GPU-Accelerated Logo Detection

Yu Luo*
University of Pennsylvania

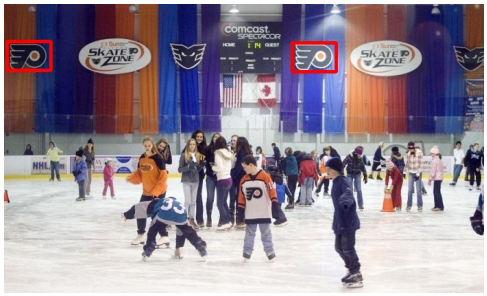

Figure 1: *Logo Detection*

Abstract

Computer vision tasks are usually computationally intensive. This impedes its applications in real world, like real-time object recognition which has high demands for performance. Though vision algorithms often deal with millions of pixels, these per-pixel operations are inherently parallel. With modern CUDA which was introduced by NVIDIA, we can get really cheap computation power by moving the whole pipeline of computer vision algorithms to GPU.

This paper studies the particular task, logo detection on GPU. Histogram of Oriented Gradients(HOG) is a very useful descriptor to represent the shape information of logos. A HOG feature descriptor is implemented on GPU in this paper, and target logos are further detected by a voting strategy. Finally, I analyze the performance of my algorithm and compare it to the same implementation using Matlab.

Keywords: logo detection, computer vision, CUDA, HOG

Links: DL PDF WEB VIDEO CODE

1 Introduction

Logo detection can be considered as a subtask of object recognition, the goal of which is trying to enable computers to automatically detect semantic objects. There are a lot of possible applications of logo detection. Typical applications include adding links on logo automatically for commercial use and replacing detected logo with new logos for image processing.

Human can detect objects easily. However, it's an extremely difficult task for computers since computers do not have any prior knowledge on target images. And it's also very hard to represent the target object using numerical descriptions which have good properties of distinctness and robustness. One of most popular methods for object detection is to use sliding windows [Chum and Zisserman 2007]. Basically, we need to construct a bounding box with a certain scale, and search the entire image using this bounding box. There are some researches on how to improve performance by making use of optimized sliding window searching, like [Lampert and C.H. 2008] which uses a branch-and-bound scheme to approximate the global optimal solution. With CUDA, however, we can improve the performance a lot without changing the underlying algorithm itself.

*e-mail:yuluo@seas.upenn.edu

2 Related Work

Though HOG features are widely used in object detection, there are some other alternatives for feature descriptors. One possible choice is Scale-invariant feature transform(SIFT) [David 1999]. The SIFT descriptor creates a scale-space pyramid by using difference of Gaussian. Interesting points (like corners) are detected in this scale-space pyramid and represented using a vector of 128 dimensions. Then for the logo detection task, we can match key points between reference logo and target image. The main advantage of this method is that logos with different scales in target images can be matched in one pass since SIFT is scale invariant. But one problem with SIFT features for logo detection is that logo may be missed due to insufficient key points. Also, the performance is more likely affected by the false matches.

One similar task to logo detection is pedestrian detection [Dalal and Triggs 2005]. Typically, it also uses HOG features to describe the shape of humans. And usually it uses sliding windows to search humans in input images and predicate the existence of humans in the window based on classification techniques in machine learning. In my paper, I did not use any training examples to train a predication model, while I use a voting method which makes each sample point in the input image vote for possible center positions of logos.

3 Histogram of Oriented Gradients

Histogram of Oriented Gradients was first introduced by [Dalal and Triggs 2005]. It accumulates gradient orientations in each portion of images. Figure 2 shows the flow chart to compute HOG descriptors.

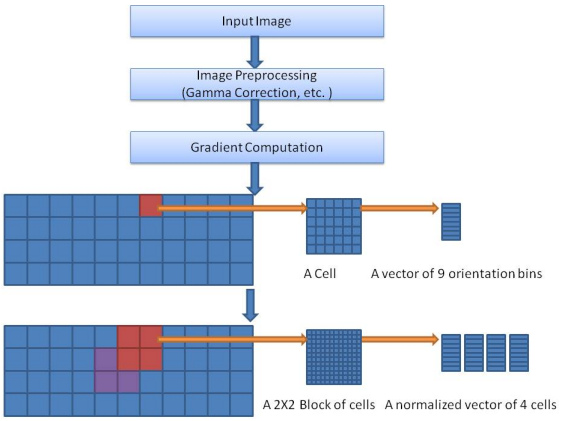

Figure 2: *Flow Chart of HOG*

To achieve better performance, the first step is usually to pre-process the input image by color normalization and gamma correction.

The second step is gradient computation. We need to get gradients along horizontal and vertical orientation. We can simply apply two 1-D kernels, $[-1, 0, 1]$ and $[-1, 0, 1]^T$ to the input image to get G_x and G_y respectively. Then the magnitude and orientation of gradients can be easily computed.

The third step is to divide the images into cells and create histograms for each cell. These cells are usually non-overlapping, and each pixel in the cell votes for a channel of histograms according to its gradient orientation. There are usually 9 channels spread over 180 degrees or 18 channels over 360 degrees. The vote weights can simply be gradient magnitude or some advanced function of magnitude.

After getting a histogram of 9 or 18 channels for each cell, we can concatenate histograms for each block of cells. Each block can contain multiple cells and blocks are usually overlapping. The concatenated vector need to be normalized to deal with small changes in illumination. Finally, we can collect HOG descriptors for all blocks of cells.

4 Approach

The basic idea is to use OpenCV to pre-process the reference and target image, and then pass the image to GPU for parallel computation by using CUDA. After getting the logo center, I can pass the center position back to CPU and display the detected logo. OpenCV is a library for computer vision algorithms. Figure 3 shows the pipeline of my approach. I will focus on algorithms on GPU part in the following sections.

4.1 Gradient Computation

The first step on GPU part is gradient computation. In my implementation, each thread computes gradient magnitude and orientation for one pixel. And based on the orientation, a tag number which indicates the bin number of histogram channel is assigned to the pixel. The step is same for both reference image and target image.

4.2 Codebook and HOG Generation

In this step, codebook is built for reference image. Each pixel computes HOG features for a block of cells. The block size I use is 3×3, and the cell size is 6×6. Basically, each thread accumulates histogram for a 18×18 region. The center point of each 18×18 region is considered as a sample point. Each HOG descriptor saved in codebook is computed around a sample point. The offset of each sample point to the center position in reference image is also saved in the codebook. In addition, I assign a weight to each HOG feature to represent the importance of each vector. This eliminate the negative effects of white areas in reference image since two white area would have exactly same HOG descriptors. And thus the white area tends to be selected as logo center in the voting procedure. The weight is just the number of non-empty channels divided by the total number of histogram channels in a block.

The HOG computation step is repeated for target image, but we don't need to save additional information like offsets to center of images.

4.3 Distance Matrix Computation

After getting HOG representation for reference images and target images, we can compute distance matrix between reference image and target image. Each entry in distance matrix is computed in a thread and it represents the distance between a HOG descriptor in reference image and a HOG descriptor in target image. I use the following formula to compute distance which is called Chi-squared distance.

$$D = \frac{(v_1 - v_2)^2}{v_1 + v_2} \tag{1}$$

where v_1 is a descriptor vector from reference image, and v_2 is a descriptor vector from target image.

4.4 Threshold and Voting

We know each HOG descriptor in reference image and target image is computed around a sample point. Based on the distance matrix, if the distance between a HOG descriptor in target image and an entry in codebook is lower than the threshold, we can vote the corresponding logo position based on the position of sample points and offsets saved in the codebook. The vote weight is computed using the following naive method.

$$W = \frac{w}{D} \tag{2}$$

where w is the weight of the HOG entry in codebook and D is the distance computed in previous step. Each thread deals with one entry in distance matrix. Hopefully, the logo center in target image would get high votes.

4.5 Blurring and Local Maximal Detection

Finally, after getting the voting matrix, we blur it to get a heat image. The intensity of each pixel represents the likelihood of logo center in target image. The blurring operation is done by applying a Gaussian kernel to each pixel in voting matrix. The kernel size is 31 and the sigma(standard deviation) of Gaussian function is 5. If the intensity of a pixel in heat image is greater than the threshold and it's a local maximal in 8-connected neighborhood, it would be considered as detected logo center.

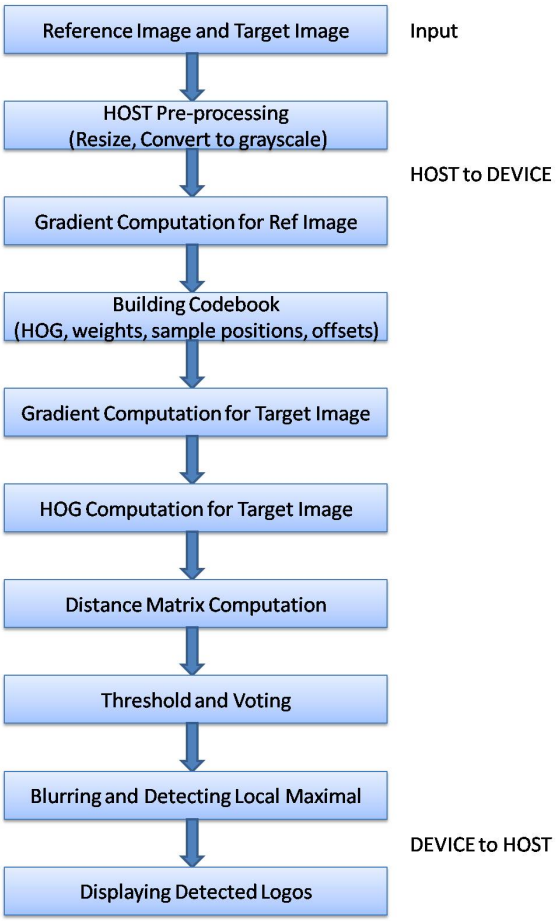

Figure 3: *Logo Detection Pipeline*

5 Results

Figure 5 shows the percentage of time each step takes when the reference image is Figure 5 and the target image is right image in Example 2. The bottleneck is distance matrix computation step due to lots of memory accesses when computing the distance between two vectors, each of which has 81 dimensions. Also, the Gaussian blur step takes a lot of time because the kernel size is too large(31). Local Maximal Detection is slow because I use Thrust::sort to get the global maximal value and transfer the logo centers to HOST.

Some other examples are showed. The left image in each example is heat image which represents the possibility of logo center. The right image is the result image which shows detected logos.

	My implementation using MATLAB	My implementation using CUDA
Image Size	700*420	700*420
Time(ms)	91849.188	1992.496

Figure 4: *Performance*

6 Future Work

The bottleneck of my implementation is Distance Matrix Computation step. There may be some better ways to optimize this step.

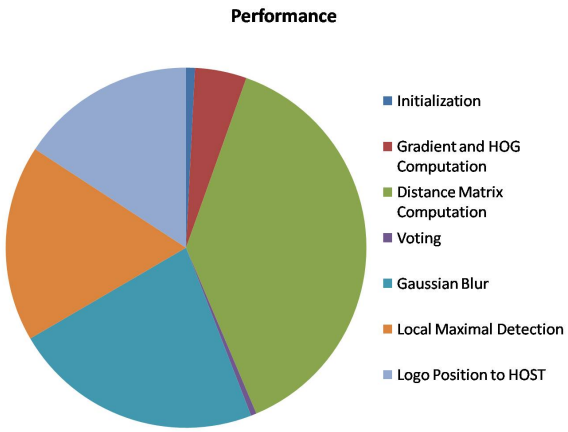

Figure 5: *Performance*

Figure 6: *Reference Logo*

Also, there are spaces for improvement in other steps. For example, the HOG computation can be splitted into two steps. One is computing histogram for each cell and the other is concatenating and normalizing histograms for each block of cells.

References

CHUM, O., AND ZISSERMAN, A. 2007. An exemplar model for learning object classes. *IEEE Conference on Computer Vision and Pattern Recognition*.

DALAL, N., AND TRIGGS, B. 2005. Histograms of oriented gradients for human detection. *IEEE Conference on Computer Vision and Pattern Recognition*, 886–893.

DAVID, L. 1999. Object recognition from local scale-invariant features. *Proceedings of the International Conference on Computer Vision*, 1150–1157.

LAMPERT, AND C.H. 2008. Beyond sliding windows: Object localization by efficient subwindow search. *IEEE Conference on Computer Vision and Pattern Recognition* (June), 1–8.

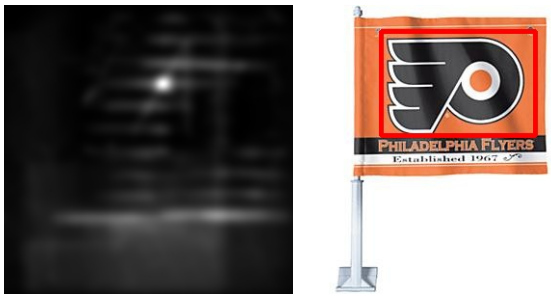

Figure 7: *Example 1*

Figure 8: *Example 2*

Figure 9: *Example 3*