2009 11th IEEE International Conference on High Performance Computing and Communications

Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs using CUDA

N. S. L. Phani Kumar, Sanjiv Satoor and Ian Buck
NVIDIA Corporation
Email: nskumar@nvidia.com, ssatoor@nvidia.com, ianbuck@nvidia.com

***Abstract*— Expectation Maximization (EM) algorithm is an iterative technique widely used in the fields of signal processing and data mining. We present a parallel implementation of EM for finding maximum likelihood estimates of parameters of Gaussian mixture models, designed for many-core architecture of Graphics Processing Units (GPU). The algorithm is implemented on NVIDIA's GPUs using CUDA, following the single instruction multiple threads model. In this paper, the emphasis is laid on exploiting the data parallelism with CUDA, thus accelerating the computations. CUDA implementation of EM is designed in such a way that the speed of computation of the algorithm scales up with the number of GPU cores. Experimental results confirm the scalability across cores. The results also show that CUDA implementation of EM when applied to an input of 230K for a 32-order mixture of 32-dimensional Gaussian model takes 264 $msec$ on Quadro FX 5800 (NVIDIA 200 series) with 240 cores to complete one iteration which is about 164 times faster when compared to a naive single threaded C implementation on CPU.**

I. Introduction

In the world of parametric estimation, Maximum Likelihood Estimation (MLE) bears a lot of importance in diverse fields. MLE is a popular statistical method of fitting a mathematical model to the data observed. Expectation Maximization (EM) algorithm is broadly applicable approach to the iterative computation of ML estimates, useful in a variety of incomplete-data problems where algorithms such as Newton-Raphson method may turn out to be more complicated. The situations where EM algorithm is applied include not only evidently incomplete-data situations, where there are missing data, but also a whole variety of situations where the incompleteness of data is not all that natural or evident. Hitherto intractable ML estimation problems for these situations have been solved or complicated ML estimation procedures have been simplified using the EM algorithm. The basic idea of the EM algorithm is to associate with the given incomplete-data problem, a complete-data problem for which ML estimation is computationally more tractable, yielding closed form estimates. In a nutshell, EM algorithm contains two steps, the E-step and the M-step. E-step consists in manufacturing log-likelihood of the complete data problem, using the observed data set of the incomplete-data problem and the current value of the parameters. M-step deals with the maximization of the log-likelihood generated by the E-step. These two steps are repeated until convergence [1] . EM has thus found applications in almost all statistical contexts and in almost all fields where statistical techniques have been applied - data mining, signal processing, medical imaging, etc.

In this paper, the application of EM for Gaussian Mixture Models (GMM) [1] is dealt with. This is because the fitting of mixture models by maximum likelihood is a classic example of a problem that is simplified considerably by the EM's conceptual unification of ML estimation from data that can be viewed as being incomplete. The motivation behind the usage of Gaussian mixture models is the fact that Gaussian mixture is shown to provide a smooth approximation to underlying long-term sample distribution in many situations. A wide variety of applications of mixture models occur in the fields of signal processing and data mining. In the area of speaker recognition [2] [3], GMM based speaker model is used to represent some acoustic classes which reflect speaker dependent vocal tract information. In the field of distributed data stream clustering [4], the data streams are treated as being generated by Gaussian Mixture Models. EM is used for learning the mixture model parameters (cluster parameters and mixture weights) in the presence of incomplete data.

The tremendous amount of computational power and availability of programming models such as CUDA have made GPU a strong candidate for performing many compute intensive tasks. With multiple cores driven by very high memory bandwidth, today's GPUs offer incredible resources not only for both graphics but also for non-graphics processing. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations implying that the same program is executed on many data elements in parallel. Data-parallel processing maps data elements to parallel processing threads. Many applications in the field of image, video and media processing that process large data sets such as image blocks and pixels can use a data-parallel programming model to speed up the computations. Besides these, many applications ranging from general signal processing or physics simulation to computational finance or computational biology are accelerated by data-parallel processing. NVIDIA's high end GPUs with CUDA environment [5] provides an easy-to-implement C based language to exploit data parallelism on the GPUs.

In this paper, we describe the CUDA implementation of EM algorithm for GMM. Initially EM starts with some guess parameters and keeps estimating the parameters for every iteration which form the guess parameters for the next iteration. The estimates at the end of each iteration are given

978-0-7695-3738-2/09 $25.00 © 2009 IEEE
DOI 10.1109/HPCC.2009.45

by update equations. The approach chosen is bottom-up by first decomposing the algorithm into a primitive level and start building upon it. At every level, the data parallelism has been extracted and exploited to a considerable extent using Single Input Multiple Thread (SIMT) architecture. In the final stage, the key for data parallelism lies in expressing the update equations as matrix transformations which best utilize the parallel processing capabilities of GPUs. The rest of the paper is organized as follows. In Section II, the EM algorithm for GMM is described. CUDA architecture of NVIDIA GPUs is discussed in Section III. In Section IV, design of EM for GMM on CUDA is presented. The effect of optimized data fetch called coalesced memory accesses on the performance is discussed in Section V. Section VI deals with the experimentation results on different series of NVIDIA GPUs with varying number of cores and analyzes the performance and scale-up with number of cores. The results also include the performance boost achieved by memory coalescing. Finally we present concluding remarks and future work to be pursued in Section VII.

II. Expectation Maximization (EM) for Gaussian Mixture Models (GMM)

Let $X \in \{x_1, x_2, ..., x_N\}$ be the data of size N supposedly drawn from a distribution with density function $p(x|\Theta)$ governed by the set of parameters Θ. If we assume that the data vectors $x_i, i \in \{1, 2, ..., N\}$ are independent and identically distributed (i.i.d) with distribution p, the resultant density for the samples is given by,

$$p(x|\Theta) = \Pi_{i=1}^{N} p(x_i|\Theta) = L(\Theta|X) \tag{1}$$

This function $L(\Theta|X)$ is called the likelihood function of parameters given the data. The ML estimates are those which maximize this likelihood function. If Θ_{MLE} are the ML estimates, then

$$\Theta_{MLE} = \underset{\Theta}{argmax} L(\Theta|X) \tag{2}$$

This is same as maximizing $log(L(\Theta|X))$, the log-likelihood function.

In the case of mixture models, the probability density function $p(x|\Theta)$ is given by,

$$p(x|\Theta) = \sum_{i=1}^{M} \alpha_i p_i(x|\theta_i) \tag{3}$$

where M is the mixture length and
$\Theta = (\alpha_1, \alpha_2, ..., \alpha_M, \theta_1, \theta_2, ...\theta_M)$ such that $\sum_{i=1}^{M} \alpha_i = 1$ are the parameters to be estimated and each p_i is density function characterized by θ_i. If the component densities are d-dimensional Gaussian densities with mean μ and covariance matrix Σ, i.e., $\theta_i = (\mu_i, \Sigma_i)$ given by,

$$p_i(x|\theta_i) = p_i(x|\mu_i, \Sigma_i) = \frac{1}{2\pi^{\frac{d}{2}} |\Sigma_i|^{1/2}} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i)} \tag{4}$$

for $i \in 1, 2..., M$, the model is called Gaussian Mixture Model.

EM starts with the assumption that X is incomplete [6]. It assumes that there exists a complete dataset $Z = (X, Y)$ whose joint density function is given by,

$$p(z|\Theta) = p(y|x, \Theta)p(x|\Theta) \tag{5}$$

With this new joint density, the complete-data log likelihood function is defined as $L(\Theta|Z) = L(\Theta|X, Y) = p(X, Y|\Theta)$. The EM algorithm first finds the expectation of the complete-data log-likelihood function $log(p(X, Y|\Theta))$ with respect to unknown data Y given the observed data X and the current parameter estimates $\Theta^{(i-1)}$ defined by,

$$Q(\Theta, \Theta^{(i-1)}) = E[log(p(X, Y|\Theta)|X, \Theta^{(i-1)})] \tag{6}$$

where Θ are the parameters to be estimated. This is called the E-step.

The second step (M-step) is to maximize the expectation computed in the E-step, i.e., to find

$$\Theta^{(i)} = \underset{\Theta}{argmax} Q(\Theta, \Theta^{(i-1)}) \tag{7}$$

These two steps are repeated as necessary. Each step is guaranteed to increase the log-likelihood and the algorithm is guaranteed to converge to a local maxima of the likelihood function.

For GMM, if X is considered incomplete and assumes the existence of unobserved data $Y = \{y_i\}, i \in \{1, 2, ...N\}$ whose values inform which component density generates each data item, the complete-data log likelihood function is simplified. That is, if $y_i = k$ and $y_i \in \{1, 2, ...M\}$, it implies that the i^{th} sample is generated by the k^{th} mixture component. The complete-data log likelihood function is given by

$$log(L(\Theta|X, Y)) = \sum_{i=1}^{N} log(\alpha_{y_i} p_{y_i}(x_i|\theta_{y_i})) \tag{8}$$

The expression for the density of the unobserved data given the observed data and the guess parameters $\Theta^g = (\alpha_1^g, ..., \alpha_M^g, \theta_1^g, ..., \theta_M^g)$ is given by,

$$p(y_i|x_i, \Theta^g) = \frac{\alpha_{y_i}^g p_{y_i}(x_i|\theta_{y_i}^g)}{\sum_{k=1}^{M} \alpha_k^g p_k(x_i|\theta_k^g)} \tag{9}$$

and

$$p(y|X, \Theta^g) = \Pi_{i=1}^{N} p(y_i|x_i, \Theta^g) \tag{10}$$

where $y = (y_1, ..., y_N)$ are independently drawn unobserved data.

The E-step applied to mixture model results in the expectation of complete-data log likelihood function given by,

$$Q(\Theta, \Theta^g) = \sum_{l=1}^{M}\sum_{i=1}^{N} log(\alpha_l)p(l|x_i, \Theta^g) + \sum_{l=1}^{M}\sum_{i=1}^{N} log(p_l(x_i|\theta_l))p(l|x_i, \Theta^g) \tag{11}$$

where $p(l|x_i, \Theta^g)$ is computed as in the Equation (9) putting $y_i = l$.

The M-step which maximizes the above expectation for the case of GMM yields the following update equations for mixture coefficients (α_l^{new}), means (μ_l^{new}) and covariance matrix (Σ_l^{new}) as [7] ,

$$\alpha_l^{new} = \frac{1}{N}\sum_{i=1}^{N} p(l|x_i, \Theta^g) \tag{12}$$

$$\mu_l^{new} = \frac{\sum_{i=1}^{N} x_i p(l|x_i, \Theta^g)}{\sum_{i=1}^{N} p(l|x_i, \Theta^g)} \tag{13}$$

$$\Sigma_l^{new} = \frac{\sum_{i=1}^{N} p(l|x_i, \Theta^g)(x_i - \mu_l)(x_i - \mu_l)^T}{\sum_{i=1}^{N} p(l|x_i, \Theta^g)} \tag{14}$$

III. CUDA on NVIDIA GPUs

CUDA [5] is a hardware-software architecture exposing the parallel data processing capabilities of GPUs. The operating system's multitasking capability is responsible for managing the access to the GPU by several CUDA and graphics applications running concurrently. CUDA allows the users to view GPU as a highly multi-threaded coprocessor capable of executing high number of threads in parallel off-loading the CPU while running compute-intensive applications. A portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the GPU as many different threads. To that effect, such a function is compiled to the instruction set of the GPU and the resulting program, called a kernel, is downloaded to the GPU.

NVIDIA's GeForce 8-series GPUs and above with the CUDA programming model provides an adequate API for non-graphics applications. CUDA provides general DRAM memory addressing on GPUs for more programming flexibility: both scatter and gather memory operations. From a programming perspective, this translates into the ability to read and write data at any location in DRAM, just like on a CPU. CUDA features a parallel data cache or on-chip shared memory with very fast general read and write access, that threads use to share data with each other. The applications can take advantage of it by minimizing over-fetch and round-trips to DRAM and therefore becoming less dependent on DRAM memory bandwidth.

The device is implemented as a set of multiprocessors as illustrated in Figure 1. Each multiprocessor has a Single Instruction, Multiple Thread architecture (SIMT). At any given

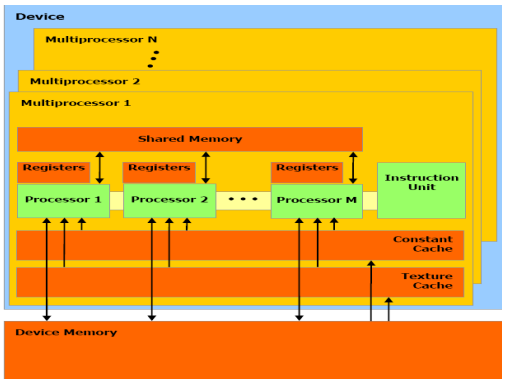

Fig. 1. Hardware Model in CUDA

clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. The batch of threads that executes a kernel is organized as a grid of thread blocks and illustrated in Figure 2. Each multiprocessor processes batches of blocks one batch after the other. A block is processed by only one multiprocessor, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses. The blocks that are processed by one multiprocessor in one batch are referred to as active. Each active block is split into SIMT groups of threads called warps. Each of these warps contains the same number of threads, called the warp size, and is executed by the multiprocessor in a SIMT fashion. Active warps i.e. all the warps from all active blocks are time-sliced. A thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessors computational resources.

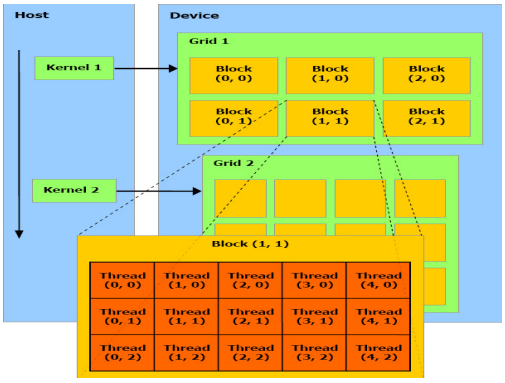

Fig. 2. Thread Batching in CUDA programming model

IV. CUDA implementation of Expectation Maximization for Gaussian Mixture Models

As discussed in Section II, for parametric estimation of GMM, EM algorithm starts with some guess parameters and at the end of every iteration, the new estimates are formed as per the update equations (12, 13, 14) which are fed as the guess parameters to the next iteration. In this work, we assumed that the covariance matrices $(\Sigma_1, ...\Sigma_M)$ are diagonal implying statistical independence of each d-dimensional Gaussian mixture component.

Following the notations of Section II, let N be the data size, M be the mixture length, D be the dimension of the mixture component Gaussian density. Let the INPUT observed data (X) in matrix form be $[\overline{x_1}...\overline{x_N}]^T$, where $\overline{x_i} = [x_{i1}....x_{iD}]^T$, $i = (1, ..., N)$. The parameters to be estimated are :

Mixture Coefficients (A) : $(\alpha_1, ..., \alpha_M)$

MEAN in matrix form : $[\overline{\mu_1}...\overline{\mu_M}]^T$, where $\overline{\mu_i} = [\mu_{i1}...\mu_{iD}]^T$, $i = (1, ..., M)$

COVARIANCE (Σ) in matrix form : $[\overline{\Sigma_1}...\overline{\Sigma_M}]^T$, where $\overline{\Sigma_i} = [\sigma_{i1}^2...\sigma_{iD}^2]^T$, $i = (1, ..., M)$

CUDA implementation starts with the computation of basis expressions for k^{th} mixture component's joint Gaussian density with respect to the i^{th} data sample $\overline{x_i}$ and guess parameters θ_k^g as,

$$p_k(\overline{x_i}|\theta_k^g = (\overline{\mu_k^g}, \overline{\Sigma_k^g})) = \frac{e^{-[\frac{(x_{i1}-\mu_{k1}^g)^2}{(\sigma_{k1}^g)^2}+...+\frac{(x_{iD}-\mu_{kD}^g)^2}{(\sigma_{kD}^g)^2}]}}{(2\pi)^{D/2}\sqrt{(\sigma_{k1}^g)^2...(\sigma_{kD}^g)^2}} \quad (15)$$

The apriori density of unobserved data samples with respect to i^{th} observed data sample and guess parameters θ_l^g defined by $p(l|\overline{x_i}, \theta_l^g)$ is given by the Equation (9) in Section II.

The CUDA implementation consists of the sequential launch of six kernels (code executing on GPU) for every iteration.

(i) The first kernel called COMPUTE-BASIS builds up a matrix called BASIS matrix of order (N x M) whose $(i,k)^{th}$ element is given by

$$BASIS_{(i,k)} = p_k(\overline{x_i}|\theta_k^g = (\overline{\mu_k^g}, \overline{\Sigma_k^g})) \quad (16)$$

for all $i = (1, 2, ..., N), k = (1, 2, ..., M)$.

This kernel is implemented on a grid of $\frac{N}{B}$ thread blocks, each consisting of B threads. The optimum block size 'B' is computed basing on the resources available for the threads like on-chip shared memory and registers. Each thread computes M densities for all the mixture components. Figure 3 illustrates pseudo CUDA implementation of COMPUTE-BASIS kernel.

(ii) The second kernel called COMPUTE-APRIORI builds up a matrix called APRIORI (P) matrix of order (N x M)

```
compute-basis <<< N/B , B >>>(input, mean, correlation, BASIS )
// Launch a grid of 'N/B' blocks and each block containing 'B' threads.
// That means N x D matrix is split into 'N/B' blocks of B x D matrices
// Each thread accessing one row of 'D' elements
// blockIndex will be the index of thread-block
// threadIndex is the index of thread in the block
{
    product = 1;
    for ( i = 0; i < M; i++ )
    {
        for ( j = 0; j < D; j++ )
        {
            // Compute the density as per Equation () as follows

            product = product * e^( -(input[threadIndex][j]-mean[i][j])^2 / 2*(correlation[i][j])^2 ) / 2*(correlation[i][j]) ;

        }
        BASIS [ blockIndex * B + threadIndex ] [ i ] = product;
    }
}
```

Fig. 3. CUDA implementation of COMPUTE-BASIS kernel

from the BASIS whose $(i,l)^{th}$ element is given by

$$APRIORI_{(i,l)} = p(l|\overline{x_i}, \theta_l^g) \quad (17)$$

for all $i = (1, 2, ..., N), l = (1, 2, ..., M)$.

This kernel is also implemented on a grid of $\frac{N}{B}$ thread blocks, each consisting of B threads. Each thread will be computing M apriori densities for all the mixture components. This matrix is crucial for computing the estimates of the parameters.

(iii) The third kernel called ESTIMATE-ALPHA estimates the values of mixture coefficients basing on the Equation (12). This kernel is implemented on a grid of $\frac{M}{B}$ thread blocks each consisting of B threads.

(iv) The fourth kernel implementation expresses the estimates of $MEAN$ matrix as a function of APRIORI (P) and INPUT (X) given by,

$$MEAN = P^T X \quad (18)$$

This is implemented by $cublasSgemm$ which is a CUBLAS library function for matrix multiplication, an efficient parallel implementation. Then dividing $i^{th}, (\forall i = 1, 2, ...M)$ row of $MEAN$ by $N\alpha_i$ gives the updated $MEAN$ matrix.

(v) To estimate the COVARIANCE matrix, expressing the update Equation (14) in terms of matrix transformations is not very straightforward. To do this, we define a new matrix called VARIANCE (V) matrix of order (N x MD) whose

$(i,(m-1)D+n)^{th}$ element, where $(1 \leq i \leq N), (1 \leq m \leq M)$ and $(1 \leq n \leq D)$ is computed as,

$$V_{(i,j)} = (x_{in} - \mu_{mn})^2 \tag{19}$$

Lets define another matrix $\hat{\Sigma}$ of order (M x MD) computed as

$$\hat{\Sigma} = P^T V \tag{20}$$

Now arranging the rows of $\hat{\Sigma}$ in the form of sequence of 'M' D-tuples, then sampling out only the diagonal D-tuples and arranging them as M rows yields COVARIANCE matrix Σ of order (M x D). The whole operation can be expressed as,

$$\Sigma_{i,j} = \hat{\Sigma}_{i,(i-1)D+j} \tag{21}$$

for all $i = (1, 2, ..., M), j = (1, 2, ..., D)$. The computation of matrix V forms the fifth kernel (COMPUTE-VARIANCE) in our CUDA implementation on a grid of $\frac{N}{B}$ thread blocks, each consisting of B threads and each thread producing MD elements.

(vi) The computation of $\hat{\Sigma}$ forms the sixth kernel which is again implemented by CUBLAS matrix multiplication library. The tapping of diagonal D-tuples followed by division of $i^{th}, (\forall i = 1, 2, ...M)$ row of Σ by $N\alpha_i$ gives the updated Σ matrix.

Table I presents the statistics of threads-blocks, threads, shared memory (SMEM), Registers/Thread (R) required by the above said kernels when $N = 153600, M = 32, D = 32$ and $B = 64$.

TABLE I
STATISTICS OF KERNELS ON CUDA FOR EM

Kernel	Thread Blocks	Threads/Block	SMEM(bytes)	R
BASIS	1200	64	8224	11
APRIORI	2400	64	8348	6
ALPHA	64	64	280	6
VARIANCE	2400	64	12316	9

While launching the kernels with the grid of thread blocks, it is made sure that the same code runs on multiple cores without any underutilization of GPU resources. For example, if the thread blocks are divided equally on x cores, when the cores have grown up to y, the equal distribution of thread blocks should be maintained or some cores will stay idle causing load imbalance which reflects badly on the performance. The input data sizes are chosen carefully in our implementation keeping in mind the load balancing.

V. EFFECT OF COALESCED MEMORY ACCESSES ON PERFORMANCE

As the shared memory bandwidth is much higher than global memory (DRAM) bandwidth, in the implementation of every kernel discussed in the last section, the data accessed by each thread is read first from the global memory into the on-chip (shared) memory before doing any arithmetic. The way data is stored in the global memory has a bearing on the latency involved while reading from global memory to shared memory. The data is reordered in global memory in such a way that the words read by consecutive threads fall into consecutive address locations. For example, if there are 32 threads, each reading 4-bytes of data from the global memory, and if the addresses they read from are randomly scattered, then it results in 32 read transactions. But if the global memory read accesses by all the threads lie in a chunk of continuous 128 bytes, the requests are coalesced into a single 128-byte read transaction. This is called memory coalescing.

For example, in the implementation of COMPUTE-BASIS kernel as shown in Figure 3, the INPUT float data matrix X of order N x D is stored in global memory. If X is stored in row-major order, each thread in a block of B-threads accesses a particular row of INPUT starting from the left. Thus the sub matrix of INPUT accessed by each block is X' of order B x D. In this row-major case, the global memory addresses the consecutive threads read from, fall into locations separated by an offset of D data elements resulting in B distinct read transactions as shown in Figure 4(a). But, if the data is stored in column-major order, each thread in a block accesses a particular column of $(X')^T$ of order D x B, starting from the top. In this case, it is guaranteed that the data read by each of the $B/32$ distinct sets of 32 adjacent threads (i.e; threads 0-31, 32-63, .., $B-32$ to $B-1$) fall into 32 contiguous locations of global memory which results in coalescing and hence giving

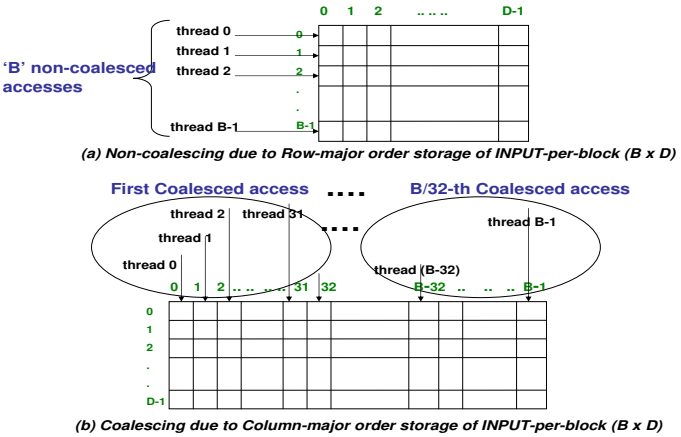

Fig. 4. Data storage affecting memory coalescing

rise to $B/32$ coalesced read transactions (assuming B is a multiple of 32) as shown in Figure 4(b). Thus by memory coalescing, the kernels can be significantly accelerated.

VI. Performance Results and Analysis

In this section, we present the experimentation results of EM algorithm for GMM on both CPU and GPU. Regarding the CPU implementation, the algorithm is implemented using non-optimized single-threaded C language. Both CPU and CUDA implementations of the algorithm have been tested on WinXP environment with DualCore 3.0 GHz Pentium IV CPU having 1 GB RAM, 2048K L2 cache and on Geforce 8800 ULTRA (NVIDIA 8 series), GTX260 and Quadro FX 5800 (NVIDIA 200 series). Table II shows the technical specifications of NVIDIA GPUs used for experimentation.

The input data fitting into Gaussian Mixture Model framework is generated by MATLAB. The algorithm has been tested for small, moderate and big datasets with increasing input data size N, mixture length M, dimension of mixture component Gaussian density D. Because of huge memory requirement, we are restricted to testing very big data set with $N = 230400, M = 32, D = 32$ only on Quadro FX 5800 with 4GB of GPU memory. The block size 'B' discussed in the last section is chosen sub-optimally to vary from $B = 64$ to $B = 256$, keeping in mind the usage of hardware resources and several trade-offs which come into play because of hardware scheduling. On Quadro FX 5800, the time taken for one iteration for very big-sized dataset ($N = 230400, M = 32, D = 32$) is 265 $msec$ which is about 164 times faster than that on the CPU. Table III, IV compares the performance of CPU and CUDA implementations with respect to time taken per iteration of EM algorithm on Geforce 8800 ULTRA and Quadro FX 5800 respectively. We observe that as the size of the dataset grows, the performance of GPU is increasing over CPU. The mean absolute difference between the CPU and GPU results comes to 0.000005. From the tables, we observe that the performance is scaling up while increasing number of GPU cores.

Figure 5 shows the variation of computing time versus the number of Floating Point Operations on Geforce 8800 ULTRA (128 $cores$) and Quadro FX 5800 (240 $cores$). We can see that as the number of floating point operations increase, the performance on 240 cores is improving over that of 128 cores. The floating point operations are computed basing on the values of N, M, D. Figure 6 shows the scalability across the cores of GPU for increasing data sets. From the figure, we

TABLE II
Specifications of NVIDIA GPUs

GPU	Cores	DRAM	Processor clock	Memory Bandwidth
8800 ULTRA	128	768 MB	612 MHz	103.7 GB/s
GTX260	192	896 MB	1.2 GHz	111.9 GB/s
Quadro FX 5800	240	4 GB	1.3 GHz	102 GB/s

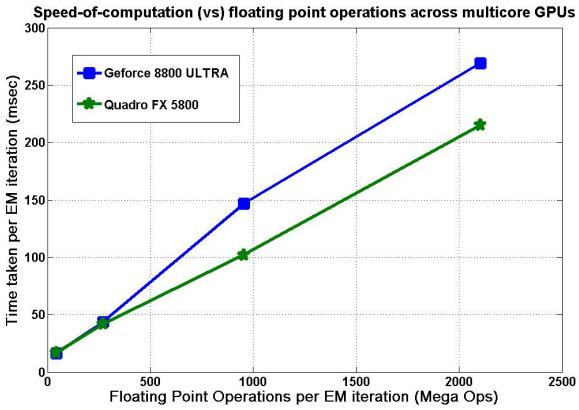

Fig. 5. Speed-of-computation (vs) floating point operations across multiple cores

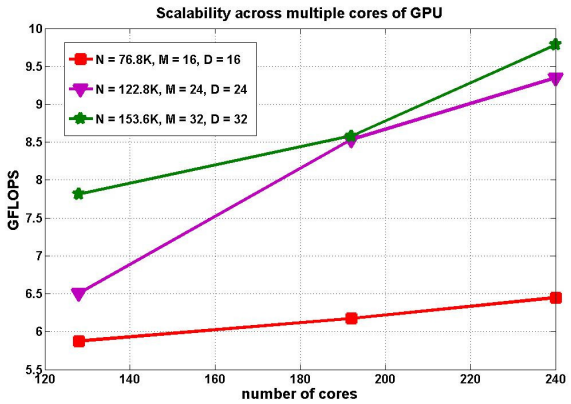

Fig. 6. Scalabilty across number of cores of GPU

can see that gigaflops (GFLOPS) are increasing with number of cores for the same data set which confirms scalability. When the data set grows, the GFLOPS achieved by the same number of cores keep increasing. For higher end GPUs (like GTX260 and Quadro FX 5800), GFLOPS increase at a slower rate compared to growth of data size. We observe that for bigger datasets, the graph appears to be piece-wise linear with varying slope. This may be due to sub-optimal block size 'B', which results in bigger queues of tasks scheduled onto the multiprocessors, in which case hardware scheduling dominates the speed of execution. Also the bent curves in

the case of 192 cores and 240 cores are due to insufficiency of threads launched on the multiprocessors to combat the shared memory and global memory access latencies while proceeding to more number of cores and bigger data sets. Table V explains the effect of memory coalescing (explained in section V) on the speed of computation of kernels discussed in the last section. From the table, we can see that the time taken by GPU drastically comes down with coalesced memory accesses.

TABLE III
PERFORMANCE COMPARISON BETWEEN CPU AND CUDA IMPLEMENTATION ON GEFORCE 8800 ULTRA WITH 128 CORES

N	M	D	CPU(t1 msec)	CUDA(t2 msec)	Speed-up (t1/t2)
46.8K	8	8	338.9	16.2	20.8x
76.8K	16	16	2401.4	43.8	54.8x
122.8K	24	24	11165.8	146.8	76.1x
153.6K	32	32	32153.0	269.3	119.3x

TABLE IV
PERFORMANCE COMPARISON BETWEEN CPU AND CUDA IMPLEMENTATION ON QUADRO FX 5800 WITH 240 CORES

N	M	D	CPU(t1 msec)	CUDA(t2 msec)	Speed-up (t1/t2)
46.8K	8	8	338.9	16.2	20.8x
76.8K	16	16	2495.8	42.4	58.8x
122.8K	24	24	11175.1	102.1	109.4x
153.6K	32	32	32225.6	215.0	150.0x
230.4K	32	32	43352.6	264.9	164.0x

TABLE V
EFFECT OF MEMORY COALESCING ON COMPUTING TIME TAKEN BY QUADRO FX 5800(MICRO SECONDS)

Kernel	GPU time without coalescing	GPU time with coalescing
BASIS	124579	497.6
APRIORI	4346.9	2350.8
VARIANCE	110962	25403.5

VII. CONCLUSIONS AND FUTURE WORK

We have presented a fast parallel implementation of EM on NVIDIA GPUs using CUDA. The results show that GPUs are well suited for implementing statistical algorithms. There are many factors to be chosen carefully for attaining greater speed-ups like on-chip shared memory usage and register usage per thread. Also, we observe that when launching less number of thread blocks making work-load per thread heavy produced better results than launching more number of light-weight thread blocks. Future work deals with the avoidance of bank conflicts when accessing on-chip shared memory. The choice of optimum block size trading off with hardware constraints is to be given much deeper thought. Also, while estimating COVARIANCE matrices from APRIORI and VARIANCE matrices, as discussed in Section IV, $cublasGemm$ library function computes each and every element of order (M x MD) which is an overhead because it is sufficient if we compute diagonal D-tuples which reduces the number of floating point operations by $MD(M-1)(2N^2-1)$. These customizations to the BLAS functions will be addressed in future. Multi-core CPU implementation of Expectation Maximization and its comparison with CUDA implementation will also be pursued as a part of future work.

REFERENCES

[1] Geoffrey J. McLachlan, Thriyambakam Krishnan, The EM Algorithm and Extensions, Wiley Series in Probability and Statistics, John Wiley and Sons, 1997.

[2] Douglas A. Reynolds and Richard C. Rose, "Robust Text-Independent Speaker Identification Using Gaussian Mixture Speaker Models", IEEE Transactions on Speech And Audio Processing, Vol. 3, No. 1, January 1995.

[3] L. Rabiner and B.H. Juang, Fundamentals of Speech Recognition, Prentice Hall Signal Processing Series, 1993.

[4] Aoying Zhou, Feng Cao, Ying Yan, Chaofeng Sha, Xiaofeng He, "Distributed Data Stream Clustering: A Fast EM-based Approach", IEEE 23rd International Conference on Data Engineering, April 2007.

[5] NVIDIA CUDA Programming Guide - version 2.1 available as `http://www.nvidia.com/object/cuda_develop.html`

[6] A.P. Dempster, N.M. Laird, and D.B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm", J. Royal Statististical Society, Series B, Vol. 39, 1977.

[7] Jeff A. Bilmes, "A Gentle Tutorial of the EM Algorithm and its Application to Parametric estimation for Gaussian Mixture and Hidden Markov Models", International Computer Science Institute, Berkeley CA.