Journal of Computer Science 10 (10): 2124-2134, 2014
ISSN: 1549-3636

doi:10.3844/jcssp.2014.2124.2134 Published Online 10 (10) 2014 (http://www.thescipub.com/jcs.toc)

PARALLEL IMPLEMENTATION OF EXPECTATION-MAXIMISATION ALGORITHM FOR THE TRAINING OF GAUSSIAN MIXTURE MODELS

[1]Araújo, G.F., [2]H.T. Macedo, [2]M.T. Chella, [2]C.A.E. Montesco and [2]M.V.O. Medeiros

[1]Núcleo de Pós-Graduação em Ciência da Computação, UFS, São Cristóvão and Brasil
[2]Departamento de Computação, UFS, São Cristóvão, Brasil

Received 2013-11-18; Revised 2013-11-22; Accepted 2014-07-07

ABSTRACT

Most machine learning algorithms need to handle large data sets. This feature often leads to limitations on processing time and memory. The Expectation-Maximization (EM) is one of such algorithms, which is used to train one of the most commonly used parametric statistical models, the Gaussian Mixture Models (GMM). All steps of the algorithm are potentially parallelizable once they iterate over the entire data set. In this study, we propose a parallel implementation of EM for training GMM using CUDA. Experiments are performed with a UCI dataset and results show a speedup of 7 if compared to the sequential version. We have also carried out modifications to the code in order to provide better access to global memory and shared memory usage. We have achieved up to 56.4% of achieved occupancy, regardless the number of Gaussians considered in the set of experiments.

Keywords: Expectation-Maximization (EM), Gaussian Mixture Models (GMM), CUDA

1. INTRODUCTION

Machine Learning (ML) algorithms are often costly, since learning is a task that requires a large amount of knowledge and constant improvement of it, thus requiring massive data computation. A major problem of massive computing is the limitation of mainstream sequential processing in older computer architectures. Such limitation can be overcome using a parallel processing of data provided on newer architectures.

One of these recent architecture is the NVIDIA™ CUDA™ architecture, which is a framework for developing general programs source code and using the power of Graphical Processing Units (GPUs) to perform execution. It is possible to use the CUDA-C programming language, for instance, to provide a parallelized source code.

GPUs have high amount of internal multiprocessors, optimized for doing several Computer Graphics calculations in parallel.

The clear advantage of using GPUs is the small costs if compared to clusters or supercomputers and its processing power if compared to multi-core processors. Even the former NVIDIATM GeForce™ 8400 GS graphics card, for instance, is able to run up to 32 threads in parallel per clock cycle, under some restrictions.

The work on CUDA to provide parallelized implementations of important algorithms in different domains can be observed in recent scientific literature (Subbaraj and Sivakumar, 2012; Tharawadee *et al*., 2013; Meng *et al*., 2013; Mielikainen *et al*., 2012; Lee and Park, 2012).

Results show average performance gains of up to 30 times compared to processing the same problem using conventional CPUs. There are also efforts to further develop the readability of CUDA programs through the development of an Application Programming Interface (API) for C/C++ which automate the processing of sequential code into parallelized code (Santos and Macedo, 2012).

Corresponding Author: Araújo, G.F., Núcleo de Pós-Graduação em Ciência da Computação, UFS, São Cristóvão and Brasil

In this study we present the CUDA parallel implementation of the Expectation-Maximization algorithm for the estimation of Gaussian Mixture Models. The Gaussian mixture model is one of the most widely used statistical models for machine learning tasks, being the most flexible parametric model.

We are particularly interested in verifying whether a more efficient global memory access can reduce the concerned overhead, providing better usage of CUDA cores and, thus, improving performance.

1.1. Expectation-Maximization Algorithm (EM)

Statistical models are used in many machine learning techniques. The maximum likelihood method (Maximum- Likelihood Estimation, or just MLE) can estimate the parameters of a statistical model from a set of sample data, for further usage in classification tasks, for instance.

An important concern is what to do when some data sample are missing. It is yet possible to perform estimation of model parameters. The Expectation-Maximization (EM) allows learning of parameters that govern the distribution of the sample data with some missing features (Sujaritha and Annadurai, 2011; Poongothai and Sathiyabama, 2012; Malarvezhi and Kumar, 2013).

The MLE is defined as Equation 1:

$$\Theta_{ML} : \sum_{k} \frac{\partial \ln\left(p_y(y_k;\Theta)\right)}{\partial\Theta} \tag{1}$$

where, y represents the full set of sample data. To deal with missing data, tough, the EM can iteratively maximize the hope of the likelihood function, given the observed samples and the estimate of the current iteration Θ.

The EM algorithm consists of two steps.

The E-step computes the hope of logarithmic likelihood, conditionally to the set of observed data and the current value of the parameters, Θ^t Equation 2:

$$Q(\Theta;\Theta^t) \equiv E\left[\sum \ln\left(p_y\left(y_k;\Theta\middle|X;\Theta^t\right)\right)\right] \tag{2}$$

The M-step computes the (t+1)-th parameter vector Θ that maximizes Q (Θ; Θ^t), given by Equation 3:

$$\Theta^{t+1} : \frac{\partial Q(\Theta;\Theta^t)}{\partial\Theta} \tag{3}$$

The algorithm starts from a Θ(0) (usually defined arbitrarily, choice) and iterates through both steps until a stop criterion is satisfied. The widely used criterion is the variation of Q between steps, defined as Equation 4:

$$\left\|\Theta^{t+1} - \Theta^t\right\| \le \varepsilon \tag{4}$$

1.2. EM for GMM Estimation

Gaussian classifiers are the most widely used methods for supervised classification. However, these methods have limitations when dealing with problems where the classes cannot be linearly separable. Also, they cannot deal with non-Gaussian data, since their discriminant functions are linear or quadratic. A workaround for such limitation is to combine probability functions (pdf's). Indeed, this approach is widely used because it is a parametric method that can be applied to non-linear classification problems. Such technique is known as Finite Mixture Model and its probability function is defined as:

$$p(x) = \sum_{j=i}^{g} \pi_j \, p(x;\Theta_j) \tag{5}$$

where, g is the number of components (pdf) of the mixture; π_j is the probability of the components (commonly known as the weight of the component), such that 1 $\sum_{j=1}^{g} \pi_j$ and p (x; Θ_j) is the pdf of the component in regards to the parameters Θ_j.

When we use Gaussian models, each component assumes a multivariate normal distribution, where $\Theta_j = \{\mu; \Sigma_j\}$. This model is known as Gaussian Mixture Model (GMM) (Shanmugapriya and Nallusamy, 2014), (Ramalingam and Dhanalakshmi, 2014). Equation 5 cant thus be rewritten as Equation 6:

$$p(x) = \sum_{j=i}^{g} \pi_j N(x;\mu_j \sum_{j}) \tag{6}$$

But how to find the parameters that maximize the likelihood of the GMM? Typically, the parameters of the components of GMMs are estimated using the EM algorithm described in the previous section. For the GMM, the EM steps are defined as follows.

E-step: Calculate for each given i:

$$w_{ij} = \frac{\pi_j^t N\left(x_i ; \pi_j^t \sum_j^t\right)}{\sum_k \pi_k^t N\left(x_i ; \pi_k^t, \sum_k^t\right)} \quad (7)$$

where, π_j,μ_j and Σ_j are the weights, means and covariance matrices of component j at step t.

M-step: For each given j, update the parameters Equation 8 to 10:

$$\pi_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij} \quad (8)$$

$$\mu_j = \frac{\sum_{i=1}^{n} w_{ij} x_j}{\sum_{i=1}^{n} w_{ij}} \quad (9)$$

$$\sum_j \frac{1}{n\pi_j}\sum_{i=1}^{n} w_{ij}\left(x_i - \mu_j\right)\left(x_i - \mu_j\right)^T \quad (10)$$

As described above, the EM algorithm iterates until the convergence of the model likelihood (stopping criterion). It is possible, though, that the algorithm becomes stuck in a local minimum, leading to nonoptimal solutions. It is thus a common practice, repeat the training process few times more, initializing the parameters with different values and in the end, choose the best solution (Webb and Copsey, 2011). Moreover, both the calculation of w and the calculation of parameters π, μ and Σ iterate over all sample data.

For a large dataset, the time of the training process can be huge, especially in cases where there are high numbers of components. Despite such limitation, calculations performed for each data are independent and thus, fully parallelizable.

1.3. Previous Work on Parallel Implementation of EM and GMM Learning

Tagare *et al*. (2010), the authors present a strategy to speed up the EM algorithm using domain reduction. The approach considers the use of three different kernels to compute the calculation of latent probabilities and the Riemann sums for the parameter updates. The EM is used for reconstructing 3D volumes from noisy Electron Cryomicroscopy images of single macromolecular particles. The work focus on problems other than GMM.

Chen *et al*. (2012), the authors derive an algorithmic method for incremental GMM learning from a hypothesis-test and merging based algorithm. EM is not used. The most time-consuming part of the algorithm is accelerated by GPU. Davies-Bouldin index is used to measure the cluster quality of the algorithm.

Pham *et al*. (2010), the authors proposes a GPU implementation of the Extended GMM to Background Subtraction (BGS), which is used in various computer vision problems.

Pangborn (2010), the author presents a strategy to speed up the EM algorithm for clustering in single and multiple GPUs. The approach consists in breaking the Estep into two kernels and the M-step into three kernels. The work includes a parallel version of the cmeans algorithm.

Kumar *et al*. (2009), the authors spread the EM algorithm over six CUDA kernels for a fast parallel parametric estimation of GMM. The work focus is in speeding up the EM algorithm through improvements of the kernels and data organization, not using it for specific problems.

Azhari and Ergün (2011), the authors implements a CUDA version of the EM algorithm for speaker verification based on Gaussian Mixture Modeling-Universal Background Modeling (GMMUBM). The major difference between this and the previous presented works is that it uses only 2 kernels for the EM, one for the E-step and one for the M-Step. There is also a parallel implementation of the k-means algorithm.

Machlica *et al*. (2011), the authors present an implementation of the parallel EM algorithm for GMM training. According to the paper, their approach offers better memory occupancy and greater speedup due to less coalesced access. Their results were obtained using adapted data taken from 2008 NIST Speaker Recognition Evaluation.

2. MATERIALS AND METHODS

The method to provide parallelized implementation of EM for GMM is greatly founded on how to deal with the specificities of CUDA. In order to provide better understanding of the approach, firstly, we depict CUDA itself.

2.1. Technological Background: CUDA

Multiprocessors are responsible for GPU internal processing and there may exist many of them, varying according to graphics card's model. Every multiprocessor is composed by smaller processors, the so-called Core processors. These core processors share the same instruction chip, which belongs to the multiprocessor.

This means that CUDA™ architecture works as a Single Instruction Multiple Data (SIMD) system, where every multiprocessor is capable of processing only one instruction at a time. The basic parallel processing element is a thread, just like CPU, but there are two others important concepts: The block and the grid. A block is a composition of up to t threads, where t is the maximum value supported by the GPU. It is also the element seen in the multiprocessors, responsible for fully process all the threads of a block, when thus a new ready block is chosen. A grid, on the other hand, is an aggregation of multiple blocks.

Both the grid of blocks and the blocks of threads can be uni-, two- or three-dimensional. A kernel call needs to specify the dimensions of grid and blocks and thus, it is possible to run kernels with different arrangements of threads within the same application.

The memory hierarchy consists of *local memory*, *global memory* and *shared memory*. The local memory is a high speed memory and private to each thread. The shared memory is larger and slower than the local memory, but it is accessible by all threads of the same block, allowing threads to work collaboratively within a single run. The global memory is the largest and slowest memory of GPU, but it is accessible by any thread, thereby allowing different kernels to share common data.

2.2. Rationale of the Parallelization Approach

The calculation of w_{ij} in *E-step* (Equation 7) and the calculations of weights π_j, means μ_j and covariances Σ_j are extremely parallelizable as they iterate over all the data and are independent of each other.

An important point to be considered is the transfer of data from the host (main memory) to the GPU memory. The bus transfer between these two memories is slow and its usage should be avoided. As the algorithm must run iteratively in order to satisfy a stopping criterion and all steps are parallelizable, it would be more effective if the whole main loop of the algorithm could run on GPU, to avoid such data transfer. However, the arrangement of threads is statically defined in the kernel. This becomes an inconvenience, since the arrangement of threads is an important setting for a better efficiency of parallelization and each step of the EM algorithm requires a different arrangement.

Similarly to the approach of (Machlica *et al*., 2011) and (Kumar *et al*., 2009), in our proposal the main loop of the algorithm is implemented sequentially and different CUDA kernels are in charge of running different steps of the algorithm.

The implemented CUDA kernels are depicted as follows:

- p-kernel: For each Gaussian component, j computes the probability of each data xi conditional to parameters Θj, multiplied by the weight of component π_j. In this kernel, the thread blocks are arranged in a grid jmx, where m blocks of line j are responsible for the calculation for the component j
- ^p-kernel: For each data xi, normalizes their probabilities computed in the previous kernel for each component j. It concerns the w_{ij} values of Equation 7. In this step m, blocks of threads are used and each block is responsible for normalizing the probabilities for a given data at a time, until the entire probability base is normalized
- m-kernel: For each Gaussian, estimates its marginal probability, that is, calculates the sum of the probabilities of the data related to each component j. j blocks of threads are used and each block is responsible for performing the sum of a component μ-kernel: For each Gaussian, re-estimates the mean vector μ that maximizes the likelihood, as described in Equation 9. Again, using j blocks of threads, each block is responsible for a component
- Σ-kernel: Re-estimates the covariance matrices Σ of the components. In this step, we use an array of 2D blocks, where blocks of threads are organized in a square matrix of order N, where N is the dimension of the data. Thus, each block is responsible for reestimate an element s_{ij} of each of the covariance matrices
- π-kernel: Re-estimates the weights π of components. Since the weight of a given component is given by the marginal probability normalized, as described in Equation 8, this step contains only a single block of j threads, which perform the summation and normalization of the marginal probabilities
- φ-kernel: Calculates determinants and inverse matrices of covariance matrices, which are used to calculate the probabilities of p-kernel step. In this step, the matrix decomposition technique called LU decomposition is used. In such technique, we rewrite the matrix as the product of a lower triangular matrix (L matrix, lower) by an upper triangular matrix (matrix U, upper). Using j blocks of one thread only, the thread executes sequentially the LU decomposition algorithm

Figure 1 summarizes the distribution of component parameters and input data on the arrangement of blocks and threads of the grid.

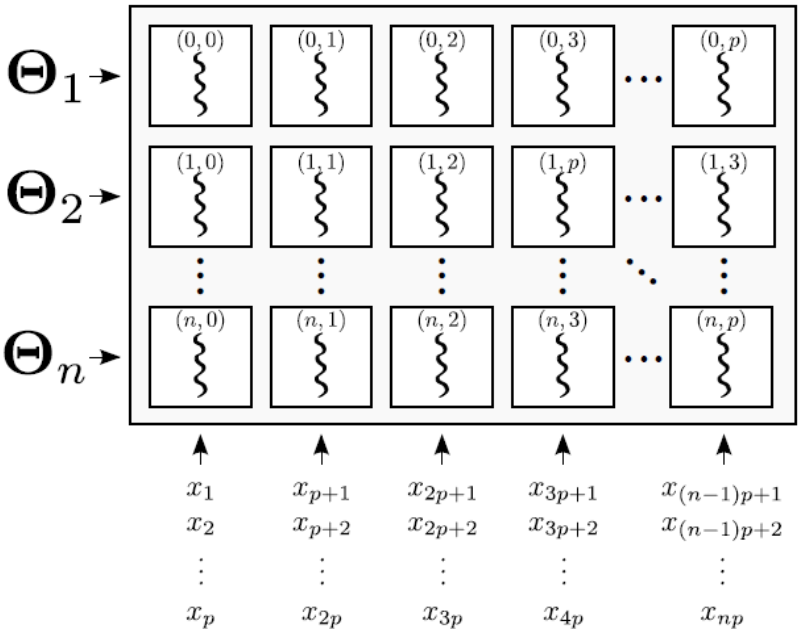

Fig. 1. Distribution of component parameters and input data on the grid

2.3. The Algorithm

The EM for GMM estimation algorithm takes as input the samples from the dataset, the number of gaussians to be estimated and the threshold as the stopping criterion. At each iteration, the algorithm initializes the parameters of each gaussian (weight π, mean μ and covariance matrix Σ). Next, for each sample, it estimates the likelihoods for each Gaussian and normalizes them. Finally, the parameters of the gaussian are re-estimated using the likelihood values. Iterations occur until the stopping criterion is satisfied (Algorithm 1). Each thread, one per block, estimates the likelihood of the sample j on Gaussian i, according to the position (i, j) of its block at the grid (Algorithm 2). The set of threads in a block performs the normalization likelihoods of values of a sample, using the reduction technique (Algorithm 3). The number of threads in a block performs the calculation of the probability of a marginal Gaussian (Algorithm 4). The number of threads in a block performs the reestimation of the parameter of a Gaussian mean μ (Algorithm 5). The number of threads in a block performs the reestimation of the parameter covariance matrix Σ of a gaussian (Algorithm 6).

Algorithm 1 EM for GMM's estimation

Input: samples, samplesnum, Gaussiannum: Thresholdmin
Output: π_I, μ_i, $\Sigma_j | i \in \{1,2\ldots., Gaussiannum\}$
for i←1, Gaussiannum do
    Initialize parameters (π_I, μ_i,Σ_i);
end for
while-stop condition () do
    for j ←1, Samplesnum do
        for i←1, Gaussiannum do
            $likelihood_{ij}$← Calculatelikelihood($Sample_j$, π_i, μ_i, Σ_i);
        end for
        $likelihood_j$←Normalize Likelihood ($likelihoods_j$);
    end for
    for i←1, $Gaussian_{num}$ do
        π_i ←UpdateWeight ($likelihood_j$);
        μ_i ←Update Mean ($likelihoods_j$, Samples, π_i);
        Σ_i ←Update Couariance($likelihoods_j$, Samples, π_i, μi);
    end for
end while

Algorithm 2 CUDA Parallel p-kernel

Input: Samples, $Samples_{num}$, π_i, μ_i, Σ_i
Output: Likelihoods
i←Block Index. Y;
j←Block Index. X;
$likelihood_{ij}$←$\pi_i \times N(samples_j, \mu_i, \Sigma_i)$;

Algorithm 3 CUDA Parallel $\hat{p}$ -kernel

Input: Liklelihoods, $Samples_{num}$
Output: Likelihoods
i←ThreadIndex. X;
j←BlockIndex. X;
cachei ←$likelihood_{ij}$;
SymchronizeThreads();
Limit ←ThreadsPerBlock/2;
While limit ≠ 0 do
If i<limit then
$cache_i$←$cache_j$+$cache_{i+limit}$;
end if
SymchronizeThreads();
limit←limit/2;
end while
$likelihood_{ij}$← $likelihood_{ij}$/$cache_0$;

Algorithm 4 CUDA Parallel m-kernel

Input: Liklelihoods, $Samples_{num}$
Output: Marginals
j←ThreadIndex. X;
i←BlockIndex. X;
cachei ←$likelihood_{ij}$;
SymchronizeThreads();
Limit ←ThreadsPerBlock/2;
While limit ≠ 0 do
If i< limit then
$cache_i$←$cache_j$+$cache_{j+limit}$;
end if
SymchronizeThreads();
limit←limit/2;
end while
marginal← $likelihood_{ij}$/$cache_0$;

Algorithm 5 CUDA Parallel μ-kernel

Input: Liklelihoods, Samples, $Samples_{num}$, marginals
Output: μ_i
j←ThreadIndex.X;
i←BlockIndex.X;
$cache_i$ ← $Sample_j$* $likelihood_{ij}$;
SymchronizeThreads();
Limit ← ThreadsPerBlock/2;
While limit ≠ 0 do
If i < limit then
Cache$_i$ ← $cache_j$+$cache_{j+limit}$;
end if
SymchronizeThreads();
limit←limit/2;
end while
μ_i← $cache_0$/$marginal_j$;

Algorithm 6 CUDA Parallel Σ-kernel

Input: Liklelihoods, Samples, Samplesnum, marginals, $Gaussian_{num}$, $Dimension_{num}$
Output: Σ_k
l←ThreadIndex.X;
i←BlockIndex.X;
j←BlockIndex.Y;
for k←1$Gaussian_{num}$ do
sub_1←$Sample_{11}$-μ_{ki};
sub_2←$Sample_{ij}$-μ_{kj};
$cache_1$←(sub_1*sub_2*$likelihood_{ki}$);
SymchronizeThreads();
limit ←ThreadsPerBlock/2;
while limit ≠ 0 do
If i< limit then
Cache$_l$←$cache_l$+$cache_{l+limit}$;
end if
SymchronizeThreads();
limit←limit/2;
end while
Σ_{kij}← $cache_0$/$marginal_k$;
End for

3. RESULTS

3.1. Dataset

We have used the dataset Arabic Spoken Digit 3 from UCI Repository in order to test the algorithm implementation. This dataset consists of instances with 13 Mel Frequency Cepstral Coefficients (MFCC), widely used to represent audio signals in speech processing systems, which commonly use GMMs to model the distribution of phones in the language. The database consists of 8800 instances: A training base with 6600 instances and a testing base with 2200 instances. These instances correspond to audios of 88 speakers (44 males and 44 females) pronouncing the digits 0 to 9 in Arabic.

3.2. Metrics

In parallel programming, Speedup (or Speed-up) is the most widely used metric to evaluate how much a parallel algorithm is faster than its sequential version. It its defined as Equation 11:

$$S_p = \frac{T_1}{T_p} \quad (11)$$

where, p is the number of processors on which the algorithm is running, T_1 is the execution time of the

algorithm and T_p is the execution time of the parallel algorithm.

3.3. Experimentation Sets

The GPU used in the first two sets of experiments was a NVidia GeForce GTS 250 with 16 processors.

In the first experimentation set, the target number of Gaussians (components) have been varied with a fixed quantity of 30 iterations to estimate parameters. **Figure 2** shows a comparison between the time of parallel execution and its respective sequential version. This execution has shown a Speedup $S_{16} = 7$. The runtime of the parallelized version varies from 0.781 to 15.094 sec.

This time results from the execution of φ-kernel step, which actually performs sequentially on the GPU. The second set of experiments was conducted varying the number of the instances in dataset: 23,344 to 263,256. In this case, the algorithms have performed 30 iterations to estimate two Gaussians. Results of this step are shown in **Fig. 3**. In this step, the Speedup of parallel algorithm was $S_{16} = 6$.

3.4. Coalesced Access to Global Memory

We have carried out modifications to the code of the kernels p-kernel and p-kernel to provide better access to global memory and shared memory usage. The kernels used in such arrangements have been rearranged according to the size of the warp in order to ensure aligned access with the “cache line” and, at the same time, maintaining all the CUDA cores employed as long as possible.

We have carried out modifications to the code in order to allow a more efficient memory access and a more appropriate array of threads. An important issue in regards to memory access is the coalesced access to the global memory, i.e., the warp threads need to access adjacent memory blocks and aligned with the "cache line" (fixed blocks of memory that are loaded once to the cache memory). In this way, the accesses required by the various warp threads are part of a single transaction, thereby reducing the overhead of the global memory access. The arrangement of threads in the grid and its blocks are also important for the coalesced access, since a better arrangement ensures that the accesses of the active warp threads are aligned on cache size.

In order to verify the performance gain achieved with the changes, tests with a NVidia GeForce GT 555M GPU have been performed. An important metric to consider when considering the arrangement and coalesced access is the *occupancy*, which refers to how effectively the hardware (the CUDA cores) is kept busy, i.e., the longer busy, best the hardware effective use.

Experiments have been performed by varying the number of Gaussians and size of the database, for both versions of the p and p kernel codes (E-Step). Firstly, we have fixed the database size of 23344 instances and varied the number of Gaussians. In the second set of experiments, we have varied the size of the database for a fixed number of Gaussians (eight). **Figure 4 and 5** show the execution time (in milliseconds) for both kernels by varying the number of Gaussians: 1, 2, 4, 8, 16, 24 and 32. **Figures 6 and 7** illustrate the runtimes of both kernels varying the size of the database. In such case, the average speedup was 19x and 30x for the kernels p and p, respectively. In both kernels, the largest observed speedups (~21x and ~32x) occurred with the database containing 152526 instances.

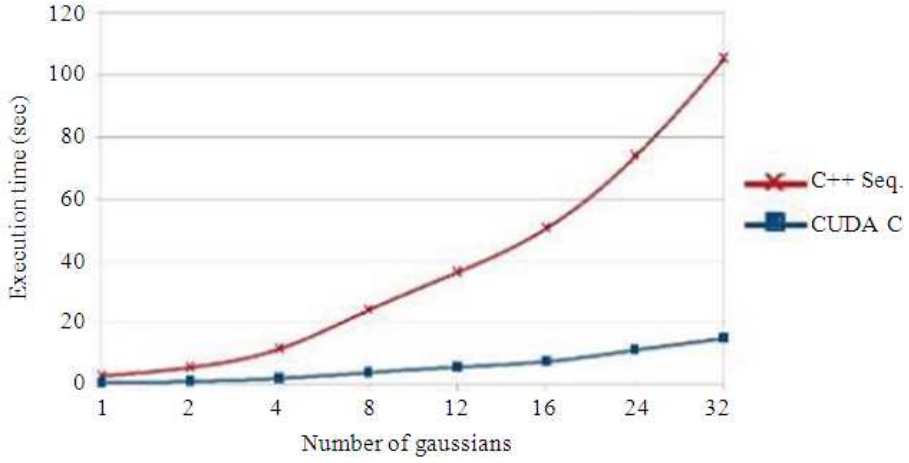

Fig. 2. Execution time for both parallel and sequential versions of EM as the number of gaussians increases

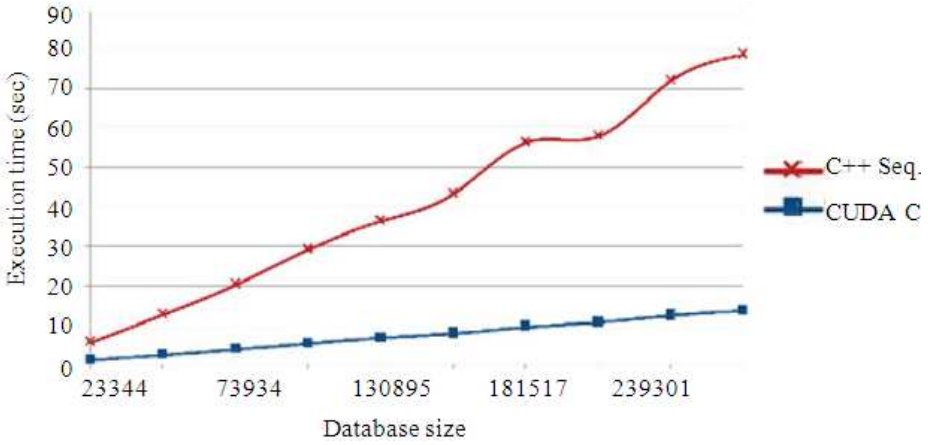

Fig. 3. Execution time for both parallel and sequential versions of EM as the number of instances of dataset varies

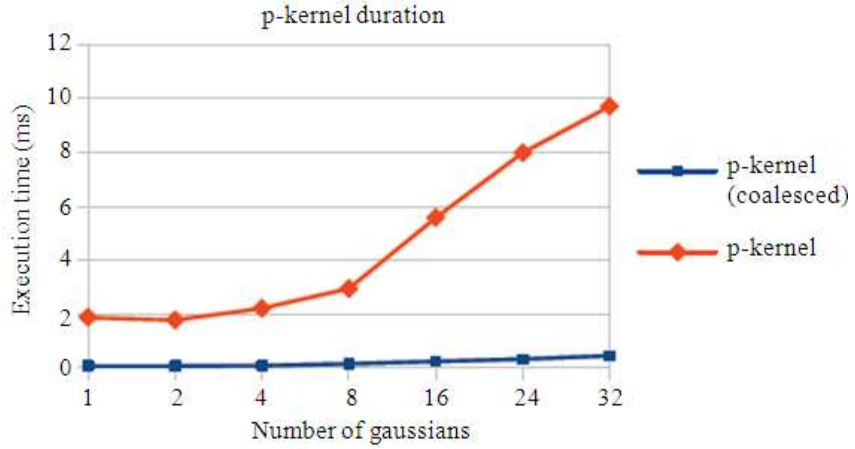

Fig. 4. Comparison of the execution time of the p-kernel with and without coalesced access by increasing the number of Gaussians

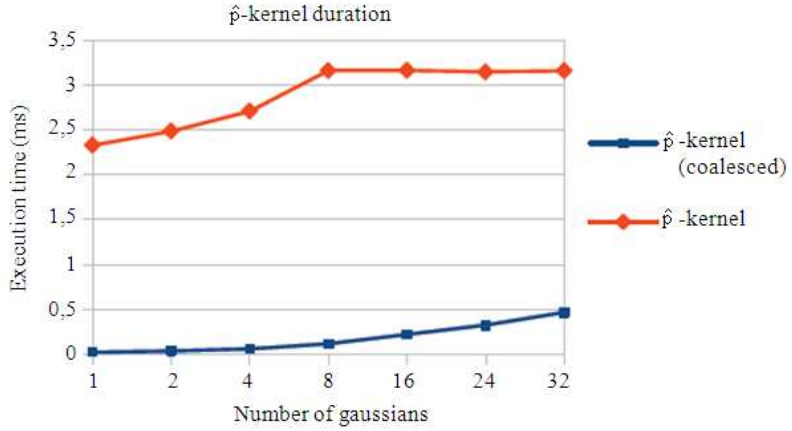

Fig. 5. Comparison of the runtime of the ^p-kernel with and without coalesced access by increasing the number of Gaussians

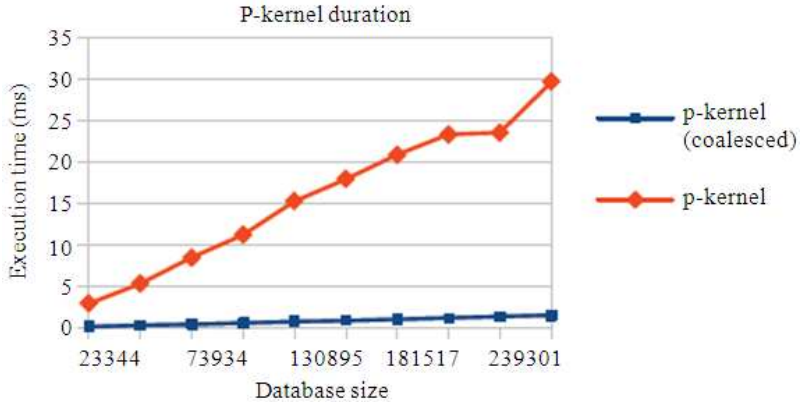

Fig. 6. Comparison of the execution time of the p-kernel with and without coalesced access by increasing the database size

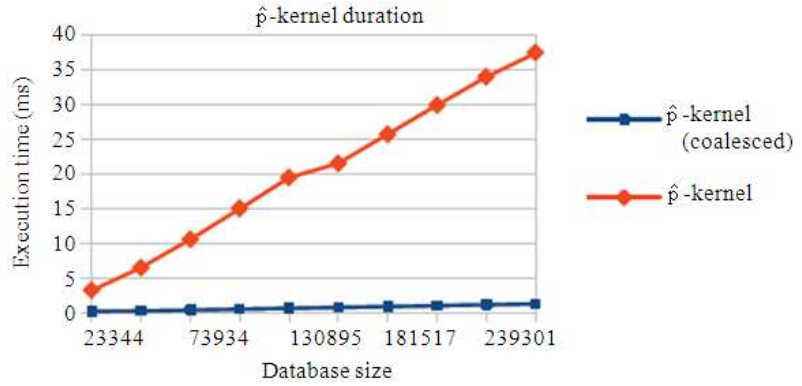

Fig. 7. Comparison of the runtime of the ˆp -kernel with and without coalesced access by increasing the database size

4. DISCUSSION

Results have shows an increase from 38.8 to 50.2% on the average achieved *occupancy* for pkernel, ranging from 16.6 to 60.1% with the growth in the number of Gaussians. In the case of p-kernel, the growth was 16.2 to 56.4% of achieved occupancy, regardless the number of Gaussians. Neither kernels have varied with the increase in the size of the database. As had been initially suspected, this increased *occupancy* directly reflects the execution time of kernels. This is clearly shown in **Fig. 4-7**.

The new version of p-kernel code, for instance, has achieved an average speed up of ~22x if compared to the previous one. In regards to the p-kernel, the average speedup is ~33x and the highest value has been observed for only one Gaussian.

These results show a clear performance gain when there is a greater control of hardware resources. Furthermore, it is important to notice that the execution settings and resource usage need to be adjustable to the capabilities of the available hardware, since the architectures of GPUs with support to CUDA have undergone changes over the years, including the cache memory management, which directly affects the transfer between the different available memories.

5. CONCLUSION

In this study, we propose an approach to provide a parallelized implementation of Expectation Maximization (EM) algorithm for training Gaussian Mixture Models (GMM). GMM is vastly used in Automatic Speech Recognition (ASR) systems, for instance.

In our approach for parallelization, the main loop of the algorithm is implemented sequentially and different CUDA kernels are in charge of running different steps of

the algorithm. Pseudocode for each CUDA kernel implementation is properly provided.

Experiments performed over a UCI database and varying number of Gaussians have shown a speedup of 7 if compared to sequential implementation of EM algorithm. We have also provided modifications in two of the CUDA kernels in order to allow more coalesced access to global memory. As a result we have achieved up to 56.4% of achieved occupancy.

The proposed approach thus contributes to the stateof-the-art of the research in ASR by providing an effective algorithm for training GMM.

Future work consists in providing modifications to the other three CUDA kernels in order to provide more coalesced access to global memory in a similar manner we have made to the first two of them.

6. ACKNOWLEDGEMENT

The researchers thank the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq-Brasil) for the financial support [Universal 14/2012, Processo 483437/2012-3] and for granting a scholarship to M.V.O. Medeiros. The authors also thank the Fundação de Apoio à Pesquisa e à Inovação Tecnológica do Estado de Sergipe (FAPITEC-Sergipe-Brasil) for granting a scholarship to G.F. Araújo.

7. REFERENCES

Azhari, M. and C. Ergün, 2011. Fast Universal Background Model (UBM) training on GPUs using Compute Unified Device Architecture (CUDA). Int. J. Elec. Comput. Sci., 11: 49-55.

Chen, C., D. Mu, H. Zhang and B. Hong, 2012. A GPUaccelerated approximate algorithm for incremental learning of gaussian mixture model. Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium, (WF‘12), pp: 1937-1943.

Kumar, N.S.L.P., S. Satoor and I. Buck, 2009. Fast parallel expectation maximization for Gaussian mixture models on GPUs Using CUDA. Proceedings of the 11th IEEE International Conference on High Performance Computing and Commu., (CC‘09), pp: 103-109.

Lee, S. and D. Park, 2012. Evaluation of CUDA for XRay Imaging System. In: Computational Intelligence and Intelligent Systems, Springer Berlin Heidelberg, pp: 621-625.

Machlica, L., J. Vanek and Z. Zajic, 2011. Fast estimation of gaussian mixture model parameters on GPU Using CUDA. Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies, (AT ‘11), pp: 167-172.

Malarvezhi P. and R. Kumar, 2013. A novel two stage carrier frequency off set estimation and compensation scheme in multiple input multiple output-orthogonal frequency division multiplexing system using expectation and maximization iteration. J. Comput. Sci., 9: 1526-1533. DOI: 10.3844/jcssp.2013.1526.1533.

Meng, C., L. Wang, Z. Cao, X. Ye and L. Feng, 2013. Acceleration of a High Order Finite-Difference WENO Scheme for Large-Scale Cosmological Simulations on GPU. Proceedings of the IEEE 27th International Symp. on Parallel and Distributed Processing Workshops and PhD Forum, (DPW’ 13), IEEE Computer Society, pp: 2071-2078

Mielikainen, J., B. Huang, H. Huang and M. Goldberg 2012. Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. IEEE J. Selected Top. Applied Earth Observ. Remote Sens., 5: 1256-1265.

Pangborn, A., 2010. Scalable data clustering using GPUs. MSc Thesis, Rochester Institute of Technology.

Pham, V., P. Vo and V. Hung, 2010. GPU implementation of extended gaussian mixture model for background subtraction. Proceedings of the International Conference on Computing and Communication Technology Research, Innovation and Vision for the Future, (VF’ 10), pp: 1-4.

Poongothai, K. and S. Sathiyabama, 2012. Efficient web usage miner using decisive induction rules. J. Comput. Sci., 8: 835-840. DOI: 10.3844/jcssp.2012.835.840.

Ramalingam, T. and P. Dhanalakshmi, 2014. Speech/ music classification using wavelet based feature extraction techniques. J. Comput. Sci., 10: 34-44. DOI: 10.3844/jcssp.2014.34.44.

Santos, B. and H. Macedo, 2012. Improving CUDA™ C/C++ encoding readability to foster parallel application development. Sigsoft Softw. Eng. Notes 37: 1-5. DOI: 10.1145/2088883.2088897

Shanmugapriya, N. and R. Nallusamy, 2014. A new content based image retrieval system using GMM and relevance feedback. J. Comput. Sci., 10: 330-340. DOI: 10.3844/jcssp.2014.330.340.

Subbaraj, P. and P. Sivakumar, 2012. Parallel memetic algorithm for VLSI circuit partitioning problem using graphical processing units. J. Comput. Sci., 8: 705-710. DOI: 10.3844/jcssp.2012.705.710,

Sujaritha, M. and S. Annadurai, 2011. A new modified gaussian mixture model for color-texture segmentation. J. Comput. Sci., 7: 279-283. DOI: 10.3844/jcssp.2011.279.283.

Tagare, H., A. Barthel and F. Sigworth, 2010. An adaptive expectation-maximization algorithm with GPU implementation for electron cryomicroscopy. J. Struct. Biol., 171: 256-265. DOI:10.1016/j.jsb.2010.06.004.

Tharawadee, N., P. Terdtoon and N. Kammuang-lue, 2013. An investigation of thermal characteristics of a sintered-wick heat pipe with double heat sources. Am. J. Applied Sci., 10: 1077-1086. DOI: 10.3844/ajassp.2013.1077.1086

Webb, A. and K. Copsey, 2011. Statistical Pattern Recognition. 3rd Edn., John Wiley and Sons, Chichester, ISBN-10: 1119952964, pp: 672.