Accelerating Visual Categorization with the GPU

Koen E.A. van de Sande, Theo Gevers, and Cees G.M. Snoek

Intelligent Systems Lab Amsterdam (ISLA),
University of Amsterdam,
Science Park 904, 1098 XH Amsterdam, The Netherlands
ksande@uva.nl

Abstract. Visual categorization is important to manage large collections of digital images and video, where textual meta-data is often incomplete or simply unavailable. The bag-of-words model has become the most powerful method for visual categorization of images and video. Despite its high accuracy, a severe drawback of this model is its high computational cost. As the trend to increase computational power in newer CPU and GPU architectures is to increase their level of parallelism, exploiting this parallelism becomes an important direction to handle the computational cost of the bag-of-words approach. In this paper, we analyze the bag-of-words model for visual categorization in terms of computational cost and identify two major bottlenecks: the quantization step and the classification step. We address these two bottlenecks by proposing two efficient algorithms for quantization and classification by exploiting the GPU hardware and the CUDA parallel programming model. The algorithms are designed to keep categorization accuracy intact and give the same numerical results.

In the experiments on large scale datasets it is shown that, by using a parallel implementation on the GPU, quantization is 28 times faster and classification is 35 faster than a single-threaded CPU version, while giving the exact same numerical results. The GPU accelerations are applicable to both the learning phase and the testing phase of visual categorization systems. For software visit http://www.colordescriptors.com/.[1]

1 Introduction

Visual categorization aims to determine whether objects or scene types are visually present in images or video segments. This is a useful prerequisite to manage large collections of digital images and video, where textual meta-data is often incomplete or simply unavailable [2]. Letting humans annotate such meta-data is expensive and infeasible for large datasets. While automatic visual categorization is not yet as accurate as a human annotation, it is a useful tool to manage large collections. The bag-of-words model [3] has become the most powerful method today for visual categorization [4,5,6,7,8,9,10,11]. The bag-of-words model computes image descriptors at specific points in the image. These descriptors are then quantized against a codebook of prototypical descriptors to obtain a fixed-length representation of an image. Although

[1] Since the workshop, an extended version of this paper has been accepted for publication in IEEE Transactions on Multimedia [1].

K.N. Kutulakos (Ed.): ECCV 2010 Workshops, Part II, LNCS 6554, pp. 436–449, 2012.

the bag-of-words model is a powerful mechanism for accurate visual categorization, a severe drawback is its high computational cost. Current state-of-the-art in visual categorization benchmarks such as TRECVID 2009 [12] require weeks of compute time on compute clusters to process 380 hours of video. However, even with weeks of compute time, most systems are still only able to process a limited subset of about 250,000 frames. In the future, more and more data needs to be processed as datasets continue to grow. To address the problem of computation, the two directions are *faster approximate methods* and *larger compute clusters*. Faster to compute descriptors (such as SURF [13,14]) and indexing mechanisms (tree-based codebooks [15,16]) have been developed. Another direction is to use large compute clusters with many CPUs [11,10] to solve the computational problem using brute force. However, both directions have their drawbacks. Faster methods will (1) suffer from reduced accuracy when they resort to increasingly coarse approximations and (2) suffer from increased complexity in the form of additional parameters and thresholds to control the approximations, all of which need to be hand-tuned. Brute force solutions based on compute clusters have the problem that (1) compute clusters are available in limited supply and (2) are expensive.

Recently, another direction for acceleration has opened up: *computing on consumer graphics hardware*. Cornelis and Van Gool [17] have implemented SURF on the GPU (Graphics Processing Unit) and obtained an order of magnitude speedup compared to a CPU implementation. These GPU implementations [17,18] build on the trend of increased parallelism. Whereas commodity CPUs currently have up to 4 cores, commodity GPUs have hundreds of cores at their disposal [19]. Together, the increased programmability and computational power of GPUs provides ample opportunities for acceleration of algorithms which can be parallelized [19]. Compared to faster approximate methods, algorithms for the GPU do not need to approximate for speedups, if they are able to exploit the parallel nature of the GPU. Compared to compute clusters, the main advantages of the GPU are their wide availability and their potential to be more energy-efficient.

When optimizing a system based on the bag-of-words model, the goal is to minimize the time it takes to process batches of images. Individual components of the bag-of-words model, such as the point sampling strategy, descriptor computation and SVM model training, have been independently studied on the GPU before [17,20,21]. These studies accelerate specific algorithms with the GPU. However, it remains unclear whether those algorithms are the real bottlenecks in accurate visual categorization with the bag-of-words model. In our overview of related work on visual categorization with the GPU, we observe that quantization and classification have remained CPU-bound so far, despite being computationally very expensive. Therefore, in this paper, the goal is to combine GPU hardware and a parallel programming model to accelerate the quantization and classification components of a visual categorization architecture. Two algorithms are proposed to accelerate these two components. The algorithms are designed to keep categorization accuracy intact and give the same numerical results.

2 Overview of Visual Categorization

The aim of this paper is to speed up state-of-the-art visual categorization systems using GPUs. In visual categorization [22], the visual presence of an object or scene of

specified type is determined. In Figure 1, an overview of the components of a visual categorization system is shown. A trained visual categorization system takes an image as input and returns the likelihood that one or more visual categories are present in the image. Visual categorization systems break down into a number of common steps:

- *Image Feature Extraction*, which takes an image as input and outputs a fixed-length feature vector representing the image.
- *Category Model Learning*, learns one model per visual category by taking all vector representations of images from the train set and the category labels associated with those images.
- *Test Image Classification*, which takes vector representations of images from the test set and applies the visual category models to these images. The output of this step is a likelihood score for each image and each visual category.

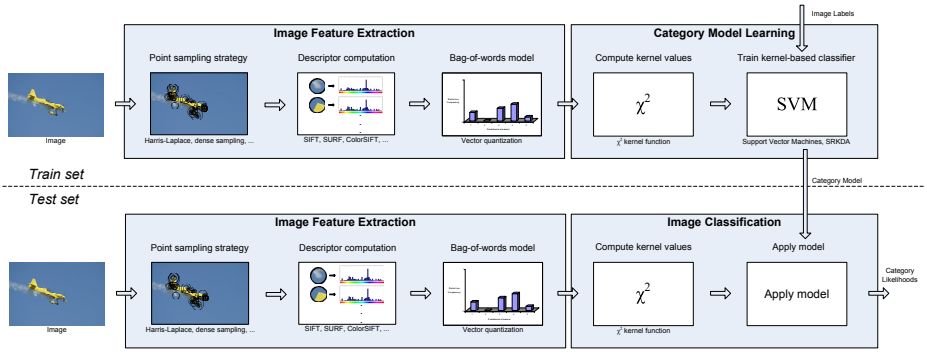

Fig. 1. The components of a state-of-the-art visual categorization system. For all images in both the train set and the test set, visual features are extracted in a number of steps. First, a point sampling method is applied to the image. Then, for every point a descriptor is computed over the area around the point. All the descriptors of an image are subsequently vector quantized against a codebook of prototypical descriptors. This results in a fixed-length feature vector representing the image. Next, the visual categorization system is trained based on the feature vectors of all training images and their category labels. To learn kernel-based classifiers, similarities between training images are needed. These similarities are computed using a kernel function. To apply a trained model to test images, the kernel function values are also needed. Given these values between a test image and the images in the train set, the category models are applied and category likelihoods are obtained.

2.1 Image Feature Extraction

Visual categorization systems which achieve state-of-the-art results on the PASCAL VOC benchmarks [5,9,6] use the bag-of-words model [3] as the underlying representation model. This model first extracts specific points in an image using a point sampling strategy. Over the area around these points, descriptors are computed which

represent the local area. The bag-of-words model performs vector quantization of the descriptors in an image against a visual codebook. A descriptor is assigned to the codebook element which is closest in Euclidean space. Figure 1 gives an overview of the steps for the bag-of-words model in the image feature extraction blocks. In Table 1, the computation times of different steps within the bag-of-words model are listed. For every step, multiple options are available. Next, we will discuss these options, their presence in related work and their computation times on the CPU and GPU.

Table 1. Computation times of different steps within the bag-of-words model on both the CPU and the GPU. For every step, multiple choices are available. CPU times obtained on AMD Opteron 250 @ 2.4GHz. GPU times obtained from the literature. One of the contributions of this paper is substantially accelerating the vector quantization step using the GPU.

Image Feature Extraction	**Times (s)**	
	CPU	*GPU*
1) Point Sampling Strategy		
• Dense Sampling	< 0.01	< 0.01
• Difference-of-Gaussians	1.4 [23]	< 0.1 [17]
• Harris-Laplace	4.4 [24]	< 0.5 [25]
2) Descriptors		
• SIFT	1.4 [23]	< 0.1 [18]
• SURF	< 1.0 [13]	< 0.01 [17]
• ColorSIFT	4.0 [6]	< 0.3 [18]
3) Bag-of-Words		
• Tree-based Codebook	< 0.5 [15,16]	< 0.01 [20]
• Vector Quantization	**5.0** [3]	$<$ **0.1 this paper**

Point Sampling Strategy. As a point sampling strategy, there are two commonly used techniques in state-of-the-art systems [9,6]: dense sampling and salient point methods. Dense sampling samples points regularly over the image at fixed pixel intervals. As it does not depend on the image contents, it is a trivial operation to perform. Typically, around 10,000 points are sampled per image. Two examples of salient point methods are the Harris-Laplace salient point detector [24] and the Difference-of-Gaussians detector [23]. See Table 1 for computation times of these point sampling strategies. The Harris-Laplace detector uses the Harris corner detector to find scale-invariant interest points. It then selects a subset of these points for which the Laplacian-of-Gaussians reaches a maximum over scale. Using recursive Gaussian filters [25], the computation of Gaussian derivatives at multiple scale required for these steps is possible at a rate of multiple images per second: computational complexity of recursive Gaussian filters is independent of the scale. As has been shown by Cornelis and Van Gool [17], running the Difference-of-Gaussians detector is possible in real-time, using a scale-space pyramid to limit computational complexity.

Descriptor Computation. To describe the area around the sampled points, the SIFT descriptor [23] and the SURF descriptor [13] are the most popular choices. Sinha *et al.* [18] compute SIFT descriptors at 10 frames per second for 640x480 images. Cornelis and Van Gool [17] compute SURF descriptors at 100 frames per second for 640x480 images. Both of these papers show that descriptor computation runs with excellent performance on the GPU, because one thread can be assigned per pixel or per descriptor, and thereby performing operations in parallel. The standard SIFT descriptor has a length of 128. Following Everingham *et al.* [5], color extensions of SIFT [6] would form a reasonable state-of-the-art baseline for future VOC challenges, due to their increased classification accuracy. ColorSIFT increases the descriptor length to 384 and the required computation time is also tripled.

Bag-of-Words. Vector quantization is computationally the most expensive part of the bag-of-words model. With n descriptors of length d in an image, the quantization against a codebook with m elements requires the full $(n \times m)$ distance matrix between all descriptors and codebook elements. For values which are common for visual categorization, $n = 10,000$, $d = 128$ and codebook size $m = 4,000$, a CPU implementation takes approximately 5 seconds per image, as the complexity is $O(ndm)$ per image. When d increases to 384, as is the case for ColorSIFT, the CPU implementation slows down to more than 10 seconds per image, which makes this a computational bottleneck.

One approach to address this bottleneck is to index using a tree-based codebook structure [15,16,14], instead of a standard codebook. A tree-based codebook replaces the comparison of each descriptor with all m codebook elements by a comparison against $\log(m)$ codebook elements. As a result, algorithmic complexity is reduced to $O(nd\log(m))$. Tree-based methods have been shown to run in real-time on the GPU [20]. However, for a tree-based codebook generally the accuracy is lower [14], especially for high-dimensional descriptors such as ColorSIFT. Therefore, tree-based codebooks conflict with our goal of keeping accuracy intact. The same argument applies to other indexing structures such as miniBOF (mini bag-of-features) [26]: accuracy is sacrificed in return for faster computation. Another drawback of tree-based codebooks and miniBOFs is that soft assignment [7,27], which improves accuracy by 5% by assigning weight to more than just the closest codebook element, requires the full distance matrix instead of only the closest elements. These methods are unable to provide this matrix. Therefore, this paper studies how to accelerate the vector quantization step using normal codebooks on the GPU, as the same accelerations are then also applicable to soft assignment.

In conclusion, in a state-of-the-art setup of the bag-of-words model, the most expensive part is the vector quantization step. Approximate methods are unable to satisfy our requirement to maintain accuracy.

2.2 Category Model Learning

To learn visual category models, supervised kernel-based learning algorithms such as Support Vector Machines (SVM) and Spectral Regression Kernel Discriminant Analysis [28] have shown good results [4,6]. Key property of a kernel-based classifier is that

it does not require the actual vector representation of the feature vector $\boldsymbol{F}$, but only a kernel function $k(\boldsymbol{F}, \boldsymbol{F'})$ which is related to the distance between the feature vectors. This is sometimes referred to as the 'kernel trick'. It has been shown experimentally [4] that the non-linear χ^2 kernel function is the best choice [9,6] for accurate visual categorization. While typical implementations compute the values of this kernel function on-the-fly and only keep a cache of the most recent evaluations, it is more efficient to compute all values in advance and store them, because then the values can be re-used for every parameter setting and for every visual category. The total number of kernel values to be computed in advance is the number of pair-wise distances between all training images, *e.g.* , it is quadratic with respect to the number of images. The benefit of precomputing kernel values is illustrated in Table 2.

Table 2. Computation times of the different steps in visual categorization. The times listed are for an image dataset (PASCAL VOC 2008), which has a training set of size 4332 and test set of size 4133. Classification times are totals for all 20 visual categories. CPU times obtained on AMD Opteron 250 @ 2.4GHz. This paper substantially accelerates the precomputation of kernel values (shown in bold) using the GPU.

Category Model Learning	**Times (s)**	
	CPU	*GPU*
Category Model Learning (without precomputed)		
Parameter Tuning (length $\boldsymbol{F} = 4,000$)	$> 1,000,000$ [29]	$> 10,000$ [21]
Train Classifier (length $\boldsymbol{F} = 4,000$)	$> 100,000$ [29]	$> 1,000$ [21]
Category Model Learning (with precomputed)		
Precompute Kernel Values (length $\boldsymbol{F} = 4,000$)	**660**	**9 this paper**
Precompute Kernel Values (length $\boldsymbol{F} = 32,000$)	**3,600**	**64 this paper**
Precompute Kernel Values (length $\boldsymbol{F} = 320,000$)	**36,000**	**650 this paper**
Parameter Tuning	1,050 [29]	60 [21]
Train Classifier	240 [29]	10 [21]
Test Image Classification (with precomputed)		
Precompute Kernel Values (length $\boldsymbol{F} = 4,000$)	**600**	**8 this paper**
Apply Classifier	< 5 [29]	< 1 [21]

The kernel-based SVM algorithm has been ported to the GPU by [30,21]. In [30], specific optimizations are made in the GPU version such that only linear kernel functions are supported. For visual categorization, however, support for the more accurate non-linear χ^2 kernel function is needed to maintain accuracy. Catanzaro *et al.* [21] perform a selection of the training samples under consideration for SVM, resulting in a speedup of up to 35 times for training models. Further speedups are possible if this GPU-SVM implementation is combined with the precomputation of kernel values. The precomputation of kernel values itself has not been investigated yet. Therefore, in section 3.3, we propose an algorithm to precompute the kernel values and investigate the speedup possibilities offered by precomputing these values.

Table 2 gives an overview of computation times on the PASCAL VOC 2008 dataset for different feature vector lengths, where the learning of visual category models is

split into a precomputation of kernel values and the actual model learning. Because the ground truth labels of all images and their extracted features are needed before training can start, it is an inherently offline process. When multiple features are used, more than 90% of computation time is spent on precomputing the kernel values. This makes it the most expensive step in category model learning.

In conclusion, the learning of category models can be split into two steps, kernel value computation and classifier training. The classifier training has been accelerated with the GPU before, but the kernel value computation is the most expensive step. This paper will study how to accelerate the computation of the kernel values on the GPU.

2.3 Test Image Classification

To classify images from a test set, feature extraction first has to be applied to the images, similar to the train set. Therefore, speed-ups obtained in the image feature extraction stage are useful for both the train set and the test set. To apply the visual category models, pair-wise kernel values between the feature vectors of the train set and those of the test set are needed. Therefore, when accelerating the computation of kernel values, this speedup will apply to both the training phase and the test phase of a visual categorization system. This speedup is made possible by processing the test set in small batches, instead of one image at a time. Timings in Table 2 show that for the test set, again, the computation of kernel values takes up the most time.

In conclusion, the speedups obtained using GPU vector quantization and GPU precomputation of kernel values also directly apply to the classification of images/frames from the test set.

3 GPU Accelerated Categorization

We start with discussing the CUDA programming model with an example of parallel programming for the GPU in section 3.1. Next, we discuss the GPU-accelerated versions of vector quantization (section 3.2) and kernel value precomputation (section 3.3). Both of these visual categorization steps take large numbers of vectors as input, and therefore are ideally suited for the data parallelism offered by the GPU.

3.1 CUDA Programming Model

A CUDA program is organized into a normal C/C++ host program, running sequentially on the host CPU, and one or more parallel procedures that are suitable for execution on a parallel processing device like the GPU. A parallel procedure[2] is a simple sequential program which is executed simultaneously on a set of parallel threads. The programmer organizes these threads into thread blocks. The threads within a thread block are allowed to synchronize and support inter-thread communication through a high-speed shared memory. Threads from different blocks coordinate only through global memory. CUDA

[2] In the CUDA documentation, parallel procedures are called parallel kernels. In this paper, we refer to them as parallel procedures to avoid using the word kernel in two different contexts.

requires that thread blocks are independent, meaning that a parallel procedure must execute correctly no matter the order in which blocks are run. This restriction on the dependencies between blocks of a parallel procedure provides scalability.

Figure 2 shows a basic example of parallel programming with CUDA. The example shows a common parallelization pattern, where a serial loop with independent iterations is executed in parallel across many threads. The results of the various threads are gathered through a parallel reduction [31], also known as the 'butterfly pattern'. With a parallel reduction, n elements are summed in $\log n$ steps.

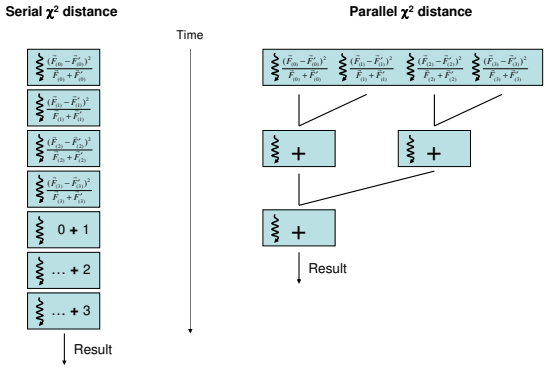

Fig. 2. Simple serial and parallel implementations of the χ^2 distance function $\frac{1}{2}\sum_{i=1}\frac{(\boldsymbol{F_i}-\boldsymbol{F_i'})^2}{\boldsymbol{F_i}+\boldsymbol{F_i'}}$ for given vectors $\boldsymbol{F}$ and $\boldsymbol{F'}$ consisting of 4 floating point numbers. The serial version on the left is a simple loop. The parallel procedure on the right executes independent iterations in parallel.

3.2 Algorithm 1: GPU-Accelerated Vector Quantization

In section 2.1, we have shown that vector quantization is computationally the most expensive step in image feature extraction. Therefore, in this section, the GPU implementation of vector quantization for an image with n descriptors against a codebook of m elements is proposed. The descriptor length is d. Quantization against a codebook requires the full $(n \times m)$ distance matrix between all descriptors and codebook elements. A descriptor is then assigned to the column which has the lowest distance in a row. By counting the number of minima occurring in each column, the vector quantized representation of the image is obtained. To be robust against changes in the number of descriptors in an image, these counts are divided by the number of descriptors n for the final feature vector.

The most expensive computational step in vector quantization is the calculation of the distance matrix. Typically, the Euclidean distance is employed:

$$||\boldsymbol{a}-\boldsymbol{b}|| = \sqrt{(a_1-b_1)^2+(a_2-b_2)^2+...+(a_q-b_q)^2}. \quad (1)$$

This formula for the Euclidean distance can be directly implemented on the GPU using loops [32]. However, such a naive implementation is not very efficient, because the same

result is obtained with fewer operations by simply vectorizing the Euclidean distance. This well-known trick [21] computes the Euclidean distance in vector form:

$$||\boldsymbol{a} - \boldsymbol{b}|| = \sqrt{||\boldsymbol{a}||^2 + ||\boldsymbol{b}||^2 - 2\boldsymbol{a} \cdot \boldsymbol{b}}. \quad (2)$$

The advantage of the vector form of the Euclidean distance is that it allows us to decompose the computation of a distance matrix between sets of vectors into several smaller steps which are faster to compute. The dot products $\boldsymbol{a} \cdot \boldsymbol{b}$ in (2) between sets of vectors can be rewritten as a matrix multiplication: AB^T contains all the dot products required for the full distance matrix, with A the matrix with all image descriptors as rows and B the matrix with all codebook elements as rows. Highly optimized BLAS linear algebra libraries exist for both the CPU and the GPU which contain matrix multiplication. On the CPU we use the ATLAS library, which we tune for every CPU architecture used. Another key insight when implementing this operation is that the squared vector lengths $||\boldsymbol{a}||^2$ and $||\boldsymbol{b}||^2$ are used multiple times and can be cached. After the compute distance matrix has been computed, assigning the descriptors to codebook elements is a matter of finding the codebook element with the lowest distance to a descriptor, which is a simple minimization over the rows of the distance matrix.

In conclusion, vector quantization involves computing the pair-wise Euclidean distances between n descriptors and m codebook elements. By simply vectorizing the computation of the Euclidean distance, the computation can be decomposed into steps which can be efficiently executed on the GPU.

3.3 Algorithm 2: GPU-Accelerated Kernel Value Precomputation

To compute kernel function values, we use the kernel function based on the χ^2 distance, which has shown the most accurate results in visual categorization (see section 2.2). Our contribution is evaluating the χ^2 kernel function on the GPU efficiently, even for very large datasets which do not fit into memory. The χ^2 distance between feature vectors F and F' is:

$$dist_{\chi^2}(\boldsymbol{F}, \boldsymbol{F'}) = \frac{1}{2} \sum_{i=1}^{s} \frac{(\boldsymbol{F_i} - \boldsymbol{F'_i})^2}{\boldsymbol{F_i} + \boldsymbol{F'_i}}, \quad (3)$$

with s the size of the feature vectors. For notational convenience, $\frac{0}{0}$ is assumed to be equal to 0 iff $\boldsymbol{F_i} = \boldsymbol{F'_i} = 0$.

The kernel function based on this χ^2 distance then is:

$$k(\boldsymbol{F}, \boldsymbol{F'}) = e^{-\frac{1}{D} dist(\boldsymbol{F}, \boldsymbol{F'})}, \quad (4)$$

where D is an optional scalar to normalizes the distances [4]. Because the χ^2 distance is already constrained to lie between 0 and 1, this normalization is unnecessary and we therefore fix D to 1.

For vector quantization, discussed in the previous section, all input data and the resulting output fits into computer memory. For kernel value precomputation, memory usage is an important problem. For example, for a dataset with $50,000$ images, the input data is 12 GB and the output data is 19 GB. Therefore, special care must be taken when designing the implementation, to avoid holding all data in memory simultaneously. We

divide the processing into evenly sized chunks. Each chunk corresponds to a square 1024x1024 subblock of the kernel matrix with all kernel function values. Because the final kernel function values only depend on the subset of feature vectors involved in the chunk, the operations are performed for every chunk separately. For every feature j, compute the χ^2 distances D between the 1024 vectors $\boldsymbol{F}_{(j)}$ and the 1024 vectors $\boldsymbol{F'}_{(j)}$. To compute the pair-wise distances between all these vectors, one thread block is created per pair (*e.g.* 1024x1024 thread blocks): $\boldsymbol{F}$ is the first input and $\boldsymbol{F'}$ is the second input to (3). The parallel procedure applied to every thread block to compute $dist_{\chi^2}(\boldsymbol{F}, \boldsymbol{F'})$ follows the parallelization pattern shown in Fig 2: one thread is assigned per data element. After the distances have been computed, they are divided by D and their exponent with base e is taken (see (4)). Repeat this operation for all chunks and the complete kernel matrix has been computed.

4 Experimental Setup

4.1 Experiment 1: Vector Quantization Speed

We measure the relative speed of two vector quantization implementations: CPU and GPU versions of the vectorized approach from section 3.2. Measured times are the median of 25 runs; an initial warm-up run is discarded to exclude initialization effects. For the experiments, realistic data sizes are used, following the state-of-the-art [6]: a codebook of size $m = 4,000$; up to $20,000$ descriptors per image and descriptor lengths of $d = 128$ (SIFT) and $d = 384$ (ColorSIFT). Because CPU architectures still improve with every generation, we include multiple CPU architectures in our comparison of CPU and GPU versions, to show the rate of development in CPU compute speeds.

4.2 Experiment 2: Kernel Value Precomputation Speed

To measure the speed of kernel value computation, we compare a CPU version and a GPU version based on the approach from section 3.3. An alternative approach besides the GPU would be to compute the kernel values on a compute cluster. Therefore, for reference, we include an MPI version which can execute on such a cluster. We compare the GPU version on the Geforce GTX275 to the single-threaded CPU version on the Xeon X5570 and the Opteron 250. To demonstrate the execution speed relative to that of a compute cluster, we also show results using 4, 16, 25, 36 and 49 Opteron CPUs. To obtain timings results, we have chosen the large Mediamill Challenge training set of 30 993 frames [33] with realistic feature vector lengths: from a single feature (total feature vector length $4,000$) up to 10 features (total feature vector length $128,000$). For a real system, the number of features might be even higher [6,10].

5 Results

5.1 Experiment 1: Vector Quantization Speed

Figure 3 shows the vector quantization speeds for SIFT descriptors using different hardware platforms and implementations. From the results, it is shown that vector quantization on CPUs takes more time than on GPUs. The difference between the fastest

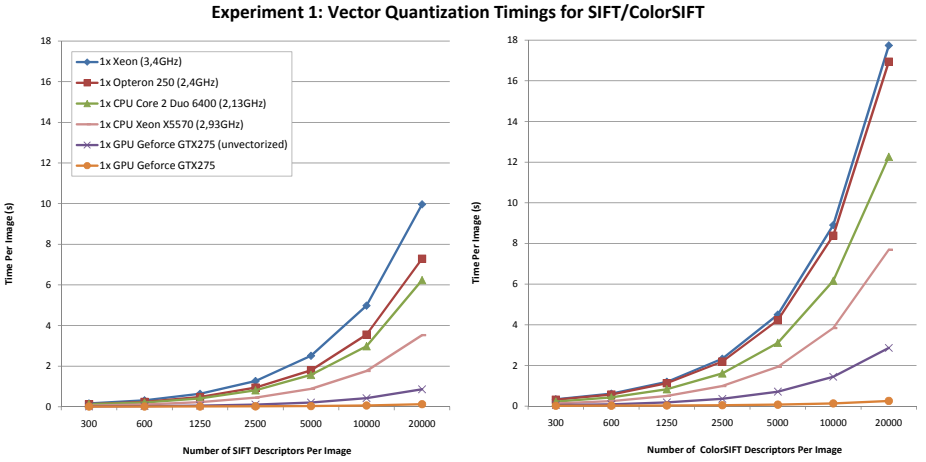

Fig. 3. Vector quantization speeds for a varying number of SIFT descriptors (on the left) or ColorSIFT descriptors (on the right). Each line represents a different hardware configuration plus appropriate implementation (CPU, GPU). The difference between the fastest single CPU core and the GPU is a factor 28.

single-threaded CPU and the fastest GPU is a factor of 28; both are using a vectorized implementation. An unvectorized GPU implementation is 6 times slower than a vectorized GPU implementation. For a typical number of SIFT descriptors per frame, 10,000, this is the difference between $0.6s$ and $0.06s$ spent *per image* in vector quantization. In the ColorSIFT results, we see the same speedup: from $1.2s$ to $0.13s$. When processing datasets of thousands or even millions of images, this is a crucial acceleration.

An interesting observation is that the CPU times can be used to roughly order them by release date. The 2004 Xeon takes about 1.4 times longer than a 2006 Core 2 Duo and 2.8 times longer than a 2009 Xeon X5570.

In conclusion, the speedup through parallelization obtained for vector quantization is an important acceleration when processing large image datasets. When combined with GPU versions of the other image feature extraction stages (see Table 1), even the most expensive feature can still be extracted in less than 1 second per image. Without GPU vector quantization, this would require an order of magnitude longer.

5.2 Experiment 2: Kernel Value Precomputation Speed

Figure 4 shows the kernel value precomputation speeds on different hardware platforms. The difference between a single GTX275 and a single Opteron CPU is a factor 90! The difference between the more recent Xeon X5570 CPU and the GPU is a factor 35. When using a bag-of-words model with features computed for four spatial pyramid levels (a total feature vector length of $120,000$), this is the difference between 2250 minutes and 170 minutes. Again, the GPU architecture results in a substantial acceleration.

When comparing the GPU implementation on a single Geforce GTX275 to the distributed CPU implementation, we see that a compute cluster with 49 Opteron CPUs is

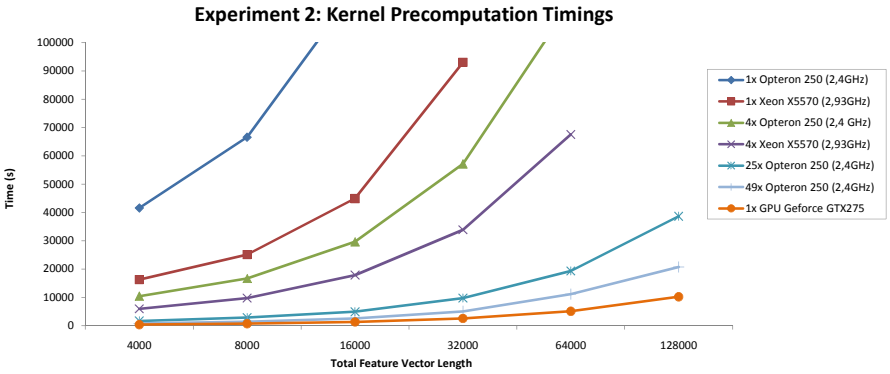

Fig. 4. Timings of kernel value precomputation on different hardware for various feature vector lengths. The difference between a single GTX275 and a single Opteron CPU is a factor 90. The difference between the more recent Xeon X5570 CPU and the GPU is a factor 35. Furthermore, a single GPU outperforms a compute cluster with 49 Opteron CPUs by a factor of 2.

still outperformed by the GPU with a factor 2. This implies that a medium-size compute cluster is insufficient to beat a single GPU when precomputing kernel values. For large datasets, consisting of tens of thousands of training images (*e.g.* , TRECVID 2009 [12], Mediamill Challenge [33]), this allows the category learning step to be performed using a single machine, instead of using an expensive compute cluster. Alternatively, the improved efficiency could be used to include more visual features (which implies even longer feature vectors) or to process additional frames from a video.

6 Conclusions

This paper provides an efficiency analysis of a state-of-the-art visual categorization pipeline based on the bag-of-words model. In this analysis, two large bottlenecks were identified: the vector quantization step in the image feature extraction and the kernel value computation in the category classification. By using a vectorized GPU implementation of vector quantization, it is 28 times faster than when it is computed on a CPU. For the classification, we exploit the intrinsic property of kernel-based classifiers that only kernel values are needed. By precomputing these kernel values, the parameter tuning and model learning stages can reuse these values, instead of computing them on the fly for every visual category and parameter setting. Also, computing these kernel values on the GPU accelerates it by a factor of 35, while giving the exact same results for visual categorization. The latter GPU acceleration is applicable to both the learning phase and the test phase. In the future, we will look at applying our GPU accelerations to other problems, such as k-means clustering and text retrieval.

References

1. van de Sande, K.E.A., Gevers, T., Snoek, C.G.M.: Empowering visual categorization with the GPU. IEEE Transactions on Multimedia (2011) (in press)

2. Hollink, L., Huurnink, B., van Liempt, M., Oomen, J., de Jong, A., de Rijke, M., Schreiber, G., Smeulders, A.W.M.: A multidisciplinary approach to unlocking television broadcast archives. Interdisciplinary Science Reviews 34, 253–267 (2009)
3. Sivic, J., Zisserman, A.: Video Google: A text retrieval approach to object matching in videos. In: IEEE International Conference on Computer Vision, pp. 1470–1477 (2003)
4. Zhang, J., Marszałek, M., Lazebnik, S., Schmid, C.: Local features and kernels for classification of texture and object categories: A comprehensive study. International Journal of Computer Vision 73, 213–238 (2007)
5. Everingham, M., Van Gool, L., Williams, C., Winn, J., Zisserman, A.: The pascal visual object classes (VOC) challenge. International Journal of Computer Vision 88, 303–338 (2010)
6. van de Sande, K.E.A., Gevers, T., Snoek, C.G.M.: Evaluating color descriptors for object and scene recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence 32, 1582–1596 (2010)
7. Jiang, Y.G., Yang, J., Ngo, C.W., Hauptmann, A.: Representations of keypoint-based semantic concept detection: A comprehensive study. IEEE Transactions on Multimedia 12, 42–53 (2010)
8. van de Sande, K.E.A., Gevers, T.: University of Amsterdam at the Visual Concept Detection and Annotation Tasks. The Information Retrieval Series: Image CLEF, vol. 32, ch. 18, pp. 343–358. Springer (2010)
9. Gaidon, A., Marszałek, M., Schmid, C.: The PASCAL visual object classes challenge 2008 submission. Technical report, INRIA-LEAR (2008)
10. Snoek, C.G.M., van de Sande, K.E.A., de Rooij, O., Huurnink, B., Uijlings, J.R.R., van Liempt, M., Bugalho, M., Trancoso, I., Yan, F., Tahir, M.A., Mikolajczyk, K., Kittler, J., de Rijke, M., Geusebroek, J.M., Gevers, T., Worring, M., Koelma, D.C., Smeulders, A.W.M.: The MediaMill TRECVID 2009 semantic video search engine. In: Proceedings of the TRECVID Workshop (2009)
11. Wang, D., Liu, X., Luo, L., Li, J., Zhang, B.: Video diver: generic video indexing with diverse features. In: ACM International Workshop on Multimedia Information Retrieval, pp. 61–70 (2007)
12. Smeaton, A.F., Over, P., Kraaij, W.: Evaluation campaigns and TRECVid. In: ACM International Workshop on Multimedia Information Retrieval, pp. 321–330 (2006)
13. Bay, H., Ess, A., Tuytelaars, T., Van Gool, L.: Speeded-up robust features (SURF). Computer Vision and Image Understanding 110, 346–359 (2008)
14. Uijlings, J.R.R., Smeulders, A.W.M., Scha, R.J.H.: Real-time bag-of-words, approximately. In: ACM International Conference on Image and Video Retrieval (2009)
15. Chang, C.C., Li, Y.C., Yeh, J.B.: Fast codebook search algorithms based on tree-structured vector quantization. Pattern Recognition Letters 27, 1077–1086 (2006)
16. Moosmann, F., Triggs, B., Jurie, F.: Fast discriminative visual codebooks using randomized clustering forests. In: Neural Information Processing Systems, pp. 985–992 (2006)
17. Cornelis, N., Van Gool, L.: Fast scale invariant feature detection and matching on programmable graphics hardware. In: IEEE Computer Vision and Pattern Recognition Workshops (2008)
18. Sinha, S.N., Frahm, J.M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Machine Vision and Applications (2007)
19. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96, 879–899 (2008)
20. Sharp, T.: Implementing Decision Trees and Forests on a GPU. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) ECCV 2008, Part IV. LNCS, vol. 5305, pp. 595–608. Springer, Heidelberg (2008)

21. Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: International Conference on Machine Learning, pp. 104–111 (2008)
22. Datta, R., Joshi, D., Li, J., Wang, J.Z.: Image retrieval: Ideas, influences, and trends of the new age. ACM Computing Surveys 40, 1–60 (2008)
23. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision 60, 91–110 (2004)
24. Mikolajczyk, K., et al.: A comparison of affine region detectors. International Journal of Computer Vision 65, 43–72 (2005)
25. Geusebroek, J.M., Smeulders, A.W.M., van de Weijer, J.: Fast anisotropic gauss filtering. IEEE Transactions on Image Processing 12, 938–943 (2003)
26. Jégou, H., Douze, M., Schmid, C.: Packing bag-of-features. In: IEEE International Conference on Computer Vision (2009)
27. van Gemert, J.C., Veenman, C.J., Smeulders, A.W.M., Geusebroek, J.M.: Visual word ambiguity. IEEE Transactions on Pattern Analysis and Machine Intelligence 32, 1271–1283 (2010)
28. Cai, D., He, X., Han, J.: Efficient kernel discriminant analysis via spectral regression. In: IEEE International Conference on Data Mining, pp. 427–432 (2007)
29. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
30. Do, T.-N., Nguyen, V.-H., Poulet, F.: Speed Up SVM Algorithm for Massive Classification Tasks. In: Tang, C., Ling, C.X., Zhou, X., Cercone, N.J., Li, X. (eds.) ADMA 2008. LNCS (LNAI), vol. 5139, pp. 147–157. Springer, Heidelberg (2008)
31. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware, pp. 97–106 (2007)
32. Chang, D., Jones, N.A., Li, D., Ouyang, M.: Compute pairwise euclidean distances of data points with GPUs. In: Intelligent Systems and Control, pp. 278–283 (2008)
33. Snoek, C.G.M., Worring, M., van Gemert, J.C., Geusebroek, J.M., Smeulders, A.W.M.: The challenge problem for automated detection of 101 semantic concepts in multimedia. In: ACM International Conference on Multimedia, pp. 421–430 (2006)