A Parallel Implementation of TLD Algorithm Using CUDA

Zhang Ping, Sun Yongqi*, Wu Yali, Zhang Rui
Beijing Key Lab of Traffic Data Analysis and Mining,
School of Computer and Information Technology,
Beijing Jiaotong University
Beijing, 100044, P. R. China
*yqsun@bjtu.edu.cn

Abstract—**Visual tracking is an important issue of computer vision, TLD is an on-line visual tracking algorithm with good robustness and high accuracy properties. However, the real-time performance of TLD is low for the large size video sequences. In this paper, we study the most time-consuming stages of TLD, and then propose a parallel algorithm based on CUDA. The experimental results show that the speedup of our algorithm reaches up to 2.59 compared to TLD while maintaining the same detection accuracy.**

Keywords-visual tracking; parallel processing; TLD; GPU; CUDA

I. INTRODUCTION

Visual tracking is an important issue in the field of computer vision and is widely used in many applications such as surveillance, perceptual user interfaces, augmented reality and driver assistance [1-4]. Visual tracking can be defined as the problem of estimating the trajectory of an object in the image plane as it is moving around a scene. Usually, there are two stages in visual tracking: object detection and object tracking, which are introduced in detail as follows.

Object detection is to find the localization of objects in a frame. There are two categories of methods: sliding window-based approaches and feature-based approaches. The sliding window-based approaches [5] scan the image by windows of various sizes and each window is detected for object. The feature-based approaches usually consist of the following steps: feature detection, feature recognition, and model fitting. There are some common features used in objection: color, edge, contour, texture and keypoint.

Object tracking is to estimate the object motion. There are mainly three kinds of methods in object tracking: optical flow, temporal difference and background subtraction [6-8]. Optical flow is computed using the brightness constraint, which assumes brightness constancy of corresponding pixels in consecutive frames. In the practice of temporal difference, the adjacent frames are subtracted to track the moving object. This method can adapt to illumination changes but is affected by the moving speed of object badly. In the practice of background subtraction, a representation of the scene called the background model is built and then the object can be tracked by finding deviations from the model for each frame.

II. PRELIMINARIES OF TLD

Tracking-learning-detection (TLD) [9] is a tracking algorithm proposed by Kalal, Mikolajczyk and Matas from University of Surrey. TLD is a framework designed for long-term tracking of an unknown object in a video stream. It consists of three stages: tracking, detection and learning. In the tracking stage, the object's motion is estimated using Median-Flow tracker [10]. The tracker is likely to fail and never recover if the object moves too fast or out of the frame. When the tracker fails, the detector scans each frame by a scanning window and decides about the presence or absence of the object for each patch. The patches are generated within all possible scales and shifts. As with other detector, there exist two types of errors in the detection: false positives and false negatives. The errors will be treated in the learning stage. Learning observes the results of both tracker and detector, handles the errors of detector, and updates the object model of the detector to avoid these errors in the future.

The TLD algorithm has been implemented by Arthurv [11] based on OpenCV using C++ programming language, however the real-time performance is still poor. In order to find the bottleneck of the real-time performance, the execution times of three stages are compared in our experiment, in which there are three kinds of images are used, their resolution are 320×240 (QVGA), 352×288 (CIF), and 640×480 (VGA) respectively. The results are shown in Table I, which shows that the detection stage is the most time-consuming stage. The three stages of detection, variance filter, ensemble classifier and nearest neighbor classifier are described as follows.

TABLE I. EXECUTION TIME COMPARISON OF EACH STAGE DURING THE RUNNING (MS)

Video sequences	Size	total	Detection	Tracking	Learning
Car	320×240	57.68	41.77	2.33	3.89
Data1	352×288	68.38	42.21	2.48	5.57
Data2	640×480	117.71	98.38	3.21	2.88

A. Variance Filter

In this stage, all patches with the gray-value variance smaller than 50 percent of the patch of the object are rejected. Let p be a patch, the variance $D(p)$ can be calculated by $D(p)=E(p^2)-E^2(p)$, where $E(p)$ is expectation of p, $E(p^2)$ is the expectation of p^2.

B. Ensemble Classifier

Ensemble classifier is the second stage of the detection. The input is the image patches that were not rejected by the variance filter. The ensemble classifier consists of n base classifiers. Each base classifier i computes many pixel comparisons on the patches, obtaining a binary code x which indexes to an array of posteriors Pi(y|x), where $y \in \{0,1\}$. For each patch, if the average posteriors of all the n base classifiers is larger than 0.5, the patch is classified as the object.

1) Pixel Comparisons

Each base classifier consists of a set of pixel comparisons. All the pixel comparisons are generated in the initialization stage at random and fixed during running. First, the image is convolved with a Gaussian kernel with standard deviation of 3 pixels to increase the robustness to shift and noise. Next, the pixel comparisons are used on the patch. Each comparison returns 0 or 1 and all the return values for each patch compose a binary code *x*.

2) Posterior Probabilities

Each base classifier i ($0 \le i \le n-1$) maintains a distribution of posteriors probabilities $P_i(y|x)$. The distribution has 2^d entries, where d is the number of pixel comparisons, and let $d = 13$ according to experience. The probability is estimated as $P_i(y \mid x) = \frac{\#p}{\#p + \#n}$, where $\#p$ and $\#n$ correspond to the number of positive and negative patches.

3) Initialization and Update

In the initialization stage, all base posterior probabilities and the values of $\#p$ and $\#n$ are set to zero. If a positive patch is classified as negative incorrectly, then $\#p=\#p+1$, similarly, if a negative patch is classified as positive, then $\#n=\#n+1$, and the corresponding probability is updated.

C. Nearest Neighbor Classifier

After the patches are filtered by ensemble classifier, the remaining patches are classified by the nearest neighbor classifier in this stage. The similarity between a patch and model is used to indicate how much the patch resembles the model. Firstly, the similarity between two patches p_i and p_j is defined as formula (1), where *NCC* is the Normalized Correlation Coefficient of two patches, it is calculated by formula (2).

$$S(p_i, p_j)=0.5(NCC(p_i, p_j)+1) \tag{1}$$

$$NCC(p_i, p_j) = \frac{\sum_{x,y}(p_i(x,y)\cdot p_j(x,y))}{\sqrt{\sum_{x,y} p_i(x,y)^2 \cdot \sum_{x,y} p_j(x,y)^2}} \tag{2}$$

Object model M is a data structure that represents the object and its surrounding observed. It is a collection of positive and negative patches,

$$M = \{q_1^+, q_2^+, ..., q_{n_1}^+, q_1^-, q_2^-, ..., q_{n_2}^-\}.$$

q^+ and q^- are the object and background patches respectively. The similarity between the patch and model is measured by S^r, and S^r is calculate by the following formula,

$$S^r(p,M) = \frac{S^+(p,M)}{S^+(p,M)+S^-(p,M)}, \tag{3}$$

where S^+ is the similarity with the positive nearest neighbor, $S^+(p,M) = \max_{p_i^+} S(p, p_i^+)$, S^- is the similarity with the negative nearest neighbor, $S^-(p,M) = \max_{p_i^-} S(p, p_i^-)$. A patch is classified as the object if $S^r(p,M) > \theta_{NN}$, where $\theta_{NN} = 0.6$.

III. Implement of CUDA-TLD

The block diagram of our implementation of TLD based on CUDA is shown in Fig. 1. The tracking, learning and integrator stages run on CPU, while the detection stage runs on GPU. We focus on the detection stage in the implement of CUDA-TLD in this section.

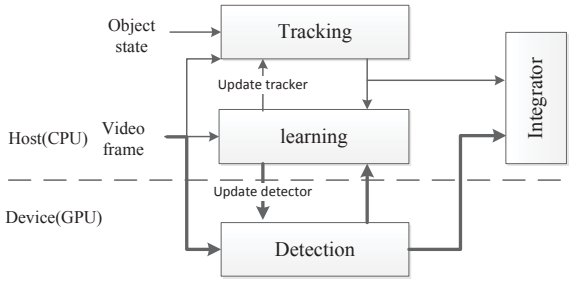

Figure 1. Detailed block diagram of the CUDA-TLD

A. CUDA Introduction

Compute Unified Device Architecture (CUDA) is a parallel computing architecture and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) [12].

In the model of CUDA programming, the function executed on the GPU is called the *kernel function*. Each CUDA thread grid typically comprises thousands to millions of lightweight GPU threads per kernel invocation, and it consists of one or more thread blocks. All blocks in a grid have the same number of threads. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.

There are five kinds of memory in CUDA: registers, shared memory, global memory, constant memory and texture memory. Registers are private for each thread. Shared memory of each thread block is visible to all threads of the block, and

has the same lifetime as the block. Variables that reside in registers and shared memories can be accessed at very high speed in a highly parallel manner. However, the capacities of both register and shared memory are very small. For example, both of them are only 16KB for the graphic card GTX 550 Ti of NVIDIA used in our experiment. Global memory occupies the most majority of memory, and it can be accessed by all threads. However its access speed is much slower than register and shared memory. The access efficiency of global memory is one of the most important factors of CUDA kernel function performance. The constant and texture memory are two read-only memory, accessible for all threads. In particular, constant memory can be accessed efficiently, however the total size of constant memory in an application is limited at 65,536 bytes.

B. Parallel Implement of Variance Filter

First, all the patches to be evaluated are generated by scanning the frame using windows of various sizes. In the implementation, we limit the minimum size of the patches to *min_window*×*min_window* pixels. The parameters *min_window* is determined by the size of object. All the bounding boxes of the patches are transferred from host to device global memory and bounded to texture memory, so are the image and integral image data. Next, all the patches are processed in a thread grid. We assign 256 threads in each thread block in the implement, and each thread calculates the variance of one patch by formula $D(p) = E(p^2) - E^2(p)$. Finally, the patches, whose variance is larger than the variance threshold, are stored into global memory.

C. Parallel implement of ensemble classifier

The ensemble consists of n base classifiers. Each base classifier consists of a set of pixel comparisons. In the implement based on CUDA, we decompose the computation into two steps: the preparation of input data and the classification of the patches.

1) The Preparation of Input Data

The input of the ensemble classifier is the image patches that were not rejected by the variance filter. These patches were stored in the global memory. For the pixel comparison, all the feature points compared are generated in the initialization stage at random and transferred from host to device global memory. In the implement, there are 12 scales, each scale has 10 base classifiers, and each base classifier has 13 pixel comparisons, each pixel comparison contains 2 feature points. So the size of the feature points selected is 12×10×13×4 bytes = 6240 bytes, it is smaller than the limit size of constant memory. Hence, the feature points are stored into the constant memory for accelerating the access speed. After the comparison, the value index to an array of posteriors, where the size of the posteriors is 10×8192. In the implement, the array of posteriors are stored on both CPU and GPU, all the elements of the posteriors are set to zero on CPU and GPU in the initialization. The elements of the posteriors updated in the learning stage are transferred from host to device in the processing of each frame.

2) The Classification of the Patches

When all the data are loaded on GPU, the patches are classified. The image is convolved with a Gaussian kernel. Assume there are *varisNum* patches that were not rejected, the grid is divided into *varisNum* thread blocks, and each thread block classifies one patch. There are 10 base classifiers and each base classifier has 13 pixel comparisons, so each thread block is divided into 10×13 threads, and each thread computes one pixel comparison.

In the implementation, each thread compares the values of two feature points and gets a code b_{ij} (0 or 1), where i is the horizontal coordinate of the thread in the thread block, j is the longitudinal coordinate. Then the thread does a shift operation of b_{ij} according to the longitudinal coordinate of thread as: $x_{ij} = b_{ij}2^j$ ($0 \le i \le 9$, $0 \le j \le 12$). Shared memory of size 13×10×4 bytes = 520 bytes is used in each thread block to store the pixel comparison result computed by each thread.

After x_{ij} is calculated by threads, all the x_{ij} of each base classifier are summed using a parallel reduction method [13] by formula $x_i = \sum_{j=0}^{12} x_{ij}$, and all the x_i are stored into global memory and transferred back to host for the use of learning. Next, the value x_i of each based classifier indexes to an array of posteriors to obtain the posterior $P_i(y|x_i)$. All the posteriors $P_i(y|x_i)$ of patch p are accumulated by $P_p = \sum_{i=0}^{9} P_i(y \mid x_i)$, and the patch p is classified as the object if $P_p > 5$, otherwise the patch is rejected. Finally, all the patches classified as object are stored into global memory.

D. Parallel Implement of Nearest Neighbor Classifier

After the patches are filtered by the variance filter and ensemble classifier, the remaining patches are classified by the neighbor classifier in this stage.

1) The Preparation of Input Data

The input date of nearest neighbor classifier is the patches classified as object by the ensemble classifier and all the patches in object model. The patches classified as object are already stored in the global memory. The patches in the model are updated in the learning stage, and the updated patches are transferred from host to device global memory and bounded to texture memory.

2) The Classification of the Patches

All the similarity measures between the object model and the patches filtered by the ensemble classifier are calculated by formula (3). Before calculating, the patches are resampled to a normalized resolution (*patch_size*×*patch_size* pixels). In order to maximize parallelism, all the patches are processed by a thread grid in one time. Suppose there are n_1 positive patches, n_2 negative patches in the object model M, and m patches that were not rejected by the ensemble classifier. The grid is divided into $(n_1+n_2)\times m$ thread blocks, where each thread block calculates the similarity measure between one input patch and one patch of the model. Note that the largest thread number of each thread block is 1024 for the graphics card GTX 550 Ti. Each thread block is divided into *threadnum* threads, where the variable *threadnum* is defined as: *threadnum*= $\min\{patch\_size \times patch\_size, 1024\}$. If *patch_size*×*patch_size* is larger than 1024, some threads need to calculate more than one point.

In the calculation of the Normalized Correlation Coefficient between one input patch p_i ($0 \le i \le m-1$) and one patch q_j ($q_j \in M$) of the model in each thread block by formula (2), firstly each thread calculates three values: $p_i(x, y) \times q_j(x, y)$, $p_i^2(x, y)$ and $q_j^2(x, y)$, where (x, y) represents the point calculated by the thread. The three values of each thread are stored into the shared memory. Then these values calculated by the threads in the same thread block are summed using the parallel reduction method as before to calculate the similarity measure $S(p_i, q_j)$ between the two patches by formula (1).

After the similarity measures between the patch p_i and all the patches in the model are calculated, the similarity measure between patch p_i and the object model $S^r(p_i, M)$ is calculated by formula (3). And then if the similarity is larger than 0.6, the patch p_i is classified as the object. All the patches that were classified as object are stored into global memory and then transferred to host for the use of learning and integrator.

IV. EXPERIMENTS

A. Environments

Hardware. The hardware is 3.30GHz Intel CPU, 4GB memory, NVIDIA GeForce GTX 550 Ti with 1.8GHz graphics core and 512MB graphics memory.

Software. The software is OpenCV 2.4.1 and CUDA 4.1. The TLD running serially on CPU is open-tld [16].

The data set. There are three kinds of sequences to be used, the resolution of them are 320×240 (QVGA), 352×288 (CIF) and 640×480 (VGA) respectively. The QVGA sequences are publicly available, the others are our own. The snapshots of the data set are shown in Fig. 2.

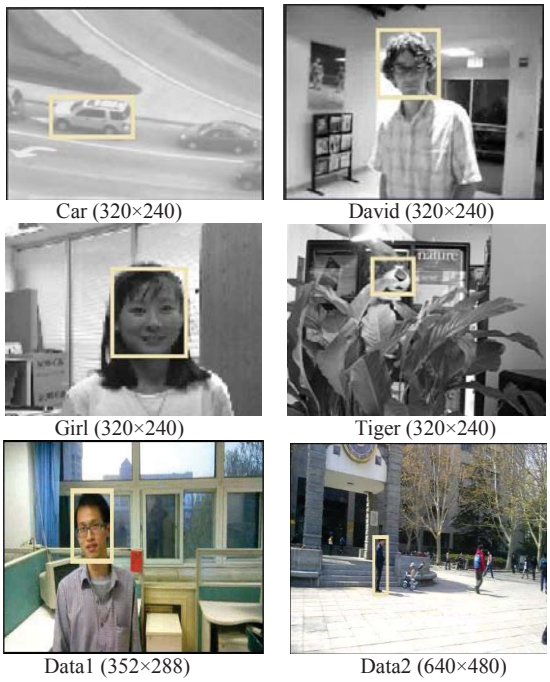

Figure 2. Snapshots of the sequences

B. Results

1) Evaluation of the Ensemble Classifier of CUDA-TLD

In the experiment, the executing times of the ensemble classifier of CUDA-TLD and TLD are compared on the data set. The results are shown in Table II, where the executing times of GPU include CPU-GPU data transfer time. We can see that the ensemble classifier of CUDA-TLD is more effective than TLD, and the speedup achieves up to 2.50.

TABLE II. COMPARISON OF THE EXECUTION TIMES OF THE ENSEMBLE CLASSIFIER (MS)

Video sequences	Size	GPU	CPU	Speedup
Car	320×240	3.81	7.46	1.96
David	320×240	3.86	9.36	2.42
Girl	320×240	7.85	17.86	2.35
Tiger	320×240	7.39	16.29	2.20
Data1	352×288	9.48	23.69	2.50
Data2	640×480	33.56	76.61	2.28

2) Evaluation of the Nearest Neighbor Classifier of CUDA-TLD

In order to evaluate the efficiency, the executing times of the nearest neighbor classifier of CUDA-TLD and TLD are compared on the data set, see Table III, where the executing times of GPU include CPU-GPU data transfer time. The results show that the nearest neighbor classifier of CUDA-TLD is more effective than TLD, and the speedup achieves up to 17.57.

TABLE III. COMPARISON OF THE EXECUTION TIMES OF THE NEAREST NEIGHBO CLASSIFIER (MS)

Video sequences	Size	GPU	CPU	Speedup
Car	320×240	1.96	32.61	16.63
David	320×240	2.51	44.12	17.57
Girl	320×240	0.87	11.61	13.34
Tiger	320×240	0.92	9.42	10.23
Data1	352×288	0.59	8.09	13.71
Data2	640×480	1.58	20.65	13.07

3) Evaluation of the Overall Performance

We compare the overall performance of CUDA-TLD and TLD on the data set. The parameter *patch_size* and *min_window* are set to 15. The comparison results are shown in Table IV, which shows that CUDA-TLD runs faster than TLD for all the three kinds of resolution, and the speedup achieves up to 2 while the tracking accuracy of CUDA-TLD is same as TLD. Moreover, for the VGA sequences, CUDA-TLD can run at about 18.7 frames per second (FPS) while TLD run at about 8.4 FPS.

TABLE IV. COMPARISON OF THE AVERAGE EXECUTION TIMES PER FRAME OF CUDA-TLD AND TLD (MS)

Video sequences	Size	TLD tracking result	CUDA-TLD tracking result	Execution time per frame of TLD	Execution time per frame of CUDA-TLD	Speedup
Car	320×240	848/945	848/945	57.68	22.24	2.59
David	320×240	693/761	693/761	58.51	27.54	2.12
Girl	320×240	385/502	385/502	58.18	28.81	2.02
Tiger	320×240	239/354	239/354	44.31	21.87	2.03
Data1	352×288	764/901	764/901	68.38	33.12	2.06
Data2	640×480	343/550	343/550	119.62	53.61	2.23

V. CONCLUSIONS

TLD is an on-line visual tracking algorithm with good robustness and high accuracy properties. In this paper, we study the most time-consuming stages of TLD, and then propose a parallel algorithm based on CUDA. In order to exploit the computational capabilities of GPU, we consider the limitations of threads, thread blocks, shared memory and texture memory of graphics card GTX 550 Ti, and take full advantage of them in our algorithm rationally. In the end, we do several experiments to evaluate our algorithm using some video sequences. As shown in the experiments, compared to TLD, the speedup of the ensemble classifier of our algorithm reaches up to 2.50, and the speedup of the nearest neighbor classifier reaches up to 17.57. For the overall performance, the experimental results demonstrate that our algorithm runs faster than TLD for all the sequences in the data set while promising the accuracy at the same time, and the speedup is between 2.02 and 2.59. In particular, for VGA sequences with large size, our algorithm can run at about 18.7 FPS while TLD runs at about 8.4 FPS, and thus satisfy requirements of the real-time performance.

ACKNOWLEDGMENT

This paper is supported by NSFC (60973011, 61272004), and the Fundamental Research Funds for the Central Universities (2013YJS033).

REFERENCES

[1] V. Kettnaker and R. Zabih, "Bayesian multi-camera surveillance," Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Fort Collins, CO, pp. 253-259, 1999.

[2] G. R. Bradski, "Real time face and object tracking as a component of a perceptual user interface," Proc. of the IEEE Workshop Applications of Computer Vision. Princeton. NJ, pp. 214-219, 1998.

[3] V. Ferrari, T. Tuytelaars and G. L. Van, "Real-time affine region tracking and coplanar grouping," Proc. of the IEEE Conf. Computer Vision and Pattern Recognition. pp. 226-233, 2001.

[4] S. Avidan, "Support vector tracking," IEEE Trans. Pattern Analysis and Machine Intelligence. 26(8), pp. 1064-1072, 2004.

[5] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," Proc. of the IEEE CS Conf. Computer Vision and Pattern Recognition. pp. 511-518, 2001.

[6] L. Wixson, "Detecting salient motion by accumulating directionally-consistent flow," IEEE Trans. on Pattern Analysis and Machine Intelligence. 22(8), pp. 774-780, 2000.

[7] H. Song, F. Shi and Y. Z. Wang, "Method of real-time face detection in video images," Computer Engineering. 2004, 30(19): pp. 23-24.

[8] K. Yamazawa and N. Yokoya, "Detecting moving objects from omnidirectional dynamic images based on adaptive background subtraction," Proc of International Conf. on Image processing. 2003, pp. 953-956.

[9] Z. Kalal, K. Mikolajczyk and J. Matas, "Tracking-Learning-Detection," IEEE Trans. On Pattern Analysis and Machine Intelligence. 34(7): pp. 1409-1422, 2012.

[10] Z. Kalal, K. Mikolajczyk and J. Matas, "Forward-Backward Error: Automatic Detection of Tracking Failures," Proc. of 20th Int'l Conf. Pattern Recognition. Istanbul, pp. 2756-2759, 2010.

[11] Arthurv, OpenTLD. https://github.com/arthurv/OpenTLD/, 2012.

[12] J. Sanders and E. Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming. Boston: Addison Wesley, 2010.

[13] NVIDIA Corporation, NVIDIA's GT200: inside a parallel processor. http://www.realworldtech.com/gt200/5/, 2008.