Efficient Parallel TLD on CPU-GPU Platform for Real-Time Tracking

Zhaoyun Chen[1,2], Dafei Huang[3], Lei Luo[1*], Mei Wen[1,2] and Chunyuan Zhang[1,2]
[1] College of Computer, National University of Defense Technology
Changsha, 410073 - China
[e-mail: chenzhaoyun@nudt.edu.cn, l.luo@nudt.edu.cn]
[2] National Key Laboratory for Parallel and Distributed Processing
Changsha, 410073 – China
[3] College of Computer, National University of Defense Technology – China
*Corresponding author: Lei Luo

Received July 25, 2018; revised April 24, 2019; accepted June 22, 2019; published January 31, 2020

Abstract

Trackers, especially long-term (LT) trackers, now have a more complex structure and more intensive computation for nowadays' endless pursuit of high accuracy and robustness. However, computing efficiency of LT trackers cannot meet the real-time requirement in various real application scenarios. Considering heterogeneous CPU-GPU platforms have been more popular than ever, it is a challenge to exploit the computing capacity of heterogeneous platform to improve the efficiency of LT trackers for real-time requirement. This paper focuses on TLD, which is the first LT tracking framework, and proposes an efficient parallel implementation based on OpenCL. In this paper, we firstly make an analysis of the TLD tracker and then optimize the computing intensive kernels, including Fern Feature Extraction, Fern Classification, NCC Calculation, Overlaps Calculation, Positive and Negative Samples Extraction. Experimental results demonstrate that our efficient parallel TLD tracker outperforms the original TLD, achieving the 3.92 speedup on CPU and GPU. Moreover, the parallel TLD tracker can run 52.9 frames per second and meet the real-time requirement.

Keywords: TLD tracker, Real-Time, Heterogeneous Platform, OpenCL, Parallel Optimizations

The authors gratefully acknowledge supports from National Key Research and Development program under No.2016YFB1000400; National Nature Science Foundation of China under NSFC No. 61872377.

http://doi.org/10.3837/tiis.2020.01.012 **ISSN : 1976-7277**

1. Introduction

Visual tracking, a remaining highly popular research area for more than thirty years, plays an important role in numerous vision-based applications. Compared with short-term (ST) trackers, long-term (LT) trackers receive far less attention. A major difference between ST and LT trackers is that LT trackers can detect the target's absence and re-appearance. Obviously, LT tracking can be more adopted in various business-critical processes, such as automatic navigation, traffic monitoring, video retrieval and human-computer interaction [1-3]. In these domains, the important requirements of LT trackers are real time and support on diverse platforms.

With the coming of big data era, video data has an explosive growth in the quantity and quality, which leads to an endless pursuit of higher accuracy, robustness and efficiency for trackers. Especially, in most real applications and cases, the requirement of real-time for LT trackers imposes computational limitations. However, as shown in **Table 1**, most state-of-the-arts trackers have more complex structures but lower computing speeds in order to achieve higher accuracy and robustness. Therefore, a high-efficient implementation of LT tracker which can meet the real-time requirement is meaningful and challenging.

Table 1. The specifications of the state-of-the-art trackers

Year	Tracker	Method	Speed (FPS)	Implementation
2015	MDNet [4]	BB-regression + Convolution Neural Network	1	Matlab, CPU+GPU
2016	SiamFC [5]	Fully-conv. Siamese Network	12.6	Matlab, CPU+GPU
2016	C-COT [6]	Discriminative Continuous Conv. Operator	2.2	Matlab, CPU+GPU
2017	LSART [7]	Kernelized Ridge Regression + Convolution Neural Network	1	Matlab, CPU+GPU
2017	ECO [8]	Conv. Operator + Compact Generative Model	8.5	Matlab, CPU+GPU
2018	LADCF [9]	Correlation Filter + Convolution Neural Network	10.8	Matlab, CPU+GPU

Moreover, LT trackers are required to process on diverse platforms and devices in real applications. Except the classic computing device CPU, heterogeneous platforms which includes at least two kinds of devices (such as CPU + GPU) have been adopted in numerous computing platforms from SoC (Qualcomm Snapdragon 835 [10]) to embedded system (Nvidia Jetson [11]), and even to supercomputer (Tianhe-1A [12]). As shown in Tab. 1, there are still tremendous optimization room left for the computing efficiency of the trackers which are implemented by Matlab. Compared with other parallel programming languages and models, such as OpenMP [13] on CPU and CUDA [14] on GPU, OpenCL [15] has the advantage of cross-platform support and portability (the ability of programs to be run with minimum modification on diverse devices). A LT tracker implemented by OpenCL can run on diverse heterogeneous platforms and tap the potential of CPU and GPU by parallel techniques, such as multicore parallelism, vector instruction and pipeline.

In this paper, we focus on a LT tracker Tracking-Learning-Detection (TLD) [16], which firstly introduces independent learning and detection module into tracking. The unique structure of TLD makes it more robust and suitable than other trackers in long-term tracking.

Most recent LT trackers [17-19] are inspired by or derived from the structure of TLD. These researches improve the accuracy and robust of TLD, but ignore the computing speed and efficiency. Therefore, a high efficient implementation of TLD based on OpenCL is the guidance for the optimization of subsequent LT trackers. Meanwhile, the official version of TLD, which is a serial implementation called OpenTLD [20], cannot make the best use of computing capacity of heterogeneous platform. Therefore, in this paper, we make a comprehensive analysis for the bottleneck of OpenTLD and propose an efficient parallel TLD tracker based on OpenCL. The design and implementation of the efficient parallel TLD can optimize the computing-intensive modules of OpenTLD, aiming to achieve real-time requirement of tracking and high utilization of CPU and GPU.

As shown in **Fig. 1**, TLD is divided into three parts: tracking module, detection module and learning module. Tracking module recursively tracks the target based on the location of the previous frame. The learning module learns the appearance of the target. The detection module is designed to locate the target when it re-enters after it gets out of the field of view. The optimized modules of TLD include random fern and nearest neighbor classifier in detection module, and sample extraction and detector updating in learning module. Moreover, the OpenCL has provided the parallel support of LK optical flow in tracking module, which part is not discussed in following sections. The experiments on a CPU-GPU platform demonstrate that our parallel TLD can both improve the speedup of original OpenTLD by up to 3.92 and meet the requirement of real-time.

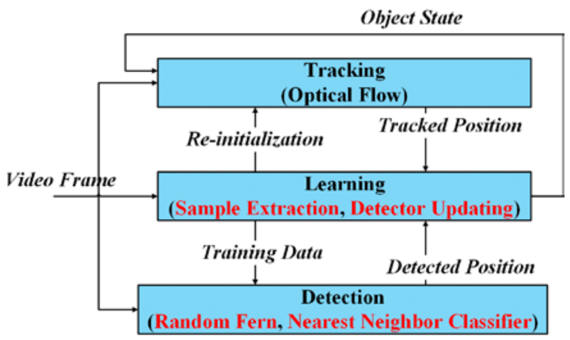


Fig. 1. TLD is compose of three parts, tracking, detection and learning. Tracking module is implemented by optical flow. Learning module includes sample extraction and detector updating. Detection module is composed of random fern and nearest neighbor classifier. Most recent LT trackers [17-19] are inspired by or derived from the structure of TLD.

The rest of this paper is organized as follows. Section 2 is the related work. Section 3 presents the overview and analysis of TLD tracker. We propose the efficient parallel designs and optimizations for TLD modules in Section 4. In Section 5, a comprehensive evaluation and analysis is performed. Finally, we draw a conclusion in Section 6.

2. Related Work

In this section, we briefly review the researches closely related to our work: (1) Recent LT tracking algorithms, (2) Parallel designs and optimizations of visual tracking.

2.1 Recent LT Tracking Algorithms

Compared with ST trackers, LT trackers are required to handle situations in which the target gets out of the field of view and re-enters. That means LT trackers can detect the target's absence and re-appearance. [18] extends the SiamRPN approach by introducing a simple but effective local-to-global search region strategy. Meanwhile, the size of search region is iteratively growing when tracking failed. [19] proposes a novel parallel tracking and verifying (PTAV) framework for long-term tracking, by taking advantage of the ubiquity of multi-thread techniques. Compared with these researches mentioned above, TLD is the first tracker combining tracking, detection and learning into one. The subsequent LT trackers [17-19] are inspired by the structure of TLD and further improve the accuracy and robustness of tracking. Therefore, parallel implementation of TLD based on OpenCL is the guidance for the optimization of subsequent LT trackers.

2.2 Parallel Designs and Optimizations of Visual Tracking

Recently, visual tracking has achieved a rapid development on accuracy and robustness rather than speed. There are few work discussing the parallel implementation and optimizations of trackers, especially on heterogeneous platform [21-24]. Research [23] proposes a high-performance version H-TLD based on OpenMP and CUDA. H-TLD exploits multicores on CPU and parallel units on GPU for load balance and decrease the transaction latency by data-compression techniques. Another version of TLD has proposed in [24]. Research [24] replaces the global searching with particle filtering and improves the tracking efficiency by the overlapped execution of the detection module and tracking module. However, aforementioned researches prefer to adopt CUDA as the programming model, which is a dedicated model on GPU. Due to the more complex scenarios and diverse platforms, the cross-platform model OpenCL can effectively decrease the difficulties of programming and transplantation.

3. Overview and Analysis of TLD Tracker

In this section, we first give an overview of TLD to show all the modules of the tracker. Then we make an analysis for the compute-intensive bottleneck of TLD in more details based on profiling to help with following optimization design.

Algorithm 1. Overview of TLD tracker

```
Algorithm 1: Overview of TLD tracker
  Input: The image of frame[i], the image of previous frame[i-1], the targe position of previous frame $lastBB$
  Output: The target position of current frame $currentBB$
$trackedBB \leftarrow$ Tracker(frame[i-1], frame[i], $lastBB$);
$detectedBBs \leftarrow$ Detector(frame[i]);
$clusteredBBs \leftarrow$ Cluster($detectedBBs$);
if $trackedBB \neq NULL$ then
    for each $clusteredBB$ in $clusteredBBs$ do
        if Overlap($clusteredBB, trackedBB$) $< 0.5$ && NNConf($clusteredBB$) $>$ NNConf($trackedBB$) then
            $confidentBBs \leftarrow clusteredBB$;
    if Sizeof($confidentBBs$) $= 1$ then
        $currentBB \leftarrow confidentBBs[0]$;
    else
        $currentBB \leftarrow$ WeightedAverage($trackedBB, detectedBBs$);
else
    if Sizeof($clusteredBBs$) $= 1$ then
        $currentBB \leftarrow clusteredBBs[0]$;
if $trackedBB \neq NULL$ && Validated($trackedBB$) then
    Learn($currentBB$, frame[i]);
```

TLD tracker is divided into three independent parts, tracking, detection and learning. The algorithm overview of TLD tracker is shown in **Algorithm. 1**. Before the calculation of each frame, TLD algorithm typically initializes to generate lots of bounding boxes. The generation of bounding boxes is only related to the size of the frame and the initial size of the target.

The function *Tracker()* belongs to the tracking module, which adopts **LK (Lucas-Kanade) optical flow** algorithm. Tracking module is charge of calculating the motion of the target between contiguous frames and proposing a candidate. Generally, in visual tracking, it is assumed that the motion changes between contiguous frames are subtle and the target is always visible. However, when the target is occluded or out of sight, the tracking module usually would fail and be unrecoverable. If the target position *trackedBB* is NULL, the tracking module fails in current frame.

The function *Detector()*, representing detection module, includes two parts: **Random Fern** and **Nearest Neighbor Classifier**. Random Fern first calculates the response value of each bounding box. The response values reflect the probability that the bounding box contains the target. The bounding boxes, whose response values are lower than the threshold, would be discarded. Nearest Neighbor Classifier adopts the remaining bounding boxes as the input. Each bounding box is compared with numerous positive and negative samples to compute a confidence probability, which is related to the similarity to positive samples and the non-similarity to negative samples. The similarity is measured by Normalized Cross Correlation (NCC) [25]. Therefore, the bounding boxes whose confidence probabilities are higher than the threshold are the output of the function *Detector()*. Furthermore, the function *Cluster()* merges the output bounding boxes by clustering to decrease the number of outputs.

Learning module estimates the target position by the outputs of tracking and detection module. When the tracking module fails and the detection module has only one output in *clusteredBBs*, the bounding box is selected as the target position of current frame. Instead, if the tracking module works, learning module compares and analyzes the *trackedBB* with *clusteredBBs*. When the *clusteredBBs* have only one output bounding box and it has the higher confidence than the *trackedBB*, the bounding box in *clusteredBB* is adopted as the target position of current frame. Otherwise, learning module calculates the weighted average result of *clusteredBB* and *trackedBB*. The weighted average bounding box is the target position.

Moreover, learning module decides the proper time to re-train the detector. The re-training function *Learn()* only works when the tracking module works and the *trackedBB* can be validated. There are two kinds of situations where the *trackedBB* is validated. In the first situation, the *trackedBB* of current frame has the high enough confidence from the Nearest Neighbor Classifier. However, in the second situation, the *trackedBB* of a certain history frame has high confidence and the tracking module never fails from this history frame.

The learning module includes two main parts, **collecting training samples** and **retraining the detector**. Before collecting samples, we calculate the overlaps between the *currentBB* and the bounding boxes of current frame by the function *Overlap()*. The overlap in this paper is defined by IoU (Intersection over Union). For Random Fern, the bounding boxes which have the high response values and the low overlaps are selected as negative samples, called hard negative mining. Moreover, the bounding boxes with high overlaps and response values are adopted as positive samples through random rotating and resizing. For Nearest Neighbor Classifier, the negative samples are selected from the bounding boxes with low overlaps in *detectedBBs*. The bounding box in *detectedBBs* with the highest overlap is the only positive sample. The retraining of Random Fern is the process of updating the weight of leaf node from each random tree. The weight updating depends on the number of positive and negative samples falling into the leaf nodes. Furthermore, the retraining of Nearest Neighbor Classifier is to updating the sample set. The positive samples with low confidence and the negative samples with high confidence are respectively added into the sample set for next NCC calculation.

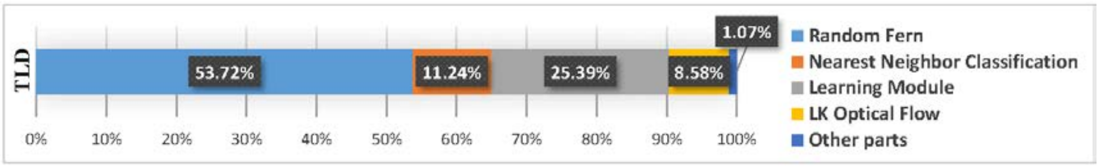


Fig. 2. Execution time distribution for all modules in TLD. The compute-intensive modules in TLD include random fern, learning module, nearest neighbor classification.

In this paper, we adopt OpenTLD [20], which is the official open-source serial version of TLD, as the baseline to analyze and optimize the compute-intensive bottleneck parts based on OpenCL. As shown in **Fig. 2**, the compute-intensive parts in OpenTLD includes **LK optical flow**, **Random Fern**, **Nearest Neighbor Classifier** and **Learning Module**. Others parts have too few computations or are not appropriate to transplant on OpenCL. Due to that LK optical flow has been supported by OpenCV based on OpenCL, tracking module is not discussed in this paper. In next section, we will make a comprehensive analysis for each compute-intensive module in OpenTLD and introduce our works about efficient parallel designs and optimizations.

4. Efficient Parallel Designs and Optimizations for TLD

In order to improve the computing efficiency of TLD on heterogeneous platform of CPU and GPU, it is important to design and optimize the compute-intensive modules, including Random Fern, Nearest Neighbor Classifier and Learning Module. We adopt various parallel techniques, such as exploiting the parallelism between calculations, and overlapping data transferring with computation, in our proposed parallel TLD. The framework of our parallel TLD based on OpenCL is shown in **Fig. 3**. In the following, we will introduce our high efficient kernels based on OpenCL for these modules mentioned above in detail. Moreover, some parts of TLD, which are not suitable for GPU, are still processed by host program. Our proposed parallel TLD has good cross-platform support and portability on both CPU and GPU.

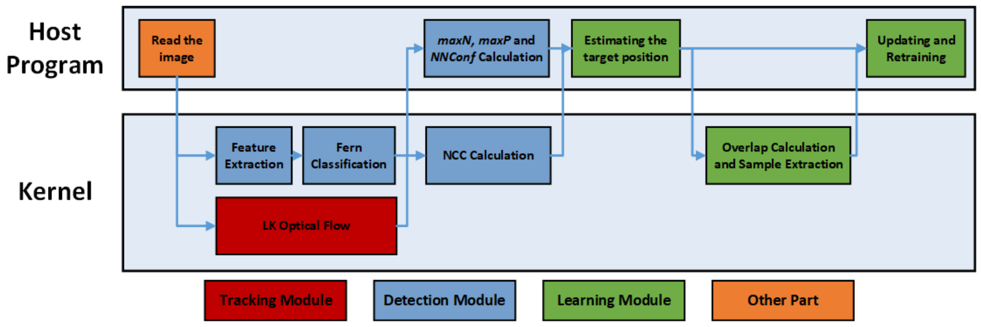


Fig. 3. The framework of our parallel TLD based on OpenCL. Most compute-intensive modules are implemented and optimized in kernel functions. Others are still processed in host program.

4.1 Random Fern Optimizations

Random Fern include fern feature extraction and fern classification, as shown in **Fig. 4**. Each bounding box, as the input of Random Fern, corresponds to an image block in the frame. In TLD, classification feature is generated by pixel comparison. More specifically, 13 pairs of pixels are sampled based on the pre-assigned pattern from an image block. The grey values of pixel pairs are used for comparison and the results are recorded by 0 or 1. Therefore, the comparison results of 13 pairs of pixels can be represented by a binary vector, called fern feature vector. Due to the diverse scales of the bounding boxes, TLD defines 10 kinds of sampling patterns for each scale. Therefore, each bounding box is represented by 10 13-bit fern feature vectors through feature extraction.

In order to classify the feature vectors, TLD builds a random tree for each kind of sampling pattern, which corresponds to a kind of feature vector. All random trees compose of a Random Forest. After that, each feature vector is classified in the corresponding random tree and falls into a certain leaf node of the tree. Each leaf node of the random tree is given the weight during the training process. And the weight represents the probability that the bounding box contains the target. For each bounding box, the response value of the Random Fern is the average of the 10 weights from the corresponding random trees.

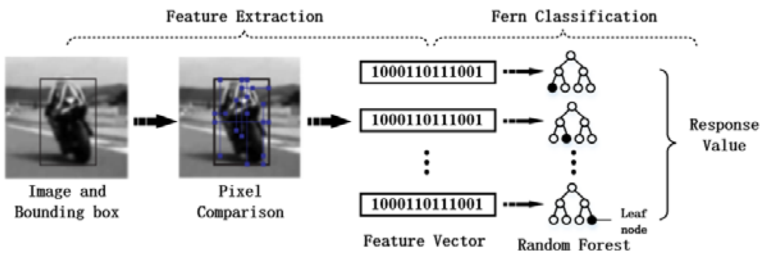


Fig. 4. The overview of Random Fern. Random fern includes two parts: feature extraction and fern classification.

4.1.1 Feature Extraction

The inputs of feature extraction include the image of the current frame, bounding boxes array, the filtered bounding box index and the sampling patterns for pixel pairs. And the outputs are the corresponding feature vectors of the bounding boxes.

Each bounding box is represented by a quintuple {BBx, BBy, BBw, BBh, scale_id}, which are respectively the coordinate of the upper left corner, the width, the height and the scale index of the bounding box. The quintuples of all candidate bounding boxes are organized as a one-dimensional array *grid*, which is stored in a constant memory of OpenCL for kernels calling. Meanwhile, the grey value of current image is also stored in a constant memory. The filtered bounding box index, which is a one-dimensional array, points out the positions of the filtered bounding boxes in the *grid*. Moreover, the sample patterns are also organized as a one-dimensional array. Each sample pattern is represented by $\{x_1, y_1, x_2, y_2\}$, which are the coordinates of the pixel pair in the bounding box. Considering that each scale of bounding boxes has 10 kinds of sampling patterns and each sampling pattern has 13 pairs of pixels, the array storing sampling patterns has {scale_num * 10 * 13 * 4} elements in total. In order to load and store efficiently, we convert the storage order from $\{x_1^1, y_1^1, x_2^1, y_2^1, \ldots\ldots, x_1^n, y_1^n, x_2^n, y_2^n\}$ to $\{x_1^1, \ldots, x_1^n, y_1^1, \ldots, y_1^n, x_2^1, \ldots, x_2^n, y_2^1, \ldots, y_2^n\}$. Similar to the *grid*, the array of sampling patterns is stored in a constant memory.

In order to optimize the process of feature extraction on OpenCL, we exploit the parallelism of this process. In our work, each work item takes charge of the complete process of a pair of pixels, which includes coordinates calculation, grey value extraction and comparison, as shown in **Fig. 5**. A work group, which is composed of 130 work items, generates 10 fern feature vectors of a bounding box. That is, the adjacent 13 work items combine their results of grey value comparison to a fern feature vector and store it into global memory. Moreover, the number of work groups is equals to the number of bounding boxes. The corresponding kernel code is shown in **Fig. 6**.

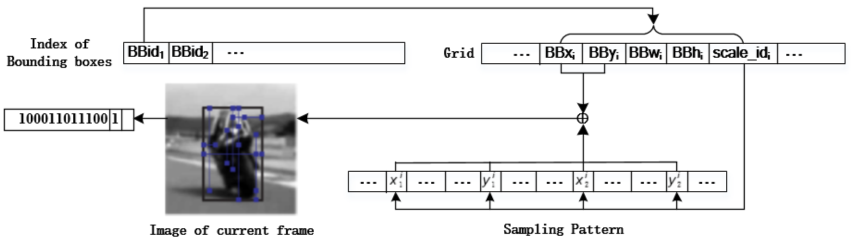
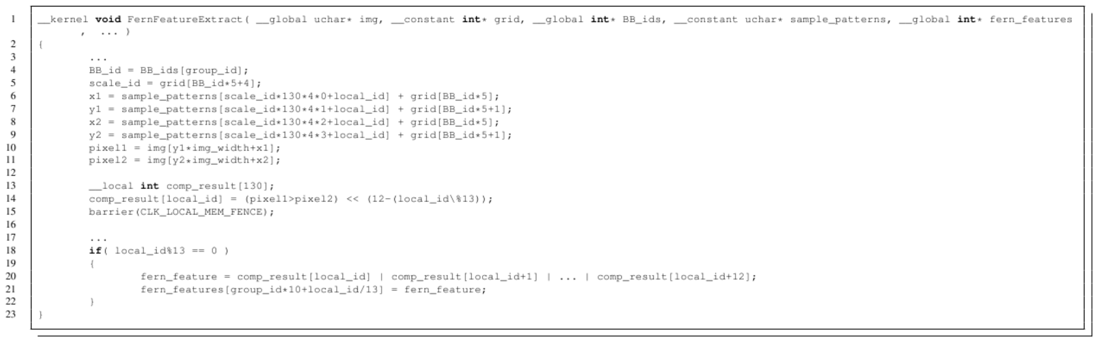


Fig. 5. Fern Feature Extraction.


```
__kernel void FernFeatureExtract( __global uchar* img, __constant int* grid, __global int* BB_ids, __constant uchar* sample_patterns, __global int* fern_features
       ,  ... )
{
        ...
        BB_id = BB_ids[group_id];
        scale_id = grid[BB_id*5+4];
        x1 = sample_patterns[scale_id*130*4*0+local_id] + grid[BB_id*5];
        y1 = sample_patterns[scale_id*130*4*1+local_id] + grid[BB_id*5+1];
        x2 = sample_patterns[scale_id*130*4*2+local_id] + grid[BB_id*5];
        y2 = sample_patterns[scale_id*130*4*3+local_id] + grid[BB_id*5+1];
        pixel1 = img[y1*img_width+x1];
        pixel2 = img[y2*img_width+x2];

        __local int comp_result[130];
        comp_result[local_id] = (pixel1>pixel2) << (12-(local_id\%13));
        barrier(CLK_LOCAL_MEM_FENCE);

        ...
        if( local_id%13 == 0 )
        {
                fern_feature = comp_result[local_id] | comp_result[local_id+1] | ... | comp_result[local_id+12];
                fern_features[group_id*10+local_id/13] = fern_feature;
        }
}
```

Fig. 6. Kernel of Fern Feature Extraction

4.1.2 Fern Classification

The 10 fern feature vectors of each bounding box are the input of the random forest to achieve the response values. The outputs are the response values of the bounding boxes. As illustrated in **Fig. 6**, a fern feature vector is stored as a 13-bit binary integer. Therefore, in order to improve the efficiency of classification, it is not necessary to build a tree structure to check each element of a feature vector. By contrast, an implementation of lookup table is adopted in this paper. A 13-bit binary integer ranges from 1 to 2^{13}=8192. We build a table containing 8192 items instead of a tree structure. Each item stores a weight of a leaf node. Due to that the random forest has 10 trees, we build 10 tables for matching. Afterwards, a fern feature vector is taken as an index to look up in the corresponding table for the weight of the leaf node. For a bounding box, 10 weights are obtained from looking up in the 10 tables by the associated fern feature vectors. The sum of the weights is the response value of the bounding box. The kernel code of classification is shown in **Fig. 7**.

```
__kernel void FernClassify( __global int* fern_features, __global float* leaf_weights, __global float* fern_responses, ... )
{
        ...
        tree_id = local_id%10;
        BB_id = group_id*20 + local_id/20;

        ...
        __local float share_weights[200];
        fern_feature = fern_features[BB_id*10+tree_id];
        share_weights[local_id] = leaf_weights[tree_id*8192+fern_feature];
        barrier(CLK_LOCAL_MEM_FENCE);

        if( ... && local_id % 10 == 0 )
        {
                fern_response = share_weights[local_id+0] + share_weights[local_id+1] + ... + share_weights[local_id+9];
                fern_responses[BB_id] = fern_response;
        }
}
```

Fig. 7. Kernel of Fern Classification

4.1.3 Overlapped Execution with Tracking Module

Because LK optical flow module is independent of Random Fern Classification, these two parts can be overlapped executed. **Fig. 8** illustrates the OpenCL host program of overlapped execution. The numbers of filtered bounding boxes are different in each frame. The array lengths of the filtered bounding box index, fern feature vector and response values are also changed during tracking. When the tracking module finishes, the command queue would be checked whether all commands in the queue have been executed. If not, the process would block and wait. At last, the memory space is released for recreation in next frame.

```
...
TLDCL.CreateMemBufferFernFeatureExtract ( BB_ids, BB_num, fern_features, BB_num*10 );
TLDCL.CreateMemBufferFernClassify ( fern_responses, BB_num );
TLDCL.SetKernelArgFernFeatureExtract( );
TLDCL.SetKernelArgFernClassify( );
TLDCL.WriteBufferFernFeatureExtract(  img, img.rows*img.cols );
TLDCL.WriteBufferFernClassify( leaf_weights, 8192*10 );
TLDCL.EnqueueKernelFernFeatureExtract( );
TLDCL.EnqueueKernelFernClassify( );
TLDCL.ReadBufferFernFeatureExtract( fern_features, BB_num*10 );
TLDCL.ReadBufferFernClassify( fern_responses, BB_num );

LKtrack(img_previous, img_current, lastBB, trackedBB);

TLDCL.SyncCommandQueue(  );
TLDCL.ReleaseMemObjectFernFeatureExtract( BB_ids, fern_features );
TLDCL.ReleaseMemObjectFernClassify( fern_responses );
...
```

Fig. 8. Overlapped Execution between LK optical flow and Random Fern

4.2 Parallel Implementations for Nearest Neighbor Classifier

After finishing the Random Fern, the response value of each bounding box has been calculated. The bounding boxes, whose response values are higher than the threshold and rank among the top 100, are adopted as the inputs of Nearest Neighbor Classifier to compute the confidence of containing the target.

4.2.1 Algorithm Flow

The positive and negative samples are composed of the model of Nearest Neighbor Classifier. Before the calculation of Nearest Neighbor Classification, an image block is extracted as the input according to the corresponding bounding box. Then the image block is compared with the positive and negative samples to calculate the similarities. We denote *maxP* and *maxN* respectively as the maximum similarity in positive samples and negative samples. And the confidence is denoted by *NNConf*. The confidence of the bounding box is calculated as:

$$NNConf = \frac{1 - maxN}{2 - maxP - maxN}. \tag{1}$$

It is clearly that *maxP* is positively correlated to the confidence and *maxN* is negatively correlated to the confidence. As described in **Algorithm 1**, the similarity is defined by Normalized Cross-Correlation (NCC). We denote f and t as two image blocks. f and t are declared as two two-dimension float arrays which respectively store the greyscale images. Each pixel in the greyscale image is represented by a float between 0 and 1. NCC of two image blocks is calculated as

$$NCC = \frac{\sum_{x,y} f(x,y)t(x,y)}{\sqrt{\sum_{x,y} f(x,y)^2 \sum_{x,y} t(x,y)^2}}, \tag{2}$$

where (x, y) is the horizontal and vertical coordinates. TLD makes a little modification for the calculation of NCC. In order to avoid the interference of illumination, the average grey value of the whole image block is subtracted from the grey value of each pixel. Meanwhile, in order to ensure the normalization of NCC, the formula is modified as:

$$NCC = \frac{1}{2}\left(\frac{\sum_{x,y} f(x,y)t(x,y)}{\sqrt{\sum_{x,y} f(x,y)^2 \sum_{x,y} t(x,y)^2}} + 1\right). \tag{3}$$

4.2.2 Parallel Implementations of NCC Calculation

Nearest Neighbor Classification includes two stages. In first stage, all image blocks are used to calculate NCC with each sample in the model. Then *maxN*, *maxP* and the confidence *NNConf* of each image block are obtained in second stage. Due to numerous bounding boxes, the number of NCC calculation is: $the\ number\ of\ bounding\ boxes \times the\ number\ of\ samples$ in the model. Obviously, the first stage is computationally intensive. However, the number of bounding box is no more than 100, which indicates the less computations of the second stage. Moreover, considering reduction operations are not appropriate to optimize on OpenCL in the second stage, we only focus on the parallel implementations of NCC calculation the first stage. The second stage is still processed in the host program.

In OpenTLD, the size of image samples in the model is 15×15. Similarly, an image block extracted from the corresponding bounding box is resized to the same size. As illustrated in **Formula (3)**, the NCC results is not related to the storage mode and access order of the pixels in an image block. Therefore, in order to exploit the parallelism in the Nearest Neighbor Classification, we build a one-dimensional array *patches* to store continuously all pixel grey values of all image blocks. In this array, the continuous $15 \times 15 = 225$ elements denote an input image block. Likewise, the positive and negative samples in the model are also stored in a one-dimensional array *p_n_samples*, where the positive samples are the first and the negative samples are the last. According to the number of the positive or negative samples, we can identify that any element in the array belongs positive or negative. These two one-dimensional arrays mentioned above are the input of NCC calculation.

We adopt three-dimensional index space in the parallel implementation of NCC calculation. As shown in **Fig. 9**, each work group takes charge of the NCC calculation between an image block and an image sample. The range of three dimensions for work groups is {1, number of bounding boxes, number of samples}. Each work item is responsible to calculate the product of the sum of squares between the same pixel position from two image blocks. The range of three dimensions for work items is {225, 1, 1}. Therefore, for the whole three-dimensional index space, the number of work items is $225 \times number\ of\ bounding\ boxes \times number\ of\ samples$. According to the **Formula (3)**, the results of each pixel position should be accumulated. In this paper, we adopt a tree reduction to improve the efficiency of accumulation. However, the number of elements should be a power of 2 in a tree reduction. We adjust the number of the work item in whole three-dimensional index space to {256, number of bounding boxes, number of samples}. The kernel code for NCC calculation is shown in **Fig. 10**.

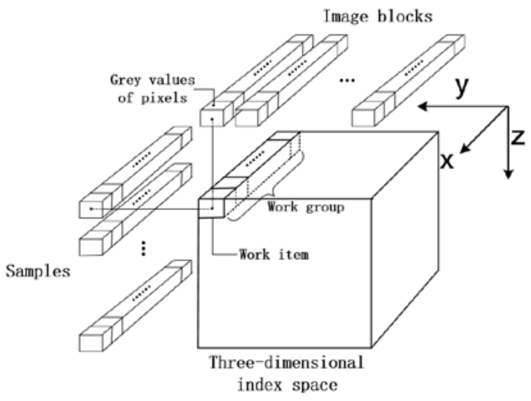


Fig. 9. OpenCL index space for NCC Calculation.

```
__kernel void NCC( __global float* patches, __global float* p_n_samples, int p_n_sample_num, __global float* nccs )
{
    ...
    if ( local_X_id < 225 )
    {
        patch_pixel = patches[group_Y_id*225+local_X_id];
        sample_pixel  = p_n_samples[group_Z_id*225+local_X_id];
    }
    else
    {
        patch_pixel = 0;
        sample_pixel  = 0;
    }

    __local float patch_squares[256];
    __local float sample_squares[256];
    __local float patch_sample_products[256];
    patch_squares[local_X_id] = patch_pixel * patch_pixel;
    sample_squares[local_X_id] = sample_pixel * sample_pixel;
    patch_sample_products[local_X_id] = patch_pixel * sample_pixel;
    barrier(CLK_LOCAL_MEM_FENCE);

    for( int offset = local_X_size/2; offset > 0; offset >>= 1 )
    {
        if ( local_X_id < offset )
        {
            patch_squares[local_X_id] = patch_squares[local_X_id+offset] + patch_squares[local_X_id];
            sample_squares[local_X_id] = sample_squares[local_X_id+offset] + sample_squares[local_X_id];
            patch_sample_products[local_X_id] = patch_sample_products[local_X_id+offset] + patch_sample_products[local_X_id];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if( local_X_id == 0 )
    {
        square_sum_product = pow( patch_squares[local_X_id]*sample_squares[local_X_id], 0.5 );
        nccs[group_Y_id*p_n_sample_num+group_Z_id] = (patch_sample_products[local_X_id]/square_sum_product + 1) * 0.5;
    }
}
```

Fig. 10. Kernel of NCC Calculation

4.3 Exploiting the Parallelism within Learning Modules

Based on an overall analysis of the output bounding boxes from tracking and detection module, host program estimates the target position in the current frame. As mentioned in **Section 3**, when the tracking module does not fail and the *trackerBB* is validated, the learning module is called for updating the detection module. The learning module composes of two stages. In the first stage, a few bounding boxes are selected as the positive and negative samples according to the overlaps with the *currentBB* and the response values. And the second stage is a retraining module, which is designed for updating the weights of leaf nodes in Random Fern and the model in Nearest Neighbor Classifier. The number of bounding boxes in the first stage is numerous. For example, a frame with size 320×240 can produce more than 60000 bounding boxes. The overlap and the response value calculations for all bounding boxes are time-consuming and inefficiency. By contrast, the number of positive and negative samples is few. Generally, the number of positive samples is about 10 and the negative samples are less than the positive. Furthermore, the calculation of the second stage is not large and there are most branch operations in the calculation. Therefore, in this section, we exploit the parallelism in the first stage based on OpenCL and the second stage is still processed by host program.

4.3.1 Extraction of Negative Samples and Calculation of Overlaps

The negative samples are the incorrectly classified bounding boxes, which have the lower overlaps than the threshold and high response values. In order to improve the efficiency, we propose to extract the negative samples and calculate the overlaps in the same kernel. The calculation of overlaps in this section refers to IoU.

Obviously, the inputs of this kernel include a one-dimension array *grid_reorg* which stores all bounding boxes, the response values and the current target position *currentBB*. The outputs include the negative samples set and the overlaps. Unlike the kernel of feature extraction, in this section, each work item is charge of the calculation of overlap between a bounding box and the *currentBB*. If the bounding boxes are stored in a one-dimension array organized as {..., BBx_i, BBy_i, BBw_i, BBh_i, $scale\_id_i$, ...} in **Section 4.1.1**, the contiguous work items access the discontinuous memory when they load the same parameter of bounding boxes (for example, BBx_i). Therefore, we declare another one-dimension array *grid_reorg* to store all bounding boxes as {BBx_1, ... BBx_n, BBy_1, ..., BBy_n, BBw_1, ... BBw_n, BBh_1, ... BBh_n}. The reorganized array *grid_reorg* substitutes for *grid* as the input of the kernel in this section. The response values are reused from the results *fern_responses* in **Section 4.1.2**. Because the number of negative samples is uncertain and the work items in different work groups cannot synchronize in OpenCL, we adopt a tag array to record the output negative samples. The length of the tag array is equal to the number of bounding boxes. An element in the tag array is labeled as 1 when the corresponding bounding box is selected as a negative sample. Otherwise, the element is labeled as 0. To exploit the parallelism in extraction of negative samples and calculation of overlaps, each work item extracts a bounding box from *grid_reorg* and calculates the overlap between the bounding box and the *currentBB*. If the overlap is lower than the threshold, the corresponding response value is checked to determine whether or not the bounding box belongs to a negative sample. In order to improve the occupancy of OpenCL devices, a work group includes 512 work items. The kernel code is shown in **Fig. 11**.

```
__kernel void GetOverlapAndNegativeSample( __constant int* grid_reorg, __global int* currentBB,  __global float* fern_responses, __global float* overlaps,
       __global uchar* negative_tags )
{
        ...
        int box1x = currentBB[0];
        int box1y = currentBB[1];
        int box1w = currentBB[2];
        int box1h = currentBB[3];

        int box2x = grid_reorg[global_id];
        int box2y = grid_reorg[global_id+grid_BB_num];
        int box2w = grid_reorg[global_id+grid_BB_num*2];
        int box2h = grid_reorg[global_id+grid_BB_num*3];
        ...

        if ( overlap != 0.0 ) {
            intersec_x =  min(box1x+box1w, box2x+box2w) - max(box1x, box2x);
            intersec_y =  min(box1y+box1h, box2y+box2h) - max(box1y, box2y);
            intersec_area = intersec_x * intersec_y;
            area1 = box1w*box1h;
            area2 = box2w*box2h;
            overlap = intersec_area / (area1+area2-intersec_area);
        }
        overlaps[global_id] = overlap;

        if ( global_id < grid_BB_num ) {
          if ( overlap < low_overlap && fern_responses[global_id]>=high_response )
            negative_tags[global_id] = 1;
          else
            negative_tags[global_id] = 0;
        }
}
```

Fig. 11. Kernel of Overlap Calculation and Negative Samples Extraction

4.3.2 Extraction of Positive Samples

In OpenTLD, the positive samples are top 10 bounding boxes with highest overlaps. Therefore, the extraction of positive samples is a process to search the bounding boxes with highest overlaps. And the process is a typical reduction which has less computation and more branches. However, due to the large number of bounding boxes in the *grid_reorg*, exploiting the parallelism in this process can significantly improve the utilization of memory bandwidth.

In order to search the top 10 bounding boxes with highest overlaps, the kernel would be executed for 10 times and in each time, an index of a bounding box in the *grid_reorg* is obtained. In the kernel, the number of work items is equals to the number of bounding boxes. Each work item loads the overlap of a bounding box and compares with other work items in the same work group. A work group contains 1024 work items and the index of the bounding box with highest overlap in the work group is obtained by a tree reduction. The index is stored into the global memory. Meanwhile, all work groups send own index to the host program. The host program achieves the final index of the bounding box with highest overlap by reduction. As shown in **Fig. 12**, the input of the kernel includes the array *overlaps* storing the overlaps which are calculated by the kernel in previous part. The output includes the index of the bounding box with the highest overlap in all work groups. The selected bounding box is added into the positive sample set. In particular, in order not to get the same output after each iteration, the overlap of the selected bounding box should be set to the minimum. Therefore, in this kernel, the index of the bounding box *top_BB_id_in_group*, which is selected from previous execution, is also adopted as an input. Before the current execution of the kernel, the overlap is set to minimum according to the index. The kernel for the extraction of positive samples is shown in **Fig. 13**.

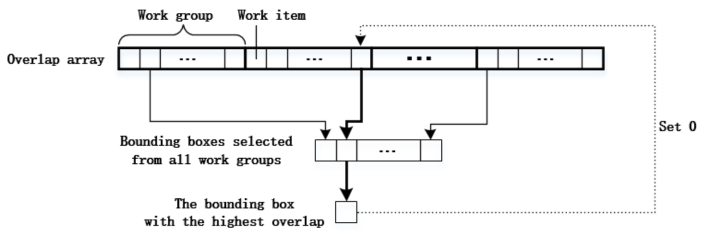

Work group
Work item
Overlap array
Set 0
Bounding boxes selected from all work groups
The bounding box with the highest overlap

Fig. 12. Positive Samples Extraction

```
__kernel void GetPositiveSample( __global float* overlaps, int current_top_BB_id, __global int* top_BB_id_in_group )
{
    ...
    __local float overlaps_local[1024];
    __local int BB_ids_local[1024];

    if ( global_id == current_top_BB_id ) {
        overlaps[current_top_BB_id] = 0.0;
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    overlaps_local[local_id] = overlaps[global_id];
    BB_ids_local[local_id] = global_id;
    barrier(CLK_LOCAL_MEM_FENCE);

    for ( int offset = local_size/2; offset > 0; offset >>= 1)
    {
        if ( local_id < offset )
        {
            if ( overlaps_local[local_id] < overlaps_local[local_id+offset] )
            {
                   BB_ids_local[local_id] = BB_ids_local[local_id+offset];
                   overlaps_local[local_id] = overlaps_local[local_id+offset];
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if ( local_id == 0 ) {
        top_BB_id_in_group[group_id] = BB_ids_local[0];
    }
}
```

Fig. 13. Kernel of Positive Samples Extraction

The array *top_BB_id_in_group* in global memory is passed back to the host program. The host program re-executes the reduction to obtain the bounding box with the highest overlap. The index of the selected bounding box is adopted as the input of the kernel for next execution until 10 bounding boxes are picked out as the positive samples. As mentioned above, after finishing the extraction of positive and negative samples, the re-training of detection module, which is not appropriate for parallelizing on OpenCL, is processed by the host program. Thus far, we have introduced the parallel implementations of OpenTLD based on OpenCL.

5. Evaluation and Analysis

In this section, we propose an exhaustive evaluation and analysis for our proposed parallel TLD tracker. We will evaluate the performance of our proposed parallel TLD from three levels. Firstly, a performance comparison is made between the original TLD and our optimized TLD at kernel level, as shown in **Section 5.2**. Secondly, besides the computational overhead, we take the data reorganization and communication overhead into consideration for evaluating. **Section 5.3** mainly shows the execution times of diverse implementations at module level. At last, we evaluate the performance of our complete parallel TLD with some baselines, including original TLD, HTLD and AATLD.

5.1 Experimental Setup

The experimental platform is based on a typical desktop platform, whose configurations are summarized in **Table 2**. The test dataset we adopted is OTB50 [3]. The baselines compared with our proposed parallel TLD include original OpenTLD, H-TLD [23] and AATLD [24]. Because this paper focuses on computing efficiency, we adopt execution time and speedup as the metric. However, our proposed parallel TLD can achieve high accuracy as the same as original OpenTLD. The tracking results on some image sequences with difficult attributes (such as occlusion, fast motion, illumination variation, deformation, out-of-view) are shown in **Fig. 14**.

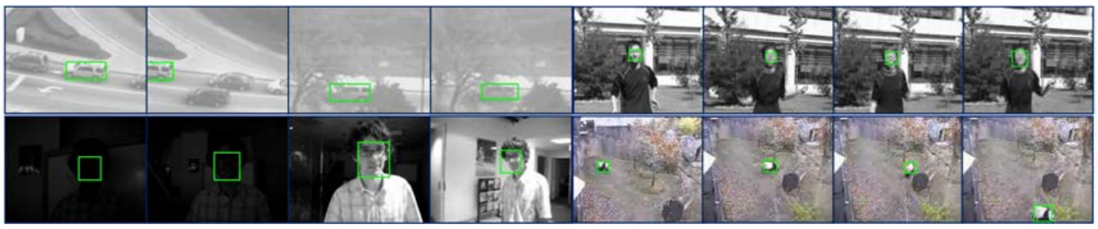


Fig. 14. Screenshots of TLD tracking results. From top to bottom, the image sequences are *Car, Jumping, David and Panda.*

Table 2. Platform Specifications

Hardware	CPU	Intel i7-5500U (2 cores@2.4GHz)
	GPU	NVIDIA GTX 960M
	Memory	16GB DDR3 1600MHz
Software	OS	Ubuntu 16.04
	Compiler	gcc/g++ 4.8
	OpenCL for CPU	OpenCL 2.0, Intel OpenCL 1.2 Driver
	OpenCL for GPU	OpenCL 1.2, CUDA 8.0
	OpenCV	OpenCV 2.4.13

5.2 Kernel Performance Analysis

We first evaluate the kernel performance separately in this part. The kernels optimized on OpenCL include Fern Feature Extraction, Fern Classification, NCC Calculation, Overlap and Negative Samples Calculation and Positive Samples Calculation.

Due to the portability of OpenCL, the aforementioned kernels can be executed on any devices and the host program can specify the device for each kernel. We respectively record the execution time of each kernel on CPU and GPU. In order to achieve the high accuracy of the statistics, we adopt the event in OpenCL to record the time overhead. In particular, the time overhead only contains the time of kernel execution, not including data transferring and kernel startup, which are considered in next part. Each image sequence in test dataset has numerous frames of images. The kernel execution time is defined as the sum of the overhead in each frame. Similarly, in this section, we extract the program segments of the same function in original OpenTLD as the baseline. The experiments are repeated for three times and the averages of the statistics for all kernels and program segments are taken for comparison.

The experiment results on different image sequences has the same trend and properties. Due to space limitations, we mainly show the results on the image sequence *Car* in OTB50. The execution times of all kernels on diverse devices are shown in **Table 3**. GPU_OCL and CPU_OCL respectively represent the execution times of the kernels we implemented and optimized on GPU and CPU. *ORG* represents the overhead of the program segment in original OpenTLD. Corresponding to the **Table 3**, the speedups based on *ORG* are shown in **Table 4**. As can be seen from the results, the performances of our proposed parallel implementations on CPU and GPU are significantly higher than that of original OpenTLD. Compared to other kernels, Fern Feature Extraction and Overlap and Negative Samples Calculation have the lower performance improvements. During overlap and negative samples calculation, most bounding boxes have no overlap with *currentBB* and cannot perform efficient overlap calculation. Therefore, the kernel of Overlap and Negative Samples Calculation has lower parallelism. Fern Feature Extraction is not compute-intensive and has complex, irregular memory accesses. Our proposed parallel implementation of this kernel cannot make the best use of computing resources and memory bandwidth.

Table 3. Execution time of all kernels on diverse devices (ms) (Lower is better. Red fonts indicate the best performance)

	FFE[1]	FC[2]	NCC[3]	ONSC[4]	PSC[5]
GPU_OCL	502.16	25.08	78.38	15.53	173.01
CPU_OCL	3735.66	281.53	400.39	96.86	990.07
ORG	24925.70	6664.20	7660.12	569.38	17189.30

1 Fern Feature Extraction
2 Fern Classification
3 NCC Calculation
4 Overlap and Negative Samples Calculation
5 Positive Samples Calculation

Table 4. Speedup of all kernels on diverse devices. The original TLD is adopted as baseline to calculate the speedup. The values in () represent the GPU speedups compared to corresponding performance of CPU_OCL. (Higher is better. Red fonts indicate the best performance)

	FFE	FC	NCC	ONSC	PSC
GPU_OCL	49.64 (7.44)	265.74 (11.23)	97.73 (5.11)	36.65 (6.24)	99.35 (5.72)
CPU_OCL	6.67	23.67	19.13	5.88	17.36
ORG	1.00	1.00	1.00	1.00	1.00

Furthermore, the kernels have higher performance on GPU than that on CPU. The kernel NCC has the lowest speedup on GPU compared to CPU. The main calculation in NCC is tree reduction, which is not appropriate for GPU. With the iteration of loops during reduction, the stream processors participating into the calculation are fewer. Until the last loop, only one work item is charge of accumulation. By contrast, the kernel Fern Classification has the highest speedup on GPU compared to CPU. Due to that each bounding box has 10 fern feature vectors to classify and the number of bounding boxes is large, the kernel Fern Classification need numerous work items, which is appropriate for parallel execution on GPU.

5.3 Module Performance Analysis

The parallel optimization based on OpenCL can significantly improve the performance, but lead to extra overhead. The main overhead results from data reorganization and transferring. For example, Before the execution of NCC, the image blocks, the positive and negative samples are reorganized into two one-dimensional arrays and transferred to the computing devices. Beyond this, the extra overhead includes maintenance of the command query and kernel startup.

In order to further evaluate the performance of our proposed kernels based on OpenCL, we consider the extra overhead into the execution times and compare with the original OpenTLD. Here we propose the comparison according to the three modules of the TLD structure:

- Random Fern: includes Fern Feature Extraction, Fern Classification, data preprocessing and transferring in front and behind of the two kernels.
- Nearest Neighbor Classifier: includes NCC Calculation, confidence calculation and data reorganization and transferring.
- Learning Module: includes Overlap Calculation and Negative Samples Calculation, Positive Samples Calculation and data transferring between the two kernels.
-

Table 5. Execution time comparison for three modules of TLD (ms) (Lower is better. Red fonts indicate the best performance. Blue fonts indicate the second-best ones.)

	Learning Module	Nearest Neighbor Classifier	Random Fern
GPU_OCL	1787.53	3184.21	6741.54
CPU_OCL	1653.49	670.25	8242.86
ORG	17760.6	7861.42	37580.6

The reason for dividing three modules is that the overheads between kernels are easy to classify into any part. Likewise, we adopt the corresponding program segments in original

OpenTLD as the baseline. The evaluation results are shown in **Table 5**. As seen from the evaluations, our proposed optimizations can still improve the performance of TLD even if the extra overheads are considered. Meanwhile, the speedups in **Table 2** slightly decrease compared with that in **Table 4**. It indicates that the performance benefits dominate the overheads for OpenCL parallelizing. For Random Fern, the kernel on GPU outperforms than that on CPU. By contrast, for Nearest Neighbor Classifier, the kernel on CPU has better performance. On one hand, the kernel of NCC Calculation has the lowest speedup on GPU compared to CPU. On the other hand, when the kernel NCC Calculation is executed on GPU, numerous image blocks, positive and negative samples are transferred to the GPU and the NCC results are transferred back to the host program. The execution on CPU does not contains these extra transferring. For the same reason, the performance of Learning Module on CPU is better than that on GPU.

5.4 Evaluation for Complete Parallel TLD Tracker

In this section, we evaluate the performance of the complete parallel TLD tracker. For the complete parallel TLD tracker, we further introduce the overlapped execution of Fern Classification and LK optical flow mentioned in **Section 4.1.3**. Based on the evaluation of previous part, we have concluded the most appropriate devices for each part of TLD. Except the original TLD, we adopt the AATLD and H-TLD as baselines for comparison on our experimental platform. Therefore, we evaluate the diverse implementations of TLD on different devices. The diverse implementations are as follows:

- Original TLD. Original OpenTLD on CPU.
- CPU Implementation. All kernels we optimized in this paper are executed on CPU.
- GPU Implementation. All kernels we optimized in this paper are executed on GPU.
- CPU/GPU Implementation. The two kernels in Random Fern are executed on GPU. The three kernels of Nearest Neighbor Classification and Learning Module are executed on CPU.
- AATLD. A heterogeneous CPU-GPU TLD solution using OpenMP, MPI and CUDA
- H-TLD. A heterogeneous CPU-GPU TLD solution using OpenMP and CUDA.

Table 6. Execution time (ms) and FPS (frame per second) comparison for diveres implementations of complete TLD tracker (Lower is better. Red fonts indicate the best performance. Blue fonts indicate the second-best ones.)

Trackers	Original TLD	CPU Imp.	GPU Imp.	CPU/GPU Imp.		AATLD	H-TLD
Execution Time	69953.2	18581.5	19842.2	17841.7		32193.9	25530.4
FPS	13.49	50.79	47.57	52.90		29.32	36.97

The evaluation results for different implementations of TLD are shown in **Table 6**. As seen from the evaluation, CPU implementation outperforms than GPU implementation. Corresponding to the conclusions of previous section, the benefits of Nearest Neighbor Classification and Learning Module on CPU dominates that of Random Fern on GPU. For CPU/GPU implementation, each kernel is executed on the most appropriate device and achieves the best performance. Compared with original OpenTLD, AATLD and H-TLD, the CPU/GPU implementation outperforms and has achieved the speedup 3.92 than original

OpenTLD. Moreover, our proposed three parallel TLD implementations can meet the real-time requirement and the CPU/GPU implementation runs at 52.9 frames per second.

6. Conclusion

TLD, which is a long-term tracking framework, has practicability and wide application prospects on heterogeneous platforms. In order to improve the computing efficiency, in this paper, we propose an efficient parallel TLD tracker based on OpenCL. We focus on parallel designs and optimizations for computing-intensive modules in TLD, including Fern Feature Extraction, Fern Classification, NCC Calculation, Overlaps Calculation, Positive and Negative Samples Extraction. Moreover, we introduce the overlapped execution between tracking module and detection module. A comprehensive evaluation demonstrates that the parallel kernels we proposed can improve the computing efficiency than original TLD. Even if considering the overhead introduced by OpenCL optimizations, our complete parallel TLD tracker has achieved a 3.92 speedup than original TLD on heterogeneous platform and meets the real-time requirement.

References

[1] D. Lee, J. Sim and C. Kim, "Visual tracking using pertinent patch selection and masking," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 3486-3493, June 23-28, 2014. Article (CrossRef Link).

[2] A. Yilmaz, O. Javed and M. Shah, "Object tracking: A survey," *ACM computing surveys (CSUR)*, vol. 38, no. 4, pp. 13, 2006. Article(CrossRefLink).

[3] Y. Wu, J. Lim and M. Yang, "Online object tracking: A Benchmark," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 2411-2418, June 23-28, 2013. Article (CrossRef Link).

[4] H. Nam and B. Han, "Learning multi-domain convolutional neural networks for visual tracking," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 4293-4302, June 27-30, 2016. Article (CrossRef Link).

[5] L. Bertinetto, J. Valmadre, J. Henriques, A. Vedaldi and P. Torr, "Fully-convolutional Siamese Networks for Object Tracking," in *Proc. of Workshop of the European Conf. on Computer Vision (ECCV)*, pp. 850-865, October 15-16, 2016. Article (CrossRef Link).

[6] M. Danelljan, A. Robinsson, F. Khan and M. Felsberg, "Beyond correlation filters: Learning continuous convolution operators for visual tracking," in *Proc. of the European Conf. on Computer Vision (ECCV)*, pp. 472-488, October 11-14, 2016. Article (CrossRef Link).

[7] C. Sun, H. Lu and M. Yang, "Learning Spatial-Aware Regressions for Visual Tracking," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 8962-8970, June 18-22, 2018. Article (CrossRef Link).

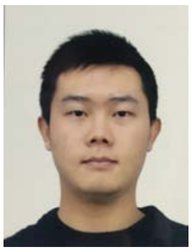
[8] M. Danelljan, G. Bhat, F. Khan and M. Felsberg, "Eco: Efficient convolution operators for tracking," in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 6931–6939, July 21-26, 2017. Article (CrossRef Link).

[9] T. Xu, Z. Feng, X. Wu and Kittler J, "Learning Adaptive Discriminative Correlation Filters via Temporal Consistency preserving Spatial Feature Selection for Robust Visual Tracking," in *arXiv:1807.11348*, 2018.

[10] Snapdragon 835 mobile platform, 2017.
https://www.qualcomm.com/products/snapdragon/processors/835

[11] J. E. Stone, M. J. Hallock, J. C. Phillips, J. R. Peterson, Z. Luthey-Schulten and K. Schulten, "Evaluation of emerging energy-efficient heterogeneous computing platforms for biomolecular and cellular simulation workloads," in *Proc. of Parallel and Distributed Processing Symposium Workshops*, pp. 89-100, May 23-27, 2016. Article (CrossRef Link).

[12] TOP500, “Top500 lists: November 2010,” 2010. https://www.top500.org/lists/2010/11/highlights
[13] Open Multi-Processing, 2015. http://www.openmp.org
[14] NVIDIA Corporation, “CUDA C Programming Guide,” 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
[15] A. Munshi, “The OpenCL Specification,” 2011. http://www.khronos.org/opencl
[16] Z. Kalal, K. Mikolajczyk and J. Matas, “Tracking-learning-detection,” *IEEE transactions on pattern analysis and machine intelligence (PAMI)*, vol. 34, no. 7, pp. 1409-1422, 2012. Article (CrossRef Link).
[17] C. Ma, X. Yang, C. Zhang and M. Yang, “Long-term correlation tracking,” in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 5388-5396, June 7-12, 2015. Article (CrossRef Link).
[18] Z. Zhu, Q. Wang, B. Li, W. Wu, J. Yan and W. Hu, “Distractor-aware siamese networks for visual object tracking,” in *Proc. of the European Conf. on Computer Vision (ECCV)*, pp. 103-119, September 8-14, 2018. Article (CrossRef Link).
[19] H. Fan and H. Ling, “Parallel tracking and verifying: A framework for real-time and high accuracy visual tracking,” in *Proc. of IEEE International Conf. on Computer Vision (ICCV)*, pp. 5487-5495, October 22-29, 2017. Article (CrossRef Link).
[20] Z. Kalal, “OpenTLD,” 2011. https://github.com/zk00006/OpenTLD
[21] D. Concha, R. Cabido, J. J. Pantrigo and A. Montemayor, “Performance evaluation of a 3D multi-view-based particle filter for visual object tracking using GPUs and multicore CPUs,” *Journal of Real-Time Image Processing*, vol. 15, no. 2, pp. 309-327, 2018. Article (CrossRef Link).
[22] J. A. Brown and D. W. Capson, “A framework for 3D model-based visual tracking using a GPU-accelerated particle filter,” *IEEE transactions on visualization and computer graphics*, vol. 18, no. 1, pp. 68-80, 2012. Article (CrossRef Link).
[23] I. Gurcan and A. Temizel, “Heterogeneous CPU-GPU tracking-learning-detection (H-TLD) for real-time object tracking,” *Journal of Real-Time Image Processing*, vol. 16, no. 2, pp. 339-353, 2019. Article (CrossRef Link).
[24] P. Guo, X. Li, S. Ding, Z. Tian and X. Zhang, “Adaptive and accelerated tracking-learning-detection,” in *Proc. of International Symposium on Photoelectronic Detection and Imaging*, June 25-27, 2013. Article (CrossRef Link).
[25] J. P. Lewis, “Fast normalized cross-correlation,” *Circuits Systems and Signal Processing*, vol. 28, no. 6, pp. 819-843, 2009. Article (CrossRef Link).



Zhaoyun Chen is a PhD candidate of College of computer, National University of Defense of Technology. He earned his master's degree in computer science and technology in 2015 from NUDT. His research interests are in the areas of system architecture and computer vision. He has authored and co-authored over 10 publications, including IJCV, INFOCOM, DATE, BMVC, ICIP, HPCC, and FITEE.

Dafei Huang is a research assistant in Southwest Electronics and Telecommunication Technology Research Institute. He received his Ph.D. in 2017. His research interests include parallel programming, compiler optimization, rumtime design and computer vision.

Lei Luo received his B.S., M.S., and Ph.D. degrees from College of Computer, National University of Defense Technology, China, in 2006, 2008, and 2013, respectively. He joined College of Computer, National University of Defense Technology, as a Lecturer, in 2013. His research interests include computer vision, system software, and machine learning.

Mei Wen is currently a professor at the Computer College at the National University of Defense Technology, China. She received her BS, MS, and PhD in Computer Science and Technology from the National University of Defense Technology in 1995, 1999 and 2006, respectively. Her research interests include computer architecture, parallel programming, and scientific computing.

Chunyuan Zhang is a professor at the Computer College at the National University of Defense Technology, China. He received his BS, MS, and PhD in Computer Science and Technology from the National University of Defense Technology, China, in 1985, 1990, and 1996, respectively. He is the director of a series of research projects including National Natural Science Foundation projects of China. His research interests include computer architecture, parallel programming, low power design, embedded systems, media processing, and scientific computing.