GMM based Fisher vector calculation on GPGPU

Erik Bodzsár Bálint Daróczy István Petrás András A. Benczúr
Data Mining and Web search Research Group, Informatics Laboratory
Computer and Automation Research Institute of the Hungarian Academy of Sciences
{erikbodzsar, daroczyb, petras, benczur}@ilab.sztaki.hu

Abstract

We describe an accurate yet very fast implementation of a visual word generation method by using general purpose graphical processors (GPUs). Visual words have recently proved to be a key tool in image classification. Best performing Pascal VOC and ImageCLEF systems use Gaussian mixtures or k-means clustering to define visual words based on the content-based features of points of interest. In many cases, Gaussian Mixture Modeling (GMM) is more accurate but computationally expensive compared to other methods, sometimes taking days to compute for the standard research tasks. We reach a 14-times speedup over a well-tuned sequential GMM implementation. We measure the accuracy of our methods over the Pascal VOC 2007 and give results comparable to the best teams with reduced computational time. Since most image processing components already have GPU implementations, we believe our results make large scale image classification with Fisher vectors scalable with the help of graphics processors.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: H.3.1 Content Analysis and Indexing; H.3.3 Information Search and Retrieval; H.3.4 Systems and Software; H.3.7 Digital Libraries; H.2.3 [**Database Management**]: Languages—*Query Languages*

General Terms

Measurement, Performance, Experimentation

Keywords

image processing, visual bag of words, HOG, Color moments, SIFT, Gaussian mixtures, Fisher kernel, Maximum Likelihood

1 Introduction

Image classification consists of assigning one or multiple labels to an image based on its semantic content. Although much progress has been made, in particular in the context of the PASCAL VOC [7] and ImageCLEF evaluation campaigns [20], the problem remains challenging. Several approaches model the distribution of low level features: bag of keypatches [5] or bag of visual terms [18], irrespective of their absolute or relative location. Categorization requires the estimation of the visual vocabulary, which is typically done by k-means [25, 5, 28], Gaussian Mixture Modeling (GMM) [23], mean-shift [14] or LDA[10]. As the starting point of our experiment, Perronnin and Dance [22] introduce Fisher Kernels over a GMM, a method that proved to be very powerful especially for concept type classes, including best performance at the ImageCLEF 2008 classification task [1].

In this paper, our main goal is to show how to add computational power to a state of the art image classification system with the help of graphics processors. As the scale of the research collections increase, researchers can less and less afford to spend days of CPU time on refined analysis and have to use simpler methods as fallback. For this purpose, we make our source code along with preprocessed visual classification data available for research purposes[1].

Commodity computer graphics chips, also known as (GP)GPUs, give probably the best available computational power for the price [21]. Not only they are fast but also accelerating quickly and promise the future for high performance computing. This is in particular true for image processing, the natural area of use for the graphics processors. While most image processing steps are already implemented on the GPU, global analysis of an image corpus does not translate to a graphics processor in the same smooth manner. The only GPU based implementation we found is a k-means based classification system [29]. As examples of GPU applications related to image classification, for matrix multiplication [9] and other matrix operations [2, 12, 15]; nearest neighbor search [11, 3]; Fourier transform [19, 26]. For a general survey see [21]. Problems of even seemingly different nature port well to GPU including graph algorithms [13] and in particular cuts [30] that could be applied for image segmentation.

Our main result speeds up GMM over graphical processors to make it feasible for processing large image collections. We believe that GMM is a key component in high quality visual word generation: both our current experiments and ImageCLEF results [1, 27] indicate that GMM is superior to k-means for concept type classes. It has been shown the k-means based methods outperform the LDA based systems[24]. As a workaround in many recent experiments, the number of mixtures had to be lowered that resulted in quality deterioration. For example at ImageCLEF 2010, a k-means based solution with very large k performed best [28]—note that the computational effort is quadratic in k that makes a good quality k-means based solution very expensive. In addition, our experiments also show that the same quality requires much higher dimensionality for k-means than for GMM, thus the GMM-based model is much more compact than a k-means based counterpart.

Although the running time of GMM has been improved by a number of heuristics, it remains by far the most time consuming step of image classification on a CPU. Among the speedup heuristics, perhaps the most important one is the hierarchical fast scoring procedure [23]. By the figures of [23], we may estimate the running time in the order of a day for 30,000 images and 128 dimensions, with all other processing costs being negligible. They use two heuristics to reduce the computational cost of GMM: they hierarchically increase the number of mixture components as the iterations proceed [17, 31] and use only the best fitting few components for each data point. We compare our GPU implementation to a very fast hierarchical CPU implementation with additional heuristics.

To illustrate GPU capabilities in the learning process of image classification, we experiment with a system believed to be most powerful, based on GMM modeling and Fisher kernels [22]. While our primary concern is running time and speedup with respect to an efficient CPU implementation, we describe a system that performs comparable to the current best ones at PASCAL VOC 2007 dataset.

1.1 Algorithms for Gaussian Mixture Modeling

Our GMM procedure is based on the standard expectation maximization (EM) algorithm shown in Algorithm 1 that we briefly describe here. The algorithm assumes that the D dimensional data points originate from K Gaussian distributions (denoted by $\mathcal{N}_1, ..., \mathcal{N}_k$) in the following way: first, we choose a random k according to the distribution $\{P_i\}_{i=1}^{K}$, and then sample a random point from the distribution $\mathcal{N}_k$. We think of these distributions as clusters.

1.2 Limitations of existing CUDA GMM implementations

Since most steps of an expectation maximization algorithm are "embarassingly" parallel, it is no surprise that there are already basic CUDA GMM implementations available. Next we review the properties of these algorithms and show that they are limited in some of the aspects below required for the image classification task:

[1] `http://datamining.sztaki.hu/?q=en/GPU-GMM`

- Data size must be allowed to exceed the GPU memory. Currently memory in a GPU chip is limited below the point where sampling already deteriorates quality. (see Figure 4)
- Dimensionality must be allowed to be in the range at leasts a hundred as, even after dimensionality reduction, lower dimensionality will result in lower classification quality.
- Numerical precision must be very accurate. In occupancy probability computations with dimension more than a hundred the 64-bit precision is insufficient.

Harp[2] provides a Matlab code with a C kernel that has two inherent limitations for our purposes:

- It uses 32-bit precision that would always underflow in the scale of an image classification task.
- It keeps all data in GPU memory, again seriously limiting the applicability as increasing the GPU memory becomes very expensive beyond certain point that is only slowly increasing with new hardware availability.

Another implementation is given by Pangborn[3] that already contains some of the basic techniques we require, including logarithmic summation, although only limited to likelihood and not the component weights. It uses 64-bit precision, which is insufficient for image classification, as we show in this paper. The memory limitation is resolved by allowing the use of more than one GPUs. Still, there is a built-in limit of 24 dimensions while our task is in the range of hundreds. A similar implementation is described in [16] who show experiments with up to 32 mixtures for 32-dimensional data points in the order of 100,000. Numerical accuracy is not addressed in these results.

Algorithm 1 The GPU GMM algorithm
Input: data points $\{x_i\}_{i=1}^N$, dimension D, mixture number K.
Output: $\{P_i\}_{i=1}^K$, $\{\mu_i\}_{i=1}^K$, $\{\sigma_i\}_{i=1}^K$, where $\mathcal{N}_i$ is normally distributed with parameters (μ_i, σ_i) and σ_i is assumed to be diagonal.

Initialize the Gaussian distributions with random parameters
repeat
  for all n and k **do**
    Expectation: Compute likelihood $p_{nk} = \frac{f_k(x_n)P_k}{\sum_{i=1}^K f_i(x_n)P_i}$ where f_i is the density of $\mathcal{N}_i$
  for all k and d **do**
    Maximization: compute $P_k = \frac{\sum_{n=1}^N p_{nk}}{N}$, $\mu_{kd} = \frac{\sum_{n=1}^N p_{nk}x_{nd}}{\sum_{n=1}^N p_{nk}}$ and $\sigma_{kd}^2 = \frac{\sum_{n=1}^N p_{nk}(x_{nd}-\mu_{kd})^2}{\sum_{n=1}^N p_{nk}}$
until until converge

2 The CUDA implementation of the GMM algorithm

Next we give the details of the algorithmic and numerical methods used in the implementation of the EM algorithm used for GMM clustering of large data sets. In our implementation we have to enhance the standard EM algorithm in multiple places, mainly to make the best use of the CUDA architecture, and also to avoid some numerical problems that occur with large data sets.

The first set of our solutions are designed to handle GPU memory limitations. First of all, we recompute most of the intermediate values when needed instead of storing them. Another important consideration is that a large data set can be cut up into smaller pieces, and the GPU can simultaneously work with one piece while loading the next one. In this way, computation hides copying time and working with data sets too large for GPU memory is not disproportionately slow (see Section 2.2).

We also resolve the known vulnerability of the EM algorithm to underflow [23]. It may happen that we round down all elements of a many-term sum to zero that results in zero variance for a component. Our

[2] `http://andrewharp.com/gmmcuda`
[3] `http://cyberaide.googlecode.com/svn/trunk/project/biostatistics/projects/mixtureModelMultiGPU/`

workaround solutions in Section 2.1 are based on the use of use the logarithm instead of the values, but this standard solution is not sufficient in itself. We implement the summation of a large number of exponentials in a novel way: roughly speaking, we sum values of different magnitudes separately.

The effect of a component that loses all of its elements is the same as an underflow: we end up with zero variance. In this case we may either delete the component or restart it with random parameters. We ruled out the latter approach and delete components instead, for the following reason:

- If we restart with large variance, we destroy a solution close to converge;
- If we restart with small variance, the new component will likely become empty again.

2.1 Ranged logarithmic summation

A naive GMM implementation would frequently underflow when computing σ_{kd}^2 and that will result in a division by zero in the next iteration. The standard solution to this problem involves computing with logarithms:

$$\log(p_{nk}) = \log(f_k(x_n)) + \log(P_k) \tag{1}$$

$$- \log \sum_{i=1}^{K} \exp(\log(f_i(x_n)) + \log(P_i)) \tag{2}$$

$$\log(P_k) = \log \sum_{n=1}^{N} \exp \log(p_{nk}) - \log N \tag{3}$$

$$\mu_{kd} = \frac{\sum_{n=1}^{N} \exp(\log(p_{nk}) + \log(x_{nd}))}{\sum_{n=1}^{N} \exp \log(p_{nk})} \tag{4}$$

$$\sigma_{kd}^2 = \frac{\sum_{n=1}^{N} \exp(\log(p_{nk}) + 2\log(x_{nd} - \mu_{kd}))}{\sum_{n=1}^{N} \exp \log(p_{nk})} \tag{5}$$

The above formulas involve expressions of the form $\sum \exp a_i$ where a_i is negative with large absolute value, e.g. $a_i \approx \log(p_{nk})$, therefore the risk of the underflow due to $\exp a_i \approx 0$ remains high. Still standard trick, we subtract $\nu_k := \max_{i=1}^{N} - \log(p_{nk})$, the maximum absolute value. We may rewrite the formulas such as

$$\log(P_k) = \log \sum_{n=1}^{N} \exp(\log(p_{nk}) + \nu_k) - \log N. \tag{6}$$

We avoid underflow within the summation since for all n at least one of $\log(p_{nk}) + \nu_k = 0$.

Unfortunately, neither of the standard tricks resolve the underflow, especially in the limited precision arithmetic of the GPU. When computing the numerator of (5), plenty of the n negative values $\log(x_{nd} - \mu_{kd})$ may be of very large absolute value. Although these values are small, there may be so many of them that we loose precision if we discard all of them.

In order to compute $\sum \exp a_i$ for a very large number of i with $\exp a_i \approx 0$, we apply a new numerical trick by splitting the sum into intervals of ranges and add the maximum absolute value separate for the ranges,

$$\sum \exp a_i = \sum_{j \in J} \sum_{a_i \in I_j} \exp a_i = \sum_{j \in J} \sum_{a_i \in I_j} \exp(a_i - m_j) \exp m_j \tag{7}$$

$$= \sum_{j \in J} \exp \left(\log \left[\sum_{a_i \in I_j} \exp(a_i - m_j) \right] + m_j \right), \tag{8}$$

where the I_j are disjoint intervals and $m_j = \max_{a_i \in I_j} a_i$. In this way, each innermost sum will have at least one term equal to 1.

We show that if we consider exponentially decreasing ranges, then it suffices to split the computation into the order of $\log N$ ranges for a relative error ε in the numerical computations. Practically for our problems, the number of ranges will be a sufficiently small parameter value so that the procedure fits into GPU local memory.

To prove, let us define I_j such that $m_j = -cj$ for some $c > 0$. By applying (6), we may assume that a_i are normalized such that $a_i < 0$. The sum of exponentials in interval ℓ is at most $N \exp(-c\ell)$. This is smaller than $\varepsilon/2$ if

$$\ell > \frac{\log(2N)}{c} - \frac{\log \varepsilon}{c}.$$

Since by using (6), we may assume $\sum \exp a_i \geq 1$, the relative error is also less than $\varepsilon/2$. Hence we use only $O(\log N)$ intervals, as required.

Finally we consider the error we incur in each of the intervals. Let the smallest number that can be represented in the GPU arithmetic be p. Since the smallest value in an interval is e^{-c}, the relative error due to rounding is at most e^{-c}/p. To guarantee that the relative error is not greater than $\varepsilon/2$, c should satisfy $c > \log(p/\varepsilon)$. The relative error is hence bounded by ε if we use $O(\log N)$ intervals where the constant depends also on the required relative error ε and the smallest number that can be represented in the (32 or 64-bit) GPU arithmetic p.

2.2 Data storage

The size of storage space needed on the GPU for the data does not necessarily depend on N, because the data can be split into multiple pieces, and we can do the computations in every step of the algorithm piecewise: the GPU can simultaneously work with piece i while loading piece $i + 1$ to GPU memory. Experiments show that while this method does produce some slowdown, part of the copying time is successfully hidden by the computations. For example, one iteration over a 900MB data set fully in GPU memory takes 165 seconds; cutting it up into 200MB parts increases the running time to 400 seconds, but using concurrent copy and execution reduces this to 320 seconds. We note that even if a GPU does not support concurrent copy and execution, we observe that the cost of copying the pieces of a large data set over and over is still acceptable.

2.3 The implementation

In one iteration of our implementation, the parameters of the different Gaussians are computed sequentially. Each Gaussian is processed by the whole GPU, and the parallel architecture is exploited for computing large sums. The data is divided evenly among blocks, and the partial sums computed by each block are summed after each block is finished. When summing using intervals as described in Section 2.1, each thread has its own set of intervals. This avoids concurrent accesses to the same interval, but requires shared memory proportional to the number of threads. When all threads are finished, the individual interval sums are merged.

Although the natural parallelization of the EM algorithm would involve computing the parameters of the different Gaussians with different blocks, it is easy to see that this is infeasible because it would require us to store every p_{nk} in GPU memory. Instead we recompute the likelihood $\log p_{nk}$ on demand from the values

$$d_n = \log \sum_{i=1}^{K} \exp\left(\log(f_i(x_n)) + \log(P_i)\right) \tag{9}$$

for every n. Using the values d_n, the $\log p_{nk}$ values for a given k can be computed locally whenever needed. This reduces the storage needed for $\log p_{nk}$ from $N \cdot K$ to N, and a given $\log p_{nk}$ value can still be calculated in $O(1)$ time.

The final form of one iteration is described in Algorithm 2.

As we have already mentioned, large data sets can be handled by concurrent copy and execution. The only requirement is that the data set should fit in the memory of the host computer. The algorithm only contains one hard limitation: the number of intervals used for numerically correct summation should not

Algorithm 2 The GPU GMM algorithm

Compute d_n for all n and ν_k for all k in one kernel call (see Sections 2.2 and 2.1)
for $1 \leq k \leq K$ **do**
  Compute $\log(p_{nk})$ for all n
  Compute $\log(P_k)$,μ_k,σ_k^2 with the methods described in Section 2.1
  if $\sigma_{kd}^2 = 0$ for some d **then**
    Delete cluster k

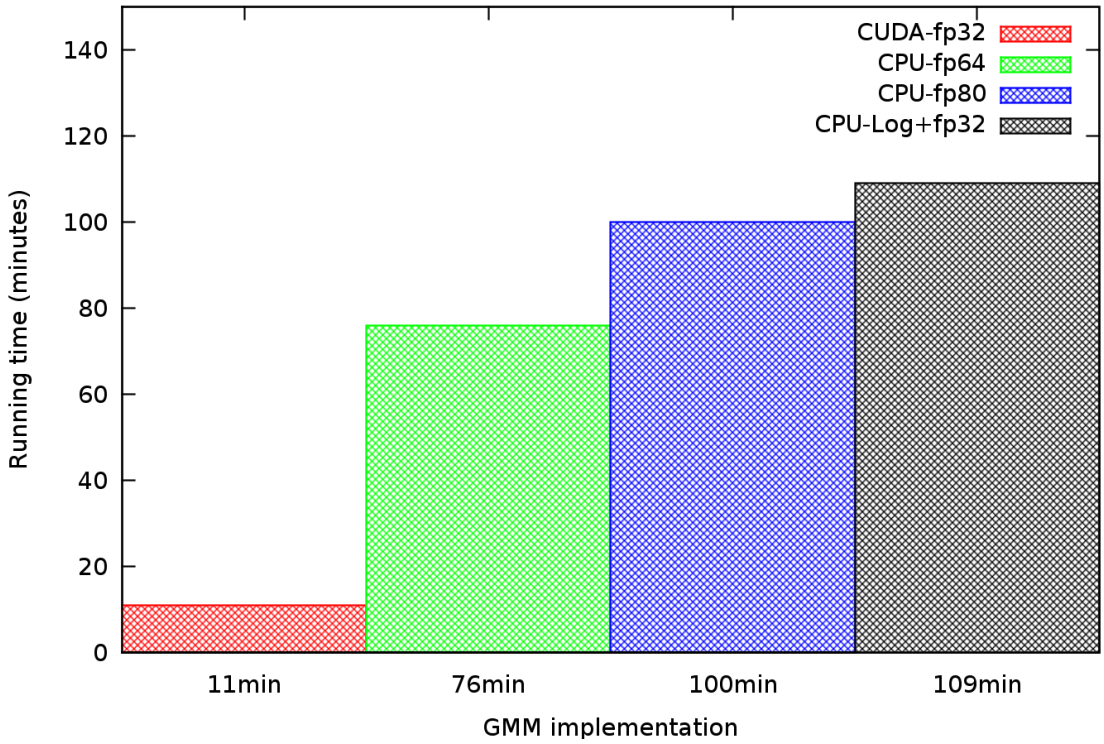

Figure 1: Running times depending on the number of Gaussians, data size, dimension

be too large, because for efficient parallelization, every thread uses its own set of intervals. But the number of intervals needed is usually sufficiently small, as shown in Section 2.1.

2.4 The hierarchical GMM algorithm

We use a hierarchical GMM procedure that is an improved version of [23], both for the CPU and the GPU implementation. Initially we model the full set of points with two Gaussians. After completing the iterations of the training procedure, we split the training set into two subsets depending on which cluster they belong to. Then we train again a GMM on each subset a with two Gaussians. We repeat this procedure until we reach the predefined number k of Gaussians. Finally we compute one EM step on the whole training set using all Gaussians.

2.5 Fisher vector calculation

Fisher vectors can be computed parallel in $k \cdot D$ independent calculations where D is the dimension of the low-level features and k is the number of Gaussians. If neither the dimension nor the number of clusters is more than the maximal number of threads for a block, the computational time depends only on the number of low-level features.

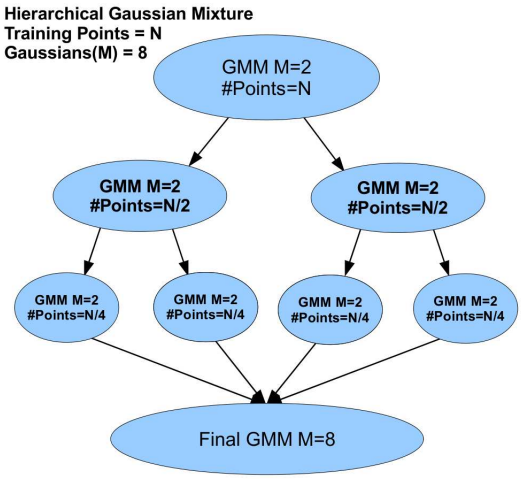

Figure 2: The idea of a hierarchical GMM procedure.

3 Experiments

We perform our experiments by using the Pascal VOC 2007 data set [6]. The Pascal VOC 2007 task uses 5011 training images and a test set with 4952 images, each image annotated manually into predefined object classes such as cat, bus, person or aeroplane.

3.1 Architecture

We conducted our experiments on a 2x1.6GHz quad-core multiprocessor Xeon box with 96GB RAM. The GPU used in the experiments was a GeForce GTX 285, with 30 multiprocessors and 2 GB global memory with concurrent copy and execution support used extensively in the GMM implementation.

3.2 Running times

The GMM task for the VCDT07 training data involved clustering nearly 700,000 data points into 128 clusters in 324 dimensions. We ran 20 iterations of the CUDA algorithm in 48 minutes. However, a large fraction of this time was spent moving and normalizing the input data; as mentioned in Section 2.2, one iteration of the algorithm took around 165 seconds (see Fig. 3). The sequential compilation of our algorithm completed the same task in 77 hours on a single core of the Xeon CPU. Notice that even if we could fully utilize a multi-core processor and the 77 hours could be divided by the number of cores, the speedup for the same hardware prize is very large. The hierarchical GMM took 20 hours to complete on the same CPU. Experiments on the same data set have shown that the running time is perfectly linear in the parameters of the GMM, except for very small dimensions where the GPU is not fully utilized. The Fisher vector calculation is also a high computational algorithm. Our CUDA implementation finished the Fisher vector calculation for the SIFT features in 92 minutes while on a single core of the CPU it took 32 hours.

3.3 Feature generation and modeling

Our approach is based on gray channel Histogram of Oriented Gradients (HOG), Scale-Invariant Feature Trainsform (SIFT), Linear Binary Pattern (LBP) and Colour moments (COL) representations. We use feature vectors to describe the visual content of an image by approximately 9000 descriptors per image per modality (we used dense multi-scale sampling and Harris-Laplace point detection). For each low-level descriptor, we trained a GMM with 256 Gaussians. Training was performed after reducing the descriptors into 64 dimensions by PCA. The normalized Fisher gradient vector computed from GMM of SIFT descriptors is a well known technique to represent an image with only one vector per pooling (we used 1x1, 2x3

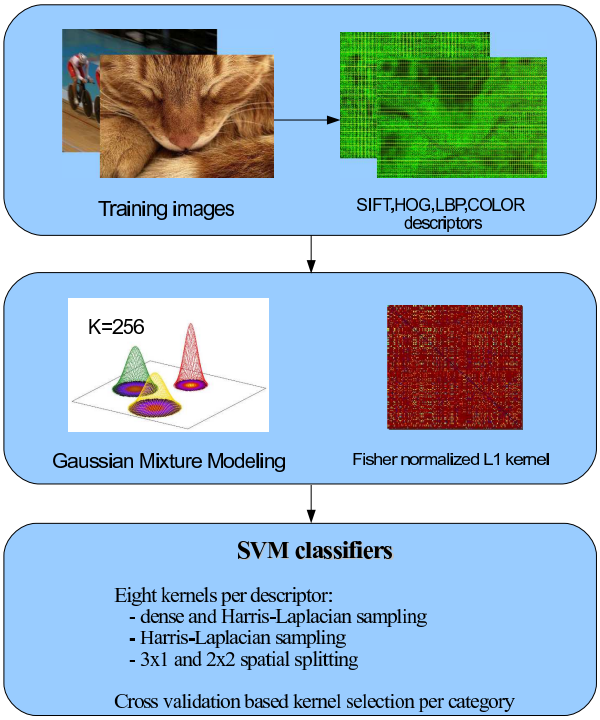

Figure 3: Our classification procedure

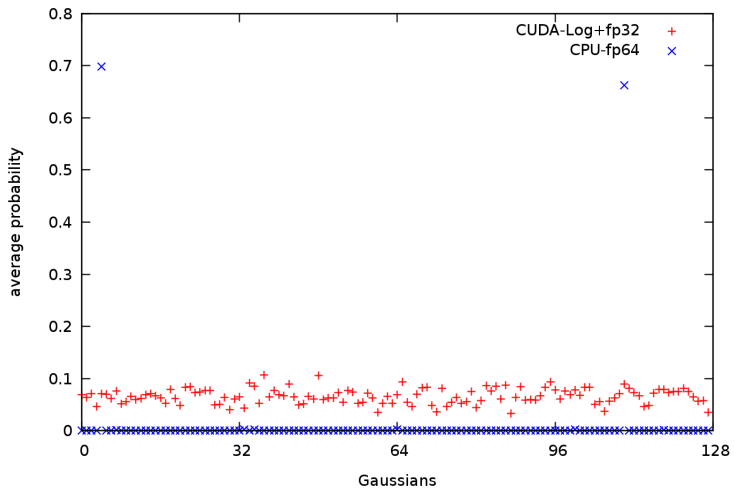

Figure 4: An example of a numerically precise GMM (+) over 750,000 points and an imprecise one over 250,000 points where the mixture degrade to two overrepresented Gaussian (X).

Table 1: Average MAP on Pascal VOC 2007 with different poolings

	Dense+HL	HL	sp1x3	sp2x2	Dense+HL+SP
SIFT	0.466	0.430	0.488	0.474	0.522
COLOUR	0.443	0.362	0.460	0.448	0.482
HOG	0.424	0.308	0.451	0.438	0.479
LBP	0.403	0.270	0.436	0.425	0.455
Combination					0.566
INRIA (genetic) [8]					0.575
INRIA (flat) [8]					0.558
XRCE'07 [8]					0.527
VOC2010 winner NUSPSL[32]					0.611

Table 2: MAP on Pascal VOC 2007 data sets

	aeroplane	bicycle	bird	boat	bottle	
Our Method	0.7846	0.5968	0.5508	0.6608	0.2318	
NUSPSL [32]	0.7890	0.6840	0.5190	0.7150	0.2980	
	bus	car	cat	chair	cow	
Our Method	0.6128	0.7490	0.5352	0.4964	0.3693	
NUSPSL [32]	0.7030	0.8160	0.6020	0.5450	0.4820	
	dining table	dog	horse	motorbike	person	
Our Method	0.4772	0.4175	0.7855	0.6408	0.8604	
NUSPSL	0.5680	0.4490	0.8080	0.6880	0.8590	
	potted plant	sheep	sofa	train	tv/monitor	AvgMAP
Our Method	0.3714	0.4488	0.4381	0.7717	0.5171	0.566
NUSPSL	0.2960	0.4770	0.5770	0.8170	0.5290	0.611

and 2x2 spatial pyramids [24]). Our overall procedure is shown in Fig. 1.

We used the resulting kernels for training linear SVM by the LibSVM package [4] for each of the 20 Pascal VOC 2007 concepts independently for each pooling. We splitted the training set into two parts to select the best kernels for each category.

4 Conclusions

In this paper we have demonstrated that graphical processors can play a key role in scaling not just in image feature generation, but also in content-based learning procedures. We described a CUDA implementation Gaussian Mixture Modeling and Fisher vector calculation, a powerful method of generating a so-called bag of visual words representation that reaches over an order of magnitude speedup over a CPU. The resulting image classification system is comparable to the best performing PASCAL VOC systems (see Table 1), in some categories outperforming the the best published system to date[32](see Table 2). Further improvment could be a Super-vector extension to our GMM based BOW generation. The Super-vector based system is a Fisher-like extension to a k-means based system with large codebook (K=2048) while our system is based on a codebook only with 256 visual words. We also plan to stengthen our combination procedure with a genetic optimization algorithm.

We make our source code along with preprocessed image classification test data available free for research use at `http://datamining.sztaki.hu/?q=en/GPU-GMM`.

References

[1] J. Ah-Pine, C. Cifarelli, S. Clinchant, G. Csurka, and J. Renders. XRCEs Participation to ImageCLEF 2008. In *Working Notes of the 2008 CLEF Workshop*, 2008.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers*, page 924. ACM, 2003.

[3] B. Bustos, O. Deussen, S. Hiller, and D. Keim. A graphics hardware accelerated algorithm for nearest neighbor search. *Computational Science–ICCS 2006*, pages 196–199, 2006.

[4] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[5] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray. Visual categorization with bags of keypoints. In *Workshop on Statistical Learning in Computer Vision, ECCV*, volume 1, page 22. Citeseer, 2004.

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2009 (VOC2009) Results. 2009.

[7] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[8] Mark Everingham. Overview and results of classification challenge. In *ICCV,Pascal VOC 2007 Workshop, 2007*, 2007.

[9] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, page 137. ACM, 2004.

[10] L. Fei-Fei and P. Perona. A Bayesian hierarchical model for learning natural scene categories. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, 2005.

[11] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, page 22. ACM, 2005.

[12] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3. IEEE Computer Society, 2005.

[13] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing–HiPC 2007*, pages 197–208, 2007.

[14] F. Jurie and B. Triggs. Creating efficient codebooks for visual recognition. In *Tenth IEEE International Conference on Computer Vision, 2005. ICCV 2005*, volume 1, 2005.

[15] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, page 234. ACM, 2005.

[16] N. Kumar, S. Satoor, and I. Buck. Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 103–109. IEEE, 2009.

[17] M. Liu, E. Chang, and B. Dai. Hierarchical Gaussian mixture model for speaker verification. In *Seventh International Conference on Spoken Language Processing*. Citeseer, 2002.

[18] F. Monay, P. Quelhas, D. Gatica-Perez, and J.M. Odobez. Constructing visual models with a latent space approach. *Lecture notes in computer science*, 3940:115–126, 2006.

[19] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, page 119. Eurographics Association, 2003.

[20] Stefanie Nowak. New Strategies for Image Annotation: Overview of the Photo Annotation Task at ImageCLEF 2010. In *Cross Language Evaluation Forum , ImageCLEF Workshop, 2010*, 2010.

[21] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[22] F. Perronnin and C. Dance. Fisher kernels on visual vocabularies for image categorization. In *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07*, pages 1–8, 2007.

[23] F. Perronnin, C. Dance, G. Csurka, and M. Bressan. Adapted vocabularies for generic visual categorization. In *Computer Vision–ECCV 2006*, pages 464–475, 2006.

[24] C. Schmid S. Lazebnik and J. Ponce. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, New York, June 2006*, 2006.

[25] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Ninth IEEE international conference on computer vision, 2003. Proceedings*, pages 1470–1477, 2003.

[26] J. Spitzer. Implementing a GPU-efficient FFT. *SIGGRAPH Course on Interactive Geometric and Scientific Computations with Graphics Hardware*, 2003.

[27] G. Csurka T. Mensink, F. Perronnin, J. SĂĄnchez, and J. Verbeek. LEAR and XRCEs participation to Visual Concept Detection Task at ImageCLEF 2010. In *Working Notes for the CLEF 2010 Workshop*, 2010.

[28] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek. Evaluating color descriptors for object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1582–1596, 2010.

[29] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek. Empowering visual categorization with the gpu. *IEEE Transactions on Multimedia*, 13(1):60–70, 2011.

[30] V. Vineet and PJ Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *CVPR Workshop on Visual Computer Vision on GPUs*, 2008.

[31] Z. Wang, H. Yi, J. Wang, and D. Feng. Hierarchical Gaussian Mixture Model for Image Annotation via PLSA. In *2009 Fifth International Conference on Image and Graphics*, pages 384–389. IEEE, 2009.

[32] Kai Yu. Tong Zhang Xi Zhou and Thomas Huang. Image Classification using Super-Vector Coding of Local Image Descriptors. In *11th ECCV,2010*, 2010.