A Systematic Method for Detecting Parallelized Software Bottlenecks and Suggesting Modifications: The Case of the Expectation Maximization Algorithm

Otávio Cordeiro Siqueira de Oliveira[1], Marco Túlio Chella[2], Hendrik Teixeira Macedo[2*], Leonardo Nogueira Matos[2]

[1] Universidade Tiradentes, Aracaju, Sergipe, Brazil.
[2] Universidade Federal de Sergipe, São Cristóvão, Sergipe, Brazil.

* Corresponding author. Tel.: (+55) (79) 31 94 66 78; email: hendrik@dcomp.ufs.br
Manuscript submitted March 13, 2017; accepted May 24, 2017.
doi: 10.17706/jsw.12.6.406-425

Abstract: Parallelized algorithms can distribute the workload on the available multi-core processors. Graphical Processing Units (GPU) began to be used in general purpose computing thanks to its ability to simultaneously perform thousands of operations in their parallel coprocessors. Unfortunately, providing parallelized versions of typical sequential routines is not a trivial task. Even with the advent of CUDA, the NVIDIA's more intuitive solution for GPU programming, developers need to acquire a deep knowledge of GPU architecture and the rationale of the target algorithms to optimize resources usage and reduce processing time. This paper proposes a systematic method for analyzing parallelized algorithms and propose guidelines for CUDA code refactoring in such a way faster and more efficient software, regarding hardware resources consumption, could be constructed. One of such kind of software is Automatic Speech Recognition (ASR) systems. Mainstream approaches for ASR use the Expectation Maximization (EM) algorithm to train Gaussian Mixture Models (GMM) to provide an Acoustic Model for ASR. These training phase is usually extensive time-consuming and so it's well suited for a parallelized solution approach. We show the feasibility of our method identifying important issues in a literature's parallelized implementation of EM and further refactoring suggestion to enhance memory occupancy and decrease processing time. The results show a processing speedup of the EM algorithm around 40x (minimum) and 61x (maximum) when compared to the control version. The method was also effective in the improvement of the values for all the concerned performance metrics for GPU-based solutions.

Key words: Expectation maximization, CUDA, parallelized code refactoring, software development process.

1. Introduction

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 which use the power of Graphical Processing Units (GPUs) to execute parallelized routines. The work on CUDA to provide parallelized implementations of important algorithms in different domains can be observed in recent scientific literature, indeed.

In general, GPU-based parallel computing provides application performance by transferring the intensive processing pieces to the GPU while the rest of the code remains running on the CPU. From the user's

perspective, applications run faster and the relatively low purchase price turns GPU platforms highly attractive.

The lack of knowledge of how the GPU really works, however, is an important issue since it can lead to impulsive purchases of equipment without a judicious analysis of its suitability to the primary goals (COOK, 2013). Another problem faced by developers in providing parallelized solutions is the absence of a well-defined method to optimize the use of GPU and CPU resources. The absence of such a method, *ad-hoc* initiatives, may hinder the learning process of inexperienced developers [1].

With the increasing demand for ever more efficient and flexible GPUs, their resources have been exploited by general purpose applications and thus, several applications with similar specificities have been identified and mapped to run on GPUs.

An important example of different attempts to parallelize their routines is the Expectation Maximization (EM) algorithm. The work of Medeiros et. al. [2] proposes a multi-kernel approach to the parallelization of EM for the estimation of Gaussian Mixture Models (GMM). In such case, the EM allows the learning of parameters that govern the distribution of the sample data with some missing features and has important application in the Automatic Speech Recognition (ASR). The work contributes to the state-of-the-art by proposing a coalesced access to CUDA global memory and providing multi dataset evaluation along four different metrics: *speedup, occupancy, number of executed instructions* and *number of global memory load transactions*. Unfortunately, the lack of a well-defined parallelization process turns further improvement attempts into a random walk on a huge hyper-dimensional space of variables.

This paper proposes a systematic method for the performance analysis of parallelized implementations based on CUDA. The method uses mainstream metrics and indicators to properly evaluate the solution and points out key parallelized code pieces that can be improved. The so-called MEPARALEL can thus assist the construction of parallelized implementations performed by developers of different expertise levels. In order to show its feasibility, MEPARALEL is applied to the parallelization work of Medeiros et al.

The rest of the paper is organized as follows. In Section 2, the basis of the EM algorithm for training GMM is summarized. Also in Section 2, the parallelized approach of Medeiros et. al. is presented. We depict the MEPARALEL method in Section 3 along with its application to performance analysis and recommendations for the work of Medeiros *et al*. Several performance testing results for each of the MEPARALEL's phases are shown. Finally, some concluding remarks are presented in Section 4.

2. Expectation Maximization for Training Gaussian Mixture

The general principle of EM algorithm is based on maximizing the likelihood function, given by the Eq. 1.

$$p(D|\theta)=\prod_{i=1}^{n} p(x_i|\theta) \tag{1}$$

where, D= $\{x_1,\ldots, x_n\}$ represents the data set, θ, the parameters of the probability density function that models its behavior and xi, an observation. The likelihood function, therefore, measure the level of alignment between the model and the data.

In general, the literature presents an alternative for obtaining the MLE, using the function:

$$l(\theta)=\ln p(D|\theta) \tag{2}$$

Since the logarithm is a monotonically crescent function inside the [0, ∞] interval, therefore the local

maximum location is not modified, and it allows representing the production in Equation (1) as a summation. By doing this, the MLE can be represented by the Eq. 3.

$$\hat{\theta} = \operatorname{argmax}_{\theta}^{l(\theta)} \tag{3}$$

The EM algorithm consists of two steps. The E-step computes conditional expectation of the logarithmic likelihood, conditionally to the set of observed data x and the current value of the parameters θ:

$$Q(\theta;\theta)^{i} = E\left[\ln p(D;\theta)|D;\theta^{i}\right] \tag{4}$$

The M-step computes the (i+1)-th parameter vector θ that maximizes Q(θt+1, θ), given by:

$$\theta^{i+1} = \operatorname{argmax}_{\theta}^{Q(\theta;\theta^{i})} \tag{5}$$

The algorithm starts from a θ(0) (usually defined arbitrarily, although there are approaches to optimize this 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:

$$\left\|Q^{t+1}\text{-}Q^{t}\right\| \leq \varepsilon \tag{6}$$

letting i be an iteration counter, and ε a preset convergence criterion.

Gaussian classifiers are one of the Hidden Markov Models applied to speech recognition. However, these methods have limitations when dealing with non-Gaussian data, since their discriminant functions are linear or quadratic. A workaround for such limitation is to combine probability distribution functions (pdf). 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 distribution function for a random variable is defined as:

$$p(y;\theta) = \sum_{k=1}^{g} p(y;\theta_k) \tag{7}$$

Satisfying

$$\Pi_k \geq 0 \quad \text{with} \quad k = 1...g \quad \text{and} \quad \sum_{k=1}^{g} \pi_k = 1 \tag{8}$$

where, g is the number of components (pdf) of the mixture; π is the probability of the components (commonly known as mixing probabilities), such that p(;θj) is the pdf of the component in regards to the parameters θj.

When we use Gaussian models, each component assumes a multivariate normal distribution, where θj = {μj; ∑j}. This model is known as Gaussian Mixture Model (GMM) [3], [4]. Eq. 5. can thus be rewritten as:

$$p(y;\mu,\Sigma) = \sum_{k=1}^{g} \pi_k N\left(y,\mu_k;\Sigma_k\right) \tag{9}$$

where ∑ is the covariance matrix and μ represents the mean of Gaussians.

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:

$$w_{ij} = \frac{\pi_j N(x_i; \pi_i; \Sigma_j)}{\sum_{k=1}^{n} N(x_k; \pi_k; \Sigma_k)} \tag{10}$$

where, πj, μj and ∑j are the weights, means and covariance matrices of component j at step t.

- M-step: for each given j, update the parameters:

$$\pi_j = \frac{1}{n} \sum_{i=1}^{n} w_{ij} \tag{11}$$

$$\mu_j = \frac{\Sigma_{i=1}^{n} w_{ij} x_j}{\sum_{i=1}^{n} w_{ij}} \tag{12}$$

$$\sum j = \frac{\Sigma_{i=1}^{n} w_{ij} (x_i - \mu_j)(x_i - \mu_j)^T}{n \Sigma_{i=1}^{n} \omega_{ij}} \tag{13}$$

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 non-optimal solutions. Repeating the training process few times more, initializing the parameters with different values and in the end, choosing the best-of-all solution is thus a common practice [5].

2.1. Parallelization Approach of Medeiros *et al.* [2]

The calculation of w_{ij} in E-step (Eq. 10.) and the calculations of weights πj, means μj and covariance ∑j are highly 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 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.

The main loop of the algorithm is implemented sequentially and different CUDA kernels concern different steps of the algorithm:

- p-kernel: For each Gaussian component j, computes the probability of each data xi conditional to parameters θj, multiplied by the weight of component πj. In this kernel, the thread blocks are

arranged in a j x m grid, 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 wij values of Eq. 10. 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;
- μ-kernel: For each Gaussian, re-estimates the mean vector μ that maximizes the likelihood, as described in Equation (1.9). Again, using j blocks of threads, each block is responsible for a component;
- Σ-kernel: re-estimates the covariance matrices ∑ 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 re-estimate an element δij 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 Eq. 11., this step contains only a single block of j threads, which perform the summation and normalization of the marginal probabilities.

Details on the parallelized algorithms can be found in the original paper of the authors.

3. Meparalel

In order to assist the developers, the method has been segmented in 6 (six) phases. This allows for an incremental analysis of the algorithmic solution. The CUDA toolkit provides a helper analytics toolset, which includes the *Visual Profiler*, *NvProf* and *HUD Launcher 4.1*. By default, these tools collect profile data throughout application runtime. Typically, though, the human analyst focuses at only one region of the application. This limitation is a way of reducing the amount of data being analyzed so it becomes easier to identify application bottlenecks and at the same time most attention is given to the code piece where optimization can indeed lead to greater performance gains. Such technique is called *profiling*.

There are several common situations where performing the *profiling* of an application region holds, for example when the application operates on a large number of iterations but the algorithm performance does not vary significantly between them.

Considering the *profiling* advantages presented so far, Fig. 1. summarizes the MEPARALEL flowchart and following subsections will detail each of the method phases along with correspondent application to the problem of Medeiros' EM implementation.

3.1. Applying Meparalel on the Parallelized Solution of Medeiros *et al.*

3.1.1. Phase 1 (configuration and execution of GPU-based algorithm)

First task is to evaluate whether the solution to be analyzed can be parallelized. The EM algorithm iterates until the convergence of the probability model (stop criterion). Steps "E" and "M" are repeated until the maximization process produces no significant improvement. For a large dataset, training processing time can be huge, especially in cases where there are many components. Despite such limitation, calculations for each of the data are independent and, so, fully parallelized [6].

Next task is to configure the algorithm so that it can efficiently consume the GPU resources. From now on, we will refer to the original version of the implementation of Medeiros et al. as the *Control* version. Further modified versions, subject to various experimentations, will be called the *Experimentation* versions. To ensure fairness of time measurements, each coding version has been executed a hundred times. Average time has been recorded. The runtime *Control* version took an average of 14.27 milliseconds (ms).

CUDA GPU Occupancy Calculator tool was used to assist the task of maximizing GPU resources. First step is to identify the CUDA GPU's capability of the target machine. Then, the kernels configuration of the *Control* version must be identified, so efficiency of resources usage could be measured.

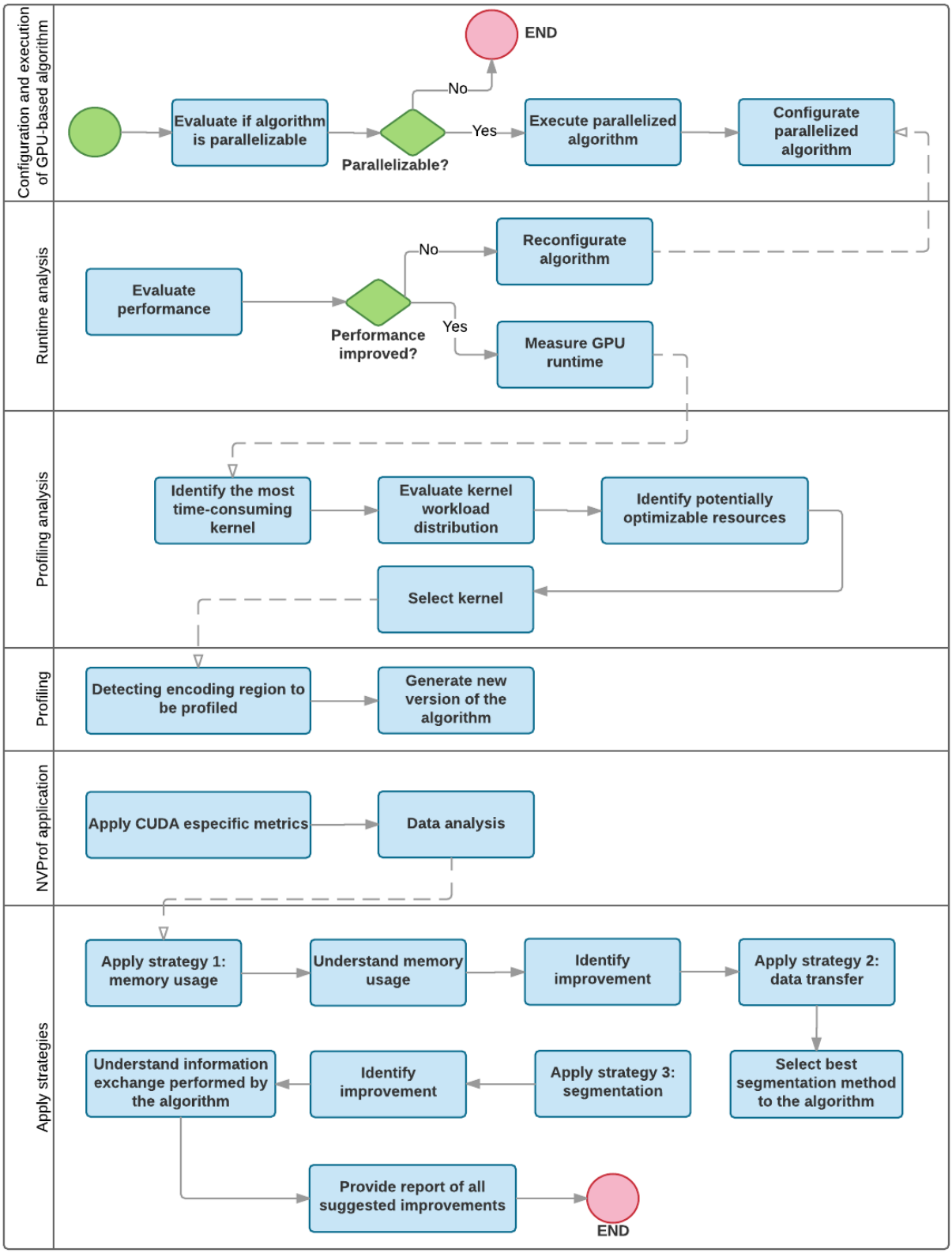


Fig. 1. Meparalel flowchart.

Although *Control* version use *dim3* variables, that works like 3-dimensional vectors passed in kernel execution directives to maximize execution optimization, the parameter configuration did not reach the physical limits established by the computer Capability of the GPU, that are shown in Table 1.

Table 1. Algorithm Configurations

Feature (GPU NVIDIA GFORCE 210)	Values
Thread per Warp	32
Warp per Multiprocessor	32
Threads per Multiprocessor	1024
Blocks of Thread per Multiprocessor	8

Control version used following configuration: 512 threads per blocks, 32 records per Threads and 4096 as the available space in bytes of shared memory per block for all the kernels. Such configuration hadn't good results. CUDA occupancy of each multiprocessor was only 50%. The maximum number of active threads block per multiprocessor was 1 (one) and the maximum number of active warp per multiprocessor was 16 (sixteen), so there is 1 Block / SM * 16 Warps / Block = 16 Warps / SM. This means that, while the limit of the GPU is 32, the number of registers per multiprocessor in the same warp in the Control version is 16 (the red triangle on the graph of Fig. 2).

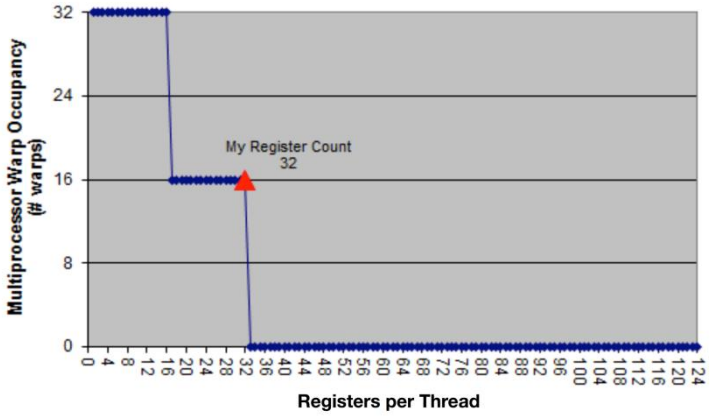


Fig. 2. Impact of warp per thread in the control version.

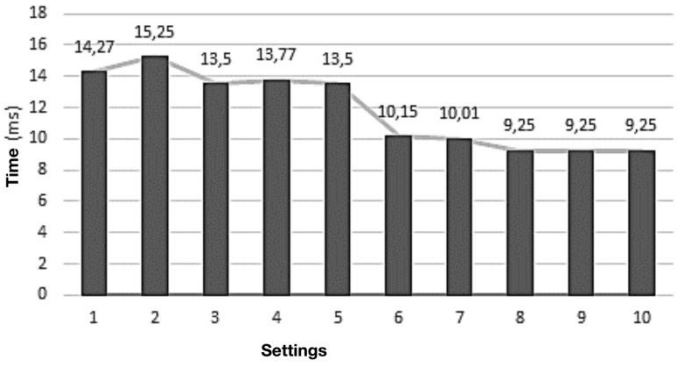


Fig. 3. Runtime of different settings that were empirically tried. setting 1 corresponds to control version.

The best suited configuration to GPU features has been achieved empirically, as shown in Fig. 3. "Setting 10" has been selected as the winner configuration. With that configuration, *Experimentation* version achieved an average runtime of 9,25 ms, which means a *speedup* gain around 37x if compared to "Configuration 1", *Control* version configuration. It has 256 threads per block and 16 registers per thread and the available size in bytes for shared memory per block has been set to 1024 bytes (respecting the

CUDA occupancy of the GPU). In such configuration, the maximum number of active multiprocessor thread blocks is 4 (four), and the maximum number of active multiprocessor warp was 8 (eight). Thus, there are 4 Blocks / SM * 8 Warps / Blocks = 32 Warps / SM. This means that the number of registers per multiprocessor in the same warp is 32. Since the GPU limit is also 32, a CUDA occupancy of 100% in each multiprocessor was achieved (see the red triangle in the graph of Fig. 4).

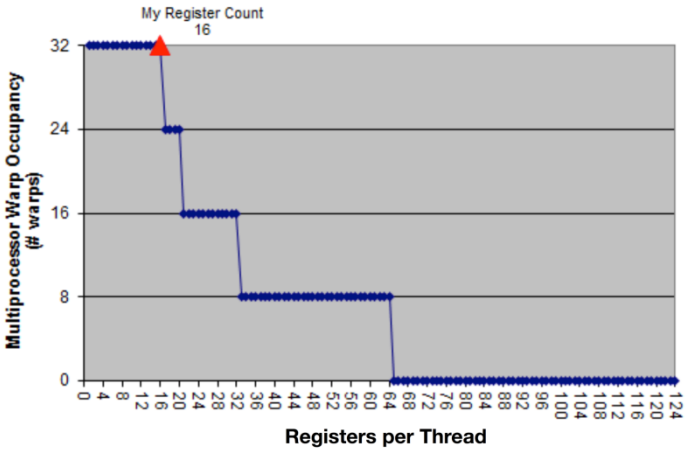


Fig. 4. Impact of warp per thread in the Experimentation version.

3.1.2. Phase 2 (runtime analysis)

The goal of this phase is to analyze execution time of *Experimentation* versions. Five datasets have been used. For each one of them, the instances with 13 Mel Frequency Cepstral Coefficients (MFCC), widely used to represent audio signals in speech processing systems and that commonly use GMMs to model the distribution of phones in the language [7] were extracted. The Arabic Spoken Digit 9 form UCI Repository [8] 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. The An4 database [9] consists of 948 training instances and 130 test instances. Each speaker was asked to spell out personal information, such as name, address etc. The CMULM Chaplain [10]: spoken dialog database comprises 4.15 hours of speech, recorded with close-talking microphone. From the CMU PDA Database [11], we have chosen the PDAm dataset. In this dataset, voice has been recorded by multiple microphones mounted around a PDA. It was used 11 speakers, each one reading about 50 sentences, resulting in 950 MB of WAV audio files. The CMU SIN Database [12], a speech in noise database, contains 500 sentences taken from the CMU ARCTIC recorded from one male US English speaker.

In order to provide a performance comparison between the two versions of the algorithm (*Control* and *Experimentation*), the present work considered the CMU PDA database with 8 Gaussian and 22334 instances. Table 2 shows the dataset setting that will be applied to the *Experimentation* version along the next steps of the MEPARALEL method.

Table 2. The Dataset Setting for the Experimentation Version

Dataset setting	Value
Dataset size (number of represented records)	23,344
Number of gaussians	8
Used setting	“Setting 10”

Command "time" available in "C" language has been used for time recording. Memory access time has been recorded with the NVIDIA INSIGHT tool which delimits the time consumed by each operation.

As shown in Table 3, although the total GPU execution time has been dropped to 9.25ms after the adjustment of the first phase, in this new kernels' execution, an intense data movement has been noticed (a time counter was triggered whenever a data movement operation was requested).

Table 3. Results for the Execution of the Experimentation Version after the Optimization of First Phase

Measures	Time (ms)
Vector filling time at CPU	0,74
Kernels execution at GPU	6,37
Copy of data from CPU to GPU	1,93
Copy of data from GPU to CPU	0,21
Complete execution	9,25

Results show that data exchange between CPU and GPU consumes 2,14ms, i. e., 23,13% of total execution time. Kernels execution time on GPU actually correspond to 68,86% of the time (Fig. 5).

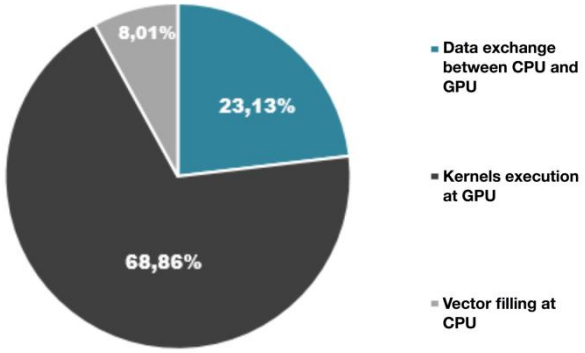


Fig. 5. Percentage of runtime per operation in the experimentation version.

3.1.3. Phase 3 (profiling analysis)

At this phase, same *Experimentation* version of both previous phase has been submitted to profiling analysis in order to evaluate the processing time of each kernel individually.

Control version implements the following six different CUDA kernels.

- p-kernel: for each Gaussian component j, it computes the probability of each data x_i conditional to parameters θ_j, multiplied by the weight of component π_j. In this kernel, the thread blocks are arranged in a j x m grid, where m blocks of line j are responsible for the calculation for the component j;
- ^p-kernel: For each data x_i, it normalizes their probabilities computed in the previous kernel for each component j. It concerns the w_{ij} values of Eq. 10. 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;
- μ-kernel: for each Gaussian, it re-estimates the mean vector μ that maximizes the likelihood, as described in Eq. 9. Again, j blocks of threads are used and each block is responsible for a component;
- Σ-kernel: it re-estimates the covariance matrices ∑ of the components. An array of 2D blocks is used, where blocks of threads are organized in a square matrix of order N, the dimension of data. Thus, each block is responsible for re-estimating an element δ_{ij} of each of the covariance matrices
- π-kernel: it re-estimates the weights π of components. Since the weight of a given component is given by the marginal probability normalized, as described in Eq. 11, this kernel contains only a single block of j threads, which perform the summation and normalization of the marginal

probabilities; and

- Φ-kernel: it computes the determinant and the inverse of the covariance matrix, which are used to calculate the probabilities represented by the "p-kernel". The LU technique for decomposing the array is used. In LU, we rewrite the matrix as the product of a lower triangular base (L matrix, lower) in an upper triangular matrix (U matrix, upper). Using J blocks from a thread, the thread sequentially executes the LU decomposition algorithm.

The results for the execution of each kernel are shown in Table 4.

Table 4. Runtime of Each Kernel

Kernel	% total runtime	Duration (ms)	Number of calls
Σ-kernel	91,6%	5,839	1
p-kernel	0,2%	0,012	1
^P-kernel	0,2%	0,012	1
μ-kernel	6%	0,408	1
π-kernel	0,2%	0,012	1
Φ-kernel	1,2%	0,076	1

Due to the observed discrepancy on the runtime contribution of a particular kernel, a further investigation was undertaken in regard to the delegated workload for each kernel. Because of the investigation, it was identified that the workload delegated the kernels is efficiently adjusted and the division of labor aims to meet each step defined by the mathematical model of EM and GMM previously presented. Given this, next task of this phase focused on to identify resources that can be optimized. Two different indicators have been analyzed in the profiling tool (Table 5).

Table 5. Gathering of Memory Access Time

Operation	Duration	Size	Average throughput
Mencpy HtoD	1,93 ms	2,50 MB	1,422 GB/s
Mencpy DtoH	0,21ms	2,60 MB	620,805 MB/s

Since it was noticed that the duration of the Mencpy DtoH operation is quite lower than the Mencpy HtoD one, optimization of Mencpy HtoD operation was prioritized. The usage percentage of each kernel in the Mencpy HtoD operation was further analyzed, as shown in the graph of Fig. 6. The graph confirms the high correlation between Mencpy HtoD operation and the total amount of processing time of a kernel.

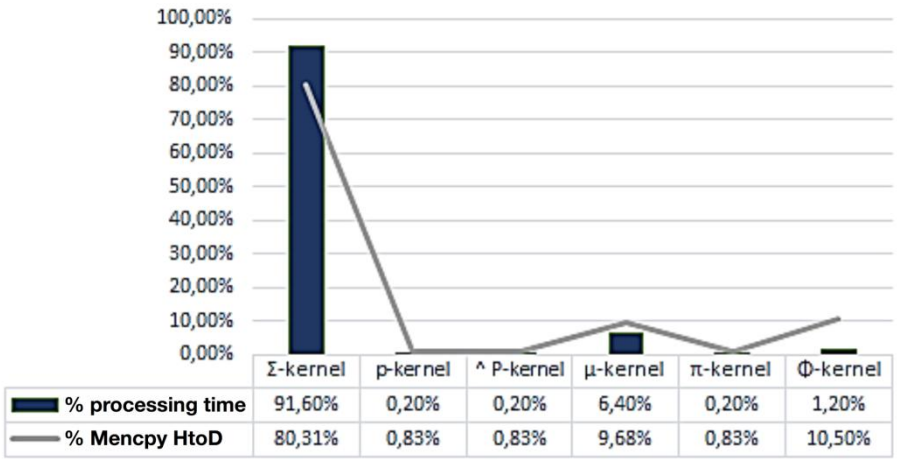


Fig. 6. Data exchange and processing time of kernels.

3.1.4. Phase 4 (profiling)

Once the high consuming kernel was identified, kernel profiling was started by selecting the excerpt of the algorithm which concerns to the Σ-kernel: *CUDAProfilerStart()* and *CUDAProfilerStop()* functions have been used for this purpose. In addition to being responsible for consuming 91.6% of the total time, the Σ-kernel has also high demanding data movement, Mencpy HtoD, as shown Table 6. Throughput analysis regarding Mencpy HtoD operation presented an important warning: communication channel was often busy for many data transfer requests.

Table 6. Time and Memory Usage of the Profiled Kernel

Operation	Start time	Duration	Size	Average throughput
Mencpy HtoD	2ms	1,55 ms	1,72 MB	1,793 GB/s

3.1.5. Phase 5 (nvprof application)

In order to explore and validate previous analysis, *Experimentation* version was undergone to performance evaluation metrics defined by MEPARALEL with the NVProf tool. Results are shown in Table 7.

The *Branch Efficiency* has an upper acceptable limit of 100%. Kernel execution achieved an average of 122% for the metric, which indicates that threads have been idle due to the delay in delivering data to be processed.

Global Memory Load Efficiency also has an upper acceptable limit of 100%. A result of 200% for this indicates that for an important percentage of the execution time, the algorithm has been idle waiting to read the data and store it in the global memory.

In regard to the metrics *Requested Global Load Throughput* and *Requested Global Store Throughput*, although they do not have an optimal threshold, results should be taken the basis for a resource optimization.

Table 7. Results for Meparalel Metrics

Metric	Average value
Branch Efficiency	122%
Global Memory Load Efficiency	200%
Global Memory Store Efficiency	200%
Requested Global Load Throughput	4.181GB/s
Requested Global Store Throughput	2.561GB

3.1.6. Phase 6 (apply strategies for data optimization)

At this phase, three data optimization strategies based on Shane Cook [13] are applied: (1) memory usage, (2) data transfer and (3) segmentation.

Memory usage

Although the execution of Σ-kernel makes use of the cache memory, it often interrupts its execution. This could be noticed because of a *for* statement that search for a new memory location so it can re-evaluate elements of the covariance matrix. If at any steps of the algorithm execution, the GPU did not access consecutive memory addresses, it was necessary to search for a new memory address and thus to continue the execution of the kernel. This search for new addresses causes a rapid decrease in bandwidth usage.

It is interesting to note that although "Setting 10" was applied in the first step of this experiment, which provided a 100% CUDA occupancy, it was not successful regarding memory usage, which reiterates the need for a more detailed analysis of the consumed resources by each kernel of the algorithm. Such analysis greatly influences the choice of the segmentation technique to be applied by the third MEPARALEL

optimization technique.

Data transfer

In this step, it was analyzed how the Σ-kernel obtains the data to perform its operations and how it updates the information in memory after performing the re-estimation of the covariance matrix. Despite the high-speed of the cache memory, readings and writes in the different types of memory must be carried out in an efficient way so that the best possible performance is obtained with GPUs.

The kernel for data readings and writing performs data transferring in a traditional manner: a data vector is transferred from the CPU memory to the GPU memory via the PCI Express bus. When the transfer ends, a kernel is invoked to perform operations on this data. At the end of the kernel run, data is copied back from the CPU to the GPU.

During kernel execution, the first step is to copy the data that has already been estimated in a previous step of the algorithm. This copy is inside a *for* statement structure that traverses and re-estimates all the items in the vector.

In order to optimize the data transfer, at each iteration loop of the *for* statement, data will be copied asynchronously to the final vector. As a consequence, smaller copies of data will be made and possibly we will obtain a time reduction in the copy of these data. The change in data copying turns it possible to employ multiple processing streams and as the data reaches the GPU memory, they can be processed and copied back to CPU asynchronously.

After identifying the possibilities for improvement, the last task of the 6th MEPARALEL phase was performed, which is to analyze the possibility of applying a segmentation technique.

Segmentation

One of the operations performed by the kernel is to sum all the probabilities computed from the data. The algorithm needs to traverse all the estimated probabilities and then store the probability in the initial position of the cache:

```
while (caheIdx < j) {
        cache[cahceIdx] += cache[cacheIdx + j]
}
```

The segmentation technique fits perfectly the operation, since it is a class of parallel algorithms that pass over an O(N) data input and generates an O(1) result computed with a "+" associative binary operator. Unless the "+" operation is extremely expensive to evaluate, reduction tends to optimize hardware resources and occupy the entire band [14]. This technique can solve the problems identified by the experiment: providying full occupation of band limits without interrupting the execution of the algorithm.

A reduction algorithm can extract a representing value from a matrix of values. This isolated value can be the sum, for instance. A reduction can be achieved by sequentially traversing each element of the array. When an element is visited, the action to be taken depends on the datatype. In our case, by reducing the sum, the value of the visited element in the current step is added to a cumulative sum as shown in Fig. 7.

Fig. 7 shows the execution of a reduced kernel. Threads and elements of the array are on the columns and the contents of the array at the end of the iterations are on the lines. Time proceeds from top to bottom. As can be see, array positions store partial sums of preceding pairs after each iteration. Following code snippet replaces previous one to assure sum reduction: each thread block has 2 * BlockDim of the vector input elements so the thread will load 2 elements into the shared memory.

```
__shared__ float partialSum[2*BLOCK_SIZE];
unsigned int t = threadIdx.x;
Unsigned int start =
2*blockIdx.x*blockDim.x;
```

```
partialSum[t] = input[start + t];
partialSum[blockDim + t] = input[start + blockDim.x + t];
```

Each of the three optimization strategies of 6th phase of MEPARALEL were applied.

We have firstly analyzed the impact of the 2nd strategy, data transfer, in the *Experimentation* version. Tables 8 and 9 show a reduction of 18% in the Σ-kernel processing time (if compared to Table 4) and of 67% for the Mencpy HtoD operation.

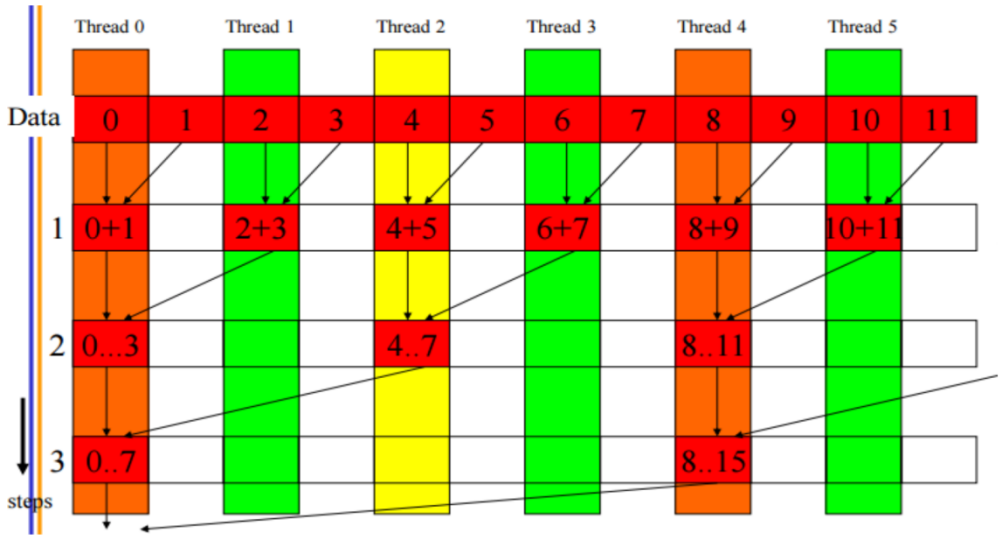


Fig. 7. Sum reduction (adapted from KIRK *et al*. (2012)).

Table 8. Runtime of Σ-kernel after 2nd Optimization Strategy

Kernel	% of execution time	Duration (ms)	Number of calls
Σ-kernel	91,6%	4,739 ms	1

Table 9. Data Movement of Σ-kernel after 2nd Optimization Strategy

Operation	Start time	Duration	Size	Average throughput
Mencpy HtoD	2ms	0,5 ms	1,72 MB	1,793 GB/s

Total runtime of *Experimentation* version decreased from 9,25ms to 8,15ms, which means a speedup of ~11.89x. If compared to *Control* version, this optimization strategy has achieved a speedup of ~42.88x, since total runtime has decreased from 14,27ms to 8,15ms.

Table 10, however, shows that despite of the considerable gain in processing time and even though the kernel has improved the metric consumption values, it has not yet been able to reach the optimal values delimited by the metrics.

Table 10. Results for Meparalel Metrics after 2nd Optimization Strategy

Metric	Average value
Branch Efficiency	107.00%
Global Memory Load Efficiency	113.00%
Global Memory Store Efficiency	113.00%
Requested Global Load Throughput	4.181GB/s
Requested Global Store Throughput	2.561GB/s

Optimization strategies number 1 and 3 were thus applied on Σ-kernel. Table 11 shows the results. It is possible to note a great improvement on the processing time, a decrease of 26% if compared to same

Experimentation version soon after 2nd optimization (see Table 8).

Table 11. Runtime of Σ-kernel after 1st and 3rd Optimization Strategies

Kernel	% of execution time	Duration (ms)	Number of calls
Σ-kernel	91,6%	3,511 ms	1

Table 11, however, shows that despite of the considerable gain in processing time and even though the kernel has improved the metric consumption values, it has not yet been able to reach the optimal values delimited by the metrics.

Experimentation version was once again undergone NVProf in order to evaluate performance metrics and the results are shown in Table 12. After optimization strategies 1 and 3, Σ-kernel could use more efficiently GPU resources.

Table 12. Results for Meparalel Metrics after 1st and 3rd Optimization Strategies

Metric	Average value
Branch Efficiency	99,70%
Global Memory Load Efficiency	100%
Global Memory Store Efficiency	100%
Requested Global Load Throughput	4.581GB/s
Requested Global Store Throughput	2.541GB

Table 13. Performance Behavior of Σ-kernel with the Increasing Number of Gaussians

Number of Gaussians	*Control* version (Σ-kernel) (ms)	*Experimentation* version (Σ-kernel) (ms)
8	5,839	3,511
32	5,911	3,516
64	5,981	3,514
128	6,201	3,517
256	6,279	5,518
512	6,338	5,522
1024	Error	5,523

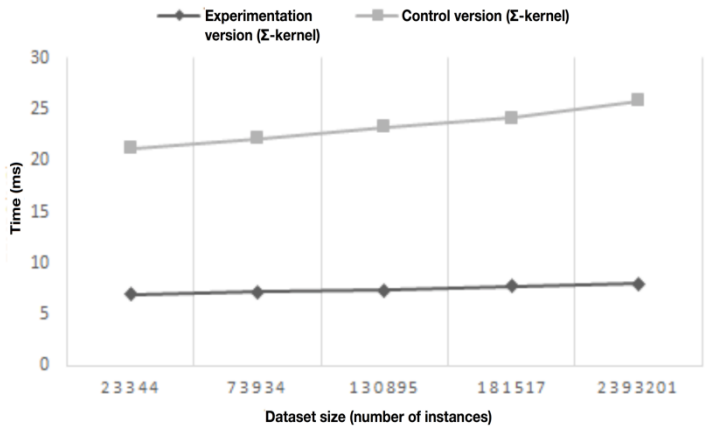


Fig. 8. Performance behavior of Σ-kernel with the increasing number of instances and fixed 8 Gaussians.

Total runtime of *Experimentation* version decreased from 8,15 ms to 6,5 ms, a speedup of ~14.72x in comparison to *Control* version.

Further performance analysis for Σ-kernel aimed at observing the runtime behavior with the increasing number of Gaussians and the size of the dataset. Table 13 shows that the algorithm seems to behave with time complexity order close to O(1) in regards to the number of Gaussians, with attention to the fact that 1024 Gaussians could not be reached in the *Control* version.

The graph of Fig. 8 shows that the *Experimentation* version achieved a speedup of ~51x to ~55x in respect to the number of instances to be processed. Results consider a fixed number of 8 (eight) Gaussians.

Next, we set the number of instances to 22,334 and observed the behavior of different datasets with the number of Gaussians. The dataset doesn't seem to interfere on *Experimentation* version results (Fig. 9). Control version, on the other hand, seems to be much more sensitive (Fig. 10).

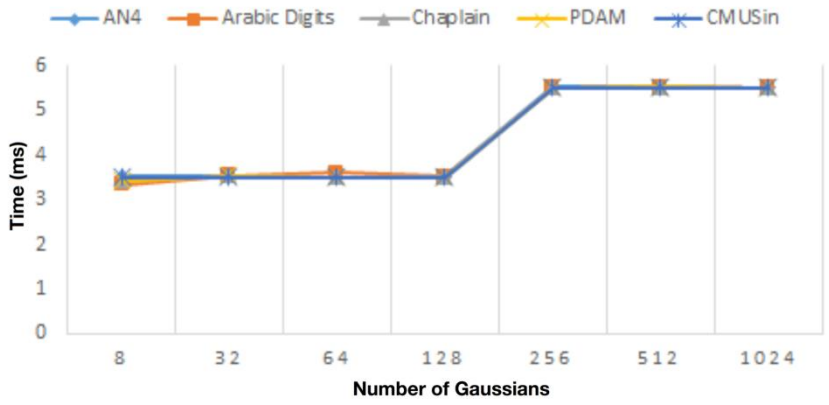


Fig. 9. Experimentation version for varying Gaussians and different datasets.

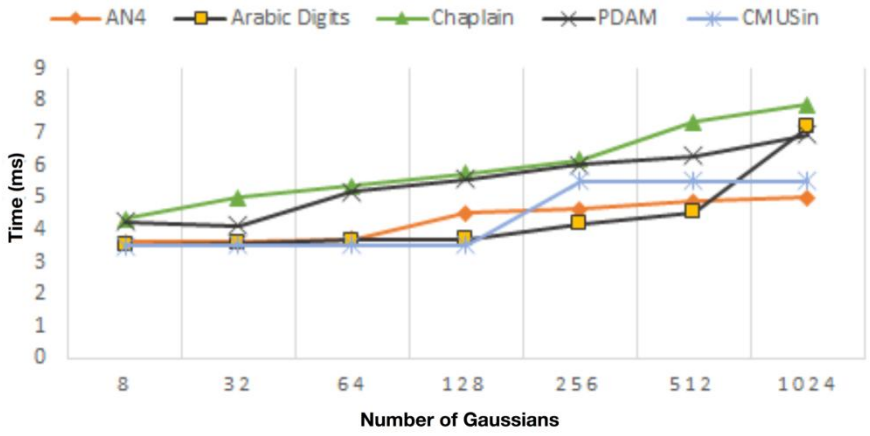


Fig. 10. Control version for varying Gaussians and different datasets.

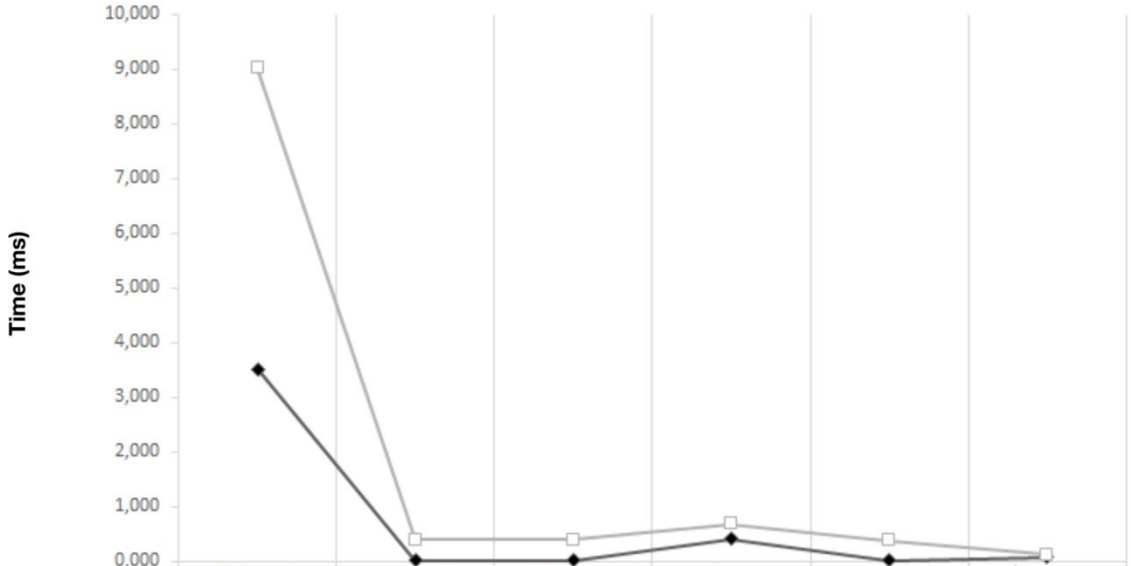
After validating all the modifications to the Σ-kernel provided by the 1st, 2nd and 3rd optimization strategies, final global runtime of EM for the training of GMM has been taken both for the *Control* and *Experimentation* versions (Table 14). *Experimentation* version achieved a speedup of ~51.01x compared to *Control* version.

Table 14. Comparison of Final Global Runtime for Σ-kernel after Three Optimization Strategies

	Control version	*Experimentation* version
Global runtime (ms)	14.27	6.95

Finally, improvement on the processing time for each kernel was analyzed. Only the times of GPU

operations were considered. These times have been captured with the aid of the NVIDIA INSIGHT tool. Fig. 11 shows the results. *Experimentation* version achieved a speedup between ~ 40x and ~ 61x.



	Σ-kernel	p-kernel	^ P-kernel	μ-kernel	π-kernel	Φ-kernel
Exp. (ms)	3,511	0,012	0,012	0,408	0,012	0,076
Control (ms)	9,011	0,401	0,394	0,687	0,389	0,121

Fig. 11. Final processing time for each kernel in both experimentation and control versions.

3.2. Related Work

Other proposals for diagnostic and recommendation of improvement of parallelized solutions exist.

First studies on performance analysis in this area began in the mid-1980s, with tools such as P.I.E, which was used for tracing-based instrumentation of applications running on workstation networks. Later in the late 1980s, the first tools for profiling on computers with multiprocessors appeared. The first profiling tools used in general-purpose parallel applications used to produce traces or execution records by thread and generated reports that could be analyzed by developers after application execution.

In [15], one of the first works of performance analysis in GPU-based general purpose applications was made. That work exploited the functionalities of the GPU Nvidia GeForce 8800 GTX in order to achieve a greater speedup. The authors provided as result of the research 4 principles for application analysis: (1) supplying the latency time of memory access using multiple threads simultaneously, (2) optimizing the use of shared memory, (3) grouping the threads of a warp to avoid the costs of getting out of the SIMD execution and (4) prevent threads that work on the same data from being executed during the same memory access, thus avoiding a possible deadlock situation where a processing unit is idle waiting for the termination of another unit that is stopped. However, these were purely theoretical result and, thus, it is useless to analyze parallel implementations; the work does not indicate in what circumstances the principles should be applied.

In [16], an automated system for the analysis of CUDA applications was created. The system analyzes two classes of problems commonly found in parallel applications: concepts of shared memory banks and the concept of speedup. The author constructed an automatic instrumentation mechanism for CUDA code analysis. This engine runs in emulation mode (running on the CPU). Although this profiler is able to identify speedup conditions and the occurrence of memory access concepts, it can not measure the impact of memory concepts on application performance, even though it does not simulate the memory hierarchy of

the GPU, since it uses CPU memory to identify concepts. Results are inefficient because it does not allow to properly identify the consumption of GPU resources by the algorithm. The proposed emulation environment does not allow developers to exploit the available GPU resources.

[17] presents a static divergence analysis that determines which program variables will have the same value being executed by different processing units. The goal of the work was to identify redundant execution flows that can be optimized. The work improves the translation of SIMD code to non-SIMD CPUs. It helps developers to manually improve their SIMD applications and also guides the compiler in optimizing SIMD programs. In order to assist the merging of identical execution streams, a new compiler optimization was created that identifies where to analyze the similarity of the sequence of steps performed by different processing units between divergent execution flows and whenever possible the compiler suggests the unification of processing flows that perform identical tasks. The work presented the following results: the analysis has a divergence of 34% from false positives if compared with the results of a dynamic profiler. The automatic optimization provided a speed increase of 3% for the parallel quicksort algorithm, a reference that is already highly optimized. The authors stated that if the developer follows the optimization manual proposed by the work, he/she would be able to improve processing time up to 10%. The analysis performed by this work proved to be useful because it analyzes in a consistent and simple manner the behavior of a variable in different processing flows and, through such analysis, it identifies redundant flows and suggests the merger of the same ones.

The work on [18] aims to analyze solutions that implement the reduction function in parallelism based on GPU resources. For the analysis of these implementations, a custom suite for parallel code analysis based on CUDA was built with a benchmark of solutions already implemented. In this suite the authors analyzed the relationship between the execution time and bandwidth when using sequential processing and parallel processing based on GPU. Lungu also analyzed the influence of the data types and the influence of the binary operator on the total execution time. Before carrying out the measurements, reduction algorithm was applied. According to the author, the reduction before measurement is the most advantageous technique for performance optimization. To justify the use of this function, he states that the execution time is proportional to the logarithm of the input vector dimension for the proposed solution. The work presented the following results: there was a reduction of data stored in the global memory before loading them in the shared memory. With the use of the reduction the number of instructions to be executed was also minimized. The analysis proposed by the author Lungu is very incomplete, since it considers only one segmentation technique and such technique can not always provide gains in processing. However, a positive point of the research is the fact that the author analyzed the reduction of the energy consumption obtained with the decrease of the processing time.

In [19], the authors performed a GPU and CPU-based implementation analysis. In the comparison of performance, two factors were taken into consideration: latency and yield. For the analysis, the authors compared the time spent executing the same task on a GPU (NVIDIA GeForce GT630M), written in CUDA C programming language, and running on a third-generation CPU (Intel I-5 3210m). In the experiment, they considered the increase in workload size and the following results were obtained: as the workload size increased, the GPU was 51% faster than the multithreaded CPU when the GPU reached 100% of occupancy and the yield obtained by the GPU was 2.1 times superior to the CPU yield. The analysis performed by the authors is important since they observed an unexplored factor so far which is the latency of the algorithm when the GPU is at its maximum occupation.

Table 15 summaries the related work concerning some criteria we chose as relevant. An “Y" indicates the presence of the feature and “N”, the absence. A dash means that no reference was found to support the information. The feature *Metric* indicates whether the proposal considers the usage of performance metrics.

The feature *Incremental analysis* indicates whether the proposal can be divided in phases. *Profiling* indicates whether the work uses the profiler technique to segment the region of analysis of the algorithm and consequently whether the proposal analyzes processing units individually. Finally, the *Large-scale testing* feature identifies whether the proposal performs experiments on a large-scale basis.

Table 15. Summary of Meparalel's Related Work

Work	Metric	Incremental analysis	Profiling	Large-scale testing
MEPARALEL	Y	Y	Y	N
Ryoo et. al [15]	N	Y	N	Y
Boyer et. al [16]	Y	N	N	—
Coutinho et al. [17]	N	Y	N	Y
Lungu et. al [18]	N	Y	Y	N
Thomas and Daruwala [19]	Y	N	N	Y

4. Conclusion

This paper proposed a method for analyzing GPU-based parallelized algorithms implementations regarding the most relevant performance metrics in the concerned literature. The goal of the method is to help beginners to proper use GPU resources and boost their parallelized solutions. The so-called MEPARALEL undergoes the target implementation to a pipeline of analysis phases to finally recommend optimization strategies to be adopted to ensure better performance metrics values.

Case study relied on the EM algorithm for the training of Gaussian Mixture Models, a mainstream approach to the development of Automatic Speech Recognition systems, for instance. In addition to the correct identification of the main bottleneck of the chosen implementation, MEPARALEL was effective in the improvement of all settled performance metrics. Also, the results showed a great reduction in the processing time of the refactored parallelized implementation, which confirms the feasibility of the proposal and indicates it should be tested in a larger basis.

An important future work is to analyze the impact of energy consumption on speedup. We intend to analyze the influence of each MEPARALEL's optimization procedure on energy consumption. For this, new metrics and performance analysis indicators might be included in the method.

Acknowledgment

The authors thank the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, Brazil) for the financial support [Universal 14/2012, Processo 483437/2012-3].

References

[1] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, *96(5)*, 879-899.

[2] Medeiros, M., Araújo, G., Macedo, H., Chella, M., & Matos, L. (2014). Multi-kernel approach to parallelization of EM algorithm for GMM training. *Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS)* (pp. 158-165).

[3] Santos, B. F., & Macedo, H. T. (2012). Improving CUDA™ C/C++ encoding readability to foster parallel application development. *ACM SIGSOFT Software Engineering Notes*, *37(1)*, 1-5.

[4] Pandy, M., & Ramamoorthy, K. (2013). A novel two stage carrier frequency offset estimation and compensation scheme in multiple input multiple output-orthogonal frequency division multiplexing system using expectation and maximization iteration. *Journal of Computer Science*, *9(11)*, 1526.

[5] Tagare, H. D., Barthel, A., & Sigworth, F. J. (2010). An adaptive expectation–maximization algorithm with GPU implementation for electron cryomicroscopy. *Journal of structural biology*, *171(3)*, 256-265.

[6] Chen, C., Mu, D., Zhang, H., & Hong, B. (2012). A GPU-accelerated approximate algorithm for incremental learning of Gaussian mixture model. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (pp. 1937-1943).

[7] Zheng, F., Zhang, G., & Song, Z. (2001). Comparison of different implementations of MFCC. *Journal of Computer Science and Technology*, *16(6)*, 582-589.

[8] Bache, K., & Lichman, M. (2014). The arabic spoken digit 9. UCI Machine Learning Repository. Disponível em. Retrieved from http://archive.ics.uci.edu/ml

[9] Acero, A. (2012). Acoustical and environmental robustness in automatic speech recognition (Vol. 201). Springer Science & Business Media.

[10] Cmulm Chaplain: Spoken dialog database. Retrieved from http://www.speech.cs.cmu.edu/Tongues/

[11] Obuchi, Y. (2003). CMU PDA Database. Retrieved from http://www.speech.cs.cmu.edu/databases/pda/

[12] Langner, B., & Black, A. W. (2004). Creating a database of speech in noise for unit selection synthesis. *In Fifth ISCA Workshop on Speech Synthesis*.

[13] Cook, S. (2012). CUDA programming: A developer's guide to parallel computing with GPUs. Newnes.

[14] Kirk, D., Wen-Mei, H., Wu, W. (2012). Parallel computation patterns – Reduction trees. Retrieved from www.imat.hwu.crhc.illinois.edu

[15] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S. Z., Stratton, J. A., & Hwu, W. M. W. (2008). Program optimization space pruning for a multithreaded GPU. *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (pp. 195-204).

[16] Boyer, M., Skadron, K., & Weimer, W. (2008). Automated dynamic analysis of CUDA programs. *Proceedings of the Third Workshop on Software Tools for MultiCore Systems* (p. 33).

[17] Coutinho, B., Sampaio, D., Pereira, F. M. Q., & Meira Jr, W. (2011). Divergence analysis and optimizations. *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (PACT) (pp. 320-329)

[18] Lungu, I., Petrosanu, D. M., & Pirjan, A. (2012). Optimization solutions for improving the performance of the parallel reduction algorithm using graphics processing units. *Informatica Economica*, *16(3)*, 72.

[19] Thomas, W., & Daruwala, R. D. (2014). Performance comparison of CPU and GPU on a discrete heterogeneous architecture. *Proceedings of the 2014 International Conference on Circuits, Systems, Communication and Information Technology Applications* (pp. 271-276).

Otávio Cordeiro Siqueira de Oliveira was graduated in information systems from Federal University of Sergipe in 2013. He obtained a master's degree in computer science from the Federal University of Sergipe in 2016. From August 2016 until the present time works as an assistent professor at the Universidade Tiradentes, Aracaju, Brazil. His current scientific research primarily focuses on distributed computing and business intelligence.

Marco Túlio Chella is an associate professor at the Department of Computer Science, Universidade Federal de Sergipe. He received his Ph.D in electrical engineering from Universidade Estadual de Campinas, Brazil, in 2006.

Hendrik Teixeira Macedo was born in Aracaju/SE, Brazil, in 1977. He graduated in computer science at the Federal University of Sergipe in 1998. He obtained a master's degree in computer science from the Universidade Federal de Pernambuco in 2001 and a doctorate in computer science also from the Universidade Federal de Pernambuco in 2006, having done an internship PhD at the University of Paris VI in 2002. From July 2006 until the present time, works as an associate professor at the Department of Computer Science, Universidade Federal de Sergipe, Brazil. His current scientific research primarily focuses on machine learning methods for natural language processing.

Leonardo Nogueira Macedo was born in Fortaleza/CE, Brasil, in 1969. He graduated in computer science at the Universidade Federal do Ceará in 1991. He obtained a master's degree in applied math from Universidade Estadual de Campinas in 1993 and a doctorate in electrical engineering from UFCG in 2004. He is an associate professor at the Department of Computer Science, Universidade Federal de Sergipe, Brazil. His current scientific research primarily focuses on machine learning methods for speech recognition and image processing.