GPU-ACCELERATED SCENE CATEGORIZATION UNDER MULTISCALE CATEGORY-SPECIFIC VISUAL WORD STRATEGY

Wei-Ta Chu and Sheng-Chung Tseng

National Chung Cheng University, Chiayi, Taiwan
wtchu@cs.ccu.edu.tw, bittertea0503@gmail.com

ABSTRACT

We utilize GPU to accelerate an essential component for computer vision and multimedia information retrieval, i.e. scene categorization. To construct bag of word models, we modify calculation of Euclidean distance so that feature clustering and visual word quantization can be processed in a parallel manner. We provide details of GPU implementations and conduct comprehensive experiments to verify the efficiency of GPU on multimedia analysis.

***Index Terms*—** GPU, scene categorization, multiscale category-specific visual words

1. INTRODUCTION

Scene information provides clues about "where" a photo was captured and "what" objects appeared. As a widespread research field, researchers work hard on scene categorization in the past decades. With the popularity of digital consumer devices, scene categorization is also used in organizing home-made photos and videos. In recent years, it is more likely considered as a basic tool for advanced analysis. Therefore, automatic scene categorization should be efficient and robust.

Some researches on scene categorization already reach good accuracy rate. However, these works often need to process tremendous amounts of data, and the time cost is high. To tackle with these problems, we modify methods for feature extraction and clustering, and take advantage of the GPU programming model to reduce execution time. Contributions of this work are listed as follows:

- Features are extracted from different scales and elaborately combined. By clustering features, visual words are then constructed in a category-specific manner, which is proven more effective in scene categorization.
- Euclidean distances are calculated in a matrix form so that GPU can be efficiently used. GPU implementations are proposed for several sub-tasks with the consideration of hardware limitations.

The reminder of this paper is organized as follows. In Section 2, we exploit the category-specific visual word model to classify scene images. GPU implementations and acceleration algorithms are introduced. We also analyze time complexity. Performance is reported in Section 3, and Section 4 concludes this paper.

2. GPU-ACCELERATED SCENE CATEGORIZATION

2.1. Overview

Figure 1 illustrates the system framework. In the training part, images in the same scene are first divided into patches at different scales, and from which SIFT features [2] are extracted. By respectively clustering feature vectors from the same scale, visual codebooks for different scales are generated. For a feature vector extracted from a patch in scale s, it is quantized into a visual word by consulting the codebook also generated from scale s. By jointly considering all patches (in different scales) in an image, a scene image is finally represented as a vector indicating appearance of various visual words. A multi-class SVM classifier is built based on a set of this representation. The same processes are applied to a test image, and the trained SVM classifier is used to determine scene type of the test image. In this paper, we focus on accelerating feature extraction and codebook construction.

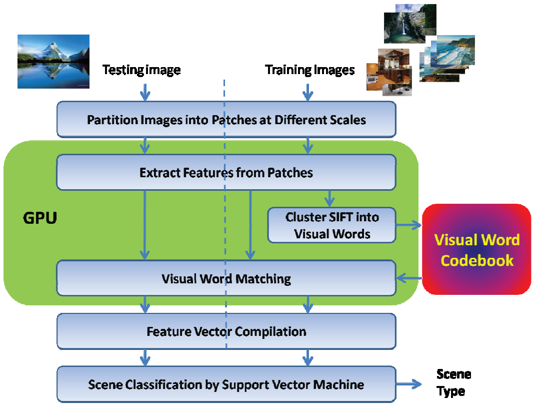
Testing image
Training Images
Partition Images into Patches at Different Scales
Extract Features from Patches
GPU
Cluster SIFT into Visual Words
Visual Word Codebook
Visual Word Matching
Feature Vector Compilation
Scene Classification by Support Vector Machine
Scene Type

Figure 1. System framework of GPU-accelerated scene categorization.

2.2. Feature Extraction

Inspired by [1], we divide an image into overlapped patches at different scales in order to capture both local and global information. At scale s, the height and width of the patch are $\frac{H}{2^{s-1}}$ and $\frac{W}{2^{s-1}}$, where H and W denote the height and width of the image, respectively. A set of patches $\mathcal{P}_s = \{P_{s,1}, P_{s,2}, ..., P_{s,n_s}\}$ are sampled from the image at scale s, where $n_s = (2^s - 1)^2$ is the number of patches. Figure 2 illustrates patch sampling at different scales. At scale 1, the whole image is considered as a patch. At scale 2, nine points are sampled and each patch has size equal to a quarter of the whole image.

978-1-4673-0046-9/12/$26.00 ©2012 IEEE

Unlike traditional SIFT descriptor extraction [2], patches may have different sizes, and by jointly considering features at different scales we can more appropriately describe a scene.

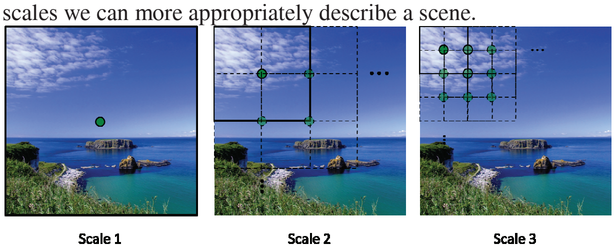

Figure 2. Patches at different scales.

```
GPU Procedure 1
Initial → blockIdx.x, blockIdx.y and threadIdx refer to the 2D block
    index and the 1D thread index, respectively.
Step 1 → Calculate the height $H_p$ and width $W_p$ of each patch.
Step 2 → Calculate the block offset $O_p$ for each patch:
    $O_p = blockIdx.x \times \frac{H}{n_s} \times W$
Step 3 → Transfer 1D thread index into a 2D coordinates threadIdx.x
    and threadIdx.y:
    $threadIdx.x = threadIdx / H_p$
    $threadIdx.y = threadIdx \mod W_p$
Step 4 → Calculate the targeted access location of the thread $O_t$ in $M$
    and $R$
    $O_t = O_p + threadIdx.x \times W + ThreadIdx.y$
Step 5 → Use $O_p$ to access $M$ and $R$, and accumulate magnitudes and
    orientations to generate the orientation histograms.
Note 1: Block offset is the offset of the first element of the patch
    relative to the first element of the image.
Note 2: $n_s = \begin{cases} 1, & \text{if } s = 1, \\ 2^s, & \text{otherwise.} \end{cases}$
```

2.3. Feature Vector Extraction

Instructions of extracting features for different patches are almost the same except for different pixel locations, and thus can be done in parallel. For a pixel at $(i, j), 1 \le i \le W, 1 \le j \le H$, in an image, its edge magnitude $M_{i,j}$ and orientation $R_{i,j}$ are first calculated. For a patch at scale s, we divide it into sixteen regions, and in each region each pixel's orientation relative to the main orientation is calculated. This relative orientation is then quantized into eight intervals, and by accumulating pixels' edge magnitudes in one of eight orientations, we construct an eight-dimensional orientation histogram for each region. Finally, a 128-dimensional feature vector ($16 \times 8 = 128$) is generated by cascading all histograms to represent a patch [2].

By utilizing GPU, we process patches at the same scale in a parallel manner. In the CUDA (Compute Unified Device Architecture) architecture [6], device can be divided into blocks and threads to access different portions of data. We use threads in a block to access pixels in a patch, and use different blocks to access different patches.

Since number of threads is limited, threads should be reused. A thread can access different memory locations by shifting the thread index. To generate an orientation histogram, a block of threads is used. Threads in different blocks access different portions of matrices M and R, where $M = [M_{i,j}]$ and $R = [R_{i,j}]$. Hence, the accessed location of every thread must be calculated precisely. The procedures to generate orientation histograms at scale s are listed in GPU Procedure 1. In this way, n_s feature vectors can be computed in parallel. Note that in step 5 atomic operations in CUDA are used to avoid a race condition.

2.4. Category-Specific Visual Word Model

In the conventional BoW model, scene images in all categories are taken as input, and features of different scenes are mixed together. This representation is less discriminative. In our work, images of different categories form independent visual words to construct the category-specific visual word model. Assume that there are K scene categories in the training set, and $C_k = \{I_k\}$ denotes a set of images in the category k. Features extracted at scale s from C_k form a feature pool $F_k^s = \{f_k^s\}$. Feature in this pool are then clustered to generate a set of visual words, which can be represented as $V_k^s = \{v_k^s\}$. Feature pools from different scales are clustered independently, and thus independent visual words are constructed. Therefore, $V_i^{s_1} \bigcap V_j^{s_2} = \emptyset$ if $s_1 \ne s_2$ or $i \ne j$.

To take advantage of GPU computing in visual word construction, we modify conventional vector quantization as follows. If we want to quantize n vectors into m target vectors, totally $n \times m$ Euclidean distances are needed to compute. For two d -dimensional vectors $\vec{u} = (u_1, u_2, ..., u_d)$ and $\vec{v} = (v_1, v_2, ..., v_d)$, we can calculate Euclidean distance as

$$\| \vec{u} - \vec{v} \| = \sqrt{\| \vec{u} \|^2 + \| \vec{v} \|^2 - 2\vec{u} \cdot \vec{v}}. \qquad (1)$$

The calculation is broken into smaller steps, and GPU threads can be utilized more effectively. With the decomposed Euclidean distance, the algorithm for quantizing n vectors into m targets is shown in GPU Procedure 2 [3].

The ith row of the $n \times d$ matrix A corresponds to the ith vector in the space. Each row of B indicates a target vector. Line 1 to line 6 calculate sum of squares of each vector in A and B. LengthsA and LengthsB are two arrays storing these vectors' squares of L2-norms. Line 7 computes the inner product term defined in equation (1). Note that this step is accomplished by a matrix multiplication $P = AB^T$. Elements of P are the dot products of the vector pairs from A and B. From line 8 to line 16, the vector from B that is closest to a vector from A is found.

```
GPU Procedure 2
1 → for $i$ = 1 to $n$ do
2 →         lengthsA[$i$] ← $\|\vec{u_i}\|^2$ // $\vec{u_i}$ is the $i$th row of $A$
3 → end for
4 → for $j$ = 1 to $m$ do
5 →         lengthsB[j] ←$\|\vec{v_j}\|^2$ // $\vec{v_j}$ is the $j$th row of $B$
6 → end for
7 → $P$ ← MatrixMultiply($A$, MatrixTranspose($B$))
8 → for $i$ = 1 to $n$ do
9 →         minDist ← ∞
10→         lengthA ← lenghsA[$i$]
11→         for $j$ = 1 to $m$ do
12→                 d ← lengthA + lengthsB[$j$] − 2 $P_{i,j}$
13→                 if d < minDist then minDist ← d, best ← $j$
14→         end for
15→         assignTo[$i$] ← best
16→ end for
17→ return assignTo
```

- GPU Implementations

Each thread on the GPU device controls different portion of data. Each $\|\overrightarrow{u_i}\|^2$ and $\|\overrightarrow{v_j}\|^2$ in this algorithm is computed by a single thread, and all threads are executed in parallel. Hence, the two loops from line 1 to line 6 can be done immediately in the GPU implementation. Moreover, the value $\|\overrightarrow{u_i}\|^2$ just needs to be computed for once, and we put it into the shared memory on the device. Accessing shared memory is much faster than that of global memory, and we can thereby save more time especially when a great number of shared memory access is needed.

The elements of P share the row index of A and the column index of B^T. With the CUDA architecture, all threads are indexed in a 2D manner, and totally $n \times m$ threads are utilized. A thread with index (i, j) can access both the ith row of A and the jth column of B^T. As a result, $n \times m$ dot products can be executed in parallel with only one loop.

Due to hardware limitations, all 2D arrays on the GPU device are simulated by a 1D array by shifting indices. Memory reading is linear when we access a row of a matrix. However, when we access a column of a matrix, memory reading becomes scattered. For this reason, we skip the matrix transpose step, and a thread will access a row of matrix B in place of a column of B^T in practice.

2.5. Visual Word Matching and Scene Classification

Quantizing feature points into visual words is done separately at different scales. That is, features extracted from a specific scale s can only be quantized into the visual words generated from the same scale s. After feature quantization, each scene image is comprised of a set of visual words from different scales.

A multi-class SVM classifier is then constructed based on features extracted from the training data:

$$D = \{(\boldsymbol{x}_i, y_i) | y_i \in \{1, 2, ..., K\}\}_{i=1}^{N}, \qquad (2)$$

where $\boldsymbol{x}_i$ is a vector representing the ith training image, and y_i is a label indicating one of the K categories. This SVM classifier predicts scene label by giving the vector representation of a test image.

2.6. Complexity Analysis

In calculating edge magnitude and orientation, for CPU two loops are used to control rows and columns, and therefore $H \times W$ calculations are needed. For GPU, these calculations can be done immediately since $H \times W$ threads are used. Magnitude and orientation of each pixel are accumulated within a patch, and complexity for this step is: $O(K_s \times \frac{H}{2^{s-1}} \times \frac{W}{2^{s-1}} + c)$, which can be simplified as $O(K_s HW)$, where K_s is the patch number in scale s. Likewise, complexity of the GPU implementation is reduced to $O(1 + c)$ since each pixel is dealt with a thread.

We analyze time costs of the following four implementations for vector quantization: (1) Original Euclidean distance with CPU; (2) Original Euclidean distance plus 1D thread indices with GPU [4]; (3) Original Euclidean distance plus 2D thread indices with GPU [4]; (4) Our implementation. Time cost is defined as number of operations per thread, and Table 1 lists the results. We see that time cost of our work is much less than settings (1) and (2). Although setting (3) has the same time complexity in terms of the big-O notation, our work actually has less time cost since our implementation contains a caching mechanism which not only reduces execution time but also memory accessing time.

Table 1. Time cost of different implementations.

	(1)	(2)	(3)	(4)
Total operations	3nmd - 1	3nmd - 1	3nmd - 1	(nm+n+m)(2d-1) + 3nm
Utilized threads	1	n	n×m	n×m + n + m
Time cost	3nmd -1	3md	3d	2d + c
Complexity	O(nmd)	O(md)	O(d)	O(d)

3. EXPERIMENTAL RESULTS

To evaluate performance, a PC with a quad-core processor (Intel Core i7) with 3.5GB RAM plus a graphics card with a GPU (NVIDIA Geforce GTX 580) is used. The GPU has 512 cores in physical, and the number of available threads can be at most 1024 × 65535 when programming. We evaluate feature extraction, vector quantization, and scene categorization based on using the CPU only, and based on using GPU.

The scene dataset "13 Natural Scene Categories" proposed by Fei-Fei [5] is used for evaluation. We employ a part of this dataset which contains totally 2688 outdoor scene images from 8 categories, including coast, forest, highway, inside city, mountain, open country, street and tall building. Resolutions of these scene images are 256×256.

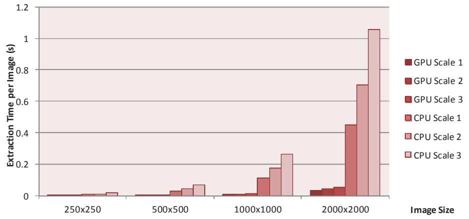

Figure 3. Average time for feature extraction.

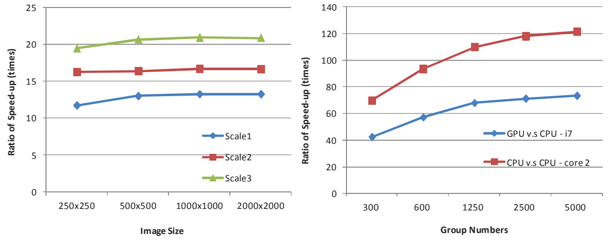

Figure 4. Ratios of speed-up for (a) feature extraction, and (b) vector quantization.

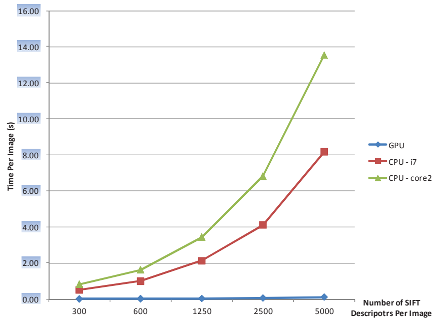

Figure 5. Time cost of vector quantization versus number of descriptors for different implementations.

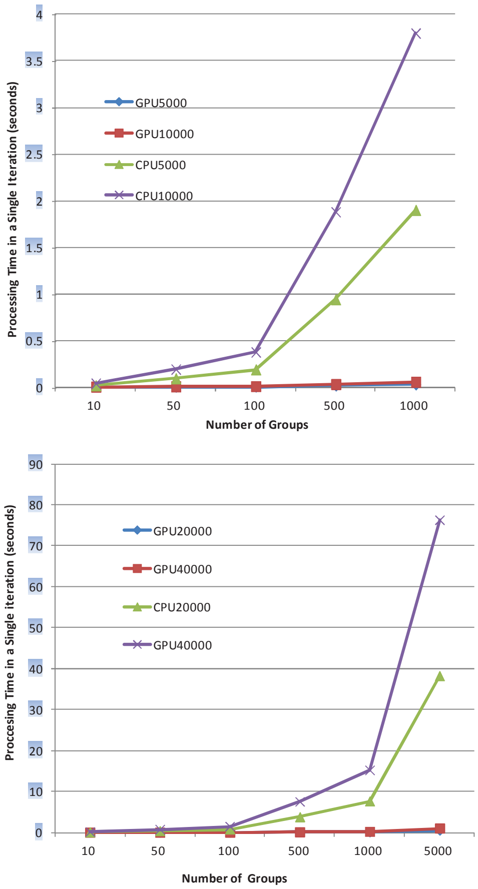

Figure 6. Processing time of K-means clustering with (a) upmost 10000 data points, (b) upmost 40000 data points.

3.1 Execution Speed

Figure 3 shows average time for feature extraction from images of different sizes. Differences between CPU and GPU increase when the image size gets larger. In fact, GPU can be at most 20 times faster than CPU in extracting features. Figure 4(a) illustrates the ratio of speed-up for two implementations at different scales.

In evaluating vector quantization, feature dimension is 128 and the codebook size is 4000. Figure 5 shows time cost of vector quantization versus number of descriptors. Performance of an implementation based on another CPU (core 2) is also shown. Time cost grows as the number of descriptor increases, and there is a huge gap between three implementations. In fact, time cost of CPU vector quantization can be at most 120 times more than that of the GPU (Figure 4(b)).

Figure 6(a) and 6(b) show execution time with a particular number of vectors in a single iteration of the K-means clustering. For example, the curve “GPU5000” shows the time cost variation for quantizing five thousands vectors into several numbers of groups based on GPUs. Curves of the GPU implementations are close to zero all along, and time cost of CPU implementations increases with the group number significantly.

3.2 Performance of Classification

The 10-fold cross validation strategy is adopted to obtain average classification performance. We compare performance between CPU and GPU implementations based on features combining scale 1 and scale 2. As Figure 7 reveals, differences of accuracies between the two implementations are quite small. This verifies that our GPU-accelerated scene classification is competitive with the CPU version without dropping in accuracy rates.

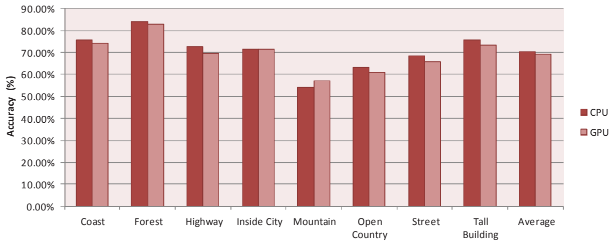

Figure 7. Classification accuracies.

4. CONCLUSION

We propose a fast and efficient scene categorization system which utilizes a category-specific visual word model and is accelerated by GPU. A multiscale feature extraction approach is applied to jointly capture global and local characteristics of a scene. Calculating of Euclidean distance is modified to facilitate vector quantization accelerated by GPUs. Theoretical analysis and experimental results reveal that the accelerated vector quantization needs fewer instructions. GPU implementations are proven to be 20 to 100 times faster than CPU does.

Acknowledgement

The work was partially supported by the National Science Council of Taiwan, Republic of China under research contract NSC 100-2221-E-194-061.

5. REFERENCES

[1] J. Qin and N. H. C. Yung. Scene Categorization with Multiscale Category-specific Visual Words. Optical Engineering, vol. 48, no. 4, 2009.

[2] D.G. Lowe. Object Recognition from Local Scale-invariant Features. IEEE International Conference on Computer Vision, vol. 2, pp. 1150-1157, 1999.

[3] K.E.A. van de Sande, T. Gevers, and C.G.M. Snoek. Empowering Visual Categorization with the GPU. IEEE Transactions on Circuits and Systems Society, pp. 60-70, 2011.

[4] D. Chang, N. A. Jones, D. Li and M. Quyang. Compute Pairwise Euclidean Distances of Data Points with GPUs. IASTD International Conference on Intelligent Systems and Control, pp 278-283, 2008.

[5] 13 Natural Scene Categories Dataset, http://vision.stanford.edu/resources_links.html

[6] J. Sanders and E. Kandrot. CUDA by Example: An Introduction to General-purpose GPU Programming. NVIDIA Corporation, 2010.