Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Ilker Gurcan · Alptekin Temizel

Heterogeneous CPU-GPU Tracking-Learning-Detection (H-TLD) for Real-Time Object Tracking

Accepted for publication in Journal of Real Time Image Processing (October 2015)
The final publication is available at Springer via `http://dx.doi.org/10.1007/s11554-015-0538-y`

Abstract The recently proposed TLD (Tracking-Learning-Detection) method has become a popular visual tracking algorithm as it was shown to provide promising long-term tracking results. On the other hand, the high computational cost of the algorithm prevents it being used at higher resolutions and frame rates. In this paper, we describe the design and implementation of a heterogeneous CPU-GPU TLD (H-TLD) solution using OpenMP and CUDA. Leveraging the advantages of the heterogeneous architecture, serial parts are run asynchronously on the CPU while the most computationally costly parts are parallelized and run on the GPU. Design of the solution ensures keeping data transfers between CPU and GPU at a minimum and applying stream compaction and overlapping data transfer with computation whenever such transfers are necessary. The workload is balanced for a uniform work distribution across the GPU multiprocessors. Results show that 10.25 times speed-up is achieved at 1920x1080 resolution compared to the baseline TLD. The source code has been made publicly available to download from the following address: http://gpuresearch.ii.metu.edu.tr/codes/.

Keywords object tracking · heterogeneous CPU-GPU Implementations · real-time · CUDA

1 Introduction

Real-time tracking of objects in video is an important problem in various domains such as robotics, defense and security. While there are many methods in the literature, most of these are based on short term tracking which often fails if the object is occluded, leaves the field of view and re-enters, changes its appearance rapidly or goes through a large displacement between consecutive frames. In most cases, tracking algorithms need to run in real-time, further imposing algorithmic and computational limitations. Recently proposed TLD (Tracking-Learning-Detection) method [14] aims long term tracking by integrating learning and detection components into the tracker. Tracking and detection are decoupled for robustness. The *tracker* recursively tracks the object based on its location in the previous frame. The *learner* continuously learns the appearance of the object to account for the changing appearance. The *detector* aims to detect the object when it reappears after it gets occluded or gets out of the field of view. TLD has been shown to have promising long-term tracking results and it is algorithmically suitable for real-time applications. On the other hand, the high computational cost of the algorithm prevents running it at higher resolutions/higher frame rates and prohibits running multiple instances of the algorithm for multi object tracking. Hence, a faster implementation of the algorithm is important to: (*i*) increase the resolutions for which the algorithm can run in real-time, (*ii*) allow running multiple instances of the algorithm to support tracking multiple objects, (*iii*) allow running the algorithm at higher accuracy: Tuning the algorithm parameters for higher tracking accuracy requires higher computation power, this results in a trade-off between object tracking quality and processing speed.

Most modern computing platforms have both CPUs and GPUs. CPUs are good at executing branched instructions, typically have higher clock frequencies and larger cache sizes. On the other hand, while GPUs are not optimized for divergent operations and typically have lower clock frequencies, they have higher number of processing cores. Taking these different properties of CPUs and GPUs into account, an optimal solution from a data processing point of view would run serial parts or parts

I. Gurcan · A. Temizel
Graduate School of Informatics
Middle East Technical University,
06800
Ankara Turkey
Tel.: +90-312-2103741
Fax: +90-312-2103745
E-mail: gurcan.ilker@gmail.com, atemizel@metu.edu.tr

requiring branching on the CPU and highly parallel or throughput oriented parts on the GPU. On the other hand, such a heterogeneous solution requires transferring partially processed data between CPU and GPU which brings additional overhead that would not be present in a pure CPU or GPU implementation.

In this work, we describe the design and implementation of a heterogeneous solution, Heterogeneous-TLD (H-TLD), which aims to use both CPU and GPU with high utilization. The proposed optimization scheme dynamically calculates the run-time parameters with respect to the GPU resources. This allows running the algorithm effectively on GPUs having different hardware capabilities. Particular attention is given in design to minimize the data transfers between the CPU and GPU and applying stream compaction whenever such transfers are inevitable. On the GPU side, memory optimizations include design of the data structures to allow coalesced access and use of shared memory whenever suitable. Load balancing between the GPU processing units is achieved by the proposed grouping structure of the data.

The rest of the paper is organized as follows. We first summarize the related work in Section 2 and then give an algorithmic description and computational analysis of TLD in Section 3. This is followed by the description of the proposed H-TLD framework design and optimizations in Section 4. We analyze the performance of the algoritm, compare its performance against the baseline TLD and state-of-the-art in Section 5 and finish with the concluding remarks in Section 6.

The source code and the different resolution videos used for benchmarking has been made publicly available to download: http://gpuresearch.ii.metu.edu.tr/codes/.

2 Related Work

A feature tracking and matching based algorithm is described in [24]. The implementation is based on OpenGL and Cg making it difficult to adapt to newer generation devices. GPU implementation of a particle filter based tracker is described in [8]. It is shown that the GPU version is more effective when the particle count is higher and it can achieve more than 12 times speed-up compared to the multi-core CPU implementation.

A video analytics system targeting video surveillance applications is described in [10]. As part of this work, a tracking algorithm was optimized on GPU. However, due to its target application field, tracking is based on background subtraction and assumes that the camera is static. Another background subtraction based algorithm is described in [15]. The proposed method achieves real-time performance on high-resolution videos for moving object detection, however it only involves detection and no object tracking method is present.

Acceleration of a boosting based face tracking algorithm, which is reported to achieve 500 frames per second on GPU is given in [11]. A particle filter based face tracking algorithm developed on a heterogeneous CPU-GPU environment is described in [17]. The latter two particularly target face tracking and as such, they are not general purpose tracking algorithms. Most of the solutions aiming to accelerate object tracking focus on analyzing and improving the optical flow part of the tracking algorithms; rather than focusing on the whole problem [19] [18] [20]. The method proposed in [18] consists of feature detection and optical flow modules based on pyramidal Lucas-Kanade (LK) tracker [5] running on GPU. They compared their work with OpenCVs feature detector, tracker implementation and showed improved performance when some parameters were tuned. In [20], CUDA was used to accelerate the LK optical flow algorithm on GPU.

These aforementioned methods in the literature can be classified as short term tracking methods. Short term tracking often fails if the object is occluded, disappears from the field of view and reappears, changes its appearance rapidly or goes through a large displacement between consecutive frames. The recently proposed TLD method [14] has become a popular visual tracking algorithm as it made robust long-term tracking possible by including learning and detection in the loop. A C++ based implementation of TLD having an extended object detection cascade and using alternative features is described in [21]. The processing speed is improved by excluding the image warping step in the learning and the object detection cascade. This modification increases the likelihood of falsely eliminating true object candidates. While there has been a recent interest in GPU implementations of tracking algorithms, our literature survey reveals that there are only two GPU based implementations of the TLD algorithm [4] [23]. Implementation in [4] is a direct porting of the CPU version and it does not involve design and optimization with regards to the particulars of such a platform. In [23], the detection part of TLD is ported to run on GPU, however no specific design or optimization is performed for this platform. Another point to note is that while CPU-GPU heterogeneous processing has a notable potential, there is limited amount of work leveraging the advantages of both CPU and GPU in a heterogeneous setting. Hence our motivation in this work is to design and implement the state-of-the-art TLD algorithm to make effective use of GPU and benefiting from a CPU-GPU heterogeneous architecture. As a fundamental design principle, we refrain from any algorithmic modifications which might improve the processing speed at the expense of the tracking performance and focus on the baseline TLD algorithm [14].

3 Algorithm Description and Analysis

In this section, we first give a general outline of the TLD algorithm and its main components. Then we analyze these components in more detail to help with the design decisions on the heterogeneous platform.

3.1 Tracking-Learning-Detection (TLD) Algorithm

TLD is composed of 3 main components; tracking, learning and detection. Tracker and detector run in parallel and the object location is estimated by the consensus of both. Aim of the learning component is to adapt the object model over time by using feedback from both tracker and detector. An overview of the TLD algorithm is provided in Algorithm 1.

Tracking is the process of predicting the displacement of each previous reliable point (pixel location) enclosed by object's Bounding Box (BB); and measuring the reliability of these newly computed displacement vectors using forward-backward tracking and NCC (Normalized Cross Correlation) scores [16]. All points with highly confident displacement vectors are the output of the tracking module.

Detection is the process of deciding upon whether the object is still in the field of camera's view and if not, detecting its presence when it re-appears. Its classifier is based on random forests [7]. Since TLD is a semi-supervised learning method; the classifier is initially trained with the labeled data. Then it starts learning different appearances of the object as each new frame is processed. Detector employs such a classifier for finding BBs where the object may potentially exist. Output of detection module is the confidence values for all BBs.

Learning component exploits the notion of P-N Experts which restrict the labeling of data [12]. The experts use output of the tracker in order to evaluate candidate BBs with high confidence values against a structural constraint. P-expert considers trajectory that has been built up by the tracker so far, it calculates the euclidean distance between each candidate BB and the trajectory. If the measured distance is less than a certain predefined threshold value and the BB was previously labeled as a negative patch by the classifier; then P-expert relabels it as positive. On the other hand, N-expert relabels a BB as negative provided that it had been classified as a positive patch but failed the structural constraint. Finally, these correctly relabeled patches are used to train the detector's classifier.

Algorithm 1 TLD Processing Overview

procedure tldProcessFrame(I)
  ▷ **Step#1 Compute optical flow. Return bounding box (BB) found by the tracker**
  $tBB \leftarrow tracker(frame[I-1], frame[I])$
  ▷ **Step#2 Run variance filter ->Ensemble Classifier ->Nearest Neighbor operations in order. Return bounding boxes (BBs) found by the detector**
  $dBBs \leftarrow detector(frame[I])$
  ▷ **Step#3 Run the integrator in order to find the trajectory**
  $cBBs \leftarrow cluster(dBBs)$
  $BBs \leftarrow Find\ clusters\ that\ have\ higher\ conf\ than\ tBB's$
  if number of BBs = 1 **then**
    Re-initialize the tracker's trajectory to this BB
  else
    $closeBBs \leftarrow tBB,\ dBBs\ with\ the\ conf > th$
    $curBB \leftarrow$ Adjust the tracker's trajectory using closeBBs
  end if
  ▷ **Step#4 Run learner. Generate positive/negative patches using the newly found BB(trajectory), and retrain the ensemble classifier**
  if trajectory is within the frame boundary **then**
    $learn(curBB, frame[I])$
  end if
end procedure

3.2 Analysis of the Algorithm

In this section, the most time consuming computational components of the TLD algorithm are defined. Then execution times of these components are further analyzed in detail to identify the performance bottlenecks. While there are also other components of the algorithm, their execution times are insignificant compared to the ones given in Table 1. Therefore, they were not taken into consideration in this computational analysis.

Main computational components of *tracking* are Lucas-Kanade (LK) optical flow and Normalized Cross Correlation (NCC) calculation. LK optical flow calculation estimates frame-to-frame optical flow as in median flow tracker [13]. It is run twice per frame for computing optical flows in forward and backward directions respectively. In the first run, it predicts positions of the tracking points in the next frame corresponding to the ones in the previous frame. Then in the second run, it tries to find out where these predicted points can be repositioned on the previous frame. Similarity measure between small rectangular regions surrounding these predicted locations and their associated original tracking points, is calculated using NCC [16]. Tracking points with results lower than a predefined threshold value are considered to be unreliable and eliminated.

Learning consists of patch warping, pattern generation, random forest update and BB overlap computation. Patch warping generates different variants of a positive patch for scale and rotational invariancy. After *tracker* estimates the position of the next BB of the object, P-N experts decide on whether BBs detected by the *detector* belongs to the object or not. Using this information, it retrains the ensemble classifier by generating positive and negative patches. Random forest is updated using these

patterns. As for BB overlap computation, it calculates the overlapping area between any two BBs.

Detection module can be subdivided into the following computational components: total recall computation, integral image calculation and image blurring. Total recall computation calculates a confidence value for each BB. Integral images are necessary to compute patch variance for each BB in an efficient way. Lastly, image blurring is the preprocessing step of each frame and smooths out the undesired details and noise.

In order to analyze the computational cost of components mentioned above, we ran the algorithm with different video resolutions (480x270, 960x540 and 1920x1080). Different resolution video sequences have been obtained from the same original 1920x1080 video having 464 frames by downsampling. As a result they have identical content. The experiments were performed on a test platform having a Intel i7 4770K 3.5 GHz 4 core CPU with 32GB RAM and running Windows 7 64-bit. Test GPU was a NVIDIA Tesla K40c of compute capability 3.5 with 15 streaming multiprocessors, each having 192 computing cores (in total of 2880 cores). It has 2 asynchronous copy engines and its Hyper-Q was enabled.

We analyzed the execution time per call as well as the total execution time of each component. Although how many times a component is called depends on the characteristics of the test video, the total execution times still give an indication of the most time consuming components. The results are reported in Table 1 as "time per call" and "time for the whole sequence" respectively.

In Figure 1, the horizontal bar illustrates execution times for each computational component. In this scenario, original-TLD implementation was tested on the 1920x1080 resolution test video.

It can be inferred from Table 1 and Figure 1 that the most computationally expensive operations are "total recall computation", "image blurring", "LK optical flow calculation", "patch warping" and "integral image computation" in descending order. All the remaining operations combined takes less time than the integral image computation which is the least time consuming one among these aforementioned components. So we concentrate on these 5 components and analyze them in more detail with an aim to implement in a heterogeneous context.

3.2.1 Total Recall Computation

Total recall computation is used in the detector module and it is the most time consuming component as highlighted by the execution time analysis. It runs an exhaustive search for the object using a sliding window approach. First, a series of BBs spanning the whole image are created at the initialization. The size of the BB is determined by the tracked object size. Then BB properties are saved into a data structure on both CPU and GPU memories to be used in the subsequent steps. A series of computationally costly sub-tasks are executed for each BB: patch variance computation, feature comparison, and confidence value calculation. The best candidate for the object is determined by the result of these computations.

Number of BBs depends on the image resolution and the object size. For a QVGA frame (320x240) and an object BB with an initial size of 25x42, 29855 BBs are required to scan the entire frame. This brings a high computational cost and the cost increases proportional to the image resolution. Due to the independent nature of calculations for each BB and high number of BBs, this part is suitable to parallelize and implement on the GPU. Hence we decided to run this part on the GPU.

3.2.2 Image Blurring

Image blurring is used in the detector module. It is a very common operation in various image and video related tasks and it was shown that considerable gains could be obtained in GPU implementations [6]. So we decided to run this part on the GPU. As there are already available optimized implementations, we decided to use the OpenCV implementation [6].

3.2.3 LK Optical Flow Calculation

LK optical flow is used in the tracker module which is based on Median Flow Tracker (MFT) [13]. In MFT, pyramidal LK optical flow is calculated for all feature points of the object. Optical flow is called twice for each image pair; first in the forward and then in the backward direction. This part lends itself well for a GPU implementation as it can be implemented in a data parallel fashion and there are a large number of data points. As in the case of image blurring, there are already available optimized implementations and we decided to use OpenCV implementation [6].

3.2.4 Patch Warping

Patch warping is a part of the learning component. The other parts of the learning component are pattern generation, random forest update and BB overlap calculation. These components do not take significant processing time as they involve calculation for a limited number of BBs and learning is invoked intermittently. As such, implementation of these parts on GPU were considered infeasible. Processing these parts on the CPU while processing patch warping on the GPU necessitates moving large amounts of data (i.e. warped patches) between CPU and GPU. As a result, we decided to keep the entire learning component on the CPU.

Table 1 Time measurements for individual components for different resolutions

Component	Time per call (ms)			Time for the whole sequence (ms)		
	480x270	960x540	1920x1080	480x270	960x540	1920x1080
Tracking						
LK Optical Flow	1.10	4.28	17.52	509	1982	8112
Normalized Cross Corr.	0.62	0.63	0.77	287	292	357
Learning						
Pattern Generation	0.01	0.02	0.08	32	65	258
Random Forest Update	0.44	1.20	1.89	141	386	608
Patch Warping	0.08	0.23	1.27	326	938	5180
BB Overlap	0.02	0.06	0.27	35	104	467
Detection						
Total Recall	5.93	20.40	62.50	2752	9466	29000
Integral Image	0.27	1.10	4.56	126	510	2116
Image Blurring	1.69	6.51	23.65	782	3021	10974

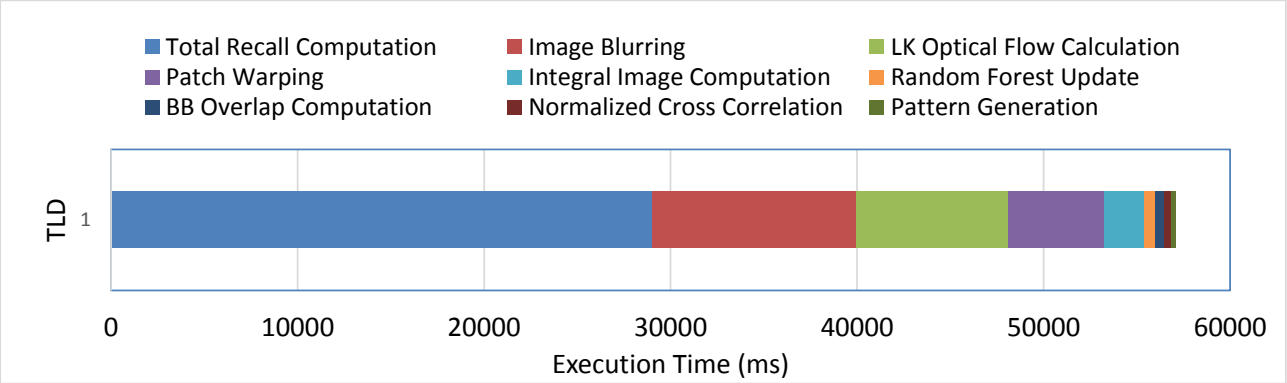

Fig. 1 Execution times for 1920x1080 video for different components in descending order

3.2.5 Integral Image Computation

Integral image computation is a part of the detector module. This task is highly parallelizable and suits well to the GPU architecture. Total recall computation step which was decided to be implemented on GPU as mentioned above, follows it and uses its output. Implementing integral image computation on the GPU also prevents moving data back and forth between GPU and CPU, hence we decided to implement this component on the GPU using [2].

As a result of this analysis we decided to move the total recall computation, image blurring and integral image computation components to the GPU. Albeit being the third most costly component, we decided to keep patch warping on the CPU as it needs to interact with the other components of the learning which are to be kept on the CPU side. In the next section, we give a detailed description of the design, implementation and optimization of these components.

4 H-TLD Framework Design

An important advantage of heterogeneous computing is its potential to utilize the resources efficiently by keeping both CPU and GPU occupied at the same time. Tracking and detection modules of TLD run independent of each other; i.e. they do not need to share any data until the integrator receives results from both to estimate the object position. This allows overlapping of various components on CPU and GPU. In addition, multi-core CPUs may be exploited in order to perform tasks that are not preferred to be executed on GPUs for particular reasons (such as serial operations and parts of the algorithm which require moving data back and forth between CPU and GPU). Moreover; some critical parts of the TLD algorithm (particularly its detector module) can be designed to overlap data transfers between CPU/GPU and kernel executions.

Figure 2 shows an overview of the heterogeneous H-TLD framework. The main design considerations of the H-TLD framework are (*i*) executing highly parallel parts on the GPU while executing serial operations on the CPU, (*ii*) keeping the data transfers between CPU and GPU at a minimum, (*iii*) applying stream compaction and overlapping data transfers with computation whenever possible, (*iv*) balancing the load for a uniform work distribution on the GPU multiprocessors. In the remainder of this section we describe the design of the individual components in accordance with these design considerations.

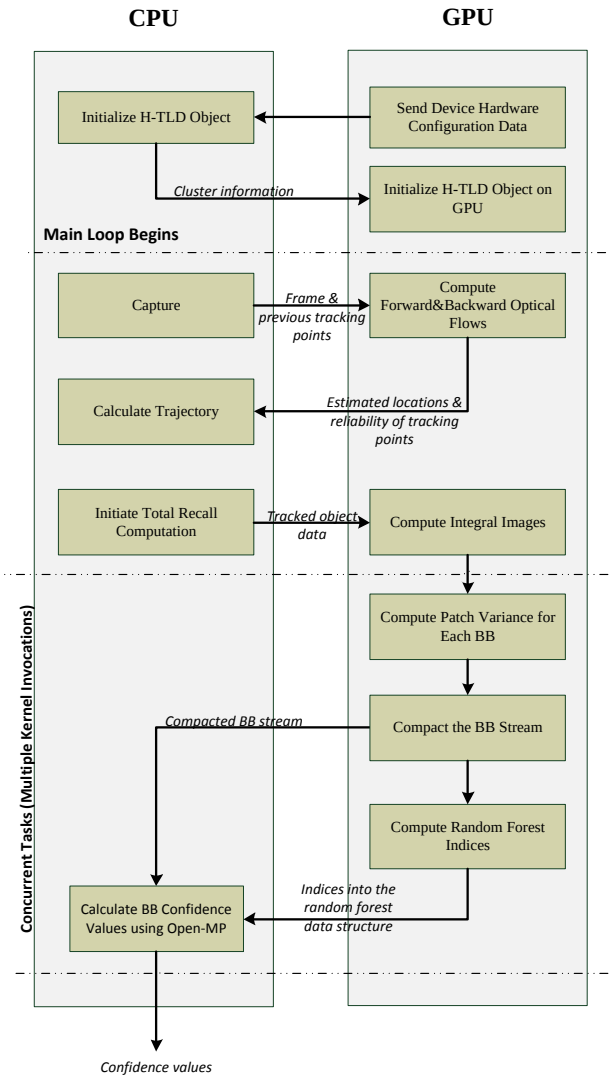

Fig. 2 H-TLD heterogeneous CPU-GPU software framework

4.1 Total Recall Computation

Total recall computation is designed to run on both processing units to efficiently utilize both CPU and GPU and it has 4 separate subcomponents executed in the order given below:

- Patch variance computation (GPU),
- Stream compaction (GPU),
- Random forest index calculation (RFI) (GPU),
- Confidence calculation (CPU).

The data organization concepts used in parallelization of total recall computation are described below and illustrated in Figure 3.
Scale Level: BBs are generated at different scaling factors for scale invariance.
Scan Line: Top and bottom lines of a group of BBs align and they lie within a single row of an integral image, we call such a row a *scan line*. Note that, one scan line may form top and/or bottom borders of BBs from different scale levels. This property will be used to reduce the number of scan lines to be read into the shared memory.
Scan Line Pair: A pair of scan lines encompasses top and bottom borders of a group of BBs at a given scale level.
Cluster: A cluster consists of a number of scan-line pairs arranged for load-balancing and this is the actual logical unit to be run by the GPU thread blocks. Each CUDA block is responsible for processing one cluster per invocation. Clusters are ensured to have approximately the same number of BBs for the purpose of load-balancing as detailed later in this section. The number of scan-line pairs in a cluster is limited by the shared memory size of the particular device.

Since GPU resources are limited, we compute the maximum number of BBs that can be processed on GPU cores at a single kernel invocation. As a result, there could be a number of asynchronous invocations. The number of kernel calls are calculated when the H-TLD object is initialized with respect to the object's BB size, video resolution, and GPU's hardware capabilities. This allows run-time optimization for a specific GPU by making effective use of its resources. Pseudo-code shown in Algorithm 2 summarizes this calculation. At Step#1, the total number of threads per each CUDA block is calculated with respect to the two hardware constraints: (*i*) maximum number of registers reserved for a block, (*ii*) maximum number of threads that can run concurrently on a single streaming multiprocessor.

At Step#2, the maximum shared memory size required to store scan-line pairs for each block is calculated. Each time a new group of BBs spanning a particular scan-line pair is added in the while loop as shown in algorithm 2 (hence more BBs will be processed at a time resulting in higher occupancy), shared memory consumed by any block will increase as well; until it reaches its maximum limited by the hardware. Although, clusters may use varying sizes of shared memory this parameter should be predefined for all blocks within a kernel. Thus, all blocks allocate the same amount of memory regardless of how much they require. After these two steps are completed, occupancy should be optimized as described in Step#3 in 2.

4.1.1 Patch Variance Computation

Patch variances of candidate patches (BBs) are calculated using integral and squared integral images. Integral images enable fast calculation of variances. Then, the patches having low variances (compared to the variance of the target object patch) are eliminated as these are not considered to be object candidates. Patch variance calculation was implemented on the GPU utilizing

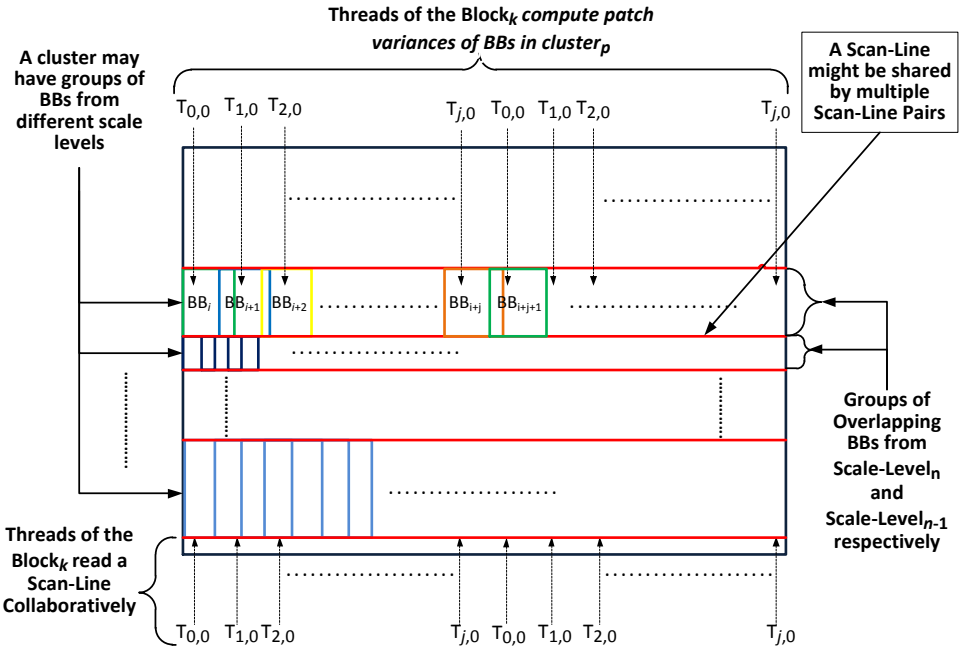

Fig. 4 A cluster and its mapping to GPU multi-threading concepts

Algorithm 2 Calculating number of asynchronous kernel invocation

procedure CALCNUMOFKERNELINVOC
▷ **Step#1 Calculate maximum number of threads per block(numofTPB)**
Maximize $numofTPB$ wrt "maximum register per block" constraint
Maximize $numofTPB$ wrt "maximum number of resident threads per multiprocessor" constraint
▷ **Step#2 Calculate required shared-memory size (sms)**
Calculate total number of scan line pairs ($totalNumofSLP$)
$clusterSize \leftarrow 1$
$sms \leftarrow 0$
while $sharedMemPerBlock > sms$ AND $clusterSize < totalNumofSLP$ **do**
Compute required sms
$clusterSize \leftarrow clusterSize + 1$
end while
▷ **Step#3 If occupancy is too low**
while $occupancy < threshold$ AND $clusterSize \geq 2$ **do**
$clusterSize \leftarrow clusterSize - 1$
Compute required sms
Calculate $activeWarps$ wrt new "sms"
$occupancy \leftarrow activeWarps/maxActiveWarps$
end while
▷ **Note that each cluster must be processed by a single CUDA block**
$numofClusters \leftarrow totalNumofSLP/clusterSize$
Calculate max number of blocks per grid(numofBPG)
$numofInv \leftarrow numofClusters/numofBPG$
return $numofInv$
end procedure

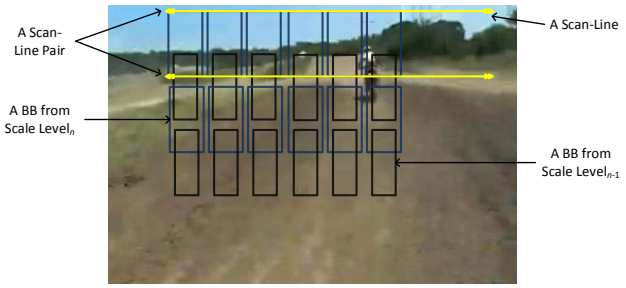

Fig. 3 Data organization for patch variance computation

the shared memory. It has the following steps: (*i*) read the scan-line pairs into the shared memory, (*ii*) compute patch variance for each BB either based on integral image or squared integral image, (*iii*) threshold each variance and assign 0 or -1 for above and below threshold respectively to use at the stream compaction stage.

The mapping between the GPU multi-threading concepts and the data organization concepts (scan line, scan line pair, etc.) for patch variance computation kernel are described below and illustrated in Figure 4.

- Each thread in a block reads various pixel locations on scan lines either from the integral image or from the squared integral image into the shared memory collaboratively.
- Patch variance computation of a BB is done by a single thread in the block. However, a thread may process more than one BB in a cluster.

For an effective implementation of this computation, data needs to be partitioned in a way to ensure uniform work distribution across the threads, and needs to be organized for parallel implementation. This is an important consideration for an effective GPU based implementation.

During patch variance computation, ideally, BBs from different scale levels should be grouped to ensure the clusters to have the same number of BBs to be processed. Exploitation of spatial locality of BBs is also important, as a careful design would allow use of scan-lines read into shared memory by higher number of BBs during the process.

Total number of BBs in a scan-line pair at a scale level n is calculated as in Equation 1

$$f(n) = \frac{W - \alpha^n w}{\beta \alpha^n w} \quad (1)$$

where, W is the width of video frame, α is the base scale level, β is the shifting factor in the range of (0, 1] and w is the width of initial bounding box of the tracked object. Note that BBs within a particular scan-line pair must be processed by a single CUDA block. Processing BBs in the same order as they are processed on the host side prevents proper load balancing on GPU. This is illustrated in Figure 5 (top). As computation proceeds from the left to the right, each scan-line pair and the BBs located within it at a certain scale level are located contiguously in the memory. Thus, blocks of computing kernels would start processing fewer numbers of BBs compared to the earlier ones at higher scale levels, resulting in many processing units to become idle after a while. To enable distribution of similar amounts of work, a pre-processing step reorders the data as illustrated in Figure 5 (bottom) where BBs from different scale levels are grouped together.

Use of integral images makes it possible to compute the patch variances for a specific group (horizontal neighborhood) of BBs by only using the scan lines spanning them. As a result, patch variance can be calculated by reading only these scan lines into the shared memory.

A pixel located on a single scan-line may be used by multiple threads in a block while patch variances are being computed. There are two cases where this might occur: (*i*) As illustrated in Figure 4, scan-line pairs may share a common scan-line (the one that spans their bottom and top borders respectively). Those BBs from different scale levels are likely to be processed in the same CUDA block. (*ii*) Neighboring BBs in a scan-line pair overlap. As a result, these BBs share most of the data resulting in multiple accesses to the same pixels.

As it can also be seen from the pseudo-code in Algorithm 3; patch variance for a BB is computed in two iterations due to the limited size of the shared memory. Thus, the scan-lines for integral image and squared-integral image are read into the shared memory separately, so that more BBs could be processed per invocation within a single CUDA block. Loading higher number of scan-lines into the shared memory results in a higher number of BBs to be processed per single kernel invocation. This leads to fewer number of kernel invocations, reducing the potential overhead of multiple kernel invocations.

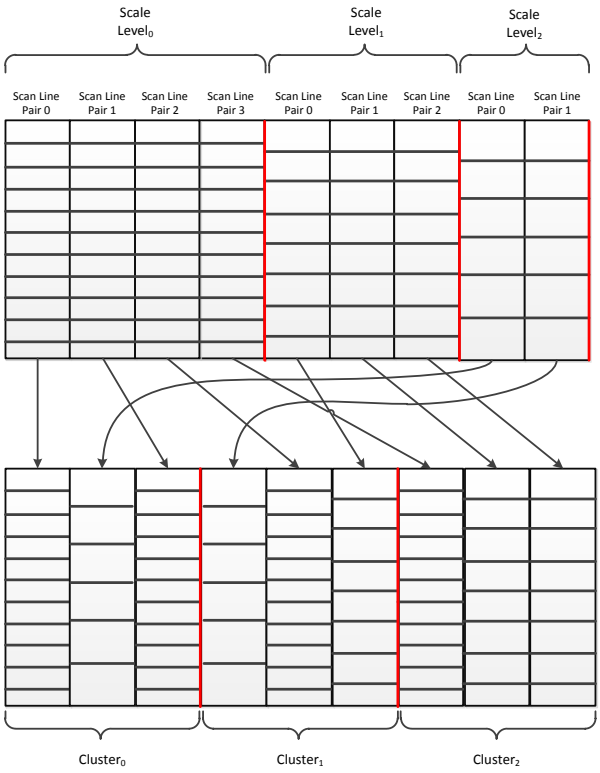

Fig. 5 Original memory ordering which causes load imbalance in processing (top), proposed BB ordering to achieve load balancing (bottom)

Furthermore, threads do not read contiguous memory locations during the computation, which has a negative impact on the performance due to uncoalesced global memory access. Reading scan-lines into shared memory also eliminates the uncoalesced access problem.

4.1.2 Stream Compaction

BBs which fail the variance test are eliminated and the remaining are passed onto the next stage. However, this results in non-sequential data access if no post-processing is applied. Thus, we propose a stream compaction step where only the remaining BBs are ordered to allow sequential access in the following steps and copy only their data back to the host side after the completion of *confidence index calculation* step.

Figure 6 (top) shows an example BB stream before the stream compaction, where BBs having attributes of -1 (BB_1,BB_2,BB_4 and BB_6) need to be eliminated. In order to compact the stream, after elimination of these,

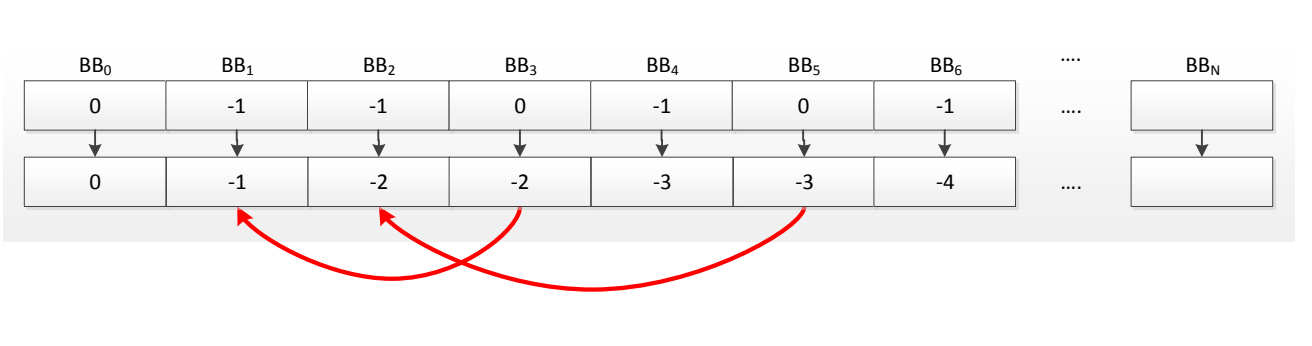

Fig. 6 BB stream after patch variance result (top), after the prefix-sum operation (bottom)

Algorithm 3 Computing Patch Variance on GPU

```
procedure COMPUTEPVONGPU(cluster_p)
    for i ← 0, 1 do
        if i = 0 then
            scanLines ← Read integral image
              into shared memory
        else
            scanLines ← Read squared integral image
              into shared memory
        end if
        Synchronize all threads in the block
        for all BBs in cluster_p do
            if i = 0 then
                pvI_k ← Compute pv using integral image.
            else
                pvSI_k ← Compute pv using squared integral
                  image.
            end if
        end for
    end for
     ▷ Decide which BBs are potential object
       candidates
    for all BBs in cluster_p do
        res ← pvSI_k − pvI_k
        if res ≥ threshold then
            pvStatus_k ← 0
        else
            pvStatus_k ← −1
        end if
    end for
end procedure
```

all succeeding BBs with attributes of 0 need to be grouped together by shifting to the left. The shift amounts can be obtained by running a prefix-sum as shown in Figure 6 (bottom). Stream compaction allows efficient use of the memory bandwidth between the host and the device by eliminating unnecessary transfers. The prefix-sum operation is performed using CUB [1]. CUB library provides software primitives and it is reported to have better performance than competing higher level libraries such as Thrust [3]. Another motivation to use CUB is that it allows using separate CUDA streams to make kernel computations overlap with the other kernels of the H-TLD framework. Issuing kernel execution followed by compacting and memory copying operations as shown in Algorithm 4 (**procedure** nonoverlapped) results in non-overlapping of kernel execution and memory transfers. In order to facilitate overlapping of multiple independent kernel invocations with memory copy operations, all independent calls for individual kernels and asynchronous memory copy operations in all streams could be grouped together. Algorithm 4 (**procedure** overlapped) shows the proposed call ordering for overlapping.

Algorithm 4 Non-overlapped and overlapped code

```
procedure NONOVERLAPPED
    for i ← 0, numofAsyncInv do
        Issue PV Computation for clusters in Group_i to
GPU
        Issue compacting BB stream for Group_i to GPU
        Issue copying of shift amounts within the BB Stream
     for each BB in Group_i from GPU to CPU
    end for
end procedure
```

```
procedure OVERLAPPED
    for i ← 0, numofAsyncInv do
        Issue PV Computation for clusters in Group_i to
GPU
    end for
    for i ← 0, numofAsyncInv do
        Issue compacting BB stream for Group_i to GPU
    end for
    for i ← 0, numofAsyncInv do
        Issue copying of shift amounts within the BB Stream
     for each BB in Group_i from GPU to CPU
    end for
end procedure
```

4.1.3 Random Forest Index Calculation

Calculation of indexes into the confidence array (i.e. array holding the weights) is the last step and does not require moving large amounts of dynamic data back-and-forth between CPU and GPU. During the calculation, some global memory accesses are not coalesced as illustrated in Figure 7. This access pattern is observed when; (*i*) accessing the feature points to be compared in the blurred image, (*ii*) accessing the absolute top-left corner positions of the remaining BBs. As shown in Table

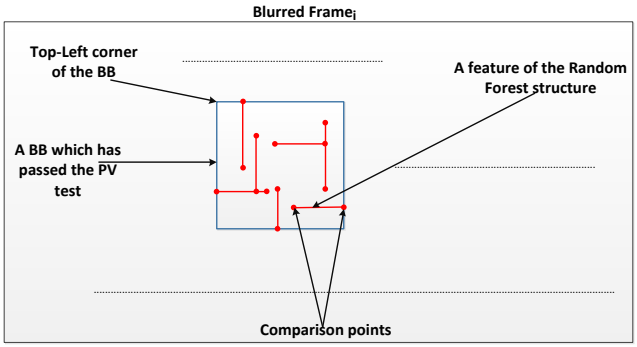

Fig. 7 Random Forest index calculation

2, even with this non-ideal access pattern, 16.55 times speed-up is achieved for random forest index calculation due to its highly parallel nature.

Algorithm 5 Calculating random forest indexes

```
procedure computeRFIndicesOnGPU(blurredImage, numofBB)
    Read all feature offsets into the shared memory collaboratively.
    Synchronize all threads in the block
    for all remaining BBs (k) in compacted BB stream do
        for i ← 1, numofForests do
            index ← 0
            for j ← 1, numofFeatures do
                index ← index << 1
                fp1 ← Calculate the absolute position of the 1st comparison point
                fp2 ← Calculate the absolute position of the 2nd comparison point
                if blurredImage[fp1] > blurredImage[fp2] then
                    index ← index|1
                end if
            end for
        ▷ Arrange the data in global memory to allow coalesced access
            confIndices[i * numofBB + bbIdx_k] ← index
        end for
    end for
    return confIndices
end procedure
```

4.1.4 Confidence Calculation

Since the random forest weights are updated by the learning component on the CPU side, moving the weights to the GPU each time they are updated would be a costly operation. Hence, it is more feasible to run this step on the CPU side. For this step, OpenMP [9] is used to employ all the CPU cores. Each group of BBs is split into equal size chunks so that each of these chunks could be processed by different CPU cores concurrently. On the other hand, the caller of this method is not aware of the data reorganization that was made due to the load balancing operation; therefore the method should return confidence values for all BBs. However, this will lead to a false cache-line sharing problem in CPU cores. As illustrated in Figure 8, cache-lines belonging to the different CPU cores need to be flushed out before another core (which is to execute a write-transaction) has access to a nearby memory location. This problem could be overcome by reordering data in separate temporary arrays for each CPU thread and then letting a single CPU thread gather those values on the final output array which is to be returned to the caller as shown in Algorithm 2. By this way it is ensured that the caller receives them in the original data ordering.

5 Experimental Results

For the experiments, the same system used for the computational analysis of the algorithm, having Intel i7 4770K CPU and Tesla K40 GPU, was used.

5.1 Comparison with the baseline TLD and analysis

In this subsection, we present our experimental results, performance comparison with the baseline TLD and a detailed analysis of the results. In addition to the overall speed-up analysis for different video resolutions, a more detailed analysis of the total recall computation is given as this is the most time consuming part of the algorithm.

As seen in Table 2 and Figure 9, there is an overall speed-up of 9.14 times for the total recall computation (TRC). Random forest index (RFI) calculation takes approximately 83% of TRC of TLD. Even though there is a speed-up of 16.55 times for the RFI calculation in H-TLD, the execution time is still dominated by RFI calculation which takes approximately 46% of TRC of H-TLD.

28% of the total execution time in H-TLD is spent on stream compaction and other overheads due to the heterogeneous architecture. Since RFIs are required in the confidence value calculation phase; data have to be transferred to the CPU per asynchronous kernel invocation at this stage. A further analysis reveals that the data transfers take up approximately 78% of total RFI calculation time. Since RFIs are used to calculate confidence values of BBs, all threads must be suspended until the transfer is completed. It might be claimed that the threads could execute other instructions to hide the memory transfers, however there are no other computationally costly operations to process at this stage.

For a comparative evaluation of TLD and H-TLD at different resolutions, we recorded a 1920x1080 video and obtained 2 other videos having resolutions 480x270 and 960x540 by downsampling. As a result, all the different

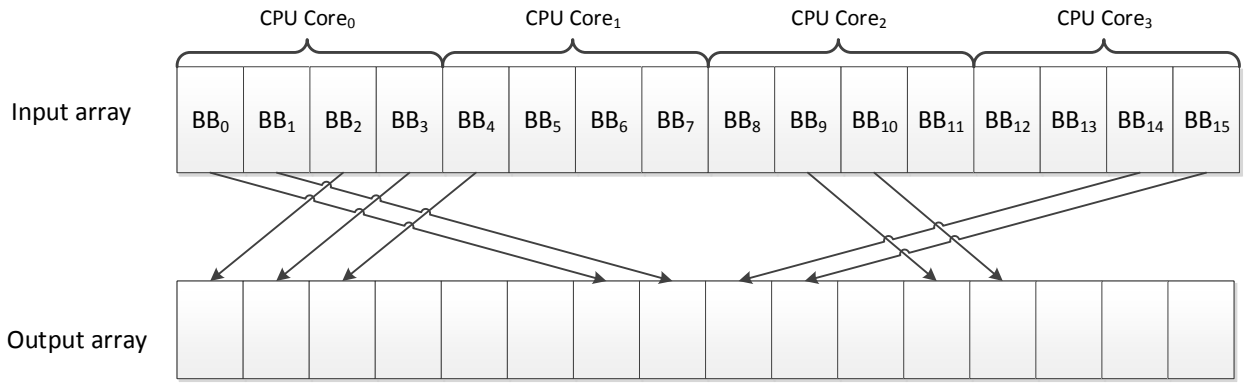

Fig. 8 Elimination of false-cache sharing in CPU cores

Table 2 Comparison of execution times (per call) of TLD and H-TLD for different stages of TRC executed on [1]CPU, [2]GPU, [1,2]CPU&GPU

Component	TLD Exec.Time (ms)	TLD % of Total Exec. Time	H-TLD Exec.Time (ms)	H-TLD % of Total Exec. Time	Speed-up
Patch variance computation	0.170[1]	2.65%	0.046[2]	6.55%	3.70
Stream compaction	-		0.046[2]	6.55%	-
RFI calculation	5.296[1]	82.54%	0.320[2]	45.58%	16.55
Confidence value calculation	0.950[1]	14.81%	0.138[1]	19.66%	6.88
Other overheads	-		0.152[1,2]	21.65%	-
Total	6.416		0.702		9.14

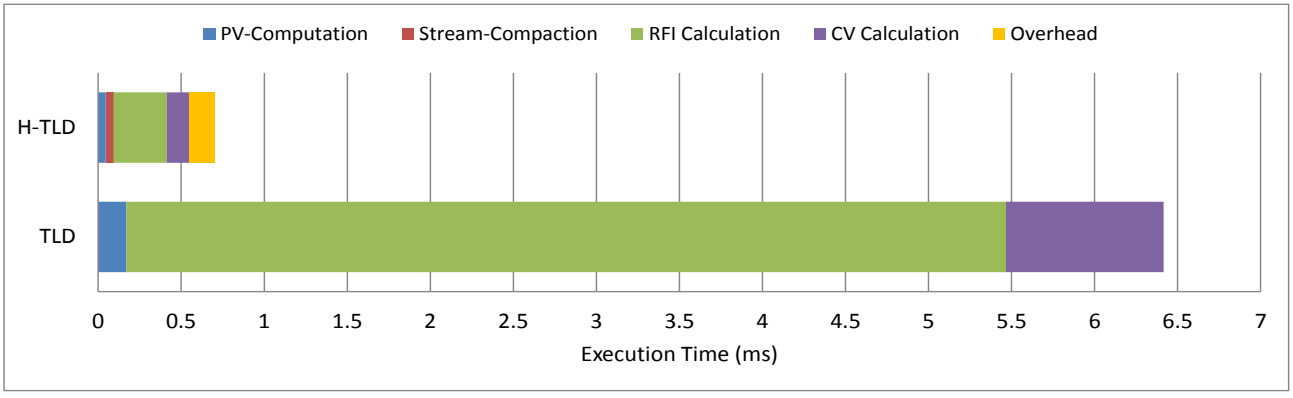

Fig. 9 Stacked view of elapsed time for each subcomponent of total recall computation for TLD and H-TLD

resolution videos have the same content, enabling analysis independent of the content. We used these videos to measure the execution time of each individual computational components separately. The results are shown in Table 3 and Figure 10. The component *others* refer to the cross correlation, pattern generation, BB overlap computation, random forest update and patch warping which are the same in both implementations and hence showing a speed-up of 1.00. The results show that the total speed up is 2.74, 6.06 and 10.25 times for 480x270, 960x540 and 1920x1080 resolutions respectively.

As expected, the total speed-up increases with the increasing resolution due to the massively parallel architecture of the GPU. The increasing resolutions of the captured videos bring the need for algorithms running at higher resolutions in real-time. In addition, use of higher

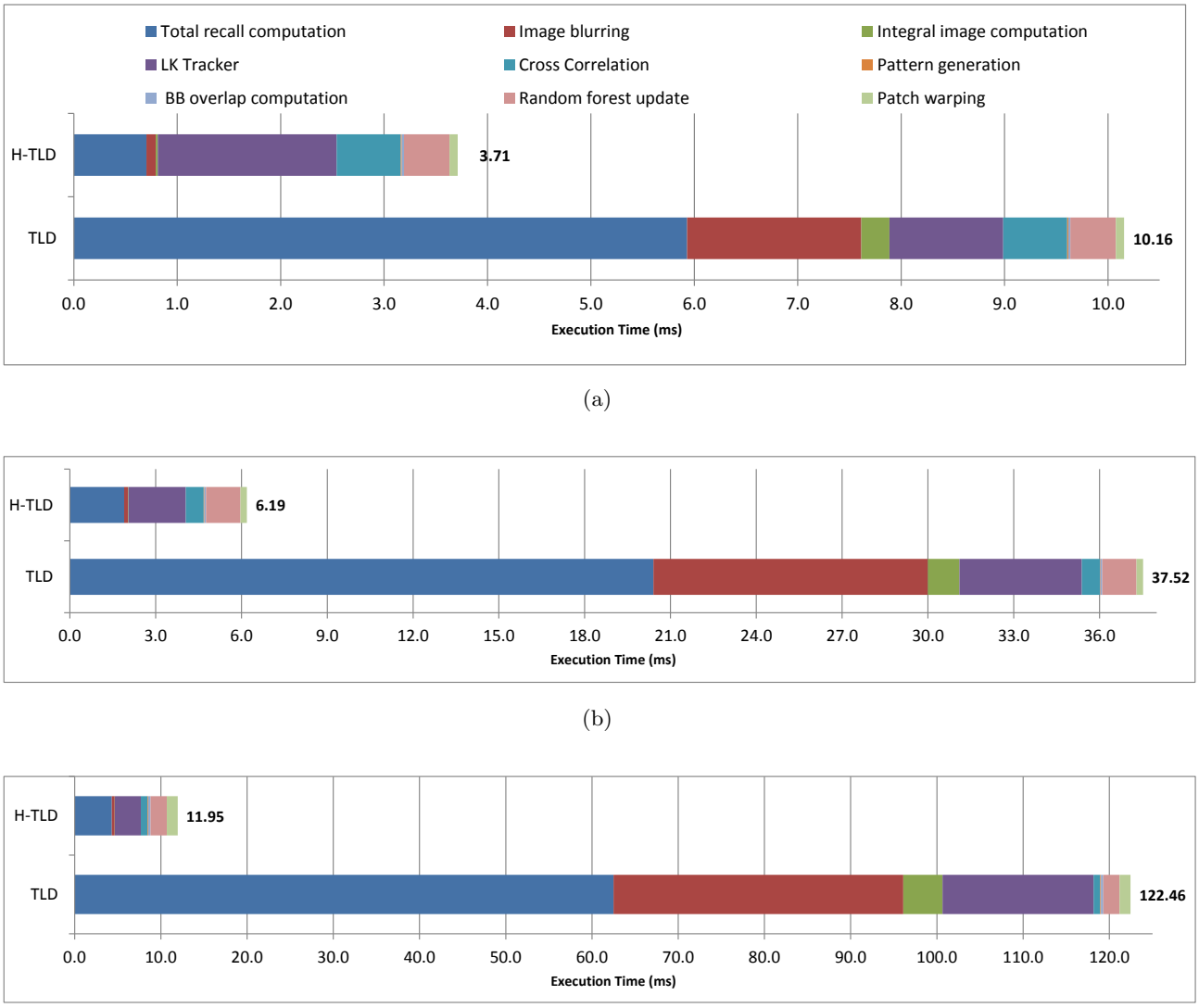

Fig. 10 Comparison of TLD and H-TLD for a)480x270, b)960x540, and c)1920x1080 videos

Table 3 Comparison of execution times of TLD and H-TLD for different components (in milliseconds)

		Total recall computation	Image blurring	Integral image computation	LK Tracker	Others	Total
480x270	TLD	5.93	1.69	0.27	1.10	1.17	10.16
	H-TLD	0.70	0.09	0.02	1.73	1.17	3.71
	Speed-up	8.45	18.85	13.55	0.64	1.00	**2.74**
960x540	TLD	20.40	9.60	1.10	4.28	2.14	37.52
	H-TLD	1.90	0.13	0.02	2.00	2.14	6.19
	Speed-up	10.74	73.85	47.83	2.14	1.00	**6.06**
1920x1080	TLD	62.50	33.60	4.56	17.52	4.28	122.46
	H-TLD	4.31	0.33	0.03	3.00	4.28	11.95
	Speed-up	14.50	101.82	152.00	5.84	1.00	**10.25**

resolution videos is desired for many applications to allow tracking of objects continuously within a wider field of view. In video capture devices, using a wide-angle lens allows monitoring a wider field-of-view uninterruptedly. On the other hand, if the capture resolution is not increased with the same proportion, the observed objects in the captured image occupy fewer pixels, making it difficult to track objects. As a result, it can be argued that the speed-up rates at higher resolutions are more relevant for future applications in which the videos are captured at higher resolutions. All the components exhibit higher speed-up rates with the increasing resolution, reaching 14.5, 101.82, and 152.00 times speed-up at 1920x1080 for total recall computation, image blurring and integral image computation respectively. On the other hand, LK tracker speed-up for this resolution is more modest at 5.84 times and at 480x270 the speed-up is less than 1, countering the positive contributions of other components and limiting the overall speed-up at lower resolutions.

Figure 11 shows sample tracking results where a water bottle is tracked. Blue bounding box indicates a highly confident detection; whereas yellow bounding box indicates moderate confidence. The bottle is initially detected and tracked. Then it is partially occluded by the seat, but the tracking is unaffected. It is then completely occluded by the seat. When it reappears, it is redetected by the detector after a few frames.

5.2 Comparison with GPU based Implementations

In this subsection, we compare our results with those of the two other GPU based implementations in the literature [4] [23]. In [4], results of the GPU implementation of TLD algorithm are presented. The baseline TLD algorithm on Xeon E31275v3 is compared against their GPU implementation on Tesla K40 GPU at 1920x1080 resolution in terms of Frames Per Second (FPS). The experiment setup which consists of Intel i7 4770K CPU and Tesla K40 GPU is similar to the one in this paper. The GPUs in both works are the same while CPU Mark scores of the CPUs are fairly close - 9714 and 10208 for Xeon and i7 respectively [22]. This allows making direct comparisons with our results. In [4], speed-up rates of approximately 5.3 to 5.6 are reported for different 1920x1080 sequences. According to these results, H-TLD achieves approximately 1.8 times speed-up on a similar platform compared to the GPU implementation in [4] at 1920x1080 resolution. While the detailed design information is not provided, the implementation is told to be a port of the CPU version to GPU and unlike H-TLD, it does not involve design and optimization with regards to the particulars of such a platform.

In [23], parts of TLD are ported to the GPU and the results are provided in terms of Frames Per Second (FPS) on GTX 550TI GPU for 320x240 and 640x480 resolution videos. In order to compare our results with this work, we ran the source code released by the authors on the same video sequences and on the same hardware platform that we benchmarked our work. The results showed that the algorithm runs at 37.62, 12.31 and 7.35 fps for 480x270, 960x540 and 1920x1080 resolutions respectively. According to these results, the proposed H-TLD implementation exhibits speed-up rates of 7.16, 13.13 and 11.39 respectively compared to the GPU implementation in [23].

6 Conclusions

In this paper, we described the design and implementation of an optimized Heterogeneous CPU-GPU TLD (H-TLD) solution using OpenMP and CUDA. The H-TLD design aims high utilization of both CPU and GPU while minimizing the data transfers. Load balancing on the GPU is achieved by the proposed grouping of the data to have a high utilization. The results show that by leveraging the advantages of the heterogeneous architecture, speed up of 10.25x could be achieved for 1920x1080 resolution. The source code of the framework is provided as a publicly available open source library. Our performance analysis of the final design highlights the data transfers between the CPU and GPU memory space as the main bottleneck. While we aimed to reduce such transfers, some are inevitable due to the heterogeneous architecture. Higher memory bandwidth between CPU and GPU and unified CPU-GPU memory spaces in future architectures are expected to have a significant positive impact on the performance without requiring any changes in design.

References

1. CUB library. Available: `http://nvlabs.github.io/cub/` (2015)
2. NPP library. Available: `https://developer.nvidia.com/NPP` (2015)
3. Thrust library. Available: `https://developer.nvidia.com/Thrust` (2015)
4. Atala, J., Bederián, C., Bordese, A., Ingaramo, G., Gaich, F., Medina, J., Rosetti, M., Sánchez, J., Tealdi, M., Wolovick, N.: Real-time FullHD Tracking-Learning-Detection on a 2-SMX GPU. GPU Technology Conference (GTC) *Poster* (2015)
5. Bouguet, J.Y.: Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. Intel Corporation **5**, 1–10 (2001)
6. Bradski, G.: OpenCV Library. Dr. Dobb's Journal of Software Tools (2008)
7. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
8. Concha, D., Cabido, R., Pantrigo, J., Montemayor, A.: 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 pp. 1–19 (2014). DOI 10.1007/s11554-014-0483-1
9. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE **5**(1), 46–55 (1998)

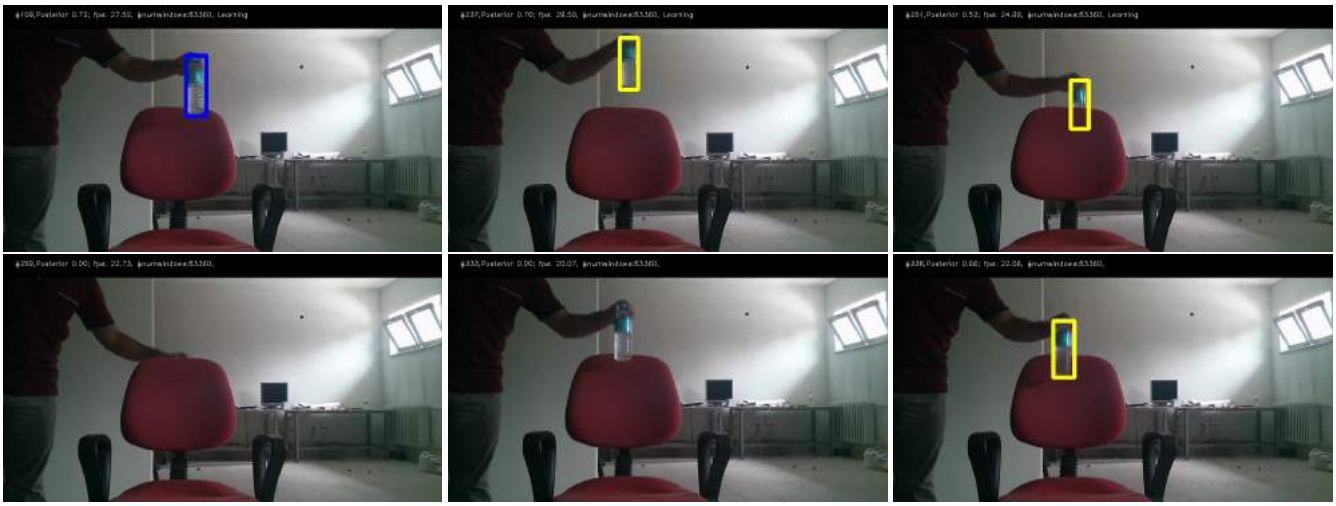

Fig. 11 Sample captured images from a sequence where a water bottle is tracked.

10. Guler, P., Emeksiz, D., Temizel, A., Teke, M., Temizel, T.T.: Real-time multi-camera video analytics system on GPU. Journal of Real-Time Image Processing pp. 1–16 (2013)
11. Ishii, I., Ichida, T., Gu, Q., Takaki, T.: 500-fps face tracking system. Journal of Real-Time Image Processing **8**(4), 379–388 (2013)
12. Kalal, Z., Matas, J., Mikolajczyk, K.: Pn learning: Bootstrapping binary classifiers by structural constraints. In: Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on, pp. 49–56. IEEE (2010)
13. Kalal, Z., Mikolajczyk, K., Matas, J.: Forward-backward error: Automatic detection of tracking failures. In: Pattern Recognition (ICPR), International Conference on, pp. 2756–2759. IEEE (2010)
14. Kalal, Z., Mikolajczyk, K., Matas, J.: Tracking-learning-detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on **34**(7), 1409–1422 (2012)
15. Kumar, P., Singhal, A., Mehta, S., Mittal, A.: Real-time moving object detection algorithm on high-resolution videos using GPUs. Journal of Real-Time Image Processing pp. 1–17 (2013). DOI 10.1007/s11554-012-0309-y
16. Lewis, J.: Fast normalized cross-correlation. In: Vision interface, vol. 10, pp. 120–123 (1995)
17. Liu, K.Y., Li, Y.H., Li, S., Tang, L., Wang, L.: A new parallel particle filter face tracking method based on heterogeneous system. Journal of Real-Time Image Processing **7**(3), 153–163 (2012)
18. Mahmoudi, S., Kierzynka, M., Manneback, P., Kurowski, K.: Real-time motion tracking using optical flow on multiple GPUs. Bulletin of the Polish Academy of Sciences: Technical Sciences **62**(1), 139–150 (2014)
19. Marzat, J., Dumortier, Y., Ducrot, A.: Real-time dense and accurate parallel optical flow using cuda. In: International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) (2009)
20. Mizukami, Y., Tadamura, K.: Optical flow computation on compute unified device architecture. In: Image Analysis and Processing, International Conference on, pp. 179–184. IEEE (2007)
21. Nebehay, G.: Robust object tracking based on Tracking-Learning-detection. Master's thesis, Faculty of Informatics, TU Vienna (2012)
22. PassMark Software: CPU Benchmarks. Available: `http://www.cpubenchmark.net/` (2015)
23. Ping, Z., Yongqi, S., Yali, W., Rui, Z.: A parallel implementation of TLD algorithm using CUDA. In: Wireless, Mobile and Multimedia Networks (ICWMMN 2013), 5th IET International Conference on, pp. 220–224 (2013)
24. Sinha, S.N., Frahm, J.M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Machine Vision and Applications **22**(1), 207–217 (2011)