Fast Estimation of Gaussian Mixture Model Parameters on GPU using CUDA

Lukáš Machlica, Jan Vaněk, Zbyněk Zajíc
Department of Cybernetics, University of West Bohemia in Pilsen, Czech Republic
Email: machlica@kky.zcu.cz, vanekyj@kky.zcu.cz, zzajic@kky.zcu.cz

***Abstract*—Gaussian Mixture Models (GMMs) are widely used among scientists e.g. in statistic toolkits and data mining procedures. In order to estimate parameters of a GMM the Maximum Likelihood (ML) training is often utilized, more precisely the Expectation-Maximization (EM) algorithm. Nowadays, a lot of tasks works with huge datasets, what makes the estimation process time consuming (mainly for complex mixture models containing hundreds of components). The paper presents an efficient and robust implementation of the estimation of GMM statistics used in the EM algorithm on GPU using NVIDIA's Compute Unified Device Architecture (CUDA). Also an augmentation of the standard CPU version is proposed utilizing SSE instructions. Time consumptions of presented methods are tested on a large dataset of real speech data from the NIST Speaker Recognition Evaluation (SRE) 2008. Estimation on GPU proves to be more than 400 times faster than the standard CPU version and 130 times faster than the SSE version, thus a huge speed up was achieved without any approximations made in the estimation formulas. Proposed implementation was also compared to other implementations developed by other departments over the world and proved to be the fastest (at least 5 times faster than the best implementation published recently).**

I. Introduction

The Expectation-Maximization (EM) algorithm, in clustering often used also with Gaussian Mixture Models (GMMs), was in [1] identified as one of the top 10 data mining algorithms. GMMs trained via EM are widely used in many state-of-the art recognition and data mining systems. They are of most importance in the speaker recognition, they are utilized in the concept of super-vectors and Support Vector Machines (SVMs) [2] and in a novel approach called Joint Factor Analysis (JFA) [3] further extended to the concept of iVectors [4]. Another usage can be found in speech recognition systems based on Hidden Markov Models with output probabilities described by GMMs [5]. Nevertheless, GMMs are utilized also by biologists and immunologists for counting, sorting, and analyzing cells suspended in a fluid [6].

All these techniques process huge amounts of data, thus demanding a superior computing power. Nowadays, parallel technologies like supercomputers, clusters, grids, and cloud infrastructures gain on importance [7]. A simpler option is to utilize the Graphics Processing Unit (GPU), which developed through time to a highly parallel and computationally powerful tool useful not only for graphics processing, but also for high performance computing [8]. The main advantage of GPUs over Central Processing Units (CPUs) is their price-performance ratio. Several manufacturers have put a lot of effort to improve their GPU's development environment in order to grant access to their GPU's computing power. This paper focuses on NVIDIA's Compute Unified Device Architecture (CUDA). It should be stated, that implementations of the GPU algorithm may be easily included also into the above mentioned parallel technologies.

Focus will be laid on GMMs described by diagonal covariance matrices. Only the core of the EM algorithm will be described, because it is the most time consuming part of the estimation process. Note that we will describe the estimation of GMM statistics rather than the estimation of new GMM parameters since the estimated statistics are more general and may be used also in other techniques, e.g. adaptation of a GMM frequently performed in speaker recognition tasks [9].

In the first part of the paper GMM statistics and basics of CUDA are described. The main part of the paper is devoted to the description of an efficient implementation of the estimation algorithm of GMM statistics on GPU. In Section IV also an augmented CPU version is proposed. It utilizes Streaming SIMD Extension (SSE) instructions, which make the estimation on CPU significantly faster, but as the results prove the GPU version performs best (in the sense of the time consumption). Note that the estimation process does not involve any approximations, GMM statistics obtained using any of the methods are equal (to some negligible rounding errors). Time consumption experiments were performed on a speech corpus NIST Speaker Recognition Evaluation (SRE) 2008 [10] containing spontaneous telephone speech. In the last part of the paper, detailed analysis of results along with comparison with other implementations are described.

II. GMM Statistics

Let us briefly introduce the GMM statistics of interest. Assume a set of feature vectors $\boldsymbol{O} = \{\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T\}$, where $\dim(\boldsymbol{o}_t) = D$, and a GMM given by a set of parameters $\boldsymbol{\lambda} = \{\boldsymbol{\lambda}_m\}_{m=1}^{M} = \{\omega_m, \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m\}_{m=1}^{M}$ containing M mixture components, their weights, mean vectors and covariance matrices, respectively. Let us define a function

$$\mathcal{L}(\boldsymbol{o}_t, \{\boldsymbol{\lambda}_b, \ldots, \boldsymbol{\lambda}_e\}) = \log \sum_{m=b}^{e} \omega_m \mathcal{N}(\boldsymbol{o}_t | \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m), \quad (1)$$

where $b \geq 1, e \leq M$. In the paper only diagonal covariance GMMs will be assumed, where $\boldsymbol{\sigma}_m^2 = \text{diag}(\boldsymbol{\Sigma}_m)$, hence

$$\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m) = \log(\omega_m) + \sum_{i=1}^{D} \log \mathcal{N}(o_{t,i}|\mu_{m,i}, \sigma_{m,i}^2), \quad (2)$$

where $\mathcal{N}(o|\mu, \sigma^2)$ denotes the Gaussian probability density function with mean μ and variance σ^2. Then

$$\gamma_m(\boldsymbol{o}_t) = \exp\left(\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m) - \mathcal{L}\left(\boldsymbol{o}_t, \{\boldsymbol{\lambda}_k\}_{k=1}^M\right)\right) \quad (3)$$

$$c_m = \sum_{t=1}^{T} \gamma_m(\boldsymbol{o}_t), \quad (4)$$

$$\boldsymbol{\varepsilon}_m = \textstyle\sum_{t=1}^{T} \gamma_m(\boldsymbol{o}_t)\boldsymbol{o}_t, \quad \boldsymbol{\varepsilon}_m^2 = \sum_{t=1}^{T} \gamma_m(\boldsymbol{o}_t)\boldsymbol{o}_t\boldsymbol{o}_t^{\mathrm{T}} \quad (5)$$

are the m^{th} component's posterior probability given a feature vector $\boldsymbol{o}_t$, the m^{th} component's soft count and the (unnormalized) first and second moment of features aligned to a component m, respectively. Note that $\mathcal{L}(\boldsymbol{o}_t, \{\boldsymbol{\lambda}_k\}_{k=1}^M) = \mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})$ represents the log-likelihood of $\boldsymbol{o}_t$ given the model $\boldsymbol{\lambda}$. The update formulas for new GMM parameters $\bar{\boldsymbol{\lambda}}$ are given as

$$\bar{\omega}_m = \tfrac{c_m}{T}, \quad \bar{\boldsymbol{\mu}}_m = \tfrac{1}{c_m}\boldsymbol{\varepsilon}_m, \quad \bar{\boldsymbol{\Sigma}}_m = \tfrac{1}{c_m}\boldsymbol{\varepsilon}_m^2 - \bar{\boldsymbol{\mu}}_m\bar{\boldsymbol{\mu}}_m^{\mathrm{T}}. \quad (6)$$

III. Estimation utilizing CUDA

GPU's CUDA may be seen as a fully parallel system operating with hundreds of threads at once. According to the GPU architecture threads are organized into *thread blocks*. Thread blocks are independent on each other (algorithm executed in each of the blocks does not depend on what is going on in other blocks), while threads in each block are allowed to cooperate. All thread blocks execute the same algorithm called a *kernel*. Note that not only threads in a block, but also several thread blocks may be executed at once. Hence, CUDA parallelism is provided at 2 levels - threads and block of threads. All thread blocks are ordered in an one- or two-dimensional *grid* (a 2 dimensional grid is depicted in Fig. 1). Each thread block carries specific information about its position within the grid (row and column position of the block in the grid).

A high GPU computing performance can be fully utilized only with proper memory management. Several memory types exist, which significantly differ in their size, access speed and access permission. *Global memory* (GM) has read/write access, has hundreds of mega bytes available and can be accessed from every block and every thread, but the access latency is relatively high. The best performance of GM can be achieved using the *Texture Memory* (TM). TM can be seen as a part of GM, but it is read only and cached, thus the access speed may be significantly faster than in the case of GM. Another type of memory is the *Shared Memory* (SM), which storage size is around kilo bytes, but the access speed is very high (very low latency). SM is visible only for threads in a thread block. In summary, one has to carefully choose the memory management according to a given task.

Other CUDA environment descriptions exceed the range of the paper, for further details and deeper understanding of the problem the reader is referred to [11].

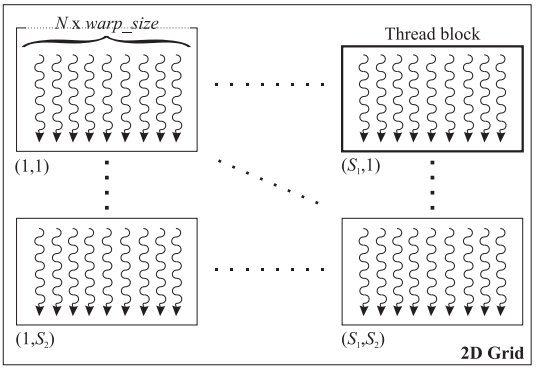

Fig. 1. Two dimensional $S_1 \times S_2$ grid with thread blocks. Each thread block contains several threads, where the optimal number of threads is a multiple of the warp size.

A. Preparing the Data

In order to make the best of the GPU computing power one has to align the data into Memory-Aligned-Blocks (MABs). The optimal size of a MAB is closely related to the number of threads in a thread block. Number of threads in a block is user dependent, but optimally has to be a multiple of the *warp size*. Warp size is hardware dependent and represents the minimum number of threads in a thread block that run at once (mostly a multiple of 32) – *run in a warp*. For the best performance threads in a warp have to access data in the memory sequentially, therefore data in MABs have to be properly organized.

We have input data (a set of feature vectors $\boldsymbol{O}$ and a set of GMM parameters $\boldsymbol{\lambda}$), temporary data (mixture component posteriors (3) together with log-likelihoods of feature vectors given $\boldsymbol{\lambda}$), and output data (first and second moments (5) and soft counts (4)) that have to be properly organized in the GPU memory. The memory storage of feature vectors, model means and posteriors of mixture components are depicted in Fig. 2, Fig. 3 and Fig. 4, respectively. Storage of GMM diagonal variances is the same as the storage of model means depicted in Fig. 3. The reason why model parameters and mixture component posteriors are stored in group of 4 is that CUDA supports *X4* (e.g. *short4, int4, float4*, etc.) data types – one can read data from the memory in quaternions.

Rather than to store only weights of a mixture component we store the precomputed normalization coefficient of each component, which logarithm is given as

$$g_m = \log(\omega_m) - 0.5D\log(2\pi) - 0.5\log|\boldsymbol{\Sigma}_m|. \quad (7)$$

Memory management of g_m, soft counts (4) and first and second moments (5) is trivial (recall that only the diagonal of second moments is stored), they are all stored sequentially in ascending order according to the number of mixture component they belong to (e.g. a vector of first moments $[\boldsymbol{\varepsilon}_1^{\mathrm{T}}, \boldsymbol{\varepsilon}_2^{\mathrm{T}}, \ldots, \boldsymbol{\varepsilon}_M^{\mathrm{T}}]$ represents one memory block). Also data log-likelihoods $\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})$ are stored sequentially in ascending order according to the position of a vector $\boldsymbol{o}_t$ in the set $\boldsymbol{O}$, thus forming a vector $[\mathcal{L}(\boldsymbol{o}_1, \boldsymbol{\lambda}), \ldots, \mathcal{L}(\boldsymbol{o}_T, \boldsymbol{\lambda})]$.

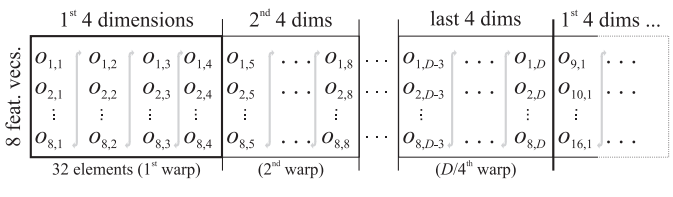

Fig. 2. Organization of feature vectors $\boldsymbol{O} = \{\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T\}$ in the GPU's global memory, where $\dim(\boldsymbol{o}_t) = D$. Data are stored column-wise – 1st dimension of first 8 feature vectors then 2nd dimension of first 8 samples, etc. In each warp a block of memory is read sequentially enabling optimal speed performance.

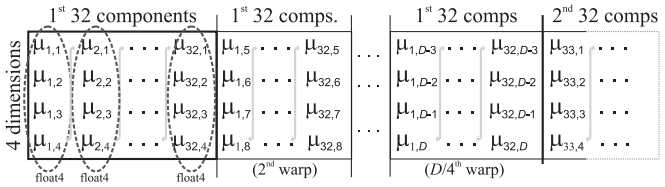

Fig. 3. Organization of model means $\boldsymbol{\mu}_i$ in the GPU's global memory. Storage of GMM diagonal variances $\boldsymbol{\sigma}_i^2$ is the same. Data are stored column-wise – 1st four dimensions of $\boldsymbol{\mu}_1$ then 1st four dimensions of $\boldsymbol{\mu}_2$, etc.

It should be stated that all the GPU's memory management of feature vectors, model parameters, temporary data (once computed) is assigned to the cached TM (all data are visible to all thread blocks and their threads). However, feature vectors are copied to the faster SM in some kernels, see next section.

B. CUDA Kernels

Kernels specify what should threads in a thread block do (number of threads is specified by the user) assuming additional information about the position of a thread block in a grid, grid dimension and given input data. The position information along with the grid dimension is utilized to properly divide input data into smaller independent portions. Each of the data portions is then handled by a separate thread block according to the specified kernel function.

Not all the tasks can be parallelized using only one kernel function since a problem can not always be divided into several fully independent parallel subtasks. More often a result of one subtask depends on a result of a different subtask. However, such tasks may have only a few points where they need to exchange their outcomes. Thus, to parallelize the task one has to employ more kernels. We have proposed 4 kernels

- $\hat{\gamma}$*-kernel* – computes $\hat{\gamma}_{m,t} = \mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m)$ for each t, m,
- $\mathcal{L}$*-kernel* – computes $\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})$ for each t,
- γ*-kernel* – normalizes each $\hat{\gamma}_{m,t}$ by $\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})$, see (3),
- ε*-kernel* – estimates first and second moments $\boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_m^2$ for each m.

In order to describe the data portions handled by distinct kernels described in the next section assume a set $\boldsymbol{\Gamma} = \{\{1, \ldots, Q_T\}, \ldots, \{T - Q_T, \ldots, T\}\} = \{\boldsymbol{\Gamma}_i\}_{i=1}^{S_1}$ containing equally large disjoint subsets of feature vector indexes, a set $\boldsymbol{\Omega} = \{\{1, \ldots, Q_M\}, \ldots, \{M - Q_M, \ldots, M\}\} = \{\boldsymbol{\Omega}_j\}_{j=1}^{S_2}$ containing equally large disjoint subsets

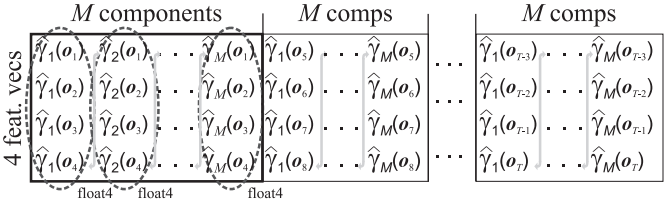

Fig. 4. Organization of unnormalized posteriors $\hat{\gamma}_m(\boldsymbol{o}_t) = \mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m)$ given in (2) of a mixture component in the GPU's global memory. Data are stored column-wise – 1st four posteriors of 1st GMM component given first four feature vectors $\{\boldsymbol{o}_1, \ldots, \boldsymbol{o}_4\}$, 1st four posteriors of 2nd component, etc.

Algorithm 1 $\hat{\gamma}$-kernel function $\rightarrow$ blocks $\boldsymbol{\Gamma}_i, \boldsymbol{\Omega}_j$

Require: Thread block position (i, j) in the grid
1: SM $\overset{\{\boldsymbol{o}_t\}_{t\in\boldsymbol{\Gamma}_i}}{\longleftarrow}$ GM
2: $m := \Omega_{j,thread\_index}$
3: $\hat{\gamma}_{m,\Gamma_{i,1}} := g_m$; $\ldots$; $\hat{\gamma}_{m,\Gamma_{i,Q_T}} := g_m$
4: **for** $d = 1$ to S_3 **do**
5: **for** $t \in \boldsymbol{\Gamma}_i$ **do**
6: $\hat{\gamma}_{m,t} := \hat{\gamma}_{m,t} + \sum_{x\in\boldsymbol{\Delta}_d}(o_{t,x} - \mu_{m,x})^2/\sigma_{m,x}^2$
7: **end for**
8: **end for**
9: GM $\longleftarrow \{\hat{\gamma}_{m,\Gamma_{i,k}}\}_{k=1}^{Q_T}$

of indexes of GMM components, and a set $\boldsymbol{\Delta} = \{\{1, \ldots, Q_D\}, \ldots, \{D - Q_D, \ldots, D\}\} = \{\boldsymbol{\Delta}_d\}_{d=1}^{S_3}$ formed by equally large disjoint subsets of dimension indexes of feature vectors. Q_T, Q_M and Q_D are user defined scalars, where $Q_T \leq T$, $Q_M \leq M$ and $Q_D \leq D$. Loosely speaking, $\Gamma_{i,j} = (i-1) \cdot Q_T + j$, $\Omega_{i,j} = (i-1) \cdot Q_M + j$, $\Delta_{i,j} = (i-1) \cdot Q_D + j$. Memory management depicted in Figs. 2 and 3 is well suited for $Q_T = 8$, $Q_M = 32$, $Q_D = 4$. In order to preserve the robustness of calculations all intermediate results are kept in logarithms (as long as possible).

1) $\hat{\gamma}$-kernel: operates on a two-dimensional $S_1 \times S_2$ grid, which rows indicate the portion of feature vectors and the columns of the grid indicate the portion of mixture components to be processed. Hence, the $(i, j)^{th}$ thread block operates with sets $\boldsymbol{\Gamma}_i$ and $\boldsymbol{\Omega}_j$, and the output of a thread block are the corresponding weighted log-likelihoods $\hat{\gamma}_{m,t} = \mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m)$ of a mixture component written to given positions in GM as illustrated in Fig. 4.

At the very beginning of the kernel execution, the complete input portion of feature vectors $\{\boldsymbol{o}_t\}_{t\in\boldsymbol{\Gamma}_i}$ (in Fig. 2 are these all the dimensions of 8 feature vectors) handled by one thread block is read sequentially (as described in Section III-A) from GM and written to SM.

Each thread estimates one mixture component's weighted log-likelihood of Q_T different feature vectors, thus a set $\{\hat{\gamma}_{m,t}\}_{t\in\boldsymbol{\Gamma}_i}$ for one specific m. Particular steps of the kernel algorithm are described in Alg. 1. The sum $\sum_{x\in\boldsymbol{\Delta}_d}(\ldots)$ across a subset of dimensions of a feature vector and across a subset of dimensions of GMM parameters in the for-loop is

caused by the fact that the model parameters are read from TM as *float4*, thus $\mathbf{\Delta}_d = \{(d-1)\cdot 4 + j\}_{j=1}^{4}$ consists of 4 indexes of 4 dimensions (see Fig. 3). The relationship between the storage of model parameters and feature vectors should be now clearer – mainly the reason why the vectors are divided to dimension blocks of 4. Also note that rather than using the for-loop through indexes in $\mathbf{\Gamma}_i = \{\Gamma_{i,1}, \ldots, \Gamma_{i,Q_T}\}$ we unroll the loop in order to boost the performance.

2) $\mathcal{L}$-kernel: is a sum kernel, it computes the overall log-likelihood of each feature vector given a GMM. Hence, the input to the kernel is the output of the $\hat{\gamma}$-kernel. The output of $\mathcal{L}$-kernel is a set $\{\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})\}_{t=1}^{T}$ written to the GM as described in Section III-A. Several efficient algorithms for a parallel sum have already been proposed, we use the implementation described in [12].

3) γ-kernel: performs the normalization of each $\hat{\gamma}_{m,t}$ with $\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})$ and produces true posteriors of GMM components, thus $\gamma_m(\boldsymbol{o}_t) = \exp\left(\hat{\gamma}_{m,t} - \mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda})\right)$. These are written to the same positions as their unnormalized counterparts. One thread block processes Q_T feature vectors from a set $\mathbf{\Gamma}_i$ and all the mixture components, and outputs $\{\gamma_m(\boldsymbol{o}_t)\}_{t\in\mathbf{\Gamma}_i, m\in\mathbf{\Omega}}$.

4) ε-kernel: operates on a two-dimensional $S_2 \times S_3$ grid, which rows indicate the portion of GMM components and the columns of the grid indicate the portion of feature dimensions to be processed. Hence, the $(i,j)^{th}$ thread block operates with sets $\mathbf{\Omega}_i$ and $\mathbf{\Delta}_j$, and the output of a thread block are the dimension blocks of first and second (diagonal) moments (5) of features aligned to a given mixture component along with the soft counts (4). The output is written to GM on positions described in Section III-A. Thus, each thread block processes the whole set of feature vectors, however only values for a specific subset of dimensions of first and second moments are estimated. The ε-kernel operates with all the data – feature vectors, model parameters and temporary data obtained as the output of the γ-kernel. The kernel algorithm is described in Alg. 2. Note that $\{o_{t,d}\}_{t\in\mathbf{\Gamma}_q, d\in\mathbf{\Delta}_j}$ is one block depicted in Fig. 2 containing 8 feature vectors ($Q_T = 8$) and their 4 dimensions ($Q_D = 4$), thus 32 elements in common. Again, to boost the performance instead of using the most inner for-loops through indexes in $\mathbf{\Gamma}_q, \mathbf{\Delta}_j$ we unroll the loops. Also note that posteriors $\boldsymbol{\gamma}_m(\boldsymbol{o}_t)$ are read from TM as *float4* data types.

Such set up is efficient mainly in cases when the number of mixture components or dimension of feature vectors is high, otherwise only a few thread blocks have to be executed what decreases the speed performance. In these cases the input portion of feature vectors is divided into Q_N blocks and each thread block accumulates statistics for one of these blocks. Hence, now the ε-kernel processes T/Q_N feature vectors, Q_M mixture components and Q_D dimensions. After all the statistics for disjoint feature sets have been accumulated an additional kernel is utilized in order to sum up the resulting Q_N distinct statistics.

The estimation of full second moments is out of the scope of this paper. The memory management stays the same, the ε-kernel has to be slightly altered, where an additional problem of distinct dimension blocks that have to be available at once has to be solved.

Algorithm 2 ε-kernel function $\rightarrow$ blocks $\mathbf{\Omega}_i, \mathbf{\Delta}_j$

Require: Thread block position (i,j) in the grid

$m := \Omega_{i,thread\_index}$
$c_m := 0$
$\varepsilon_{m,\Delta_{j,1}} := 0; \ldots; \varepsilon_{m,\Delta_{j,Q_D}} := 0$
$\varepsilon^2_{m,\Delta_{j,1}} := 0; \ldots; \varepsilon^2_{m,\Delta_{j,Q_D}} := 0$
for $q = 1$ to S_1 **do**
  SM $\overleftarrow{\{o_{t,d}\}_{t\in\mathbf{\Gamma}_q, d\in\mathbf{\Delta}_j}}$ GM
  for all $t \in \mathbf{\Gamma}_q$ **do**
    $c_m := c_m + \gamma_m(\boldsymbol{o}_t)$
    for all $d \in \mathbf{\Delta}_j$ **do**
      $\varepsilon_{m,d} := \varepsilon_{m,d} + \gamma_m(\boldsymbol{o}_t) \cdot o_{t,d}$
      $\varepsilon^2_{m,d} := \varepsilon^2_{m,d} + \gamma_m(\boldsymbol{o}_t) \cdot o^2_{t,d}$
    end for
  end for
end for
GM $\longleftarrow \{\varepsilon_{m,\Delta_{j,k}}, \varepsilon^2_{m,\Delta_{j,k}}\}_{k=1}^{Q_D}$
GM $\longleftarrow c_m$

IV. Estimation utilizing SSE

We have tried to speed-up also the estimation on CPU utilizing Streaming SIMD Extensions (SSE), where SIMD stands for Single Instruction, Multiple Data. The power of SSE is that it can perform several instructions (addition, subtraction, multiplication, etc.) at once using 128-bit registers. Thus, assuming 32-bit single-precision floating point (SPFP) numbers one can perform 4 operations at a time.

We have incorporated the SSE instructions into the estimation of $\mathcal{L}(\boldsymbol{o}_t, \boldsymbol{\lambda}_m)$ given in (2), which is the most frequent, thus most time consuming. More precisely, SSE is used when computing the exponential part of the normal distribution $\sum_{d=1}^{D}(o_d - \mu_d)^2/\sigma_d^2$. Using SSE and SPFP such sum can be added up in $D/4$ steps. In situations where D is not a multiple of 4 one has to correctly align the memory (pad ends with zeros) where GMM parameters (means and variances) and feature vectors are stored. Additional less significant speed bursts may be acquired extending the SSE instructions into the accumulation process of moments given in (5).

V. Experiments

Experiments were performed on a single EM iteration. Data were taken from NIST SRE 2008 [10], only training data were used for adaptation. More precisely, it was the short2 training condition and only male telephone speech of approximately five minutes total duration was used (non-speech events were discarded during feature extraction). In common 648 speakers were involved, approximately 54 hours of speech were used. In summary, the training data consisted of 3,125,506 (3125.6k) feature vectors of dimension 40.

In our implementation we used only floating point arithmetic. The user defined constants used in Section III-B were

set to $Q_T = 8$, $Q_M = 32$, $Q_D = 4$ (such settings correspond with Figs 2-4), and $Q_N = 8$. The number of threads in each thread block was set to 32.

CPU and SSE implementations were tested on 2.39 GHz Intel 4 GB RAM PC, the GPU implementation was tested on low-end NVIDIA GeForce GTX 280 video card and the algorithms were developed in CUDA toolkit 3.1. All the GPU time consumptions were computed as the sum of times of all the executed kernels.

A. Analysis of the implementation performance

Comparison of time consumptions of the proposed implementation can be found in Tab. I. Only the time needed to accumulate statistics was measured, and just one thread was used in all CPU implementations. Results are given in seconds, the data set consists of 3125.6k feature vectors of dimension 40. GPU is approximately 150 times faster than the CPU-SSE implementation and more than 400 times faster than the naive CPU implementation. The relative GPU execution times of particular kernels can be found in Fig. 5.

The comparison of GPU and CPU version strongly depends on the implementation of the CPU version. In order to express only the performance of the GPU implementation one can evaluate the number of floating point operations per second (GFLOPS). Counts of operations executed in each kernel are

- $\hat{\gamma}$*-kernel* – $4 \times D \times T \times M$ operations,
- $\mathcal{L}$*-kernel* – $T \times M$ of logarithmic addition functions (which we rated as 13 operations), thus $T \times M \times 13$ operations,
- γ*-kernel* – $T \times M$ of 5 simple operations + one exponential operation (equal to 4 simple operations), thus $T \times M \times 9$
- ε*-kernel* – $4 \times D \times T \times M + T \times M$ operations.

We distinguish simple operations as addition, subtraction, and multiplication from operations as logarithms and exponentials, which are on GPUs calculated using special function unit that have four times lower throughput. Therefore we rate logarithms and exponentials as 4 simple operations. Hence, when number of mixture components M, number of feature vectors T, and dimension of feature vectors D are known, the overall number of operations of the estimation can be computed. In our case this is the sum of operations of all 4 kernels. Tab. II contains the number of operations per second (giga FLOPS = GFLOPS), which are computed as the number of operations needed to estimate GMM statistics for various number of components divided by the estimation times from Tab. I. GFLOPS increase from 163.4 to 242.1 because larger models utilize GPU cores better, and the overhead of kernel executions is relatively lower in cases of larger models. The theoretical peak of the GTX 280 GPU is 933 GFLOPS (according to specifications of the manufacturer), which is in comparison to the performance on real tasks significantly overstated. In a benchmark task performed in [13], where a well optimized task of multiplication of two large matrices on GTX 280 GPU is carried out, the achieved performance varies between 190 – 375 GFLOPS in dependence on matrix sizes. Hence, GFLOPS of our implementation are in the range of such a well optimized task. Nevertheless, matrix-matrix multiplications consist only of fused multiplication/addition instructions, which are evaluated in a single GPU clock (doubles the GFLOPS performance). However, our task consists also from other instructions, which are not as efficient.

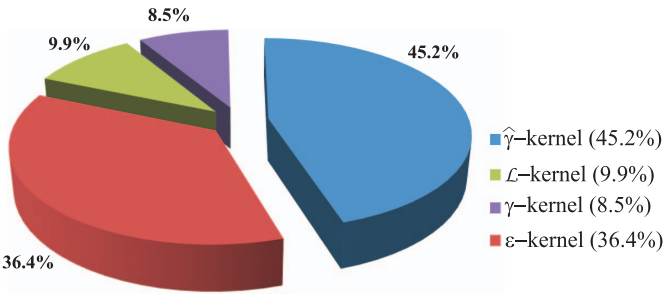

Fig. 5. Relative GPU execution times for all of the kernels described in Section III-B.

TABLE I
THE AMOUNT OF TIME IN SECONDS NEEDED TO ESTIMATE STATISTICS OF 3125.6K FEATURE VECTORS OF DIMENSION 40 FOR VARIOUS NUMBER OF GMM COMPONENTS.

#Comps	32	64	128	256	512	1024	2048
CPU	78	150	287	559	1094	2174	4289
SSE	32	54	95	171	325	619	1199
GPU	0.21	0.32	0.63	1.18	2.34	4.57	9.07

TABLE II
PERFORMANCE OF THE ALGORITHM RUNNING ON NVIDIA GEFORCE GTX 280 IN GFLOPS WHEN PROCESSING 3125.6K FEATURE VECTORS OF DIMENSION 40 FOR VARIOUS NUMBER OF GMM COMPONENTS.

#Comps	32	64	128	256	512	1024	2048
GFLOPS	163.4	214.4	217.8	232.6	234.6	240.2	242.1

B. Comparison with previous works

We have tried to compare the time consumptions also with other implementations. We have tested several freely available implementations, but all of them failed (lack of numerical stability) on our large dataset of high dimensional real data. We have found two remarkable recent publications interested in the GPU implementation of the EM algorithm focusing on GMMs with diagonal covariances, namely a publication by Kumar et al. [14] and a master thesis from Andrew Pangborn [6].

Experiments performed by Kumar et al. used NVIDIA Quadro FX 5800, which is almost identical to the NVIDIA TESLA C1060 on which the experiments of Andrew Pangborn were performed, and to NVIDIA GeForce GTX 280 on which our experiments were performed. Time consumptions of both implementations were taken from Tab. 5.8 from [6]. In order to compare the implementations to ours we set up same conditions as in [14] and [6]. Hence, we reduced the dimension of our data to 32 and took only 153.6k and 230.4k feature vectors. Tab. III is the extended table containing also our results denoted as UWB and the CPU reference computed

TABLE III
COMPARISON GIVEN IN MILLISECONDS OF DIFFERENT IMPLEMENTATIONS FOR DIFFERENT AMOUNTS OF TRAINING DATA ASSUMING FEATURE VECTORS OF DIMENSION 32 AND GMM WITH 32 COMPONENTS.

#samples	Kumar et al.	A.Pangborn	UWB	MATLAB®
153.6k	215.0	51.1	9.25	10936.0
230.4k	264.9	71.1	13.99	16461.0

in MATLAB® 7.5.0.342 (R2007b) utilizing the Statistics Toolbox function `gmdistribution.fit()`.

As can be seen from Tab. III, our proposed implementation outperformed the others. It is more than 5 times faster than A. Pangborn's implementation and more than 20 times faster than Kumar's implementation.

The key part of the speed up is the proper memory management of the data adhering to the rules of coalesced access. In addition, data loaded to the kernels are reused as much as possible (higher degree of parallelization), e.g. the log-likelihoods are estimated for several frames and several mixture components at once in each kernel, the same principle holds for the accumulation kernel (see descriptions of $\hat{\gamma}$- and ε-kernel in Section III-B). Another important performance related technique lays in the use of the Texture Memory (TM) with *float4* data types for read-only data. Data shared across a thread block or data that are accessed repeatedly should be copied into the Shared Memory (SM) in advance. The mentioned advices are of course well known, but it is quite difficult to integrate them to a specific task.

Another drawback of Kumar et al. implementation is that the GPU memory requirements are very high, this holds particularly also for A. Pangborn. The most of the memory is occupied by the intermediate results. Since the statistics are additive, the computation can be divided into smaller parts that require significantly lower amount of memory. In our case the most of the memory is occupied only by the input data, thus we are able to fit up to 6 millions of 40 dimensional feature vectors into the GPU memory of size 1 GB. However, even in cases where a huge data set containing several hundreds of millions of feature vectors needs to be processed, the memory problem can be solved efficiently. Most of GPUs dispose of concurrent copy and execution feature, thus additional data can be uploaded to the GPU memory while already uploaded data are processed.

VI. CONCLUSION

The paper presented a novel approach to the estimation of GMM statistics using GPU and CUDA environment. Since the EM algorithm does converge only locally it is often convenient to run EM several times with different initializations. Hence, in order to train a reliable GMM via EM one has to perform a lot of reestimations. With increasing amount of training data and increasing complexity of models, the training of GMMs becomes very time consuming. As has been shown (see Tab. I), the GPU implementation offers a huge increase in the speed of GMM training. However, the final speed up strongly depends not only on the GPU hardware, but also on a proper implementation itself (see Tab. III). The estimation process can be easily parallelized also on the CPU (e.g. dividing feature vectors to smaller disjoint sets, estimating statistics for each set, and at the end adding the statistics up), but the resources spent on the hardware are much higher than in the case of GPU, which is parallel inherently.

We have focused on the estimation of GMM statistics with diagonal covariances since often full covariance GMMs can be accurately replaced by diagonal covariance GMMs with higher number of components. The estimation of diagonal covariances is more robust mainly with increasing dimension of feature vectors. This is often the case in tasks of speech and speaker recognition, where frequently only the diagonal covariances are used. Note that one of the outputs of the algorithm produced by the $\mathcal{L}$-kernel is also the log-likelihood of feature vectors given a GMM, which is required in the classification phase.

ACKNOWLEDGMENT

This work was supported by the Grant Agency of the Czech Republic, project No.GAČR 102/08/0707, by the Ministry of Education of the Czech Republic, project No.MŠMT LC536 and by the grant of the University of West Bohemia, project No. SGS-2010-054.

REFERENCES

[1] W. Xindong, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, et al., "Top 10 algorithms in data mining," Knowledge and Information Systems, Volume 14, 2007.

[2] W. M. Campbell, D. E. Sturim, D. A. Reynolds and A. Solomonoff, "SVM based speaker verification using a GMM supervector kernel and NAP variability compensation," Acoustics, Speech and Signal Processing, 2006.

[3] P. Kenny, "Joint Factor Analysis of speaker and session variability: Theory and algorithms," technical report, Centre de Recherche Informatique de Montral (CRIM), 2006.

[4] N. Dehak, P. Kenny, R. Dehak, P. Dumouchel and P. Ouellet, "Front-end Factor Analysis for speaker verification," IEEE Transactions on Audio, Speech and Language Processing, 2010.

[5] J. Vaněk, J. Trmal, J. V. Psutka and J. Psutka, "Optimization of the Gaussian Mixture Model Evaluation on GPU," Interspeech 2011, in press.

[6] A. D. Pangborn, "Scalable data clustering using GPUs," Masters thesis, Rochester Institute of Technology, 2010.

[7] C. Plant and C. Bohm, "Parallel EM-clustering: Fast convergence by asynchronous model updates," International Conference on Data Mining Workshops, 2010.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," Journal of Parallel and Distributed Computing, Volume 68, 2008.

[9] D. A. Reynolds and R. C. Rose, "Robust text-independent speaker identification using Gaussian Mixture speaker models," IEEE Transactions on Speech and Audio Processing, 1995.

[10] "NIST SRE 2008 evaluation plan," http://www.itl.nist.gov/iad/mig/tests/sre/2008/sre08_evalplan_release4.pdf

[11] "NVIDIA CUDA™, NVIDIA CUDA C programming guide version 3.2, 11.9.2010. Online: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[12] D. B. Kirk and W. Hwu, "Programming massively parallel processors: A hands-on approach," Morgan Kaufmann, San Francisco, 2010.

[13] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", ACM/IEEE Conference on Supercomputing (SC08), 2008.

[14] N. Kumar, S. Satoor and I. Buck, "Fast parallel Expectation Maximization for Gaussian Mixture Models on GPUs using CUDA," 11th IEEE International Conference on High Performance Computing and Communications, 2009.