A CUDA based Parallel Implementation of Speaker Verification System

Mohammad Azhari

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
August 2011
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Elvan Yılmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Muhammed Salamah
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Asst. Prof. Dr. Cem Ergün
Supervisor

Examining Committee

1. Asst. Prof. Dr. Adnan Acan
2. Asst. Prof. Dr. Ahmet Ünveren
3. Asst. Prof. Dr. Cem Ergün

ABSTRACT

Speaker Verification (SV) is a type of speaker recognition that validates the identity of a claimed person by his/her voice. Training the models from large speech data requires a significant amount of memory and computational load. In this thesis we present a parallel implementation of speaker verification system based on Gaussian Mixture Modeling – Universal Background Modeling (GMM – UBM) designed for many-core architecture of NVIDIA's Graphics Processing Units (GPU) using CUDA single instruction multiple threads (SIMT) model. CUDA implementation of these algorithms is designed in such a way that the speed of computation of the algorithm increases with number of GPU cores. In our experiments we have achieved 30 times speedup for k-means clustering and 65 times speedup for Expectation Maximization (EM) for an input of about 350K frames of 16 dimensions and 1024-2048 mixtures on GeForce GTX 570 (NVIDIA Fermi Series) with 480 cores when compared to a single threaded implementation on the traditional CPU.

Keywords: Speaker Verification, Gaussian Mixture Models, Parallel Computing, Compute Unified Device Architecture, General-purpose computing on graphics processing units

ÖZ

Konuşmacı tanıma işlemlerinden olan konuşmacı doğrulama sisteminde iddia edilen konuşmacının sesinin doğruluğu onaylanır. Konuşmacıların modelleri eğitilirken önemli miktarda bellek ve işlem yükü gerektirir. Bu tezde biz konuşmacı dogrulama sistemini Gauss Karışım Modeli- Evrensel Arkaplan Modelleme tekniği (UBM-GMM) kullanılarak eğittik. Eğitim aşmasını hızlandırmak için seçilen paralel uygulama modeli CUDA teknolojili, tek komutlu çok izgeli (SIMT) işlemci sistemini destekleyen ve çoklu çekirdek desteği olan NVIDIA Grafik İşleme Üniteleri (GPU) kullanılarak gerçekleştirilmiştir. CUDA kullanılarak tasarlanan uygulamarın hesaplama hızı, GPU daki çekirdek sayısına bağlı olarak artmaktadır. Deneysel sonuçlara göre, 350K penceresi ve 16 boyutu olan öznitelik vektörleri k-ortalamala kümeleme algoritmasının paralelleştirilmesi ile elde edilen hızlanma faktörü 65 kat, aynı sayıda öznitelik vektörlerinin 2048 karışımlı GMM datasının Enbüyütme Beklentisi Algoritmasına sokulmasıyla elde edilen hızlanma faktörü 65 kat olarak gerçekleşmiştir.

Anahtar Kelimeler: Konuşmacı Doğrulama, Gauss Karışım Modelleri, Paralel Hesaplama, Hesap Birleşik Aygıt Mimarisi, Grafik İşleme Ünitelerinde Genel-Amaçlı Hesapla

ACKNOWLEDGMENT

I would like to thank Asst. Prof. Dr. Cem Ergün for his continuous support and guidance in the preparation of this study. Without his invaluable supervision and guidance, further development was not possible.

I would also like to thank all the members of staff at Department of Computer Engineering and Computer Center for the facilities they provided that helped me achieve the results in this dissertation.

I must also express my gratitude to Nick Kopp for his continued support over the used technologies and who responded to my questions and queries so promptly.

Above all I would like to thank my family who sponsored me and devoted their love to help me reach my prosperities. I love them all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Speaker Verification (SV) is a type of speaker recognition that validates the identity of a claimed person by his/her voice. The main aim is not to validate the sentence spoken itself but to validate the speaker voice characteristic. There are various methods used to identify a person using SV. These methods involve techniques such as frequency estimation, hidden Markov models, Gaussian mixture models, pattern matching algorithms and more [1]. In this thesis, our research is narrowed down to Gaussian Mixture Modeling (GMM).

Optimization of the SV systems can help to decrease development time. It should be kept in mind that the mixture based modeling techniques can be easily optimized using parallel implementation. The existence of a massively parallel computing technology such as Compute Unified Device Architecture (CUDA) has changed the face of computing in the past years. Currently having some of the most efficient ratings on Green500 list [2] indicates that today’s GPUs are more suited for scientific computing with less power consumption.

In this thesis we perform a research on parallelizing SV using CUDA technology and we measure its benefits compared to the traditional single-thread or parallel CPU implementations.

The rest of the thesis is organized in the following way. In chapter 2 we will explain the speaker verification system that was chosen as our target problem. In chapter 3 we will describe the CUDA technology. In chapter 4 we review some of the related works. Following that, we will continue with the actual implementation of the system in chapter 5. In chapter 6 the results of the experimentations will be presented with analysis. Finally in chapter 7 we will conclude the thesis.

Chapter 2

SPEAKER VERIFICATION

2 Gaussian Mixture Model

GMM is a parametric probability density modeling system typically used to make decision with respect to mixture models. This method is usually used as a model for probability distribution of feature of speaker verification system. The parameters of GMM are trained and refined usually using Maximum Likelihood (ML). In this thesis Expectation Maximization (EM) is used to estimate the parameters [3].

2.1 Universal Background Modeling

Universal Background Modeling (UBM) is a method used to represent speaker-independent feature characteristics of all of our speakers. In order to reflect the correct type and composition of our speech we train the system for male and female speakers separately. For example for verification of telephone speech of male speakers, we train our system using the pool of male speakers. UBM is trained for likelihood ratio test. Given an observation, *O*, and a hypothesized person, *P*, the task of verification is to determine if *O* was from *P*. This verification task can be restated as a basic hypothesis test between [4]:

$$H_0 : O \text{ is from person } P$$

$$H_1 : O \text{ is not from person } P$$

The optimum test to the ratio can be written as:

$$\frac{p(O|H_0)}{p(O|H_1)} \quad \begin{cases} \geq \theta & accept\ H_0 \\ < \theta & reject\ H_0 \end{cases} \tag{2-1}$$

Where $p(O|H_i), i = 0$ is the probability density function or also called “likelihood” of the hypothesis H_i and θ is the decision threshold. However the optimality is a rare case since the maximum likelihood functions are not usually known [4]. The first step of SV is to extract parameters of speakers in form of vector of features. In our case Mel-Frequency Cepstrum Coefficients Filter (MFCC) is used to extract the features. The result of this step is then used to calculate H_0 *and* H_1. Given a set of N background speaker models $\{\lambda_1, \dots, \lambda_N\}$ the alternative hypothesis model is represented by:

$$p(X|\lambda_{\bar{C}}) = \mathcal{F}(p(X|\lambda_1), \dots, p(X|\lambda_N)) \tag{2-2}$$

Where $\mathcal{F}()$ is some average or maximum function of likelihood values. Another method to approximate the imposter model is the use of UBM. From a large number of speakers a pooled training is used to generate a single large mixture (2048) model. The likelihood of background speakers used as a reference set is as follows:

$$p(X|\lambda_{\bar{C}}) = \ p(X|\lambda_{UBM}) \tag{2-3}$$

Where $p(X|\lambda_{\bar{C}})$ is the claimant speaker GMM likelihood for a sequence of the feature vectors, X and the UBM model parameters are $\lambda_{UBM} = \{m_i, \mu_i, \Sigma_i\}_{i=1}^{M}$. The UBM model parameters are trained using EM algorithm. As a benefit, for use in a task for all hypothesized speakers, a single speaker-independent model can be trained only once. In this thesis UBM is always used as the reference likelihood.

2.2 Mel-Frequency Cepstrum Coefficients

MFCC are a collection of coefficients that represent the audio on a non-linear mel-scale of frequencies. MFCC are commonly used as reduced-dimensional features of an audio which represents the human vocal characteristics. MFCC are also used in music information retrieval applications [5]. One thing to point is that MFCCs are not additive-noise invariant and therefore their values are usually normalized to be less affected by low energy components [6]. The block diagram of MFCC procedure can be seen in the following figure:

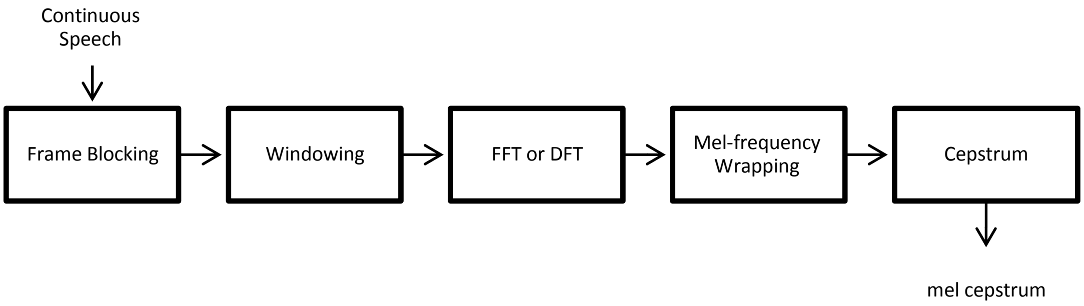

Figure 2-1. Block diagram of the MFCC processor [7]

K-means Clustering

K-means Clustering is a method that helps to accelerate convergence [8]. We have implemented this algorithm solely for MFCC in UBM step. In this method with a set of observations (x_1, x_2, …, x_n), each of which a vector of d dimensions are clustered into k sets ($k \leq n$) S = {S_1, S_2, …, S_k} such that the sum of square in each cluster is minimized:

$$arg_s \min \sum_{i=1}^{k} \sum_{x_j \in s_i} \left\| x_j - \mu_i \right\|^2 \tag{2-4}$$

where μ_i is the mean of point in S_i.

2.3 Parameter Estimation: Expectation Maximization Algorithm

Expectation-maximization (EM) algorithm is a method that iteratively estimates the likelihood [9]. It is similar to the k-means clustering algorithm for Gaussian mixtures since they both look for the center of clusters and refinement is done iteratively.

For a given set of T observation vectors $X = \{x_1, x_2, \dots, x_T\}$ and assuming x_t vectors are independent and identically distributed (iid), the best fitting model λ is the one that maximizes,

$$p(X|\lambda) = \prod_{t=1}^{T} p(\overrightarrow{x_t}|\lambda). \tag{2-5}$$

This is a non-linear problem; therefore λ cannot be directly calculated. The EM algorithm consists of the following steps:

Algorithm 2-1. Expectation Maximization Procedure [10]

1. Choose an initial model λ .
2. Find a new model λ' so that $p(X \mid \lambda') > p(X \mid \lambda)$.
3. Repeat step 2 until the difference
4. $p(X \mid \lambda') - p(X \mid \lambda)$ has reached a convergence threshold or you have reached the maximum number of iterations.

In each EM iteration, the ML estimates for the means, variances and weights (a priori mixture probability) for a particular speaker model are computed as follows [10]:

Mixture Weight:

$$\overline{m_\imath} = \frac{1}{T}\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda) \tag{2-6}$$

Means:

$$\overline{\overrightarrow{\mu_\imath}} = \frac{\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda)\,\overrightarrow{x_t}}{\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda)} \tag{2-7}$$

Variances:

$$\overline{\sigma_\imath^2} = \frac{\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda)x_t^2}{\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda)} - \overline{\mu_\imath^2} \tag{2-8}$$

where$\overline{m_\imath}$, $\overline{\overrightarrow{\mu_\imath}}$ and $\overline{\sigma_\imath^2}$ are updated values of m_i, $\overrightarrow{\mu_\imath}$ and $\sigma_\imath^2$ respectively, and $\sigma_\imath^2$, μ_i and x_t refer to arbitrary elements of the vectors in $\overline{\sigma_\imath^2}$, $\overrightarrow{\mu_\imath}$ and$\overrightarrow{x_t}$ respectively and, $\overrightarrow{x_t^2}$ is the shorthand for dialog $(\overrightarrow{x_t},\overrightarrow{\acute{x}_t})$. The posterior probability for the i_{th} acoustic class is given by,

$$P(i|\overline{x}_t,\lambda) = \frac{m_i b_i(\overrightarrow{x_t})}{\sum_{k=1}^{M} m_k b_k(\overrightarrow{x_t})} \tag{2-9}$$

Where m_i and m_k are the mixture weight of the i_{th} and the k_{th} mixture component and $b_i(\overrightarrow{x_t})$ and $b_k(\overrightarrow{x_t})$ are the component densities of i_{th} and k_{th} mixture component.

2.4 Speaker Model Adaptation

In adaptation stage, instead of constructing the model from the training data, we adapt the trained UBM parameters using Bayesian adaptation to claimant speaker and the Maximum A Posteriori (MAP) estimation (*see* Figure 2-2). The MAP is done in two steps. The first step is to calculate the estimates of the statistics of training data for every mixture in the prior model. In the second step these estimates are combined with UBM mixture parameters to create adapted claimant models [3].

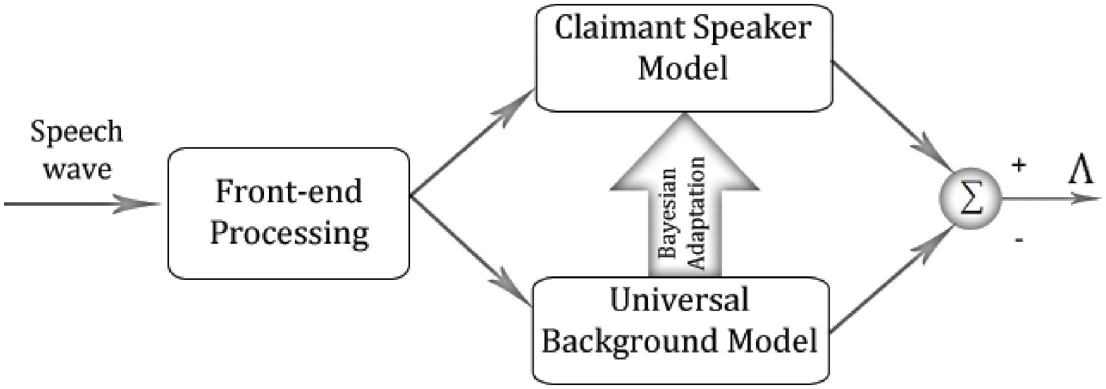

Figure 2-2. GMM-UBM likelihood ratio detector. [10]

In adaptation process, at first basic statistics are estimated to compute the desired parameters for the next part of adaptation. The training feature from a client is used to perform this process. The probability of the mixture component *i* in the UBM is defined as,

$$\Pr(i|\overrightarrow{x_t}) = \frac{m_i b_i(\overrightarrow{x_t})}{\sum_{k=1}^{M} m_k b_k(\overrightarrow{x_t})} \tag{2-10}$$

Where m_i and m_k are the mixture weight of the i_{th} and the k_{th} mixture component and $b_i(\overrightarrow{x_t})$ and $b_k(\overrightarrow{x_t})$ are the component densities of i_{th} and k_{th} mixture component.

Using $\Pr(i|\overrightarrow{x_t})$ and $\overrightarrow{x_t}$, the statistics for weight, mean and variance can be found follows:

$$n_i = \sum_{t=1}^{T} \Pr(i|\overrightarrow{x_t}) \tag{2-11}$$

$$E_i(x) = \frac{1}{n_i}\sum_{t=1}^{T} \Pr(i|\overrightarrow{x_t})\overrightarrow{x_t} \tag{2-12}$$

$$E_i(x^2) = \frac{1}{n_i}\sum_{t=1}^{T} \Pr(i|\overrightarrow{x_t})\overrightarrow{x_t}^2 \tag{2-13}$$

Then, the adapted speaker model parameters $\lambda_c = \{\widehat{m}_i, \widehat{\vec{\mu}}_i, \hat{\Sigma}_i\}_{i=1}^{M}$ are computed as,

$$\widehat{m}_i = [\frac{\alpha_i n_i}{T} + (1-\alpha_i)m_i]\gamma \tag{2-14}$$

$$\hat{\mu}_i = \alpha_i E_i(x) + (1-\alpha_i)\mu_i \tag{2-15}$$

$$\hat{\sigma}_i{}^2 = \alpha_i E_i(x^2) + (1-\alpha_i)(\sigma_i^2 + \mu_i^2) - \hat{\mu}_i^2 \tag{2-16}$$

Where γ is a scale factor to make sure that the mixture weights sum to unity. $\alpha_i = \frac{n_i}{n_i + r}$ is a data dependent adaptation coefficient which controls the balance between the old and new estimates where *r* is the fixed relevance factor (as suggested by [11] *r* is typically between 8 and 20). An important factor is that the adaptation of the UBM mixture components is highly dependent on the fact that there is sufficient correspondence with the client training data.

Chapter 3

COMPUTE UNIFIED DEVICE ARCHITECTURE

3 A short introduction to GPU Computing and CUDA

Since the first time Graphics Processing Units (GPU) was invented back in 1999 by NVIDIA [12], the GPU technology has evolved and has been through major changes. Currently, GPUs are available with a much higher arithmetic power and greatly higher memory bandwidth than CPUs. Since 2003, GPUs were available for non-graphics applications in the form of high level shading languages in DirectX, OpenGL and others. Because of that, several algorithms were ported to GPU and problems such as protein folding, stock options pricing, SQL queries, and MRI reconstruction gained considerable speedups [13]. These efforts were called GPGPU (General-purpose computing on graphics processing units).

There were limitations with those architectures. First of all, it was very difficult for general programmers to adopt their programs into these graphics APIs. Also the complexity of expressing those problems in form of vertices and textures were hard to manage. Additionally, random reads and writes on the memory were not possible because of architectural limitations. The other limitation was that double precision floating points were not supported, which concluded to no use for some scientific computations until recently. Both ATI and NVIDIA companies have introduced GPU architectures that can be benefited by programmers.

3.1 G80 and GT200 Architectures

NVIDIA having a head-start on the new GPU technology introduced a pair of G80 unified graphics and compute architecture and CUDA software/hardware architecture that allows development on different high level languages.

G80 was the first architecture to support C language and also the first to introduce Single Instruction Multiple Threads (SIMT) execution model. In SIMT multiple independent threads execute concurrently using a single instruction. Shared memory and barrier synchronization were also features of G80.

On 2008, NVIDIA introduced GT200 architecture which increased the number of stream processor from 128 to 240 and the size of register file per processor were also doubled as well as adding double precision support. The GT200 supports up to 512 threads per execution block.

3.2 The Fermi Architecture

Fermi is the world's first computational GPU. This is the main target architecture in this thesis. The advantages of Fermi over previous architectures are as follows:

- **Improved double precision performance**
- **World's first GPU ECC memory**: Error Correction Codes (ECC) insures that important applications like finance and medical imaging are performing their calculations without any errors.
- **Increased shared memory**: The Fermi supports up to 48Kbytes of shared memory per block.

- **Faster context switching and atomic operations**
- **A true cache architecture**: Due to the request of users a true cache hierarchy was implemented as an alternative to shared memory when users are not able to use shared memory.
- **More threads per block:** Fermi supports up to 1024 threads per block.

3.3 Compute Unified Device Architecture

CUDA [14] is the parallel computing architecture developed by NVIDIA and is the computing engine in NVIDIA graphical processing units (GPUs). This architecture is available through different programming languages and supports other application programming interfaces, such as CUDA FORTRAN, OpenCL, and DirectCompute. CUDA helps to solve many complex computational problems in a more efficient way than on a CPU. The CUDA parallel programming model is designed to overcome the challenge of developing application software that transparently scales its parallelism while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

CUDA has several advantages [15] over traditional general purpose computation on GPUs. One of them is that the GPU code can access different addresses in memory. Another advantage is availability of shared memory which is a locally accessible memory that is shared among a group of threads. This helps to reduce the global memory accesses and as a result providing a higher bandwidth.

However, there are some limitations. One is that the data transfer between the device and the host memory is slower that within-device memory transfers. Although [16] discusses that this may not always be the case, including [17] and our implementation. We have used a small number of threads per block group to get better results. Threads are activated in groups of 32, with thousands of them running in total. Another limitation is that CUDA technology is only available for NVIDIA GPUs. It only supports round-to-nearest mode of IEEE 754 [18] standard for double precision calculations.

On CUDA architecture, the problems are divided into sub-problems and each sub-problem into finer pieces that can cooperatively run in parallel by all threads within the block. Each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 3-1.

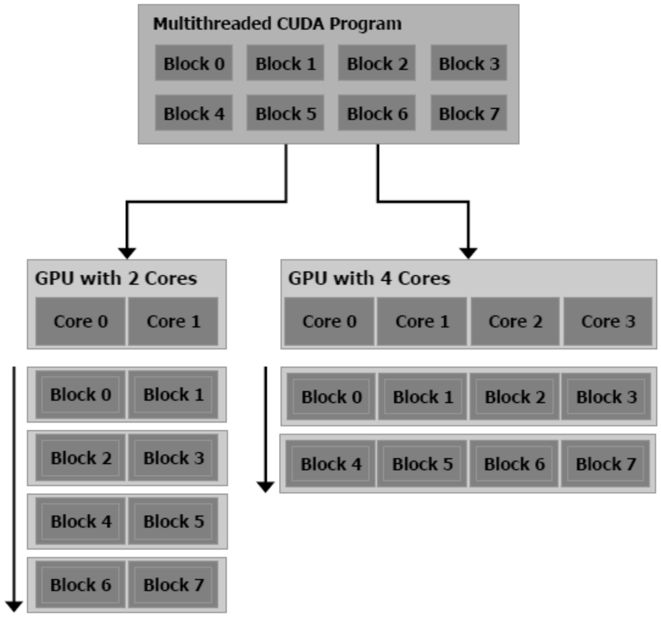

Figure 3-1. Automatic Scalability [14]

3.4 CUDA Kernels

Kernels are C functions that are extended by CUDA programming language. The kernels are defined by a declaration identifier "__global__". The number of threads that will execute such kernel function is defined using the new syntax <<< >>> called *execution configuration*. For every thread there is a unique Thread ID that can be accessed from within the thread. The threadIdx is a 3-dimensional variable which is accessible locally in each thread and it represents a logical location of the thread that runs the kernel. As an example, Algorithm 3-1 performs $f(\vec{x}) = a\vec{x} + b$ and stores the result into a variable named *result*:

Algorithm 3-1. Kernel FofX is executed on N threads

```
// Kernel definition
__global__ void FofX(float* X, int A, int B,float* result)
{
  int i= threadIdx.x; // The X dimension of Thread index.
  result[i] = A * X[i] + B;
}
// Main function
int main()
{
  ...
  // Kernel invocation with N threads
  FofX<<<1, N>>>(X, A, B, result);
}
```

Blocks of threads are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 3-2. The number of thread blocks in a grid is usually decided depending on the size of our data being processed or the number of processors in the system.

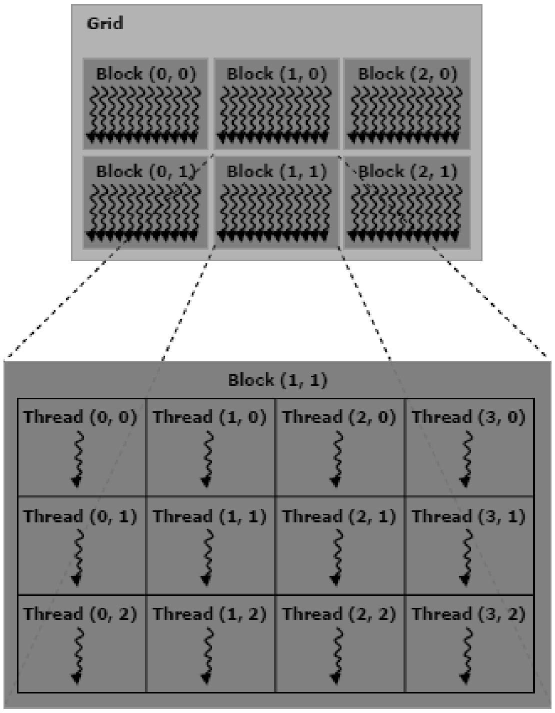

Figure 3-2. Grid of Thread Blocks [14]

Each running block is divided into Single Instruction Multiple Threads groups called warps. The size of these warps are equal. The running warps are scheduled in a timely manner (time-sliced). The thread scheduler switches between warps to maximize the use of the multiprocessors computational resources.

3.5 Memory Hierarchy

The threads in the CUDA environment can benefit from different types of memory. Each thread has local memory, a shared memory within the block and global memory which is available to all threads in all blocks. Additionally, threads can access two read-only memory types called constant memory and texture memory. The global, constant and texture memories are suited for different memory usages. The general memory hierarchy of CUDA GPUs is illustrated in Figure 3-3.

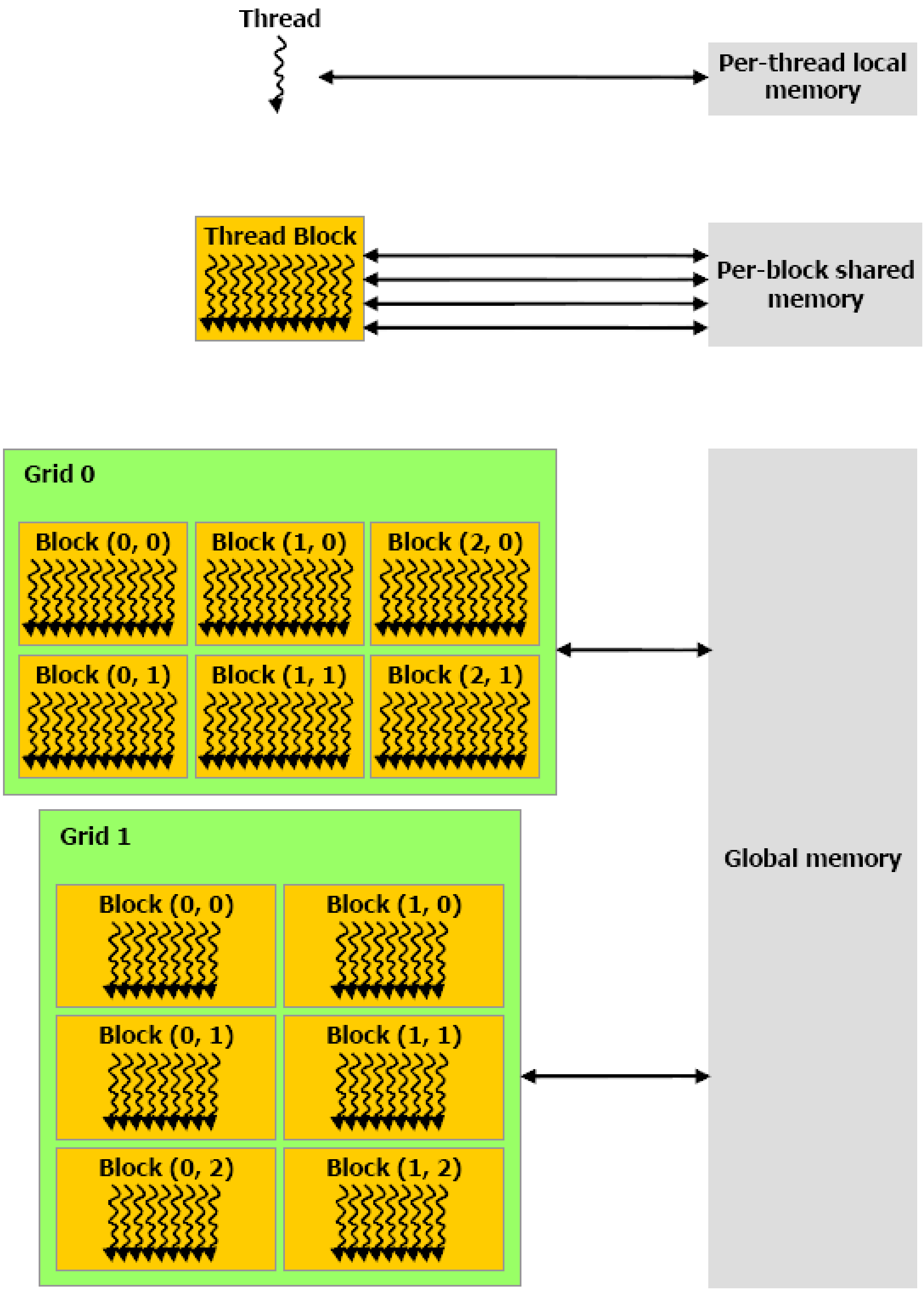

Figure 3-3. Memory Hierarchy [14]

Chapter 4

RELATED WORKS

To the best of our knowledge, there are no full implementations of Speaker Verification systems on CUDA technology. The focus is usually on a particular module of the system rather than targeting the whole process. Therefore direct comparison to this thesis is not possible. However, in this chapter we will introduce the related works in a modular form. Note that:

- Most of the publications do not mention the exact conditions the experimentations were performed.
- In most cases the performance of CUDA increases with the increase in the data size or number of processing cores.
- The architectures targeted by those publications mostly belong to G80 and GT200 architectures while we target GT200 and Fermi.
- Most experiments were performed on data with the type of single-precision floating points due to architecture limitation.

4 Related modules

4.1 MFCC Feature Extraction

In [19] a GPU based implementation of different feature extraction methods were introduced. In their implementation for MFCC, the kernel function executes a single transformation. Each thread block consists of a window of 512 samples and since there is no data dependency, the window samples can run simultaneously on GPU. This is not easily achievable for our thesis since we consider a method of discarding silence periods while in [19] there is no mention of Voice Activation Data (VAD). In their experiment they have used FFT algorithm that is provided by NVIDIA CUFFT Library [20] to maximize the performance of FFT. In this thesis the problem is implemented using DFT. The other optimization technique they have applied is the use of parallel reduction where summation of a variable was needed. There is no mention of what type of data was introduced in matters of precision. But since the operations were also performed on G80 architecture we implicitly assume the data was of type single-precision floating point. In most cases their implementation shows better performance when more cores are introduced, except for MFCC which GT200 architecture shows better performance than G80 and Fermi.

4.2 K-means Algorithm

There are several implementations of K-means over CUDA technology. We briefly review those publications.

In BAI Hong-Taoa et al [21] publication the experimentation were performed on G80 architecture which lacks the support of double-precision floating points, but the authors have produced samples of 32-bit float numbers from 100K to 1M between 0 and 1 which in the iteration they have simulated 64-bit floating point manipulations. This is possible because CUDA follows the IEEE 754 standard. In their problem they have achieved a result of 40 times faster than CPU k-means. However the increase in the size of data increased their execution time on GPU. Their GPU based K-means is illustrated in Figure 4-1:

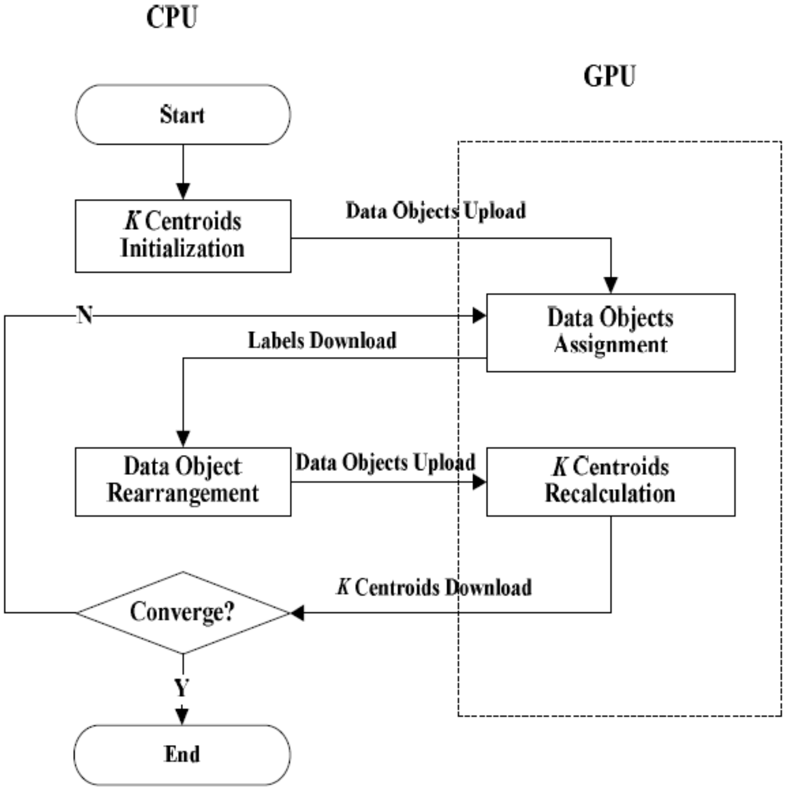

Figure 4-1. BAI Hong-tao el al, K-Means on commodity GPUs with CUDA Diagram [21]

In Mario Zechner el ta [22] experimentation is performed on G80 architecture which again implies single-precision float data manipulation. Their strategy involves cooperation of CPU and GPU similar to [21] except after rearrangement of the labels the centroid reallocation is done on CPU rather than GPU. They did not benefit from shared memory since their data size of 4000 dimensions restricts the usage of shared memory up to 16 Kbytes due to G80 specifications. They performed the process of loading and calculating the distance from a data point to a centroid in parts. The achieved a speedup of 14x. It is also mentioned for some tests there are variations in the resulting centroids.

It is observed that this is because of combined multiplication and addition operations (MADD) that results in rounding errors.

In You Li et al publication [23], the adopted method is *divide and conquer*. The data is first divided into groups and each group is reduced to get temporary centroids. Then those centroids are divided and reduced iteratively. This process is repeated until the number of groups to be divided is smaller than the multiple of the number of steaming multiprocessors (SM). In other words the process is continued until GPU has no advantage over computing the rest of calculation on CPU. The target architecture of this publication is GT200 and they claim to achieve speedup of 3 to 8 times faster than the best GPU based k-means implementations.

4.3 Expectation Maximization Algorithm

We have observed that currently there is only one publication [24] available for CUDA implementation of EM algorithm for SV. This is authored by NVIDIA Corporation [12] which is the designer of CUDA technology itself. They have targeted G80 and GT200 architectures with 128 and 240 cores respectively. The problem in had is input of 230K with 32 mixtures of 32 dimensional Gaussian model. The data type accuracy is single-precision floating points. The location of data loop in each iteration is different from the implementation in this thesis. While our outer loop is the mixtures' loop, their outer loop is the data loop. They have utilized the shared memory to maximize the performance. In their implementation they have divided the EM algorithm into multiple kernels, a method that we have also applied in this thesis. Their results show that performance increases with the number of cores and/or the data size. On GT200 architecture they achieved the speedup of 164 compared to CPU single-threaded implementation.

Chapter 5

IMPLEMENTATION OF ALGORITHMS

5 Implemented Modules

In this chapter we discuss the development of SV. In order to have a fair comparison, we implemented the most important modules of SV in single-threaded CPU, multi-threaded CPU and Parallel GPU modes. In nature, all parts of our original problem from UBM parameter maximization to adaptation and testing include common algorithms that are slightly different. Therefore, we focus on explaining those common modules and later in chapter 6 we will illustrate the results separately for each step of SV.

5.1 Mel-Frequency Cepstrum Coefficients

Due to the nature of speech data, it is very normal to have unvoiced sound segments. Here, Voice Activation Data is used to discard silence periods [10]. The other front-end operations like segmentation and windowing are also applied to each frame in order to produce the corresponding feature set. The complexity of MFCC extraction is mainly dominated by Discrete Fourier Transform (DFT) of each window segment which is $O(N_s^2)$ where N_s is the window size. In this thesis we only parallelize the DFT step of MFCC as shown in Figure 2-1. The CPU DFT pseudo-code for window size of 240 is as illustrated in Algorithm 5-1.

In order to have load balancing, we have split the execution of the loop at line 5 of Algorithm 5-1 into I number of cores. Each core is responsible for executing operations of a range of window-size frames. In our case we have used an Intel® Core™ i7-920 [25] CPU to perform the parallel tasks on 4 cores. Note that the *cosine* and *sine* values are pre-computed to decrease the redundancy of the calculations.

Algorithm 5-1. CPU DFT Parallel

```
$Cores$ = 4;
$N$ = 240; // Winowsize
$RangeSize$ = $N$ / Cores;
For core $I$ (concurrently)
    For $k$ from [$RangeSize * I$] to [$RangeSize * ( I + 1 )$ ]
        Initialize Real and Imaginary variables
        For n from 0 to 2 * $N$
            Calculate $Real$ & $Imaginary$
    Calculate $DFT$
```

In this algorithm, the time complexity is $O(N^2)$. On CPU Parallelism, the complexity is only divided by a number much smaller than N. Therefore the complexity remains the same. The CUDA DFT implementation in nature also follows the CPU model with the exception that all iterations of loop *k* are run on separate threads. In this case the complexity is reduced to $O(N)$. There are also memory transfers between host and CUDA device as shown in Algorithm 5-2.

Algorithm 5-2. CUDA DFT Parallel

```
Allocate & Transfer Sine and Cosine Tables from host to device memory.
Allocate & Transfer Current Window Frames from host to device memory.
Allocate space for DFT output on device memory
Perform DFT on CUDA with grid size of 16 and block size of 16
Copy the output from device to host memory
```

The MFCC module is used in all steps of SV. In UBM feature extraction, MFCC is applied to the pool of frames from all speakers of each gender separately while in adaptation and testing steps, MFCC features are extracted for every speaker separately.

5.2 K-means Clustering

The k-mean clustering pseudo-code in a nutshell is as shown in Algorithm 5-3. Since initial selection of mean is random, we choose mean vectors of size M mixtures by D dimensions from the feature vectors.

Algorithm 5-3. K-mean Pseudo-Code

```
Random initialization of mean vectors.
$T$= number of frames;
$M$= order of mixtures
$D$= number of dimensions;
For each frame $T$
    For each mixture $M$
        For each feature $D$
            Calculate minimum distance
Calculate average of selected feature vectors belonging to the same centroid.
```

In parallel implementation it is advised to give more work load to each processor to maximize the utilization. In the case of k-means clustering, the major loop that is parallelizable is loop T which includes inner loop M and D. The computation complexity of K-mean clustering is $O(n^{dk+1}\log n)$. Since k and d are fixed, the problem will come down to n number of entities to be clustered [10]. In both CPU and GPU approach we have parallelized the most outer loop which in this case is T given the fact that the distances can be calculated concurrently. For CUDA device implementation we will perform the calculations for each feature vector simultaneously. As a result the complexity will be $O(\acute{n}^{dk+1}\log \acute{n})$ where $\acute{n} < n$. Additionally, there are memory transfers from and to device. The order of CUDA actions for k-mean algorithm is shown in Algorithm 5-4.

Algorithm 5-4. K-means CUDA Implementation

```
1. Random initialization of mean vectors.
2. $T$= number of frames;
3. $M$= order of mixtures
4. $D$= number of dimensions;
5. Transfer initialized random mean vectors to device.
6. Transfer feature vectors to device.
7. Allocate label memory on device.
8. Allocate mean average memory on device.
9. Allocate distance memory on device.
10. Perform k-mean on CUDA device with grid size of $T$/20+1 and block size of 20.
11. Transfer label memory back to host.
12. Transfer mean-average memory back to host.
13. Calculate average of selected feature vectors belonging to the same centroid.
```

Note that all of our 2-dimensional jagged arrays are first converted into 1-dimensional arrays for device memory allocation. Also, the k-means algorithm is only applied to refine UBM parameters and is only concern of UBM feature extraction step. The other notable point is that in this thesis once we find the centroids, we don't update them; therefore running k-means only for one iteration.

5.3 Expectation Maximization

As explained in section 2.3, the first step of expectation maximization is to initialize λ parameters mixture weights (Equation (2-6)), means (Equation (2-7)) and variances (Equation (2-8)). The computation complexity for background speaker model is directly extracted from Algorithm 5-5 as $O(N_T MD)$ where N_T is the number of feature vectors each having dimension of D and each speaker having M mixtures. In order to prevent redundant calculations we first calculate the common part of those parameters which is:

$$\sum_{t=1}^{T} P(i|\overrightarrow{x_t},\lambda) \tag{5-1}$$

where i is the number of mixtures. Following the Equation (2-9), we calculate the denominator of Equation (2-9), so later it can be applied for the posterior probability. Then we continue with calculating posterior probability, sum of means and sum of variances. Then we apply Equations (2-6), (2-7) and (2-8) to calculate the updated values $\overline{m_i}$, $\overline{\vec{\mu_i}}$ and $\overline{\sigma_i^2}$. This concludes one iteration of EM Algorithm. In our implementation the iterations are repeated 50 times. The pseudo-code for EM algorithm is illustrated in Algorithm 5-5.

Algorithm 5-5. Expectation Maximization Pseudo-Code for CPU

```
Initialize mean, variance and weight values.
T = Number of feature vectors.
M = Order of mixtures
D = Number of dimensions in a feature vector
For iteration between 0 and 50 {
For k between 0 and M
        Calculate determinant of sigma[k]
For t between 0 and T
    For k between 0 and M
        For l between 0 and D
            prepare sum.
        calculate sum_p_denominator
For k between 0 and M
    For l between 0 and D
        Initialize Sum_mean and Sum_variance
    For t between 0 and T
        Initialize Sum
        For l between 0 and D
            Calculate Sum
        Calculate Sum_p
        For l between 0 and D
            Calculate Sum_mean
    Calculate Sum_variance
    For l between 0 and D
        Calculate Mean and Variance
    Calculate k
}
```

The mentionable points of parallelization in Algorithm 5-5 are where we have loops of size *T*. Since the convergence iterations are dependent and have to be executed consecutively, the most dominant parallelizable loop is *T*. In case of CPU parallelism, we apply a similar method as seen in Algorithm 5-1 to distribute the load. In a nutshell the following tasks are performed in CPU Parallel mode:

Algorithm 5-6. EM Pseudo-Code for Parallel CPU

```
Initialize mean, variance and weight values.
$T$ = Number of feature vectors. (around 350K)
$M$ = Order of mixtures (1024)
$D$ = Number of dimensions in a feature vector (16)
For iteration between 0 and 50 {
Parallel For $k$ between 0 and $M$
        Calculate determinant sigma[$k$];
Parallel For $t$ between 0 and $T$
        Calculate sum_p_denominator;
// Updating mean, weight and variance
For $k$ between 0 and $M$
    For $l$ between 0 and $D$
        sum_mean, sum_var initialization
    Parallel For $t$ between 0 and $T$
        Calculate sum_mean and sum_var
    For $l$ between 0 and $D$
        Update mean,variance
    Update weight
}
```

The *D* loops are left sequential because in the small loops the cost of creating multiple threads is more than the calculation cost in a single thread. On the other hand the *K* loop is not parallelized for the fact that the inner loop *T* is big enough to cover the cost of thread activation.

In CUDA Parallel mode we have sliced the procedures into smaller groups to handle the number of threads better. Since the CUDA code is SIMT all threads run identical code only on different parts of the memory. For example in the case of initializing sum of means and sum of variances we will only need *D* threads while for updating them we need *T* threads. This calls for task separation. It can be understood from Algorithm 5-6 that for EM algorithm there are more memory transfers/allocations are involved. It is also known that the CUDA code will run on device as kernels. They will be lunched from the host code to run on many CUDA threads. We will discuss the importance of grid size and block size in Chapter 6. The following is the EM algorithm designed to run on CUDA device:

Algorithm 5-7. EM Pseudo-Code for Parallel CUDA

```
Initialize mean, variance and weight values.
$T$ = Number of feature vectors. // around 350K for each gender
$M$ = Order of mixtures // 1024 for each gender
$D$ = Number of dimensions in a feature vector (16)
Allocate memory on device and transfer the initialized mean, variance and weight.
Transfer feature vectors to device memory.
For iteration between 0 and 50 {
        Lunch Det_Sigma () on 2 blocks of size $M/2$;
        Lunch Create_Sum_p_denom () on $T / 20 + 1$ blocks of size 20;
    For $k$ between 0 and $M$
        Lunch SumMeanVarInit () on 1 block of size $D$;
        Lunch Calc_Sum_Mean_SumVar () on $T / 32 + 1$ blocks of size 32;
    Lunch UpdateMeanVar () on 1 block of size $D$;
    Lunch UpdateWeight () on 1 block of size 1;
}
Transfer mean, weight and variance from device to host memory
```

The complexity of our EM algorithm is reduced to $O(MD)$. Note that there are slight differences for different steps of SV. In UBM, all speakers are participating into the EM together, so the size of T will be almost as large as 350,000 for each gender. While in adaptation process speakers are applied to maximization algorithm separately and size of T ranges between 4000 and 5000 for each speaker. The testing step is also similar to EM in adaptation process except the fact that two likelihoods (UBM Model and Speaker Model) are calculated for every speaker. This however, will not change the computation complexity of EM algorithm.

Chapter 6

EXPERIMENTATION AND RESULTS ANALYSIS

6 Experimentation Specifications

We have implemented our parallel algorithms using CUDA version 3.2. Our experiments were performed on a PC with GTX 285 and GTX 570 GPUs and an Intel® Core™ i7-920 CPU. The CPU has 4 cores (8 hyper-threads) running at 2.66 GHz. The main memory is 3 GB (DDR3-1600) with the peak bandwidth of 12.8 GB/sec. The specification of our GPUs can be seen in Table 6-1.

Table 6-1. Specifications of NVIDIA GPUs.

GPU	Architecture	Cores	DRAM	Processor Clock	Memory Bandwidth
GTX 285	GT200	240	1 GB	1.47 GHz	159 GB/s
GTX 570	Fermi	480	1.25 GB	1.46 GHz	152 GB/s

The SV problem we have chosen for this thesis is based on a part of [10] which was originally implemented on C++ language in Linux platform. We have converted that implementation to C# on Windows platform. The database used in this dissertation is NIST [ref]TODO. There are three modes of SV available in our application:

- *CPU Single-threaded*: This is the traditional CPU implementation which was performed solely on .NET Framework without any external libraries.

- *CPU Multi-core*: Multi-threaded implementation was programmed using .NET 4.0 Task Parallel Library which relatively corresponds to the available number of cores in the target machine.
- *CUDA*: GPU parallel implementation was done using two .NET interfaces for the original CUDA SDK called CUDA.NET [26] and CUDAfy [27]. Note that the CUDA code runs on GPU and has a separate CUDA C compiler by NVIDIA.

We will present the results of calculations omitting the file read/writes to concentrate on algorithm's speedup. The UBM experiments were performed on 92 female and 92 male speakers. The adaptation process was performed over 396 speakers of both genders. The testing step was performed on 100 speakers of both genders.

6.1 MFCC feature extraction and K-Means Clustering Modules

The Table 6-2 shows the actual time taken to perform MFCC feature extraction as well as refining its parameters using k-means clustering on UBM.

Table 6-2. Time of MFCC and K-means clustering in seconds.

Algorithm	CPU	CPU Parallel	CUDA (GTX 285)	CUDA (GTX 570)
MFCC	216.6	111.47	310	254
K-means	59.36	14.25	2	2

6.1.1 MFCC

We observe that our MFCC algorithm has poor performance compared to traditional CPU algorithm. The reasons for this are the following:

- The only parallelized function within MFCC was discrete flourier transform algorithm. Although we reduced the complexity in theory, given the size of our problem for a single DFT run, it is not feasible to distribute the sub-problem into too many threads. For our experiment the size of outer loop was 240 and hence 240 threads were activated, but because of the low occupancy level the time needed to read from/write on device global memory affected the performance.

- The other reason is low throughput of memory transfer between host and device. The DFT algorithm is called almost 4000 times for every speaker and each time it copies the window frames from host to device and copies the results of DFT back to host memory.

6.1.2 K-means clustering

In K-means clustering we see a considerable performance improvement when data-parallelism is compared to single threaded CPU. However the time taken on both GPU architectures is the same. The cause of such performance anomaly can be analyzed in the following way:

- *The lack of use of shared memory to benefit from the architecture specific advantages*. The problem targeted in this thesis requires more shared memory than the available amount to perform faster operations. There are two ways to resolve this. One is to activate fewer threads per block so the total required memory to fit our data will be less than the size of available shared memory. The other method is to split the operations inside of each thread into multiple rounds, caching each round for the next round. This requires us to create several kernel functions that represent the original kernel. The second method by itself will cause introducing more memory transfers and it implicitly introduces more of shared memory demand. Therefore this is to be evaluated on problem-specific basis.

- Our algorithm may suffer from *memory partition camping* [28]. Global memory accesses go through partitions. Successive 256-byte regions of global memory are assigned to successive partitions. The problem of partition camping is when global memory accesses at an instant use a subset of partitions. This may hide the true potential of speedup scalability across cores. For optimal performance GPU accesses should be distributed evenly among partitions.

6.2 Expectation Maximization Module

The results of EM algorithm are measured in parts to show the execution times more clearly. The logical behavior of this algorithm is similar in all stages of SV. However the speedups differ relative to the size of input. The difference between EM in UBM modeling and EM in Adaptation or Testing is the size of the problem introduced to each stage as well as the number of executions. UBM deals with all speakers of a gender as one universal model and iterates through them many times. In this thesis our iteration limit was 50 to satisfy the level of convergence (*see* Algorithm 2-1). In adaptation and testing the EM is performed separately for each speaker and it is one iteration per speaker. Table 6-3, Table 6-4 and Table 6-5 illustrate the results of EM in UBM, Adaptation and Test respectively.

Table 6-3. Expectation Maximization of UBM in seconds.

EM Step	CPU	CPU Parallel	CUDA GTX 285	CUDA GTX 570
SUM_P Per EM Iteration	56	17	3	2
Updating Mean,Var & Weight Per EM Iteration	159	45.27	27	10
EM Total Per Iteration	215	62.27	30	12

Table 6-4. Average time of EM of Adaptation for every speaker in seconds.

EM Step	CPU	CPU Parallel	CUDA GTX285	CUDA GTX570
SUM_P and P_i	13.311	3.55	0.309	0.121
Adaptation for mean, weight & variance	24.027	7.45	0.527	0.318
Adaptation for all mixutres	0.015	0.15	0.048	0.047
EM Total	24.042	7.6	0.575	0.365

Table 6-5. Average time of EM in Testing stage for every speaker in seconds.

EM Step	CPU	CPU Parallel	CUDA GTX285	CUDA GTX570
Likelihood UBM	13.921	4.473	3.165	3.004
Likelihood Speaker	5.601	0.724	0.032	0.031
EM Total	19.522	5.197	3.197	3.035

Again we can see the faster calculations on more cores and further on many cores of GPU. This proves the scalability of cores.

6.3 Speedup

The speedup for each module is calculated using:

$$S_p = \frac{T_1}{T_p} \tag{6-1}$$

where:

- p is the number of processors
- T_1 is the execution time of the sequential algorithm
- T_p is the execution time of the parallel algorithm with p processors

The speedup of UBM, Adaptation and Test steps are shown in Table 6-6, Table 6-7 and Table 6-8 respectively. Figure 6-1, Figure 6-2 and Figure 6-3 are graphs corresponding to the latter tables.

Table 6-6. Speedup of UBM step.

Step	CPU	CPU Parallel	CUDA GTX 285	CUDA GTX 570
MFCC	1	1.94	0.69	0.85
K-Mean	1	4.16	29.68	29.68
EM	1	3.45	7.16	17.91

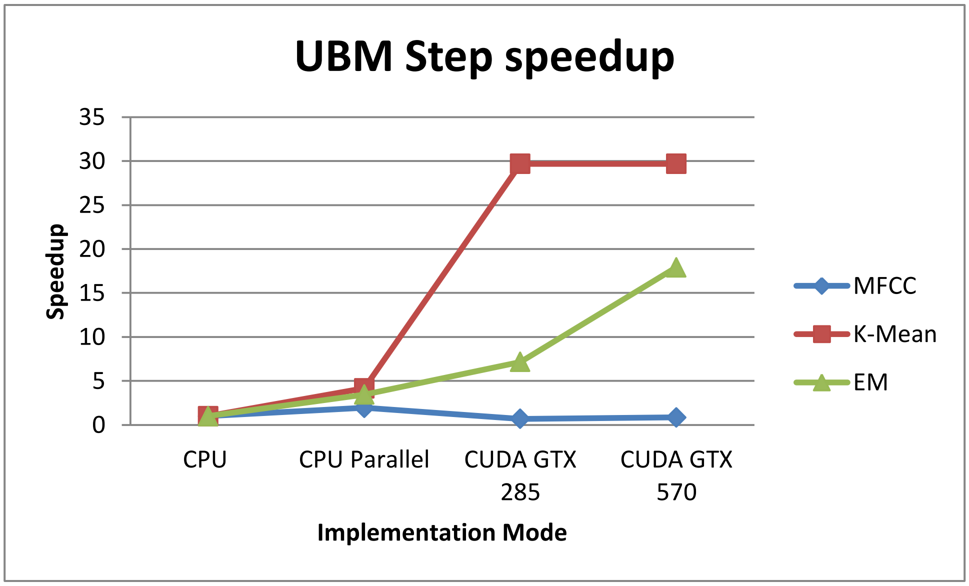

Figure 6-1. UBM Speedup

Table 6-7. Speedup of Adaptation Step

Step	CPU	CPU Parallel	CUDA GTX 285	CUDA GTX 570
MFCC	1	3.51	0.77	0.84
EM	1	3.16	41.81	65.86

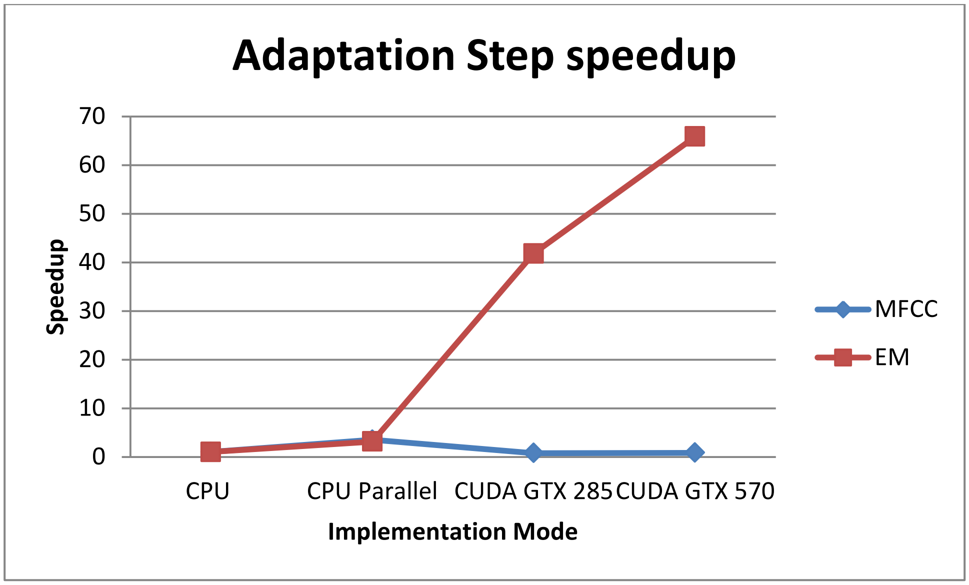

Figure 6-2. Adaptation Speedup

Table 6-8. Speedup of Test step

Step	CPU	CPU Parallel	CUDA GTX 285	CUDA GTX 570
MFCC	1	3.64	0.82	0.93
EM	1	3.75	6.1	6.43

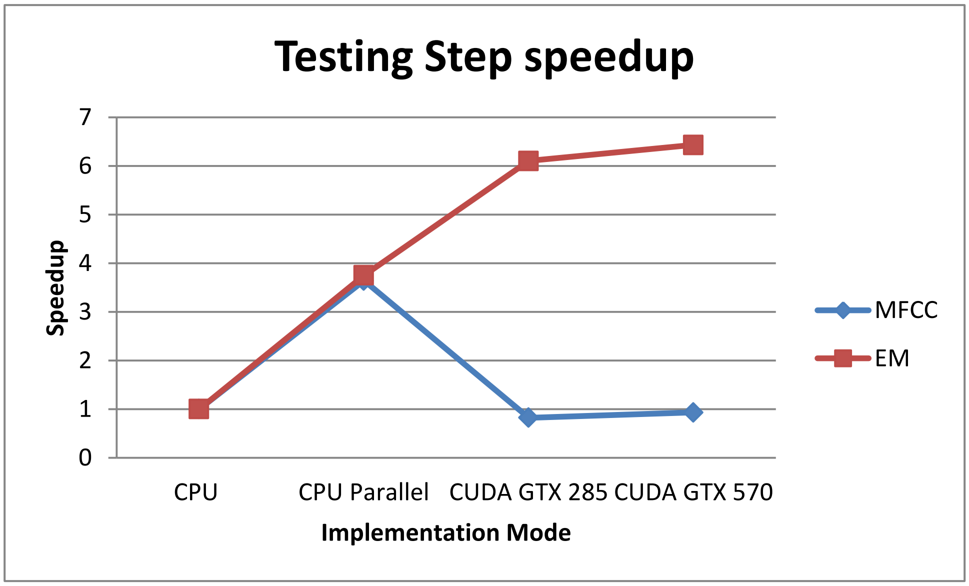

Figure 6-3. Test speedup

6.4 Further Result Analysis

We would like to bring the attention to speed up of K-Means clustering for the case of CPU parallelism in Table 6-6. As we have presented the specification of our machine at the start of this chapter, the number of cores available in Core i7 cpu is 4 (*p*=4). From (6-1), the theoretical maximum speedup should be $S_p = p$ in ideal situation [29] while in our case it is *4.16* which is greater than 4. In some cases we may get speedup larger than p which may be confusing. This is called super speedup and it may happen for various reasons. One known cause is caching effect. Modern machines have advanced techniques for caching data into different memory hierarchies and therefore preventing the redundant calculations.

As can be seen in latter figures, the increase in the number of processing cores considerably increases the speedup of Expectation Maximization algorithm. This shows that the nature of EM is suitable for parallelization.

However, this can be even further optimized by applying advanced GPU techniques that are not extensively discussed in this thesis. An example of such techniques are presented in [24] which the calculations are improved by utilizing shared memory. The speedup in their case is 164 times faster than a single-threaded CPU. However, the problem that was introduced in [24] is not directly comparable to this thesis. One is that their target architecture is G80 and GT200 while our target is GT200 and Fermi. The other fact is that the number of mixtures in their model is only 32 while in this thesis it is discussed to be 2048 which concludes in the different problem sizes.

In CPU architecture most of the execution steps including the context-switching are out of control of the users. They are mostly handled by the processor vendors and operating systems. When programming over GPU architectures such as CUDA, execution steps can be explicitly manipulated or in some cases they can be modified. Therefore, the hardware characteristics should be taken into consideration. In the rest of this chapter, we will discuss issues that will directly affect the performance of our system.

6.4.1 Number of Threads in a Block

The strategy of choosing the correct grid and block sizes is a vital factor in performance. Too little threads in a block may race against the occupancy of the active blocks [14]. Too many threads in a block may prevent us to use shared memory since there is a limit of 16KB on GTX 285 and 48KB on GTX 570 models per block. In our study, we provided a naïve GPU code that uses the global memory. The number of threads we have used per CUDA block varies from 1 to 32. That is because our algorithms performed better in low block dimension sizes. According to [16] in some cases such as [17] a better performance may be achieved in very low block sizes.

In order to find the optimal number of threads per block we have experimented with different block sizes of threads. Note that only the core kernels are shown since the total performance of some algorithms like MFCC feature extraction may not fully demonstrate the effect of different block sizes. In MFCC we focused on Discrete Fourier Transform (DFT). Since the window size is fixed, the DFT performance does not differ in different steps of SV.

The following figures illustrate the achieved results with different block sizes.

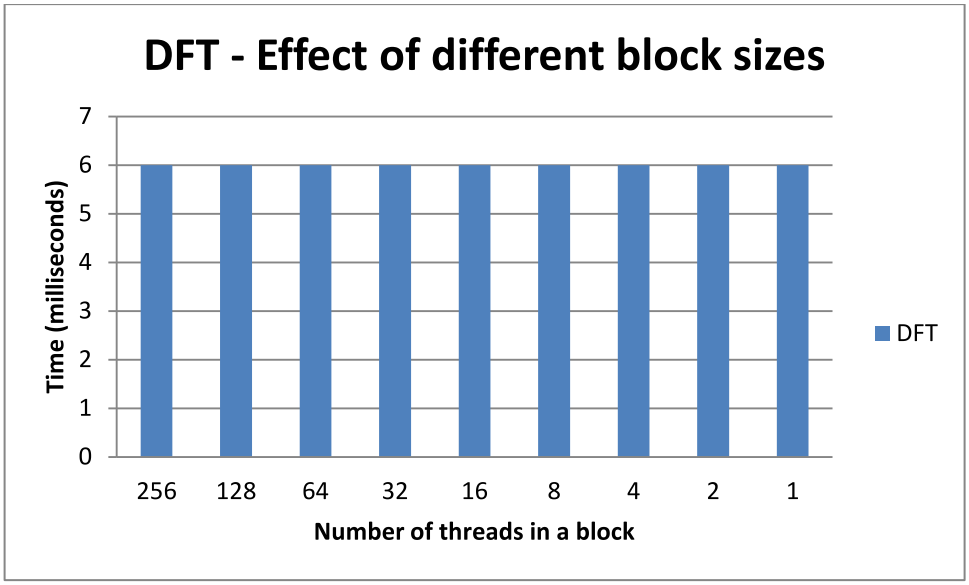

Figure 6-4. Effect of different block sizes on DFT Algorithm

As you may observe from Figure 6-4 there is no effect on DFT algorithm when changing block size. This means that the performance is dominated by some other factors. One could be the excessive memory reads during its execution. We have already calculated and cached our sine and cosine tables on CPU and transferred the result to GPU memory to be accessed by those threads. In detail, we have around 3072 memory read/writes from sine-cosine tables and windowed frame on every thread of DFT kernel.

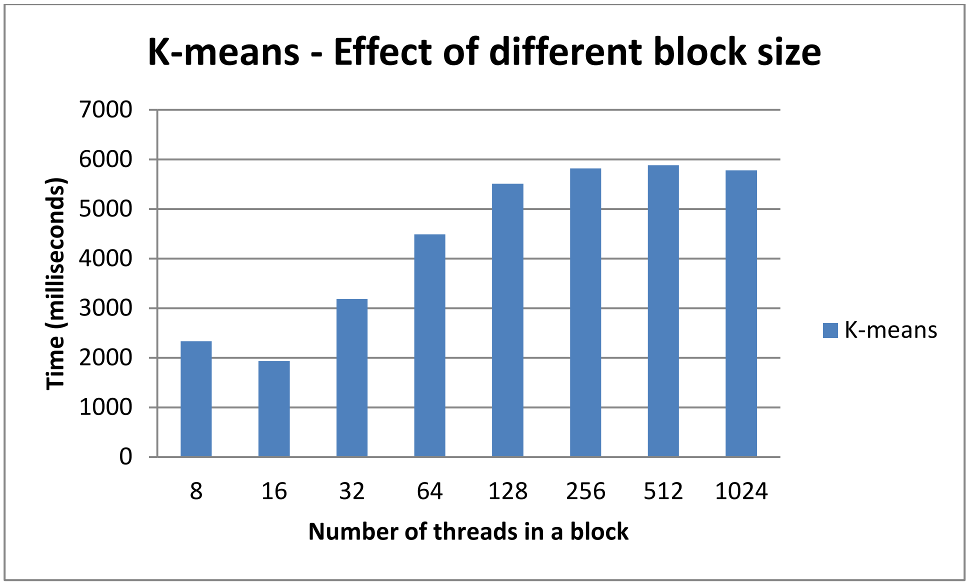

Figure 6-5. Effect of different block sizes in K-means clustering algorithm

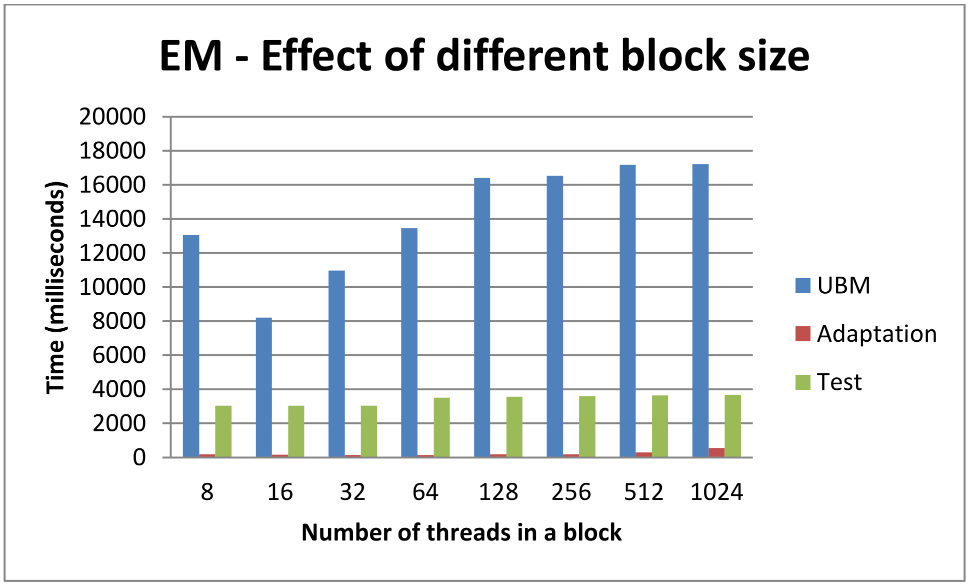

Figure 6-6. Effect of different block sizes in Expectation Maximization Algorithm

Figure 6-5 and Figure 6-6 show the increase of calculation time when block size exceeds 32 in all stages of SV. Several reasons for better utilization in lower block sizes are discussed in section 6.1.2. There are also other factors for this choice:

- *Register Pressure*: Since the number of registers per block is fixed. An ideal compiler tries to maximize the number of registers used in all architectures.

Register pressure is an issue when the number of available registers is less than that of optimal. This forces the compiler to perform register spilling and reloading [14]. Register spilling is the task of copying the registers to the local memory of the thread so higher priority variables can be loaded to registers for that particular moment. In the end the original register values are again reloaded from local memory. This drastically increases global memory accesses and therefore slowing down the performance of the system. By choosing smaller block sizes we reduce the register pressure resulting in higher speeds.

- Block sizes of smaller than 16 will result in slowdown since the cost of activating blocks will exceed the calculation cost per block. This is also referred to as occupancy of a block. In almost most cases the performance is memory bound. This means threads are usually awaiting data from memory to perform operations.

6.4.2 Shared Memory and Memory coalescing

In this thesis we did not use shared memory for our calculation due to various reasons explained through the thesis. One issue of using shared memory is the read access. The memory coalescing is a technique that is used to improve read accesses to global memory. Each thread first reads data from global memory into shared memory before performing arithmetic on them. When multiple memory reads is requested, the data in the global memory is reordered in a way that the words read by consecutive threads fall into consecutive address locations. For example if there are 64 threads and if each thread requests to read 4-bytes (float type) from global memory and if the addresses are sequential then it results in 2 read transactions of 128 bytes chunks, instead of 64 transactions. This is advantageous if the threads have sequential memory accesses. It even becomes more suitable when the size of the requested data is smaller. For example the short data type is only 2 bytes and consecutive reads of 64 short numbers only needs a single read transaction of 128 bytes and this greatly improves the performance. However, in our thesis most of our data is in double size (8 bytes) which decreases the chance of coalescing and it may even become counterproductive since a misaligned 8 byte word is more difficult to fit than a 4 byte word.

6.4.3 Accuracy

Unfortunately the only double-precision floating point arithmetic followed by CUDA technology is IEEE 754 [18]. There *round-to-nearest-even* is the only supported rounding mode for reciprocal, division, and square root. In [30] it is shown that the difference in accuracy error is small enough to neglect. However, in the case of this thesis the results were identical to CPU implementation and did not affect the output of speaker verification.

Chapter 7

CONCLUSION

In this thesis, we introduced a CUDA based implementation of a speaker verification system. We also presented the CPU parallelism of the mentioned algorithms for better comparison. The results demonstrate the advantage of parallelization, specifically using GPUs to speed up calculation of k-means algorithm to almost 30 times and EM to 65 times faster than a single threaded CPU. It is noted that the use of GPU will benefit applications with less cost as the amount of DP computational power is to be increased on GPUs to 6 gigaflops per watt on 2011 and to 15 gigaflops per watt by the end of 2013 which is much faster than the increase rate on traditional CPUs [31].

It should be kept in mind that the GPU implementation of current technologies is very architecture specific and a slight modification in the strategy may drastically change the performance of the system. Therefore, analysis of the architecture is advised.

In the future, the effects of using shared memory on the performance of Speaker Verification will be studied. Additionally, running the algorithms concurrently on multiple GPUs will be presented to show how GPUs can benefit from cross device memory access and how the load balancing should be applied for optimal performance [14]. Advantage of multi-stream processing on CUDA will also be observed. Furthermore, a comparison between the performance hits of single-precision and double-precision floating point calculation will be observed and different data sizes will be experimented.

REFENRECES

[1] Wikipedia, The Free Encyclopedia. (2011, June) Speaker Recognition. [Online]. http://en.wikipedia.org/wiki/Speaker_recognition

[2] Green500. (2011, June) Green500. [Online]. http://www.green500.org/lists.php

[3] Douglas Reynolds. (2011, June) Gaussian Mixture Modeling, MIT. [Online]. http://www.ll.mit.edu/mission/communications/ist/publications/0802_Reynolds_Biometrics-GMM.pdf

[4] Douglas Reynolds. (2011, June) Universal Background Models, MIT. [Online]. http://www.ll.mit.edu/mission/communications/ist/publications/0802_Reynolds_Biometrics_UBM.pdf

[5] Wikipedia, The free encyclopedia. (2011, June) Mel-frequency cepstrum. [Online]. http://en.wikipedia.org/wiki/Mel-frequency_cepstral_coefficient

[6] V. Tyagi and C. Wellekens, "On desensitizing the Mel-Cepstrum to spurious spectral components for Robust Speech Recognition," *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, vol. 1, pp. 529-532, 2005.

[7] Mustafa Jamil, Md. Golam Rabbani Md. Saifur Rahman Md. Rashidul Hasan, "SPEAKER IDENTIFICATION USING MEL FREQUENCY CEPSTRAL

COEFFICIENTS," in *3rd International Conference on Electrical & Computer Engineering*, Dhaka, Bangladesh, 2004.

[8] Richard O. Duda, Peter E. Hart, and David G. Strok, *Pattern Classification*, 2nd ed.: John Wiley and Sons, 2001.

[9] Wikipedia. (2011, June) Expectation-maximization algorithm. [Online]. http://en.wikipedia.org/wiki/Expectation_maximization_algorithm

[10] Cem Ergün, Speech Codec Detector Based Speaker Verification System in a Multi-Coder Environment, doctoral dissertation, 2004.

[11] T. F. Quatieri and R. B. Dunn D. A. Reynolds, "Speaker Verification using Adapted Gaussian Mixture Models," *Digital Signal Processing Review Journal*, vol. 10, pp. 19-41, January 2000.

[12] NVIDIA Corporation. (2011) NVIDIA official website. [Online]. www.nvidia.com

[13] NVIDIA. (2009) NVIDIA's Next Generation CUDA Compute Architecture "Fermi". [Online]. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[14] NVIDIA. (2011, March) NVIDIA CUDA Programming Guide - Version 4.0. [Online]. http://developer.nvidia.com/cuda-toolkit-40

[15] Wikipedia. (2011, June) CUDA. [Online]. http://en.wikipedia.org/wiki/CUDA

[16] V. Volkov. (2011, June) Better Performance at Lower Occupancy, GTC On-Demand: GTC 2010. [Online]. http://www.gputechconf.com/page/gtc-on-demand.html#session2238

[17] Anas Mohd Nazlee and Noohul Basheer Zain Ali Fawnizu Azmadi Hussin, "Tranformation of CPU-based Applications To Leverage on Graphics Processors using CUDA," *IJECS: International Journal of Electrical & Computer Sciences*, vol. 10, no. 1, pp. 40-47, February 2010.

[18] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1-58, August 2008.

[19] Schmadecke, I.; Morschbach, J.; Blume, H.; Inst. of Microeletronic Syst., Leibniz Univ., "GPU-based acoustic feature extraction for electronic media processing," in *Electronic Media Technology (CEMT), 2011 14th ITG Conference on*, Dortmund, 2011, pp. 1-6.

[20] NVIDIA. (2007) CUFFT Library. [Online]. http://developer.download.nvidia.com/compute/cuda/1_0/CUFFT_Library_1.0.pdf

[21] Bai Hong-tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He, "K-Means on Commodity GPUs with CUDA," *Computer Science and Information Engineering, 2009 WRI World Congress on*, vol. 3, pp. 651-655, March 2009.

[22] M. Zechner and M. Granitzer, "Accelerating K-Means on the Graphics Processor via CUDA," in *Intensive Applications and Services, 2009. INTENSIVE '09. First*

International Conference on, Valencia, 2009, pp. 7-15.

[23] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu, "Speeding up K-Means Algorithm by GPUs," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, Bradford, 2010, pp. 115-122.

[24] N. Kumar, S. Satoor, and I. Buck, "Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA," in *High Performance Computing and Communication, 11th IEEE International Conference on*, 2009, pp. 103-109.

[25] Intel. (2011, June) Previous Generation Intel® Core™ i7 Processor. [Online]. http://www.intel.com/products/processor/previousgenerationcorei7/index.htm

[26] Company for Advanced Supercomputing Solutions GASS. (2008) CUDA.NET. [Online]. http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx

[27] Hybrid DSP. (2011) CUDAfy.NET – General Purpose GPU on.NET. [Online]. http://www.hybriddsp.com/Products/CUDAfyNET.aspx

[28] NVIDIA. (2009, March) Optimizing CUDA, Austrialian National University: CUDA Tutorial. [Online]. http://cs.anu.edu.au/files/systems/GPUWksp/PDFs/04_OptimizingCUDA_full.pdf

[29] Gita Alaghband Harry F. Jordan, *Fundamentals of Parallel Processing*.: Prentice Hall, 2003.

[30] Jeremy Meredith - TU Dresden - ZIH Guido Juckeland. (2010) Analyzing CUDA Accelerated Application Performance at 20 PFLOP/s, GPU Technology Conference. [Online]. http://www.gputechconf.com/page/gtc-on-demand.html

[31] NVIDIA Corparation. (2010) nVIDIA CUDA GPU roadmap, GPU Technology Conference 2010. [Online]. http://www.gputechconf.com/page/home.html