Accelerated Dictionary Learning with GPU/Multi-core CPU and Its Application to Music Classification

Boyang Gao, Emmanuel Dellandréa, Liming Chen
Université de Lyon, CNRS, Ecole Centrale Lyon, LIRIS, UMR5205, F-69134, France
E-mail: {Boyang.Gao, Emmanuel.Dellandrea, Liming.Chen}@ec-lyon.fr

Abstract— **K-means clustering and GMM training, as dictionary learning procedures, lie at the heart of many signal processing applications. Increasing data scale requires more efficient ways to perform this process. In this paper a new GPU and multi-core CPU accelerated k-means clustering and GMM training is proposed. We show that both methods can be concisely reformulated into matrix multiplications which allows the application of NVIDIA Compute Unified Device Architecture (CUDA) implemented Basic Linear Algebra Subprograms (CUBLAS) and AMD Core Math Library (ACML) that are highly optimized matrix operation libraries for GPU and multi-core CPU. Experimentations on music genre and mood representation and classification have shown that the acceleration for learning dictionary is achieved by factors of 38.0 and 209.5 for k-means clustering and GMM training, compared with single thread CPU execution while the difference between the average classification accuracies is less than 1%.**

Keywords-GPU acceleration; k-means; GMM; bag-of-words

I. INTRODUCTION

Digitalized audio data has exploded with the development of information technology. For example only in the on-line music store of iTunes, there have been over 20 million songs[1] available and the number is still growing fast. Such a huge amount of audio data drives the development of automatic analysis and retrieval methods. Audio data like music contains various information, however, hidden under signal appearance. Automatic analysis seeks to extract effective features to describe the essence of the signal. Many signal level or low level features for example MFCC are developed to transform waves into parameterized vectors which still preserve complete information of the signal. However, low level feature itself contains too much detail of signal's randomness so that another layer of model has to be built to depict low level feature's characteristic.

K-means or GMM based bag-of-words framework has been demonstrated to have superior ability to model low level features and has been successfully applied to many classification tasks [1][2]. Bag-of-words method first construct clusters of assembled low level features through k-means or GMM training, then provide cluster information based on the feature distribution. This procedure helps translate signal level information into meaningful audio word, that is k-means clusters and Gaussian mixtures, which convey more definite information for example genre or mood for music signal. In re-representation of low level features, k-means clustering and GMM training are the most crucial and computational expensive steps. The quality of dictionary learnt also determines for example final classification performance. With drastically increased data scale, the dictionary learning becomes computational bottle neck. When scrutinizing k-means clustering and GMM training process carefully we can find that both processes contain massive repeated calculations between feature vectors and words in dictionary, in which huge parallelizability is hidden. GPU and multi-core CPU naturally come and fit into the picture. GPU has successively accelerated many problems which contains parallelizable calculations for example [3][4]. GPU's efficiency is attributed to its multiple core structure and parallelable instruction execution ability.

There are several previous works employing GPU to accelerate k-means clustering and EM for GMM training. [5-10] have implemented EM algorithm on CUDA to train GMM. Wu in [10] also mentioned to use individual CUBLAS function to update means and variance matrix. Azhari in [7] implemented MFCC extraction in CUDA too. Gonina in [8] also provided Python interface to GPU based GMM training. Machlica in [9] used cached textual memory and carefully configured memory usage to elevate performance. In previous works user defined kernel functions undertake main computation work. However, GPU kernel function requires careful design; hardware related tuning and is hard to translate into other languages. In this paper we shows that k-means clustering and GMM training with EM algorithm can be mostly transformed into two standard matrix multiplications so that highly optimized matrix operation library CUBLAS or ACML can be employed to speedup overall calculations. The proposed computation structure can also be easy translated into other languages with BLAS support for example MATLAB and FORTRAN.

The rest of the paper is organized as follow: section II shows matrix multiplication format of k-means clustering and EM algorithm for GMM. Section III introduces performance tuning for GPU implementations. In experiment section, music genre and mood classification tasks are performed with ACML-based and CUBLAS-based implementations. The execution speed and quality of learnt dictionary are then examined. Final conclusion is drawn in section VI.

II. K-MEANS AND EM ALGORITHM IN MATRIX FORMAT

In this section k-means and EM algorithm are shown in matrix multiplication format. Block-wise training structure is also developed to fit huge data matrix into limited GPU

[1] http://www.apple.com/itunes/features/

memory, for example in NVIDIA's GTX 285 used in this paper only 750MB display memory can be used for computation. Because block-wise training divide computation into independent unit, with multiple graphic cards installed, the calculation load can be yet arranged parallelly onto multiple devices even onto local or remote GPU or CPUs, which can further boost overall computation speed. In following sections II.A and II.B data block based k-means and GMM training in matrix format are introduced in detail.

Several symbol are firstly defined: D is number of dimension for feature vectors, k-means cluster centers and Gaussian mixtures; N is the number of feature vector; M is number of k-means clusters or GMM mixtures; $\boldsymbol{X}$ is feature matrix in which each column represents one feature vector; $\boldsymbol{C}$ is k-means cluster centers matrix in which each column represents one center; $\boldsymbol{V}$ is variance matrix, in contrast to covariance matrix $\boldsymbol{\Sigma}$, $\boldsymbol{V}(i,j)$ is the variance of dimension i for cluster or mixture j. In GMM, $\boldsymbol{w} = (\alpha_1, \alpha_2 \cdots \alpha_M)^T$ denotes the priori probabilities or weights of every mixture, $\boldsymbol{M} = [\boldsymbol{\mu}_1, \boldsymbol{\mu}_2 \cdots \boldsymbol{\mu}_M]$ is the means matrix, $\boldsymbol{\Sigma} = \{\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2 \cdots \boldsymbol{\Sigma}_M\}$ is the covariance matrix set. Other symbols are defined when used.

A. *K-means in Matrix Format*

K-means clustering iteration contains two steps: cluster decision and cluster center updating. The first is to find the nearest center to which each feature vector is closed. The second is to re-estimate cluster centers according to the nearest relationship.

To measure the closeness between data set $\boldsymbol{X}$ and centers $\boldsymbol{C}$, squared Euclid distance is commonly used, defined as

$$\begin{aligned}\boldsymbol{D}_s(\boldsymbol{x}_i, \boldsymbol{c}_j) &= (\boldsymbol{x}_i - \boldsymbol{c}_j)^T(\boldsymbol{x}_i - \boldsymbol{c}_j) \\ &= \boldsymbol{c}_j^T\boldsymbol{c}_j - 2\boldsymbol{x}_i^T\boldsymbol{c}_j + \boldsymbol{x}_i^T\boldsymbol{x}_i\end{aligned}$$

In matrix format, squared distance matrix can be written as

$$\begin{aligned}\boldsymbol{D}_s &= \boldsymbol{1}\cdot\boldsymbol{c}_s^T - 2\boldsymbol{X}^T\boldsymbol{C} + \boldsymbol{x}_s\cdot\boldsymbol{1}^T \\ &= [\boldsymbol{1}, \boldsymbol{X}^T, \boldsymbol{x}_s]\cdot\begin{bmatrix}\boldsymbol{c}_s^T \\ -2\boldsymbol{C} \\ \boldsymbol{1}^T\end{bmatrix}\end{aligned}$$

where $\boldsymbol{x}_s = (\boldsymbol{x}_1^T\boldsymbol{x}_1, \boldsymbol{x}_2^T\boldsymbol{x}_2 \cdots \boldsymbol{x}_N^T\boldsymbol{x}_N)^T$ and $\boldsymbol{c}_s = (\boldsymbol{c}_1^T\boldsymbol{c}_1, \boldsymbol{c}_2^T\boldsymbol{c}_2 \cdots \boldsymbol{c}_N^T\boldsymbol{c}_N)^T$.

For the center updating step the new center matrix can be written as $\boldsymbol{C}^{new} = \boldsymbol{X}\cdot\boldsymbol{O}$, where occupation matrix $\boldsymbol{O}$ is defined as

$$\boldsymbol{O}(i,j) = \begin{cases}\dfrac{1}{R_j}, & \text{if } \boldsymbol{x}_i^{\prime}\text{s nearest center is } \boldsymbol{c}_j \\ 0, & \text{otherwise}\end{cases}$$

R_j is the number of associated feature vector to center j. Variance matrix can be computed similarly as $\boldsymbol{V}^{new} = \boldsymbol{X}_s\cdot\boldsymbol{O} - \boldsymbol{C}_s^{new}$, where $\boldsymbol{X}_s(i,j) = \boldsymbol{X}(i,j)^2$ and $\boldsymbol{C}_s^{new}(i,j) = \boldsymbol{C}^{new}(i,j)^2$.

For large data set, feature matrix $\boldsymbol{X}$ is too large to entirely load into limited display memory. Therefore large $\boldsymbol{X}$ is cut into blocks $\boldsymbol{X} = [\boldsymbol{X}^{blk_1}, \boldsymbol{X}^{blk_2} \cdots \boldsymbol{X}^{blk_B}]$. In each iteration, sub-results on $\boldsymbol{X}^{blk}$ are computed and then aggregated to form the overall new cluster centers.

The block-wise cluster center updating step is as follow. For each feature block, define occupation matrix $\boldsymbol{O}^{blk}$ as

$$\boldsymbol{O}^{blk}(i,j) = \begin{cases}1, & \text{if } \boldsymbol{x}_i^{\prime}\text{s nearest center is } \boldsymbol{c}_j \\ 0, & \text{otherwise}\end{cases}$$

For each feature block, weight feature vector $\boldsymbol{w}^{blk}$ in which w_j indicates the number of related feature vector for center j, center matrix $\boldsymbol{C}^{blk}$ and variance matrix $\boldsymbol{V}^{blk}$ are computed as

$$\begin{bmatrix}\boldsymbol{w}^{(blk)T} \\ \boldsymbol{C}^{blk} \\ \boldsymbol{V}^{blk}\end{bmatrix} = \begin{bmatrix}\boldsymbol{1}^T \\ \boldsymbol{X}^{blk} \\ \boldsymbol{X}_s^{blk}\end{bmatrix}\cdot\boldsymbol{O}^{blk}$$

The overall accumulated vector and matrices for B feature blocks are $\boldsymbol{w}^{acc} = \sum_{b=1}^{B}\boldsymbol{w}_b^{blk}$, $\boldsymbol{C}^{acc} = \sum_{b=1}^{B}\boldsymbol{C}_b^{blk}$ and $\boldsymbol{V}^{acc} = \sum_{b=1}^{B}\boldsymbol{V}_b^{blk}$. In the end of an iteration the updated cluster center j is calculated as $\boldsymbol{c}_j^{new} = \boldsymbol{c}_j^{acc}/w_j$ and variance vector $\boldsymbol{\sigma}_j^{2(new)} = \boldsymbol{\sigma}_j^{2(acc)}/w_j - \boldsymbol{c}_j^{2(new)}$, where $\boldsymbol{c}_j^{2(new)} = \left((c_{j,1}^{new})^2, (c_{j,2}^{new})^2 \cdots (c_{j,M}^{new})^2\right)^T$. Note that $\boldsymbol{V}^{blk}$ does not affect cluster center updating; therefore it is only computed in the last iteration. The matrix version k-means clustering main steps are summarized as follows.

- For each iteration
  - For each feature block $\boldsymbol{X}^{blk}$
    - Calculate $\boldsymbol{D}_s^{blk}$ with one sgemm()
    - Calculate $\boldsymbol{O}^{blk}$ from $\boldsymbol{D}_s^{blk}$
    - Calculate $\boldsymbol{C}^{blk}$ and $\boldsymbol{V}^{blk}$ with one sgemm()
  - Aggregate $\boldsymbol{C}^{blk}$ and $\boldsymbol{V}^{blk}$ to update new $\boldsymbol{C}$

B. *EM for GMM in Matrix Format*

In GMM, probability of feature vector $\boldsymbol{x}_i$ is defined as

$$p(\boldsymbol{x}_i|\,\boldsymbol{\theta}) = \sum_{m=1}^{M}\alpha_m p(\boldsymbol{x}_i|m, \boldsymbol{\theta})$$

where $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{M}, \boldsymbol{\Sigma}\}$ is the GMM parameter set and $p(\boldsymbol{x}_i|m, \boldsymbol{\theta})$ is the probability of feature vector $\boldsymbol{x}_i$ under single Gaussian mixture m, defined as

$$p(\boldsymbol{x}_i|m, \boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}_m|}}e^{-\frac{1}{2}(\boldsymbol{x}_i-\boldsymbol{\mu}_m)^T\boldsymbol{\Sigma}_m^{-1}(\boldsymbol{x}_i-\boldsymbol{\mu}_m)}$$

Like k-means, EM iteration for GMM training includes two main steps: probability calculation and parameter updating. The first step decides relation between feature vector and Gaussian mixtures in terms of probability; the second step updates GMM parameters with feature matrix according to how likely each mixture can generate observed feature vectors.

In the first step $p(\boldsymbol{x}_i|m, \boldsymbol{\theta})$ are computed. To avoid floating point number overflow, the natural logarithm of $p(\boldsymbol{x}_i|m, \boldsymbol{\theta})$ is preserved during calculation

$$ln\big(p(\boldsymbol{x}_i|m,\boldsymbol{\theta})\big) = -\frac{1}{2}(g_m + \boldsymbol{x}_i^T\boldsymbol{\Sigma}_m^{-1}\boldsymbol{x}_i - 2\boldsymbol{x}_i^T\boldsymbol{\Sigma}_m^{-1}\boldsymbol{\mu}_m)$$

where $g_m = D \cdot ln(2\pi) + ln\big(\sigma_{m,1}^2 \cdot \sigma_{m,2}^2 \cdot \cdots \cdot \sigma_{m,D}^2\big) + \boldsymbol{\mu}_m^T\boldsymbol{\Sigma}_m^{-1}\boldsymbol{\mu}_m$. In practice covariance matrix $\boldsymbol{\Sigma}_m$ is often treated as diagonal, in order to simplify computation. Let $\boldsymbol{V} = [\boldsymbol{diag}(\boldsymbol{\Sigma}_1), \boldsymbol{diag}(\boldsymbol{\Sigma}_2) \cdots \boldsymbol{diag}(\boldsymbol{\Sigma}_M)]$ denotes the GMM variance matrix; Let $\boldsymbol{P}$ denote corresponding logarithm probability matrix in which $\boldsymbol{P}(i,j) = ln\big(p(\boldsymbol{x}_i|j,\boldsymbol{\theta})\big)$, then $\boldsymbol{P}$ can be written as

$$\begin{aligned}\boldsymbol{P} &= -\frac{1}{2}(\boldsymbol{1}\cdot\boldsymbol{g}^T - 2\boldsymbol{X}^T\boldsymbol{M}_v + \boldsymbol{X}_s^T\boldsymbol{V}_r) \\ &= -\frac{1}{2}[\boldsymbol{1}, \boldsymbol{X}^T, \boldsymbol{X}_s^T]\cdot\begin{bmatrix}\boldsymbol{g}^T \\ -2\boldsymbol{M}_v \\ \boldsymbol{V}_r\end{bmatrix}\end{aligned}$$

where $\boldsymbol{g} = (g_1, g_2 \cdots g_M)^T$, $\boldsymbol{V}_r$ is the reciprocal matrix of $\boldsymbol{V}$ in which $\boldsymbol{V}_r(i,j) = 1/\boldsymbol{V}(i,j)$ and $\boldsymbol{M}_v$ is $\boldsymbol{V}_r$ weighted mean matrix in which $\boldsymbol{M}_v(i,j) = \boldsymbol{M}(i,j)\cdot\boldsymbol{V}_r(i,j)$. Then $p(j|\boldsymbol{x}_i,\boldsymbol{\theta})$ is computed according to Bayes' rule

$$p(m|\boldsymbol{x}_i,\boldsymbol{\theta}) = \frac{\alpha_m p(\boldsymbol{x}_i|m,\boldsymbol{\theta})}{\sum_{j=1}^M \alpha_j p(\boldsymbol{x}_i|j,\boldsymbol{\theta})}$$

Posterior probability is firstly computed in logarithm scale, that is $\ln\big(p(m|\boldsymbol{x}_i,\boldsymbol{\theta})\big) = \ln(\alpha_m) + \boldsymbol{P}(i,m) - \ln\,(p(\boldsymbol{x}_i|\,\boldsymbol{\theta}))$, and then converted into linear scale when needed. $\ln(p(\boldsymbol{x}_i|\,\boldsymbol{\theta}))$ are also aggregated directly in logarithm. When posterior probability is ready EM updating procedure can be performed to re-estimate GMM parameters as follow

$$\begin{aligned}\alpha_m^{new} &= \frac{1}{N}\sum_{i=1}^N p(m|\boldsymbol{x}_i,\boldsymbol{\theta}) \\ \boldsymbol{\mu}_m^{new} &= \frac{\sum_{i=1}^N p(m|\boldsymbol{x}_i,\boldsymbol{\theta})\boldsymbol{x}_i}{\sum_{i=1}^N p(m|\boldsymbol{x}_i,\boldsymbol{\theta})} \\ \boldsymbol{\Sigma}_m^{new} &= \frac{\sum_{i=1}^N p(m|\boldsymbol{x}_i,\boldsymbol{\theta})(\boldsymbol{x}_i - \boldsymbol{\mu}_m^{new})(\boldsymbol{x}_i - \boldsymbol{\mu}_m^{new})^T}{\sum_{i=1}^N p(m|\boldsymbol{x}_i,\boldsymbol{\theta})}\end{aligned}$$

Let $\boldsymbol{P}_p$ denote posterior probability matrix, in which $\boldsymbol{P}_p(i,j) = p(j|\boldsymbol{x}_i,\boldsymbol{\theta})$ and let $\boldsymbol{O}$ denote the occupation matrix for all mixtures, in which

$$\boldsymbol{O}(i,j) = \frac{p(j|\boldsymbol{x}_i,\boldsymbol{\theta})}{\sum_{i=1}^N p(j|\boldsymbol{x}_i,\boldsymbol{\theta})}$$

The EM updating formulas then can be written as

$$\begin{aligned}\boldsymbol{w}^{\boldsymbol{T}} &= \frac{1}{N}\boldsymbol{1}^T\cdot\boldsymbol{P}_p \\ \boldsymbol{M} &= \boldsymbol{X}\cdot\boldsymbol{O} \\ \boldsymbol{V} &= \boldsymbol{X}_s\cdot\boldsymbol{O} - \boldsymbol{M}_s\end{aligned}$$

where $\boldsymbol{X}_s$ is the squared data matrix in which $\boldsymbol{X}_s(i,j) = \boldsymbol{X}(i,j)^2$ and $\boldsymbol{M}_s$ is the squared mean matrix of $\boldsymbol{M}$ just updated.

To deal with large feature matrix $\boldsymbol{X}$ the similar block-wise scheme as for k-means is used. For each feature block, weight vector, mean matrix and variance matrix are computed as

$$\begin{bmatrix}\boldsymbol{w}^{(blk)T} \\ \boldsymbol{M}^{blk} \\ \boldsymbol{V}^{blk}\end{bmatrix} = \begin{bmatrix}\boldsymbol{1}^T \\ \boldsymbol{X}^{blk} \\ \boldsymbol{X}_s^{blk}\end{bmatrix}\cdot\boldsymbol{P}_p^{blk}$$

The overall accumulated matrices for B feature blocks are $\boldsymbol{w}^{acc} = \sum_{b=1}^B \boldsymbol{w}_b^{blk}$, $\boldsymbol{M}^{acc} = \sum_{b=1}^B \boldsymbol{M}_b^{blk}$ and $\boldsymbol{V}^{acc} = \sum_{b=1}^B \boldsymbol{V}_b^{blk}$. The updated GMM parameters are finally calculated as weights $\boldsymbol{w}^{new} = \boldsymbol{w}^{acc}/N$, mean $\boldsymbol{\mu}_j^{new} = \boldsymbol{\mu}_j^{acc}/\alpha_j^{acc}$ and variance $\boldsymbol{\sigma}_j^{2(new)} = \boldsymbol{\sigma}_j^{2(acc)}/\alpha_j^{acc} - \boldsymbol{\mu}_j^{2(new)}$, where $\boldsymbol{\mu}_j^{2(new)} = \big((\mu_{j,1}^{new})^2, (\mu_{j,2}^{new})^2 \cdots (\mu_{j,M}^{new})^2\big)^T$. The matrix version main steps are summarized as follow.

- For each iteration
  - For each feature block $\boldsymbol{X}^{blk}$
    - Calculate $\boldsymbol{P}^{blk}$ with one sgemm()
    - Calculate $\boldsymbol{P}_p^{blk}$ from $\boldsymbol{P}^{blk}$
    - Calculate $\boldsymbol{w}^{(blk)T}$, $\boldsymbol{M}^{blk}$ and $\boldsymbol{V}^{blk}$ with one sgemm()
  - Aggregate $\boldsymbol{w}^{(blk)T}$, $\boldsymbol{M}^{blk}$ and $\boldsymbol{V}^{blk}$ to update new GMM parameters

III. Performance Tuning

ACML based implementation for k-means and GMM is quite straight forward. To take full advantage of multi-core system, calculation of $\boldsymbol{O}^{blk}$ and $\boldsymbol{P}_p^{blk}$ is implemented with OpenMP[2] which takes charge of dividing for-loop into multiple threads. Similarly on GPU version, calculation of $\boldsymbol{O}^{blk}$ and $\boldsymbol{P}_p^{blk}$ is implemented with CUDA kernel function. However, kernel functions need to be well tuned to achieve peak performance.

CUDA adopts single instruction multiple threads (SIMT) architecture. 32 threads form one warp which is unit of GPU thread management. When GPUs are idle thread manager tries to arrange one or half warp of threads to execute at same time with the same instruction. Therefore dividing problem into threads block whose dimension is multiple of warp will run faster, otherwise some threads will be wasted when computing the edge cases of problem.

Memory in CUDA can roughly be divided into local and global category. Local memory is faster, however, of too limited size for example GTX285 has only 16KB. Local memory is only accessible within thread block and may cause bank conflicts. Despite of the limitations, local memory should be used whenever shared data exists within thread block in order to boost memory throughput. Compared with local memory, global memory is larger, up to hundreds mega bytes, yet slower. For our tasks, most memory access happens in global domain and for computation ability less than 2.0 the

[2] http://openmp.org/wp/

global memory does not have cache (GTX285 has computation ability of 1.3). Therefore memory access pattern affects computation throughput tremendously. In many scenarios memory access becomes the computational bottle neck. In CUDA, global memory access is divided into memory transactions which can load up to 16 bytes at one time. Memory access scheme and SIMT architecture determines coalesced memory access is the fastest way which requires neighbor threads in a block access neighbor data. Although threads block can be configured in 2D or 3D, the memory neighborhood is still confined in linear due to lack of cache.

TABLE I. EXECUTION TIME AND ACCURACY OF DIFFERENT IMPLEMENTATION FOR MUSIC GENRE CLASSIFICATION.

# Centers/# Mixtures = 2048 # Vectors = 2.7M # Dimension = 39	K-means			GMM		
	time per iteration (sec)	speed-up (times)	GTZAN genre (%)	time per iteration (sec)	speed-up (times)	GTZAN genre (%)
1 thread CPU	278.2	1.0	n/a	2,288.4	1.0	n/a
32-cored CPU Yael(INRIA)	37.2	7.5	68.1	153.2	14.9	78.0
32-cored CPU (this paper)	7.3	38.0	69.6	70.1	32.6	78.7
240-cored GPU	9.1	30.4	69.5	10.9	209.5	77.8
UWB[9]				9.1*		

*only kernel execution time counted for GMM with 2048 mixtures trained by 3.125M 40-dimensional vectors

To determine minimum distance for each feature vector from matrix $\boldsymbol{D}_s$, we have tested calculation performance on $\boldsymbol{D}_s$ and on its transpose $\boldsymbol{D}_s^T$. It can be observed that same kernel function runs 7 times faster on $\boldsymbol{D}_s$ than on $\boldsymbol{D}_s^T$. In former case the computation task is parallelized on each vector so that speed gain is obtained from coalesced memory access.

Another example is to compute posterior probability matrix $\boldsymbol{P}_p$ from $\boldsymbol{P}$. Thread block dimension of (1, 256) is 30% faster than of (16, 16). This is also due to coalesced memory access. Because although take exponential for each matrix element is a 2D parallelable operation, the matrix data is actually stored linearly in global memory. Therefore (16, 16) configuration causes non-consecutive global memory access whereas (1,256) does.

In calling CUBLAS SDK functions, using non-transposed matrix function call whenever possible can improve the execution speed, because transpose operation is expensive and from NVIDIA's profiler we can find when cublasSgemm() is performed on matrix who needs to transpose the multiplication is actually performed by a CUDA kernel function, which is slower. In experiments we observe 4 times speed-up when calling cublasSgemm() with non-transposed arguments.

Merging matrix operations whenever possible, like multiplication, can save overhead caused by function calling and repeated memory access. For example when block mean and variance matrix updating is combined into one matrix multiplication 10% execution time is saved.

IV. MUSIC GENRE AND MOOD CLASSIFICATION

To evaluate speed and quality of our implementations, music genre and mood classification have been conducted on both GPU and multi-core CPU systems.

A. Experiment setups

The single thread implementation based on clapack[3] ran on a pc with Intel® Core™ i7-940 2.93GHz as a baseline. The implementation based on ACML ran on a server with 4 8-cored AMD Opteron™ Processor 6128 2GHz. The implementation based on CUBLAS ran on 240-cored NVIDIA GTX 285. Yael[4] is also tested on the server as a benchmark for multi-threading implementation.

GTZAN [11] data set is used for genre classification. GTZAN contains 10 genres of music. Each genre contains 100 30-sec segments. MFCC with delta and delta delta features are extracted. MFCC analysis window shift is 10ms long, resulting 100 39-dimensional feature vectors per second. To test k-means clusters and GMM quality 10-fold cross validation is performed. In each fold K-means and GMM with 2048 clusters and mixtures are trained by 2.7 million 39-dimentional feature vectors.

Music mood classification is performed on the dataset provided by Xiao et al. [12] consists of 416 16s long pure classical music segments, manually divided into 4 moods as shown in Table II. OpenSMILE [13] large emotional feature set is used which contains 57 low level features plus delta and delta delta. In mood classification experiments, 5 times 2-fold cross-validation is performed in order to compare with result in previous literature. Before dictionary learning, 171 dimensional feature vectors are normalized by training set global mean and standard derivation to avoid different scale effect. In each fold, GMM with 1024 mixture is trained by 0.33 million 171-dimentional feature vectors for 1024 mixtures.

After dictionary learning, low level features of each music segments are converted to dictionary words histogram and normalized. The normalized histograms as final feature vectors are used to train one-versus-others SVM classifier. To compare the mood classification performance, standard OpenSMILE low level feature with statistic functions are extracted and used to train SVM as benchmark. The running time and accuracy results of the two experiments are shown in Table I and Table IV-VI.

[3] http://www.netlib.org/clapack/

[4] https://gforge.inria.fr/projects/yael

TABLE II. CLASSICAL MUSIC DATASET MOOD DISTRIBUTION

	Anxious	Content	Depressed	Exuberant
# music	81	124	120	91

B. *Results and Analysis*

From Table I, we can find that 1) high performance library based implementations proposed in this paper achieve up to 5 times faster than multi-threading based implementation (Yael). 2) GPU based implementation executes 5 times faster than multi-core CPU based one on GMM training whereas multi-core version outperforms GPU on k-means clustering by 22% in terms of speed. GPU based k-means is slower than multi-core CPU in that CPU version executes $O(ND)$ operations for $\boldsymbol{C}^{blk}$ calculation while GPU version actually executes matrix multiplication containing $O(NDM)$ operations. We choose matrix multiplication for $\boldsymbol{C}^{blk}$ calculation in GPU because $O(ND)$ operations consume as twice time as $O(NDM)$ matrix multiplication on GPU architecture. This abnormal phenomenon is due to random memory access pattern of $O(ND)$ operations on GPU. Therefore in this scenario matrix multiplication is the optimal but still slower way for GPU.

In [9] Machlica et al. reported a faster result of comparable data set, however, the duration profile in [9] only recorded kernel function's running time without data IO and data preparation duration. If only GPU execution duration is considered, our method consumes 7 seconds compared with 9.1 in [9]. From genre classification accuracy column in Table I, we can find that the quality of dictionary trained by CPU is little better than GPU by less than 1% in terms of average classification accuracy. This is due to the less floating point error of CPU and double precision floating point numbers used during summation procedure on CPU.

From Table III, we can find that with the same number of mixture; the same types of GPU and comparable data scale in [7], our CUBLAS based implementation is 10 times faster than CUDA kernel implementation described in [7]. From Table IV-VI we can find 1024 dimensional bag-of-words histogram features outperforms standard 6552 dimensional OpenSMILE emotional features by 3% in terms of average classification accuracy.

TABLE III. RUNNING TIME PER ITERATION FOR MOOD GMM TRAINING

	32-cored CPU	240-cored GPU	Azhari [7]
time (sec)	9.6	2.9	30.0

TABLE IV. CUBLAS MOOD CLASSIFICATION CONFUSION MATRIX

%	Anx	Con	Dep	Exu
Anx	81.0 ±8.9	1.2 ±2.0	1.0 ±1.7	16.8 ±7.4
Con	0.0 ±0.0	91.1 ±2.9	8.7 ±2.9	0.2 ±0.5
Dep	0.0 ±0.0	8.8 ±3.7	90.5 ±3.2	0.7 ±1.5
Exu	10.2 ±4.7	3.8 ±2.0	0.0 ±0.0	86.0 ±4.2
Average	87.2			

TABLE V. ACML MOOD CLASSIFICATION CONFUSION MATRIX

%	Anx	Con	Dep	Exu
Anx	81.0 ±7.8	1.2 ±2.0	1.2 ±1.7	16.5 ±6.2
Con	0.0 ±0.0	91.9 ±3.0	8.1 ±3.0	0.0 ±0.0
Dep	0.0 ±0.0	8.3 ±3.5	91.0 ±3.2	0.7 ±1.5
Exu	11.3 ±5.3	3.6 ±1.8	0.0 ±0.0	85.1 ±5.0
Average	87.3			

TABLE VI. OPENSMILE MOOD CLASSIFICATION CONFUSION MATRIX

%	Anx	Con	Dep	Exu
Anx	85.5 ±7.1	2.5 ±1.6	0.5 ±1.0	11.5 ±6.8
Con	5.5 ±4.7	86.9 ±4.2	6.3 ±3.0	1.3 ±2.7
Dep	9.0 ±6.3	6.2 ±3.9	83.5 ±7.9	1.3 ±3.0
Exu	17.6 ±5.0	2.2 ±1.4	0.2 ±0.7	80.0 ±5.9
Average	84.0			

V. CONCLUSION

In this paper we have proposed to use GPU and multi-core CPU to accelerate bag-of-words method especially dictionary learning of k-means clustering and GMM training. To employ high performance matrix operation library of ACML and CUBLAS we propose to reformulate k-means and EM algorithm into matrix multiplications, which is also convenient to implement with other languages for example MATLAB and FORTRAN. Experiments on music genre and mood classification tasks show that the proposed implementations achieve 38 to 209 times acceleration, compared with single threaded CPU version. 240-cored GPU can run up to 5 times faster than 4 8-cored CPUs with just less 1% performance decline.

As k-means and GMM training are widely used for learning dictionaries in many applications, we will propose to the community our optimized implementation of these procedures presented in this paper within an open-source toolbox that will be freely available to the public.

ACKNOWLEDGMENT

This work is partly supported by the French ANR under the project VideoSense ANR-09-CORD-026.

REFERENCES

[1] H. Jégou, M. Douze and C. Schmid, "Histopathology image classification using bag of features and kernel functions", *Artificial Intelligence in Medicine*, pp. 126-135, Springer, 2009.

[2] Y. Zheng, H. Lu, C. Jin, and X. Xue, "Incorporating spatial correlogram into bag-of-features model for scene categorization", *Proceeding of Asian Conference on Computer Vision*, 2009.

[3] J. Kong, "GPU accelerated face detection", *Proceeding of Intelligent Control and Information Processing (ICICIP)*, pp.584-588, 2010.

[4] K. van de Sande and T. Gevers, "Empowering visual categorization with the GPU," *IEEE Transactions on Multimedia*, Volume 13 (1), page 60-70, 2011.

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

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

[7] M. Azhari and C. Ergün, “Fast Universal Background Model (UBM) Training on GPUs using Compute Unified Device Architecture (CUDA),” *International Journal of Electrical & Computer Sciences IJECS-IJENS*, vol. 11, no. 4, pp. 49-55, 2011.

[8] E. Gonina, “Fast Speaker Diarization Using a Specialization Framework for Gaussian Mixture Model Training,” master thesis, University of California Berkeley, 2011.

[9] L. Machlica, J. Vanek, and Z. Zajic, “Fast Estimation of Gaussian Mixture Model Parameters on GPU Using CUDA”, *Proceeding of 12th International Conference on Parallel and Distributed Computing Applications and Technologies*, no. 1, pp. 167-172, 2011.

[10] K. Wu, Y. Song and L. Dai, “CUDA-Based Fast GMM Model Training Method and Its Application,” *Journal of Data Acquisition & Processing*, vol. 27, no. 1, pp. 85-90, 2012.

[11] G. Tzanetakis and P. Cook, “Music genre classification ofaudio signals”, *IEEE Transactions on Speech and Audio Processing*, 10(5):293–302, 2002.

[12] Z. Xiao, E. Dellandréa, W. Dou and L. Chen, “What is the best segment duration for music mood analysis ?,” in *Proc. International Workshop on Content-Based Multimedia Indexing (CBMI)*, London, 2008.

[13] F. Eyben, M. Wöllmer and B. Schulle, “OpenSMILE - the munich versatile and fast open-source audio feature extractor,” in *Proc. ACM Multimedia (MM)*, Florence, 2010.