LETTER

Implementation of Scale and Rotation Invariant On-Line Object Tracking Based on CUDA

Quan MIAO[†a)], ***Student Member***, **Guijin WANG**[†b)], ***Member***, ***and*** **Xinggang LIN**[†c)], ***Nonmember***

SUMMARY Object tracking is a major technique in image processing and computer vision. Tracking speed will directly determine the quality of applications. This paper presents a parallel implementation for a recently proposed scale- and rotation-invariant on-line object tracking system. The algorithm is based on NVIDIA's Graphics Processing Units (GPU) using Compute Unified Device Architecture (CUDA), following the model of single instruction multiple threads. Specifically, we analyze the original algorithm and propose the GPU-based parallel design. Emphasis is placed on exploiting the data parallelism and memory usage. In addition, we apply optimization technique to maximize the utilization of NVIDIA's GPU and reduce the data transfer time. Experimental results show that our GPGPU-based method running on a GTX480 graphics card could achieve up to 12X speed-up compared with the efficiency equivalence on an Intel E8400 3.0 GHz CPU, including I/O time.

key words: *object tracking, classifier updating, GPGPU, CUDA*

1. Introduction

Robust object tracking is an important task with many applications, ranging from visual surveillance to human-computer interfaces. The difficulties of object tracking include complicated object appearance variations, such as illumination change, partial occlusion and cluttered scenes.

In our earlier work [1], we proposed a new object tracking scheme which employs the invariance of local features to guide an on-line boosting technique. The resulting tracker has been proven to achieve robust and accurate object tracking, especially under complex appearance changes. However, the discriminative on-line learning of local features becomes a major bottleneck with respect to computing time. The incorporation of local features' invariance in the weak classifiers and the subsequent updating consume significant CPU resources, limiting its usage in applications with real-time constraints.

Fortunately, modern GPGPU has evolved into a highly parallel, multithreaded processor with huge computational power and high memory bandwidth [2]. A serial processing problem can often be partitioned into coarse sub problems solved independently in parallel. Now GPU parallel computing has seen increasing applications in the domain of image processing and computer vision, such as face detection [3], feature matching [4], human detection [5] and pattern classification [6]. Despite the tremendous computing capacity of GPU, utilizing it to deal with practical problems is non-trivial. It is obviously unprofessional to only concern the definition of blocks and threads without utilizing all the GPU's resources to maximize the computing efficiency. On one hand, how to interpret data structure and handle different memory spaces based on the algorithm itself has to be considered. On the other hand, data loading between CPU and GPU is the major factor which significantly detracts from the advantages of parallel computing. The way to overcome it merits discussion.

This paper performs efficient parallel design on the basis of analyzing the on-line object tracking system. Through collaborative interaction of different kernels and suitable data parallelism, we can efficiently achieve scale- and rotation-invariance. Algorithmic adjustments and implementation details are described. For best overall object tracking performance, we employ stream technique based on page-locked memory to improve concurrency and optimize the data transfer process between CPU and GPU.

2. On-Line Boosting for Object Tracking

In [1], we propose a new feature-based tracking scheme by employing adaptive classifiers to match the detected keypoints in consecutive frames (see Fig. 1). The object region is transformed using the homography estimated based on the matching candidates. In [7], we further expand the idea and provide in detail a new framework.

Specifically, each classifier C is composed of J selectors h_j^{sel} and holds a weak classifier pool X from which the training procedure selects the selectors with the minimal estimated error. The classifier wishes to predict the matching confidence of a point $\mathbf{x}$ by:

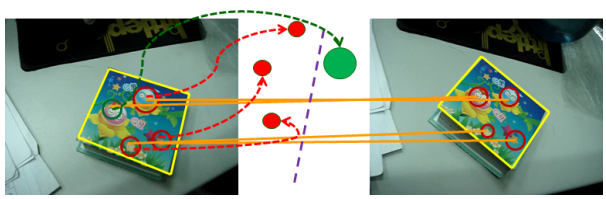

Fig. 1 Establish feature correspondences by classifier-based keypoint matching. The object region is transformed based on estimated homography.

Manuscript received July 6, 2011.
Manuscript revised September 1, 2011.

[†]The authors are with the Department of Electronic Engineering, Tsinghua University, Beijing 100084, China.
a) E-mail: miaoq07@mails.tsinghua.edu.cn
b) E-mail: wangguijin@tsinghua.edu.cn
c) E-mail: xglin@tsinghua.edu.cn

DOI: 10.1587/transinf.E94.D.2549

$$C(\mathbf{x}) = \sum_{j=1}^{J} \alpha_j \cdot h_j^{sel}(\mathbf{x}) \Big/ \sum_{j=1}^{J} \alpha_j. \quad (1)$$

As new samples arrive sequentially, each selector h_j^{sel} re-selects the best weak classifier and the corresponding voting weight α_j is updated (see Fig. 2). In [8], Grabner et al. correspond each weak classifier to a simple Haar feature computed in a fixed bounding patch centered at the corresponding keypoint. Despite the simple computation, such approaches can only deal with ordinary object changes such as pure translations and slight rotations. Under other more complex changes in the target object, the updates of each classifier will fail, causing poor tracking performance and few applications of such techniques in reality.

The main novelty of [1] is that the scale and dominant orientation of each SURF feature is used to guide the on-line learning process. Each weak classifier seeks its corresponding Haar feature within the scale- and rotation-invariant window. The size of the Haar feature changes with the current scale. Furthermore, the responses are distributed in horizontal and vertical direction in relation to the dominant orientation. Experiments show the resulted tracker is able to handle viewpoint change, partial occlusion and complex transformations [7].

However, updating the whole classifier pool and constructing the final strong classifiers result in high computation complexity. Unlike the situation of [9] in which a certain response (horizontal or vertical) is sufficient to compute a Haar feature, both the horizontal and vertical responses have to be computed in our scheme and then combined in relation to the dominant orientation. In addition, the center of each weak Haar feature is no longer fixed; it has to be reconsidered with respect to the scale and dominant orientation. The shape of Haar feature's rectangular region must be also scale normalized. For 60 strong classifiers each consisting of 20 selectors and 250 weak features, it will cost about 215 ms to tackle object tracking for each frame, whereas the on-line updating using 40 samples would require up to 180 ms, which occupies more than 80% of the whole time. Therefore, accelerating the on-line boosting algorithm is of great significance to improve the efficiency of object tracking. This paper focuses on a GPGPU accelerated parallel solution for the on-line updating technique, to achieve efficient tracking.

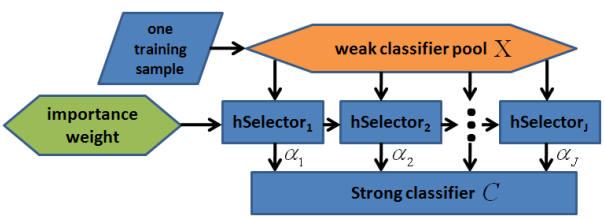

Fig. 2 On-line boosting for a strong classifier.

3. GPU-Boosted On-Line Object Tracking

Boosting algorithm may seem intuitively suitable for parallel processing. One option is to roughly assign each keypoint to one block and each weak classifier (250 in total for each strong classifier) to one thread. In this case, each thread has to independently take charge of the complex scale- and rotation-based computation, followed by classifier's updating. If we alternatively assign each thread to only one sample and every M threads correspond to a weak classifier, $250 \times 40 = 10000$ threads will be needed, which completely exceed the upper limit (1536 for GTX480 card) of the threads each block could contain. In addition, the corresponding memory allocation is non-trivial. It is optimal to store the Haar feature pool in the shared memory, rather than the global memory. Nevertheless, the size of shared memory leads to several further limitations. All these ingredients make this proposed problem different from the previous boosting techniques, and thus form new difficulties to be resolved. This section performs efficient parallel design to overcome difficulties mentioned above. The ultimate aim is to make the whole tracking system applicable in real-time tasks.

3.1 Implementation on GPU Platform

As is mentioned above, the series of operations to achieve scale- and rotation-normalization make the mechanism rather complex. Generally, we employ three kernels working collaboratively on the on-line updating scheme. Figure 3 shows the assignment of the kernels.

The first kernel called Initial-HaarLocation extracts each sample's patch and establishes the new location and size of each weak feature according to the sample's scale and orientation. This kernel is implemented on a grid of 40 thread blocks, each in charge of processing one sample. Each block consists of 250 threads finding the new location and size of Haar rectangles. The pseudo CUDA implementation of Initial-HaarLocation kernel is illustrated in Table 1.

The second kernel called Update-WeakClassifier updates the weak classifiers of each strong classifier. The ker-

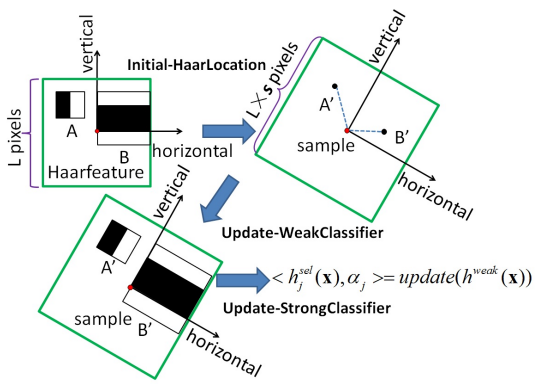

Fig. 3 The process of classifier learning based on sequential kernels.

Table 1 CUDA implementation of Initial-HaarLocation kernel.

Initial-HaarLocation<<< 40, 250 >>>(Haarfeature, Samples) // Launch a grid of 40 blocks and each block contains 250 threads // **blockIndex** will be the index of thread block // **threadIndex** will be the index of thread in the block
$Samples(blockIndex).size = L \times Samples(blockIndex).scale$; $ori = Samples(blockIndex).orientation$; $Samples(blockIndex).freature(threadIndex).x = Haarfeature(threadIndex).x \times \cos(ori) - Haarfeature(threadIndex).y \times \sin(ori)$; $Samples(blockIndex).freature(threadIndex).y = Haarfeature(threadIndex).x \times \sin(ori) + Haarfeature(threadIndex).y \times \cos(ori)$; $Samples(blockIndex).freature(threadIndex).size = Haarfeature(threadIndex).size \times Samples(blockIndex).scale$;

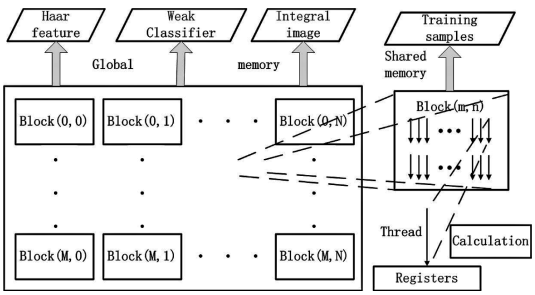

Fig. 4 Storage structure of Update-WeakClassifier kernel on GPU.

nel is implemented on a grid of 60 thread blocks, each corresponding to a strong classifier. Inside each block every two thread correspond to one weak classifier, so there will be 500 threads in total. The two threads respectively compute the horizontal and vertical Haar responses according to the weak feature's location and shape the first kernel outputs. After synchronizing all the threads, the consecutive two features are scale- and rotation-normalized and combined to estimate the feature value corresponding to the current weak classifier. Furthermore, we employ the threads with even index to update the parameters of each weak classifier's classifying model, such as the mean and variance, for subsequent tracking.

Figure 4 shows the storage structure. The Haar feature pool, the computed integral image and the classifiers are written into the global memory because the size of shared memory is limited (48 KB for GTX480). These arrays are ordered in a coalesced way to maximize the speed of reading data from global memory to shared memory. Take the Haar feature pool for example, as is illustrated in Fig. 5. Each thread corresponds to a Haar feature; the parity of the current thread's index determines whether horizontal or vertical one is used. If the horizontal feature and the vertical feature are stored in row major (see Fig. 5 (a)), the global memory accesses between consecutive threads fall into locations separated by an offset of 250 (column number) data elements. But, if the feature is stored in an interaction way (see Fig. 5 (b)), the data read by each set of 32 adjacent threads fall into 32 consecutive locations of global memory which meets the coalesced access requirement and hence gives rise to 500/32 coalesced read transactions. Thus the kernel can

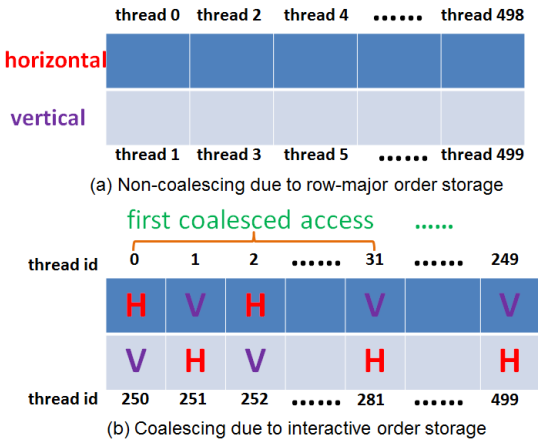

Fig. 5 Data arrangement involving memory coalescing.

be accelerated.

The third kernel called Update-StrongClassifier updates all the strong classifiers by selecting the 20 selectors from their updated weak classifiers. The kernel is implemented on a grid of 60 thread blocks, each consisting of 250 threads. First each thread updates the accumulated estimated error of its corresponding weak classifier with respect to the importance weight of each sample. Then the weak classifier with smallest error is selected by the selector; the corresponding voting weight α_j is updated.

3.2 Optimization Methods for CPU-GPU Memory Access

As the peak bandwidth between the device memory and the GPU is much higher than the peak bandwidth between host memory and device memory, minimizing data transfer between the host and the device is very important for saving the I/O time. Our approach is optimized in this aspect.

Since the process of data transfer between host and device is unavoidable, the solution can be converted to overlap kernel execution with data transfer, which is called concurrent copy and execute. On devices that have this capability, page-locked (or pinned) memory [10] is required to achieve higher bandwidth between host and device. In addition, the data transfer and kernel must use different, nonzero streams. A stream is a sequence of commands that execute in order. Classifier updating in different streams can be interleaved and overlapped with stream transferring from CPU to GPU. Thus, part of the I/O time is "hided" by the device computations.

4. Experimental Results

We now present the experimental results of applying our algorithm on various image sequences with the size of 640×480. The platforms used are NVIDIA's GTX480 card and Intel E8400 3.0 GHz CPU, respectively. Comparisons are carried out as follows.

Figure 6 shows the speedups when running our parallel approach. To better illustrate the soundness of our design,

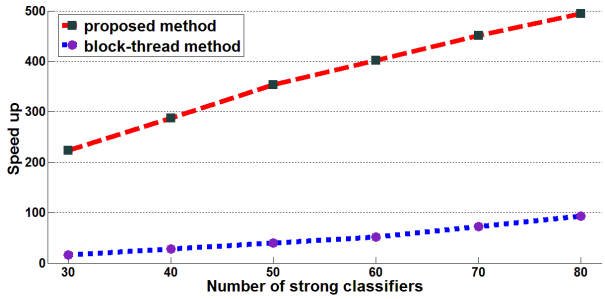

Fig. 6 Speedups with varying number of strong classifiers.

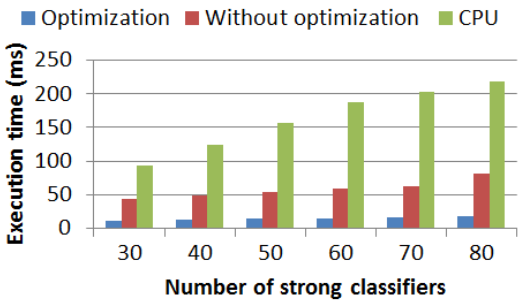

Fig. 7 Execution time comparison.

we perform another GPU-based implementation which directly assigns each keypoint to one block and each weak classifier to one thread. We call this approach the block-thread method. As the number of strong classifiers increases, this block-thread method has not shown obvious speedups. In contrast, the proposed method exhibits good scalability, validating the superiority of our implementation. For 60 strong classifiers, our GPU-based on-line boosting achieves 400X compared to its counterpart implementation on CPU.

Although our design of on-line boosting shows very high degree of parallelization, data transfer between CPU and GPU is still a major factor which negatively affects the overall speed of object tracking. For 60 strong classifiers, the parallel computing time is 0.45 ms, while the data transfer time will cost nearly 60 ms. Hence, we employ the optimization technique mentioned above. Figure 7 verifies the performance of optimization. For 60 strong classifiers, the optimized version takes only 15 ms (including I/O time) to execute on-line learning, realizing 12X speedup compared to CPU implementation (180 ms). With the rest CPU computational modules (about 35 ms), our CPU-GPU cooperative platform needs only 50 ms for the overall object tracking, achieving a real-time speed of 20 fps.

5. Conclusion

This paper developes a GPU-based parallel implementation of a novel on-line tracking system. Three kernels work collaboratively to compute the scale- and rotation-normalized Haar features and update the strong classifiers. Reasonable organization of data construction based on memory hierarchy further improves computing efficiency. Through design optimization, our parallel on-line tracking presents up to 12X speedup, including I/O time. The resulted tracker can process 640×480 sized videos in real time.

This work shows a new way of implementing high performance computing systems. We believe the GPU-based parallel computing will keep providing compelling benefits for tracking and other real-world industrial tasks. Future work will involve performing multi-GPGPU associated processing in one computer to achieve multiple camera tracking.

References

[1] Q. Miao, G. Wang, X. Lin et al., "Scale and rotation invariant feature-based object tracking via modified on-line boosting," ICIP, pp.3929–3932, 2010.
[2] NVIDIA, NVIDIA CUDA programming guide version 2.3, NVIDIA, 2009.
[3] J. Kong and Y. Deng, "GPU accelerated face detection," ICIP, pp.584–588, 2010.
[4] N. Cornelis and L. Van Gool, "Fast scale invariant feature detection and matching on programmable graphics hardware," CVPR, 2008.
[5] S. Tang and S. Goto, "Accurate human detection by appearance and motion," IEICE Trans. Inf. & Syst., vol.E93-D, no.10, pp.2728–2736, Oct. 2010.
[6] S. Liang, Y. Liu, C. Wang, and L. Jian, "A CUDA-based parallel implementation of K-nearest neighbor algorithm," CyberC, pp.291–296, 2009.
[7] Q. Miao, G. Wang, C. Shi, X. Lin et al., "A new framework for on-line object tracking based on SURF," Pattern Recognit. Lett., vol.32, no.13, pp.1564–1571, 2011.
[8] M. Grabner, H. Grabner, and H. Bischof, "Learning features for tracking," CVPR, 2007.
[9] H. Grabner and H. Bischof, "On-line boosting and vision," CVPR, 2006.
[10] NVIDIA Corp., NVIDIA CUDA C program best practices guide, Santa Clara, CA, NVIDIA, 2009.