, "Eigenfaces for Recognition", *Journal of Cognitive Neuroscience,* pp.71-86, 1991

[3] OpenCV Library, http://opencv.org

[4] R Hewitt, "Face Recognition with Eigenface", *SERVO Magazine*, April 2007

[5] J. Nickolls et al., "Scalable Parallel Programming with CUDA," ACM Queue, vol. 6, no. 2, Mar./Apr. 2008, pp. 40-53

[6] C. M. Wittenbrink, E. Kilgariff, A. Prabhu, "Fermi GF100 GPU Architecture", IEEE Micro, vol. 31, no. 2, Mar. 2011, pp. 50-59

[7] "CUDA C Programing Guide version 4.1", pp. 98-100

[8] P. Phillips, H. Wechsler, J. Huang, P. Rauss, "The FERET Database and Evaluation Procedure for Face-recognition Algorithms", *Image Vision Computing*, no. 16, vol. 5, pp. 295-306, 1998 ,

[9] S. C. Tek, M. Gokmen, "CUDA Accelerated Face Recognition Using Local Binary Patterns", Technical Report, Istanbul Technical University, Turkey, 2012

[10] N. Ashraf, Sibi. A "CUDA-Accelerated Face Recognition", poster presentation, *GPU Technology Conference*, 2010