

HYBRID CPU-GPU IMPLEMENTATION OF
TRACKING-LEARNING-DETECTION ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

İLKER GÜRCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2014

**HYBRID CPU-GPU IMPLEMENTATION OF
TRACKING-LEARNING-DETECTION ALGORITHM**

Submitted by **İlker GÜRCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Director, **Informatics Institute**

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department, **Information Systems**

Assoc. Prof. Dr. Alptekin Temizel
Supervisor, **Work Based Learning Studies, METU**

Examining Committee Members:

Assoc. Prof. Aysu Betin Can
IS, METU

Assoc. Prof. Dr. Alptekin Temizel
WBL, METU

Assist. Prof. Erhan Eren
IS, METU

Assist. Prof. Sinan Kalkan
CENG, METU

Assoc. Prof. Altan Koçyiğit
IS, METU

Date: 16.09.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: İlker GÜRCAN

Signature : _____

ABSTRACT

Hybrid CPU-GPU Implementation of Tracking-Learning-Detection Algorithm

GÜRCAN, İlker

M.S., Department of Information Systems

Supervisor: Assoc. Prof. Alptekin TEMİZEL

Tracking objects in a video stream is an important problem in robot learning (learning an object's visual features from different perspectives as it moves, rotates, scales, and is subjected to some morphological changes such as erosion), defense, public security and many other various domains. In this thesis, we focus on a recently proposed tracking framework called TLD (Tracking-Learning-Detection). While having promising tracking results, the algorithm has high computational cost. The computational cost of the algorithm prevents running it at higher resolutions as well as running multiple instances of the algorithm to track multiple objects on CPU. In this thesis, we analyzed this framework with an aim to optimize it computationally on a CPU-GPU hybrid setting and developed a solution via using GP-GPU (General Purpose GPU) programming using Open-MP and CUDA. Our results show that 2.82 times speed-up at 480x270 resolution can be achieved. The speed-ups are higher at higher resolutions as expected in a massively parallel GPU platform, increasing to 10.25 times speed-up at 1920x1080 resolution. The resulting performance of the algorithm enables the algorithm to track multiple objects at higher frame rates in real-time and improving detection and tracking quality by allowing selection of configuration parameters requiring higher processing power.

Keywords: Computer Vision, Long-term Tracking, GP-GPU Programming, Multiprocessing, Real-Time

ÖZ

TAKİP ETME-ÖĞRENME-TESPİT ALGORİTMASININ HİBRİD CPU-GPU GERÇEKLEMESİ

GÜRCAN, İlker

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Danışmanı: Assoc. Prof. Alptekin TEMİZEL

Bir video görüntüsünde var olan nesnelerin takibi; robotların öğrenme mekanizması (bir nesnenin görsel özelliklerinin robot tarafından; hareket etme, şekilsel değişikliğe uğrama, ölçek değişimi, ve/veya dönme gibi zaman içinde nesnede meydana gelen değişikliklerin anbean takip edilerek öğrenilmesi), savunma, kamu güvenliği, ve bunlara benzer diğer birçok alanda önemli bir yere sahiptir. Bu tezde yakın bir zamanda önerilmiş olan, TLD (Takip Etme-Öğrenme-Tespit) isimli bir nesne takip algoritmasına odaklandık. TLD başarılı sonuçlar üretmesine karşın, çok yüksek hesaplama gücüne ihtiyaç duyan bir yöntemdir. Bu yüksek hesaplama gücüne duyulan ihtiyaç; CPU üzerinde yüksek çözünürlüklerdeki video’larda tek bir nesnenin takibini ya da bir video’da birden fazla nesnenin takip edilebilmesini engellemektedir. Biz de bu özgün takip algoritmasının hızını arttırmaya yönelik bir dizi çalışmalar yaptık ve GP-GPU (GPU üzerinde genel amaçlı programlama) ile Open-MP ve CUDA teknolojilerini kullanarak hibrid bir çözüm gerçekleştirdik. Sonuçlarımız gösteriyor ki 480x270 çözünürlükte 2.82 kat kadar hızlanma sağlanmaktadır. Çok büyük bir ölçekte paralel bir sistemden beklendiği üzere hızlanma daha yüksek çözünürlüklerde daha fazla olmaktadır ve 1920x1080 çözünürlüğünde 10.25 kata kadar yükselmektedir. Bu hızlanma, yüksek çözünürlüklerde nesne takibine ve çoklu nesne takibine imkan sağlamakta ve takip

algoritmasının kalitesini arttıracak şekilde kurulum deęiřkenlerinin belirlenmesine olanak saęlamaktadır.

Anahtar Kelimeler: Bilgisayar Görüntüsü, Uzun Süreli Nesne Takibi, Genel Amaçlı GPU Programlama, Çoklu-İřleme, Gerçek Zamanlı

TABLE OF CONTENTS

ABSTRACT	v
ÖZ.....	vi
1 INTRODUCTION.....	1
1.1 Motivation	3
2 LITERATURE REVIEW	7
2.1 Object Tracking Algorithms	8
2.2 Machine Learning.....	11
2.3 Tracking-Learning-Detection (TLD).....	13
2.4 Tracking Algorithms Implemented on GPU or Hybrid Platforms	15
3 OPEN TLD ALGORITHM AND ANALYSIS OF COMPUTATIONAL BOTTLENECKS	19
4 IMPLEMENTATION DETAILS.....	25
4.1 TLD Object & H-TLD Modules.....	25
4.2 Implementation of H-TLD Detection Module	30
4.2.1 PV-Computation (GPU Only).....	34
4.2.2 BB Stream Compaction (GPU Only).....	46
4.2.3 RFI Calculation (GPU Only)	48
4.2.4 Confidence Calculation (CPU Only)	51

4.3 Implementation of H-TLD Tracking Module	54
5 RESULTS.....	59
5.1 Detection	62
5.1.1 Integral Image (II) Computation on GPU	62
5.1.2 Image Blurring (Open-CV Used)	63
5.1.3 Total Recall Computation on CPU and GPU Collaboratively	64
5.2 Tracking Module	67
5.2.1 Optical Flow	67
5.3 Effect of Different Display Resolutions on the Performance	69
5.4 NVIDIA's Visual Profiler & Overlapping Data Transfer and Kernel Executions	72
5.5 Discussions	75
6 CONCLUSION AND FUTURE WORK.....	81
7 REFERENCES	85

LIST OF FIGURES

Figure 1-1 CPU-GPU Comparison	2
Figure 1-2 CUDA Architecture	4
Figure 2-1 TLD Framework [2]	13
Figure 2-2 Overview of Parallel Algorithm in [13]	15
Figure 2-3 GPU Timing for Each Separate Operation [14]	17
Figure 4-1 Communication with H-TLD via TLDObjets	26
Figure 4-2 Class Diagram for H-TLD Objects and Structures.....	27
Figure 4-3 A subset of a Rectangular Region on an II.....	31
Figure 4-4 Pseudo-Code for Total Recall Computation.....	33
Figure 4-5 Terms Used in PV-Computation	34
Figure 4-6 Pseudo-Code for PV-Computation on GPU	38
Figure 4-7 Mapping Multi-Threaded Concepts to PV-Computation Ones.....	39
Figure 4-8 BB Organization on RAM in Serial Code Context	41
Figure 4-9 Access Pattern for BB Offs without Conversion on Device Memory.....	42
Figure 4-10 Conversion from AoS to SoA.....	43
Figure 4-11 Accessing BB Offsets on GPU's Global Memory after Conversion	43
Figure 4-12 Relationship between Scan Line Pairs and Scale Levels	44
Figure 4-13 Processing of BBs without Loading Balancing.....	45
Figure 4-14 BB Ordering for Load Balancing	46
Figure 4-15 BB Stream after Running PV-Computation	47
Figure 4-16 Left-Shift Amounts after Running "Prefix-Sum".....	47
Figure 4-17 Feature of a Random Forest's Tree	49
Figure 4-18 Pseudo-Code for RFI Calculation on GPU	51
Figure 4-19 Confidence Values Written to Output Array.....	53
Figure 5-1 A Sample Screen-shot Captured from the Test Video	62

Figure 5-2 Average Time per Call for II-Computation.....	63
Figure 5-3 Average Time per Call for Image Blurring	64
Figure 5-4 Average Time per Call for Total Recall Computation	65
Figure 5-5 Stacked View for Elapsed Time of TRC.....	66
Figure 5-6 Data Transfer and Execution Time for RFI Calculation	67
Figure 5-7 LK-Tracker GPU vs. CPU Implementations.....	68
Figure 5-8 Effect of Resolution Change on TRC.....	70
Figure 5-9 Total Gain of H-TLD	71
Figure 5-10 Timeline View of Visual Profiler	73
Figure 5-11 Visual Profiler Timeline View for TRC.....	74
Figure 5-12 Timeline That Shows Overlapping Behavior of Multiple Kernel Invocations	74
Figure 5-13 Tricky Code for a Compiler	76
Figure 6-1 Change in Computation Power of CPU vs. GPU over Years	82

LIST OF EQUATIONS

(2-1).....	8
(2-2).....	8
(2-3).....	9
(2-4).....	9
(2-5).....	10
(2-6).....	11
(2-7).....	16
(4-1).....	30
(4-2).....	31
(4-3).....	36
(4-4).....	37
(4-5).....	37
(4-6).....	44
(4-7).....	48
(4-8).....	50
(4-9).....	50
(4-10).....	54
(4-11).....	56
(5-1).....	63
(5-2).....	64
(5-3).....	65
(5-4).....	66
(5-5).....	68

(5-6).....	68
(5-7).....	68
(5-8).....	70
(5-9).....	70
(5-10)	70
(5-11)	78
(6-1).....	81

LIST OF TABLES

Table 3-1 Performance Statistics for Each Repetitive Task of Open TLD	21
Table 4-1 HETLDError Class' Public Fields and Their Definitions	29
Table 4-2 Building Pyramids for LK Tracker	56
Table 5-1 System Specs of the Test Platform	60
Table 5-2 Method Used to Measure Time on Host Side	61
Table 5-3 Method Used to Measure Time on Device Side	61
Table 5-4 Elapsed Time for Each Stage of TRC	65
Table 5-5 Wrong Call Order for GPU to Overlap Kernel Invocations	75
Table 5-6 Correct Call Order for GPU to Overlap Kernel Invocations	75
Table 5-7 Object Detection Sequence against the Medium Size Video	78

ABBREVIATIONS

- Open-TLD: Open Tracking Learning Detection
- CUDA: Computational Unified Device Architecture
- SMP: Symmetric Multi-Processing
- Open-MP: Open Multi-Processing
- SDK: Software Development Kit
- API: Application Programming Interface
- CPU: Central Processing Unit
- GPU: Graphical Processing Unit
- BB: Bounding Box
- PV-Computation: Patch Variance Computation
- II: Integral Image
- TRC: Total Recall Computation
- RFI: Random Forest Index
- NPP: NVIDIA Performance Primitives Library
- NPPI: NVIDIA Performance Primitives Library for Image Processing
- AoS: Array of Structures
- SoA: Structure of Arrays
- WDDM: Windows Display Driver Model

-TCC: Tesla Compute Cluster

-GP-GPU Programming: General Purpose GPU Programming

-MFT: Median Flow Tracker

CHAPTER 1

INTRODUCTION

This thesis is based on “Open TLD”, which is an algorithm for long-term tracking of objects throughout a series of video frames. The aim of this thesis is to increase the processing speed of this algorithm by parallelizing it to run on hybrid CPU-GPU environment.

Open TLD algorithm has high computational complexity. For instance, its detection module runs an extensive search (a kind of a sliding window method) for detecting location of the object within each frame, which is a highly compute-bound operation (due to running a single operation on thousands or even hundreds of thousands bounding boxes depending on the video resolution and number of scale levels object’s patches have). While modern hardware is capable of achieving such goals, real-time tracking is only possible for low-resolution images and for a single object due to its high computational cost. Parallelization of the algorithm and increasing its processing speed is expected to allow running multiple instances of the algorithm to track multiple objects, which opens up possibilities for other applications. Optimization of the algorithm to run at higher resolutions is also desirable. Open TLD has also some configuration parameters to tune the accuracy of its tracking/detection operations. Tuning those parameters for higher tracking accuracy; eventually leads to performance degradation. For example, for the sake of being invariant to scale and rotational changes; the framework creates a number of patches of the object under observation, differing in scale and rotation angle. That number is specified at configuration time and should be kept small, if anyone wants to run its application at higher frame rates. This raises an issue of trade between object tracking quality and processing speed. By incorporation of GPU technology in particular; now speed of video/image processing applications that even run on a

single machine rather than a cluster of machines, might be accelerated significantly. A faster Open TLD implementation on the GPU is important in the sense that:

- Increase the resolutions for which the algorithm can run in real-time,
- Allow running multiple instances of the algorithm to enable real-time multiple object tracking,
- Allow running the algorithm at higher accuracy by making it possible to run the algorithm with parameters resulting in higher accuracy while increasing the computational cost.

The aim of this thesis is to design and implement an efficient hybrid CPU-GPU processing framework which makes use of the specific properties of CPU and GPU (i.e. using CPUs for operations for which CPU has advantages and vice versa).

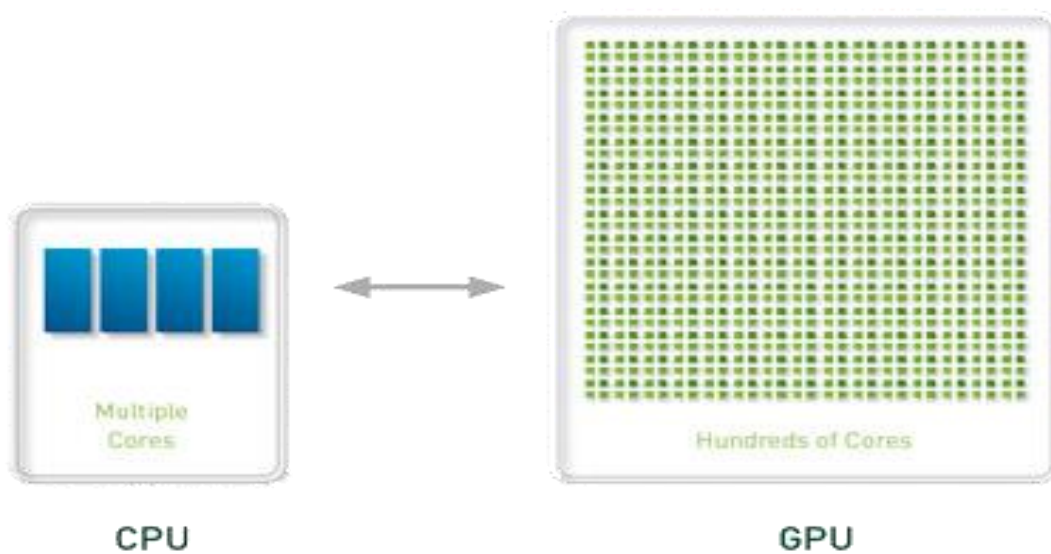


Figure 1-1 CPU-GPU Comparison

Most modern computer platforms have both processing units, CPUs and GPUs. Thus, in this thesis, one of the aims is to exploit both processing units. Each has its own advantages and disadvantages by their design. For instance, CPUs are good at executing branched-instructions and have higher clock frequencies; while GPUs are not optimized for divergent operations requiring branching and typically have lower clock frequencies. On the other hand, GPUs have hundreds of cores devoted to data

processing rather than data caching and control flow (branched-instructions); therefore they may run a single instruction on more data simultaneously in comparison with the CPU (See Figure 1-1 for a comparison).

The aim of the thesis is not to modify or improve Open TLD in algorithmic sense; but to decrease its execution time. By improving its execution time, its detection and tracking quality will be increased implicitly.

1.1 Motivation

Especially in the last decade, there have been significant improvements in terms of GPUs and they have been started to be used in general purpose programming. At the beginning, GPUs had always been used to process graphical data in order to display geometrical shapes on the screen along with a proper perspective and depth. However, while initially driven by gaming and graphics applications there has been an interest to leverage this high processing power of GPUs in other application areas. As a result, many software development platforms have emerged to ease programming of GPUs and make their massive processing power available for general purpose applications. Among those, the most prominent ones are Open-CL, Open-ACC, C++ AMP and, CUDA.

CUDA is a combination of software tools (GPU accelerated libraries such as CUBLAS, a driver API, and a run-time API) that enables developers to develop and run applications on the GPUs leveraging their massively parallel architectures. See Figure 1-2 for an illustration of software layers of CUDA in hierarchical order.

Incorporating GPUs into such application programming in order to have them cooperate with CPUs is called (a kind of) heterogeneous computing or programming. Since data and instructions must be transferred to GPU's memory; they mostly use PCI-Express bus to transmit data back and forth. However that is the most critical bottleneck of heterogeneous computing (GPU architects have been trying to minimize this overhead and it is still a hot topic for many of these GPU architectures). In order to eliminate this cost, application developers are encouraged

to execute as many instructions as possible on the data residing on GPU memory currently so that whenever another portion of the data is to be moved to GPU memory (this operation is completely independent of execution and carried out by loading units), time required to transmit it could be hidden out by aforementioned execution time. There are many other restrictions and bottlenecks specified in manuals of corresponding technologies.

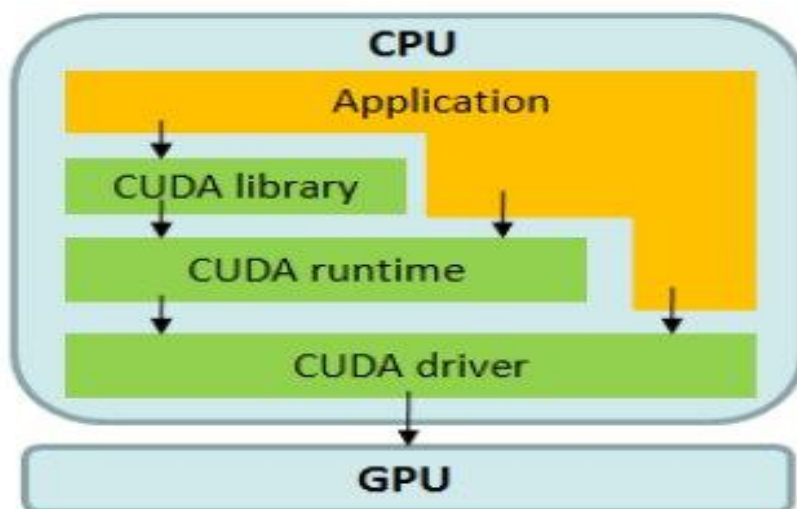


Figure 1-2 CUDA Architecture

CUDA is highly popular for several reasons, some of which are listed below:

- Unlike technologies such as Open-CL, CUDA is developed for a specific hardware from NVIDIA eliminating the overhead caused by interoperability issues; hence resulting in a better performance.
- CUDA is supported by many teaching centers in universities, companies and even by individuals. Thus, there are many 3rd party libraries provided to application developers. Besides, its evolution is faster than other heterogeneous parallel computing tools.
- Application developers are completely isolated from graphics APIs and they can develop their programs without considering the pipeline.

- CUDA allows many C++ features like classes, templates, etc. while many others are based on C99.

CUDA also has some very powerful libraries that could be incorporated into video/image processing applications; such as [1]. In addition to those libraries, some open source communities such as Open-CV has recently started to provide application developers with GPU-accelerated versions of its already built-in methods. Due to these reasons, CUDA is chosen for the implementation phase in this thesis.

Various video/image processing algorithms suits well to SIMD architectures provided that they are based on running a single instruction on individual pixels. GPU architectures are good candidates for such purposes; because of having small fast memories and registers attached to cores; but having thousands of those cores dedicated to data processing. Although GPU cores are not specialized for executing branching instructions; many video/image processing algorithms' computationally complex parts do not contain flow control structures (even if they do, many of them do not cause branch divergence among threads). That also leads developers of such applications to run such computationally intensive fractions of the algorithm on GPU.

Open TLD's tracking and detection modules run independent of each other; i.e. they do not need to share any common data until the integrator receives results from both in order to estimate the location of the object. An important advantage of heterogeneous computing is to keep CPU and GPU busy as much as possible and hence utilize the resources efficiently. A detailed study of the algorithm in [2] reveals that it could be run on CPU and GPU independently and might be overlapped. Moreover; using CUDA streams/events [3] enables developers to overlap data transfer between host/device; and some very critical parts of Open TLD fits well to this pattern (its detector module in particular). Each "TLDOject" (defined in Section 4) is independent of each other in the sense that they may run different methods from same or different modules at the same time without any interference.

For instance, while TLD Object “A” is running tracking module’s method “B”; in the meantime object “C” might be running detection module’s method “D”. Finally, multi-core CPUs may be exploited in order to perform tasks that are not desired to be executed on GPUs for particular reasons (such as parts of the algorithm which require moving data back and forth between CPU and GPU very frequently).

CHAPTER 2

LITERATURE REVIEW

In this chapter, an overview of object tracking algorithms and the related work focusing on acceleration of object tracking algorithms as well as the rationale behind selection of the TLD algorithm as the main focus in this thesis are given.

Tracking a particular object in a sequence of continuous video frames is an important concept in computer vision domain due to the wide range of applications. Object tracking systems are expected to track an object until it moves out of the camera's field of view. It is a challenging task due to many reasons such as changes in illumination, noise, rotation or scaling, and object appearance. Besides, some systems aim tracking on moving cameras; rather than stationary ones. This adds additional complexity to object tracking algorithms. There are also some real time limits which affect the continuous tracking of the object, therefore they demand high processing power in order to track any object uninterruptedly.

In this chapter, two different types of past works are covered:

- Object Tracking Algorithms,
- Accelerating Object Tracking Algorithms via Heterogeneous Computing.

GP-GPU programming is relatively new topic compared to object tracking. Thus, many different approaches to track an object in 2D and 3D scenes have been proposed; while there is relatively less number of GPU-based or hybrid approaches.

In [4], object tracking is grouped into 3 different categories: recursive tracking, tracking-by-detection, and adaptive tracking-by-detection. According to [4], a 4th category has emerged recently, which is called tracking-learning-detection. Our work

is based on a tracking algorithm that could be classified in this new group of tracking algorithms.

As for increasing effectiveness and speed of object tracking algorithms via hybrid solutions, almost all of these solutions focus on analyzing and improving optical flow part of tracking algorithms; rather than focusing on the whole problem.

2.1 Object Tracking Algorithms

As just mentioned before there are 4 types of tracking algorithms. First group is recursive tracking. This type of algorithms estimates the approximate location state x_t of the object on the current frame in accordance with its previous location x_{t-1} state by applying a certain transformation to this previous location. CONDENSATION [5] is one such popular recursive tracking algorithm. Unlike “Kalman Filtering” method which outputs a single estimate of position and covariance, it estimates entire probability distribution of likely object positions; increasing its robustness against distracting clutter. Note that, it assumes that all observations (frames) in a temporal image sequence are mutually independent of each other; therefore previous observations has no effect on determining next state x_t , and it is called Markov process as shown in Equation(2-1).

$$\rho_t(x_t) \equiv \rho(x_t|z_t) \quad (2-1)$$

Where ρ_t is the conditional probability density function of state x_t , given that z_t has already been observed at time t .

$$\rho(x_t|z_t) = k_t \rho(z_t|x_t) \rho(x_t|z_{t-1}) \quad (2-2)$$

Where,

$$\rho(x_t|z_{t-1}) = \int_{x_{t-1}} \rho(x_t|x_{t-1}) \rho(x_{t-1}|z_{t-1})$$

Where k_t is normalization constant, $p(z_t | x_t)$ is the probability density function that weighting new samples in state x_t , and $p(x_t | x_{t-1})$ is the conditional probability density function which proves the next step is conditioned directly only on the previous state.

Another popular recursive technique is “mean-shift tracker”. In this method, tracker tries to find the mean-shift vector by maximizing “Bhattacharyya coefficient” [6] shown in Equation (2-3). First it finds the weighted pdf (probability density function) of object model (q) centered at the location y_0 in the previous frame and the pdf of the candidate centered at the location y on the next frame. Then it calculates the similarity between these two PDFs.

$$\rho(y) = \sum_{u=1}^m \sqrt{\widehat{p}_u(y) \cdot q_u} \quad (2-3)$$

Where $\rho(y)$ is Bhattacharyya coefficient, $\widehat{p}_u(y)$ is the m -bins color histogram vector of the candidate on the next frame centered at the location y , and q_u is the m -bins color histogram vector of the target on the previous frame centered at the location y_0 .

After calculating the first Taylor expansion of this function around y_0 ; mean-shift vector (shown in Equation (2-4)), that will be added up to the previous location vector y_0 (in order to find the new center location of the object), could be obtained. This is shown in Equation (2-5). This will lead to error accumulation, if the previous locations are calculated wrong.

$$M_h(y_0) = \frac{\sum_{i=1}^n w_i(y_0) x_i}{\sum_{i=1}^n w_i(y_0)} - y_0 \quad (2-4)$$

Where M_h is the mean-shift vector, w_i is the weighted function which depends on p and q (probability distribution functions), x_i is the location inside the ROI (Region of

Interest) centered at the location y on the next frame, and y_0 is the center of the object on the previous frame.

$$\hat{y} = y_0 + M_h(y_0) \quad (2-5)$$

Where M_h is the mean-shift vector maximizing Bhattacharyya coefficient in Equation (2-3), and \hat{y} is the new center location of the object on the next frame.

However, in [7] it is mentioned that the recursive methods accumulate the error due to dependence on the previous states of the object as depicted in Equation (2-2) (where next state always depends on the immediately preceding state). If object leaves the scene, and comes back into the view; they produce a significant error that will propagate to the following frames eventually.

Second group of tracking algorithms is tracking-by-detection. However these trackers have detectors that are trained once initially, and are never updated again. As a consequence of this, they require to learn many different aspects of the object beforehand (offline learning). In [8], a bunch of different patches are produced by applying affine warping techniques; which are then used to train a classifier. Although this tracker is effective; it cannot learn unseen appearances of the object; because it is difficult to estimate all variances of the object's appearance in the beginning.

Third group is adaptive-tracking-by-detection. It was developed for updating the classifier online. In [9] tracking problem is treated as a binary classification in which there are two different classes, object and background. The method works on color video frames, and uses self-learning (discussed in the next subsection) in order to train its classifier throughout the temporal sequence of video frames. The features are defined as a weighted linear combination of R, G, and B channels of any pixel as shown in Equation (2-6).

$$F_i = \{\omega_1 R + \omega_2 G + \omega_3 B \mid \omega_i \in [-2, -1, 0, 1, 2]\} \quad (2-6)$$

Where F_i is equal to any feature.

This methodology creates two m-bins histograms for both object and background; then tries to maximize the likelihood ratio of each feature using the probability density function based on those histograms. Features with higher likelihood ratios are picked up and used to model both object and the background each time a new frame is processed. However, in [10] it argues that self-learning algorithm causes drift due to using its own inferences to update itself.

The last group is TLD (which the algorithm we use in this thesis is based on) and discussed in more details in subsection 2.3.

2.2 Machine Learning

Machine Learning is an important subfield of computer science in object tracking domain; because a spatial-temporal model is required in order to detect the object's presence throughout a video stream, in turn this model should shape itself in the course of time as unseen samples are captured from that stream. There are 3 types of machine learning techniques in general: unsupervised, supervised learning, and semi-supervised.

- **Unsupervised Learning**: This type of learning techniques tries to find a hidden structure in which each group of unlabeled data belongs to a particular class (label). It uses methods such as k-means, mixture models, hierarchical clustering, etc. in order to group data with respect to the features the input data have.
- **Supervised Learning**: It uses labeled training data to infer a function (model). In supervised learning, each training example is a pair of object (typically a vector) and a label that it is classified as. However, extra caution should be taken when training data are picked to create the model. For instance if the heterogeneity of training data is high; while some algorithms such as SVM

(Support Vector Machine) which requires the input features to be numerical and scaled to a certain range, are easier to apply; others like nearest neighbor methods are sensitive to such data. There are many other constraints to be considered in choosing training set. Main problem with supervised learning is that it could lose its generalization power in the course of time due to bias-variance trade-off. If bias is low, then it will fit each incoming data differently, hence will result in a high variance (over-fitting); and vice-versa. A system expert may label the input data from time to time and keep the balance between bias and variance via updating parameters of the model.

- **Semi-supervised Learning**: This learning technique stands in between unsupervised and supervised learning. It exploits both unlabeled and labeled input data for updating its model on its own. Initially, it is fed with an independently identically distributed (i.i.d.) training data like in supervised learning. Unlike supervised learning, there is no validation step to tune the parameters of the model. It combines unlabeled and labeled data to enhance the classification performance either by discarding the unlabeled data and doing supervised learning, or discarding the labels and doing unsupervised learning (hierarchical clustering, k-means, etc.).

Semi-supervised learning has made a significant progress recently. There has been implemented a great variety of learners included in this group. In [11], some algorithms were discussed in this context, such as expectation maximization (EM), self-learning, and co-training.

EM maximizes the likelihood (which is soft-clustering. Unlike k-means in where objects are classified as either inlier or outlier of a certain class; rather than assigning a probability to it as to how much likely it belongs to that class); hence if the posterior probability functions of two different classes are not separate on their low boundaries, EM will eventually lose its discrimination power. In self-learning, after classifier is trained, unlabeled data are evaluated by the classifier; and classifier picks up the unlabeled data with the highest confidence value (hard-clustering), and

eventually this kind of schema in which the classifier uses its own decisions to train itself, will lead decision boundaries to be pushed away from the unlabeled data [10]; as a result it will cause drift.

In learning algorithms, unimodality means that there is more than one different resource that contributes to the object model. On the other hand, co-training splits the vector that describes examples into two; then it trains two separate classifiers using those separate features. At the learning step, each classifier evaluates unlabeled data and enhances the training set of another. As a result, each classifier learns a different group of features resulting in two modalities (multimodality). However in object detection problem, samples (patches) are generally (for instance [9] is an exception to this) obtained from a single modality. Thus, in [11] it is claimed that co-training is not a good candidate for detecting objects within a temporal sequence of frames.

In [11] and [2], it was shown that P-N Learning, which is also a type of semi-supervised learning, is superior to other semi-supervised learning techniques in detecting objects under observation. This technique is discussed in the next sub section in detail.

2.3 Tracking-Learning-Detection (TLD)

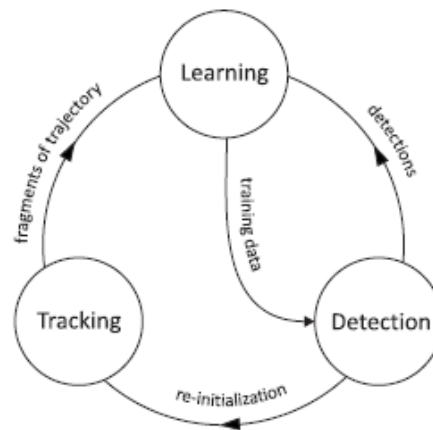


Figure 2-1 TLD Framework [2]

In Figure 2-1 TLD Framework, the whole framework of TLD algorithm is summarized. In this subsection, these components and their roles in this framework are explained.

The novel approach in TLD [2] is that its classifier does not use its own inference; but another component (P/N experts) helps it in updating the model. In addition to this improvement, it has some structural constraints. In [11] it is assumed that an object cannot be located in more than one position on a single frame (i.e. at a time) and that object should follow a particular trajectory.

In TLD, tracking is the process of predicting next locations of previous reliable points located in the BB (bounding box) of the object; and determining their reliability based on forward-backward and NCC (Normalized Cross Correlation) scores as well [12]. As a result of those, it finds the trajectory (which is used by P/N experts) as well.

Task of the detection module is to find the reliable BBs (based on classifier's scoring) where the object may exist. Then those BBs with high scores are sent to the P-N learner for evaluation (discussed in next paragraphs). Main purpose of the detector is to decide whether the object is still in the field of camera's view and if not, it tries to detect the object when it comes back into the scene. This is a significant improvement over many other tracking algorithms.

TLD uses random forest to describe its model and P-N learning for updating that model. As is the case with any semi-supervised learning, the classifier is initially trained with some labeled data. Then the classifier evaluates unlabeled data. Finally, P-N learning decides whether classifier's decisions on unlabeled data are correct or not by following steps:

- P expert checks each sample labeled as negative (background) against the trajectory. If that sample is nearby the trajectory, then it re-labels it as positive and adds it to the positive training set. This will eventually increase generalization power of the classifier.

- N expert checks each sample labeled as positive (object) against the trajectory. If that sample is far away from the trajectory, then it re-labels it as negative and adds it to the negative training set. This will eventually increase discrimination power of the classifier.

If any of the two steps (or both) above occurs, then the classifier is updated; otherwise it remains intact as it was before.

2.4 Tracking Algorithms Implemented on GPU or Hybrid Platforms

In this subsection some GPU implementations for tracking algorithms are mentioned. Many algorithms implemented on GPU or hybrid platforms focused on calculating optical flow as it is the common requirement for almost all modern tracking algorithms. On the other hand, by the time H-TLD has been implemented, no hybrid implementation for TLD was proposed.

In [13], CUDA was used to accelerate the speed of LK (Lucas-Kanade) optical flow on GPU. It splits the algorithm into some sequential steps as shown in Figure 2-2.

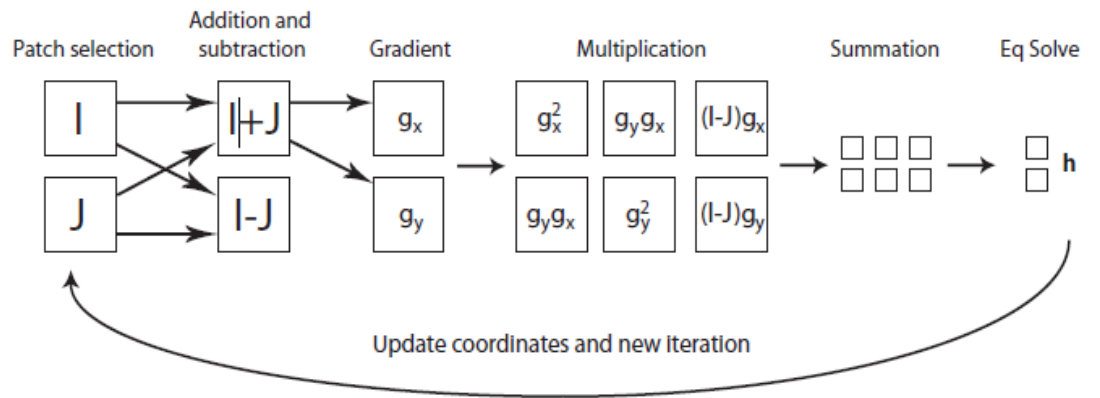


Figure 2-2 Overview of Parallel Algorithm in [13]

It uses Harris operator to figure out which patch on the frame is better for finding disparity (motion) vector h (this step is called patch selection step). Then it adds and

subtracts patches for further steps that will be run on GPU. It runs Sobel operator (kernel) on $(I + J)$ patch in order to obtain g_x and g_y gradient matrices in parallel. It fuses multiplication and summation step into one and run parallel reduction algorithm. After running equation solver to find vector h , it generates a new image J' by adding h with J ; then it runs all steps for several times again until vector h converges. However, since it did not consider pyramidal case, it fails to capture fast motion changes. Any speed-up was not given in the discussion part of [13].

In [14], it also includes pyramidal implementation from [15] in order to capture fast motion changes between two consecutive frame contrary to [13]. They also split into several steps; but they have more steps due to pyramidal implementation. They showed that most time consuming operation among them is “LK Optical Flow” as shown in Figure 2-3. They achieved a good performance in that thesis, and compared their results with Bruhn’s CPU implementation and Horn & Schunck (HSCuda)’s GPU implementation, based on a measurement that reflects the trade-off between execution time and accuracy (ETATO). This is shown in Equation (2-7). Their algorithm outperformed other two algorithms.

$$ETATO = exeTime \div pixelAngError \quad (2-7)$$

A video analytics system targeting video surveillance applications is described in [16]. 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 cameras are static.

Lastly, another parallel tracking algorithm has been published recently in 2014. It [17] has two big modules, one for feature detection and another for the optical flow. In this work, Bouguet’s corner detector [15] based on Harris detector [18] was used to detect features to be tracked and LK optical flow was exploited for tracking those detected features. They also used multiple GPUs (as CUDA allows developers to enumerate and use multiple GPUs concurrently) to increase their speed gain. They

compared their work with Open-CV's feature detector, tracker implementation; but they only outran Open-CV under some certain circumstances (when some parameters are tuned).

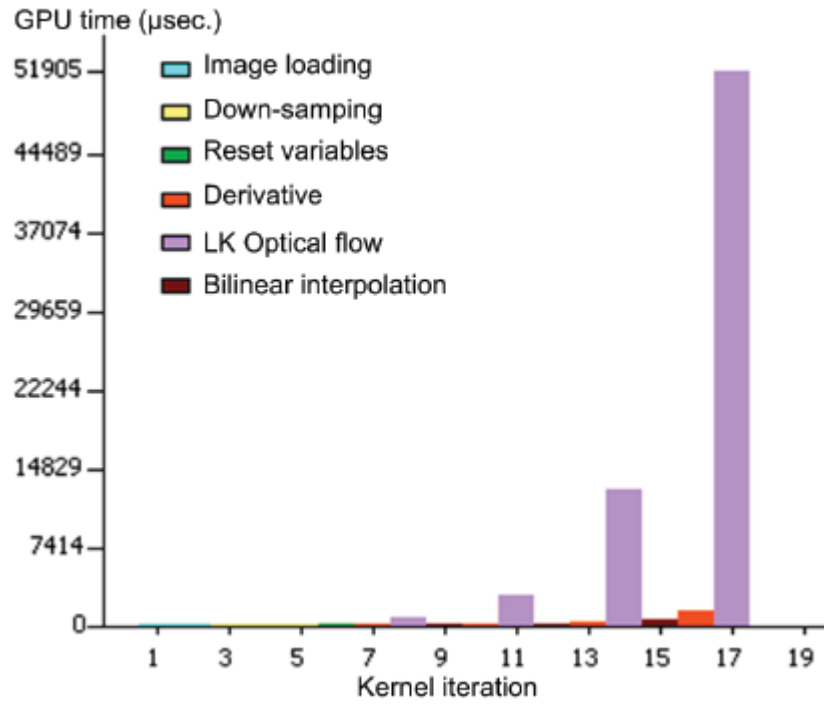


Figure 2-3 GPU Timing for Each Separate Operation [14]

Note that, all parallel algorithms mentioned in this subsection may perform faster in conjunction with the development of new GPU architectures in the future.

CHAPTER 3

OPEN TLD ALGORITHM AND ANALYSIS OF COMPUTATIONAL BOTTLENECKS

In this section, execution times of the TLD algorithm are analyzed to detect the performance bottlenecks. Execution times were calculated by the methodology explained in Chapter 5 (in Table 5-2 Method Used to Measure Time on Host Side). Test platform and its specs could be found in Table 5-1 System Specs of the Test Platform in Chapter 5 as well.

The algorithm is composed of the following components:

- **LK-Tracker (LKT)**: It is a part of the tracking module and it calculates frame-to-frame optical flow as was explained in median flow tracker [12]. It is run twice per frame in order to find reliable tracking points with more confidence values.
- **Total Recall Computation (TRC)**: It is run in the detection module. Its aim is to find a confidence value for each BB before the object- tracking phase begins. The confidence value is then used to check the existence of the tracked object in the current frame to find out whether it is still in the field of camera's view or not and detect its position.
- **II-Computation (IIC)**: It computes integral and squared integral images for each frame. These images are used within PV-Computation (patch variance computation) of each BB. PV-Computation is the first stage of Total Recall Computation, and called once before TRC begins.
- **Blurring the Image (BI)**: Like II-Computation, this is called before TRC is executed to smooth out the details in the image. This blurred image is used in

TRC for feature comparisons in order to find indices to random forest data structure.

- **Pattern Generation (PG)**: After tracking module estimates the location of the next bounding box pertaining to the object, P-Expert decides on whether the patch which that bounding box represents belongs to the object or not. If it belongs to the object, then it retrains ensemble classifier by generating positive patches (i.e. bounding box or scanning window). In the sense of detection module (i.e. ensemble classifier), training data are simply patterns generated via feature comparisons made on those generated positive patches.
- **Computing BB Overlap (CBO)**: It is used by learning component in 3 different situations to find the ratio between two different bounding boxes ($intersection / (area1 + area2 - intersection)$). After bounding boxes with high confidence values are determined by detection module, learning component cluster these detections and form a greater bounding box encompassing all bounding boxes owned by each cluster. Then it tries to find how much of the bounding box defined by tracking module overlaps with each of those bounding boxes. Second case is in which learning generates positive and negative patches. The third and the last one is when learning component harnesses detections (bounding boxes) with high confidence in order to adjust trajectory which is intrinsically defined by tracking component.
- **Random Forest Update (RFU)**: Ensemble Classifier classifies labeled example, and if this classification is proved to be incorrect by P-N Experts; posterior probabilities of feature comparisons for that labeled patch are updated (this operation was explained in [2] in more details and is out of this thesis' scope). This operation is not performed for every frame. It interferes with the execution whenever detection or tracking module makes a mistake.
- **Warping a Patch (WP)**: This operation is used to generate different version of a positive patch to make it scale and rotation invariant. After a patch is captured, it rotates, rescales that patch to create variants of it. It is called more

than once by learning component; but note that the learning component is not run for every time a frame is generated.

- **Norm Cross Correlation Computation (NCC)**: In the tracking module, after finding the next points via median flow tracker [12], similarity between sub rectangular regions, which are centered on original point on the previous frame and centered on next (predicted) point on the next frame, with a predefined size (explained in section 4.3), is calculated via Open-CV's [19] template matching procedure.

While there are other components of the algorithm, their execution times are insignificant compared to the ones above. Thus, they were not considered in the execution time analysis. In Table 3-1 Performance Statistics for Each Repetitive Task of Open TLD, execution time per call and the average number of calls per frame for each task mentioned above are given.

Component	Time per call (ms)			Time for whole sequence (ms)		
	480x270	960x540	1920x1080	480x270	960x540	1920x1080
<i>Tracking</i>						
LK Tracker	1.100	4.280	17.520	509	1982	8112
Norm Cross Correlation	0.620	0.630	0.770	287	292	357
<i>Learning</i>						
Pattern Generation	0.010	0.020	0.080	32	65	258
Random Forest Update	0.440	1.200	1.890	141	386	608
Patch Warping	0.080	0.230	1.270	326	938	5180
BB Overlap Computation	0.020	0.060	0.270	35	104	467
<i>Detection</i>						
Total Recall Computation	5.930	20.400	62.500	2752	9466	29000
Integral Image Computation	0.271	1.100	4.560	126	510	2116
Image Blurring	1.685	6.509	23.649	782	3021	10974

Table 3-1 Performance Statistics for Each Repetitive Task of Open TLD

As seen in Table 3-1, there are some tasks which are not called per frame (average number of calls lesser than 1) and which are called for couple of times per frame (average number of calls greater than 1). All tasks that seem they are called for couple of times per frame (PG, CBO, WP) are parts of learning module, and they are executed when the learning module interferes with the process (Thus, it could be inferred that they are called couple of times by the learning component when the learning component runs). On the other hand, only RFU's call statistic shows the exact average number of calls to the learning component (because RFU is called once per call to the learning component).

It can also be seen that there are some tasks which are called every time a frame is processed, and are costly. These are TRC, BI and LKT which are computationally expensive. Moreover, as the resolution of the video is increased, those three tasks dominate the execution time.

TRC is the most time consuming component. The reason is that it performs an exhaustive search for the object using a sliding window method. This task is performed as follow:

- A series of BBs based on object size are initially created and saved into a data structure.
- Then TRC treats each one of these BBs as a search window and carries out a series of costly sub tasks (PV (Patch Variance)-Computation, feature comparison, and CV (Confidence Value)-Calculation sequentially) on this single window (BB) to figure out which one of them is the best for the object to fit into (or it does not exist in the field of camera's view).

According to [2], there could be 50k BBs for a QVGA image (240x320) depending on the initial size of the object's BB. Therefore, this high number of BBs causes this high computational cost.

Although BI is not a heavy operation, yet it could become problematic as the resolution increases. In [2]’s case, it runs in a virtual machine (VM) and this brings another extra overhead for this task.

LKT’s sparse mode is used in [2]. Since Median Flow Tracker (MFT) mentioned in [12] calls LKT twice to calculate the forward-backward error for all points; it doubles the cost. Moreover, Open TLD [2] uses pyramidal LK tracker [15] in order to capture large motions. As a consequence of those two facts, tracking good features (points) of the object under observation becomes an expensive operation; even though it does not compute dense optical flow.

Finally, computational cost of IIC increases linearly with the number of pixels in video frame. Besides, the first sub task (PV-Computation) of TRC was implemented on GPU (discussed in Chapter 4); hence there are two options: either both (plain and squared) IIs had to be calculated on CPU and then moved to GPU; or only the current frame had to be moved to GPU and then IIC should have been performed on GPU as well. First option is more expensive than is the second one; because moving data back and forth between GPU and CPU is expensive compared to computational operations that could be done on GPU. Furthermore, IIC task suits well to the GPU architecture, hence we decided to run this component on the GPU.

CHAPTER 4

IMPLEMENTATION DETAILS

In this section, the design of the H-TLD library and the implementation details are provided.

4.1 TLD Object & H-TLD Modules

In this subsection, the general software design and main components of the Hybrid-TLD (H-TLD) algorithm are given. H-TLD has been designed for easy injection into any serial context. The context is not restricted only to C/C++ based native OS processes; but it may also be a JVM, .NET framework, or another container. Since the output of this code is a DLL or SO file, it can be used within various applications on different platforms.

A C++ object called `TLDOject` is the key to communicate with the library. Each `TLDOject` refers exactly to one object under observation in the current video stream, and is pure state holder which means that it does not have any behavioral implementation. `TLDOjects` are passed to H-TLD modules (tracking, detection) in order to exploit H-TLD capabilities. For this reason, there might be more than one `TLDOjects` while there is only one instance of each H-TLD module (i.e. one instance for tracking and one for detection) in a single process. Each `TLDOject` consists of 2 different module states: one state for tracking and one for detection. The learning component is not included in this object. Reasons for excluding the learning component from this object are explained in Chapter 3.

`TLDOject` is called “active” whenever a `TLDOject` binds to (calls) an API method of any module from H-TLD. Note that, when one of those objects is busy (i.e. active)

with any of those modules' methods (either tracking's or detection's); it excludes all other objects from using that method for thread safety. That is because many GPUkernels use file-scoped variables residing on GPU's constant memory (which stores data per TLDOBJECT) until module it belongs to is done with that active TLDOBJECT. Nevertheless, other unbound methods of that module and all other modules' methods can be used by any other TLDOBJECT simultaneously without worrying about any kind of race condition that may occur in between two active TLDOBJECTs. Figure 4-1 depicts this notion.

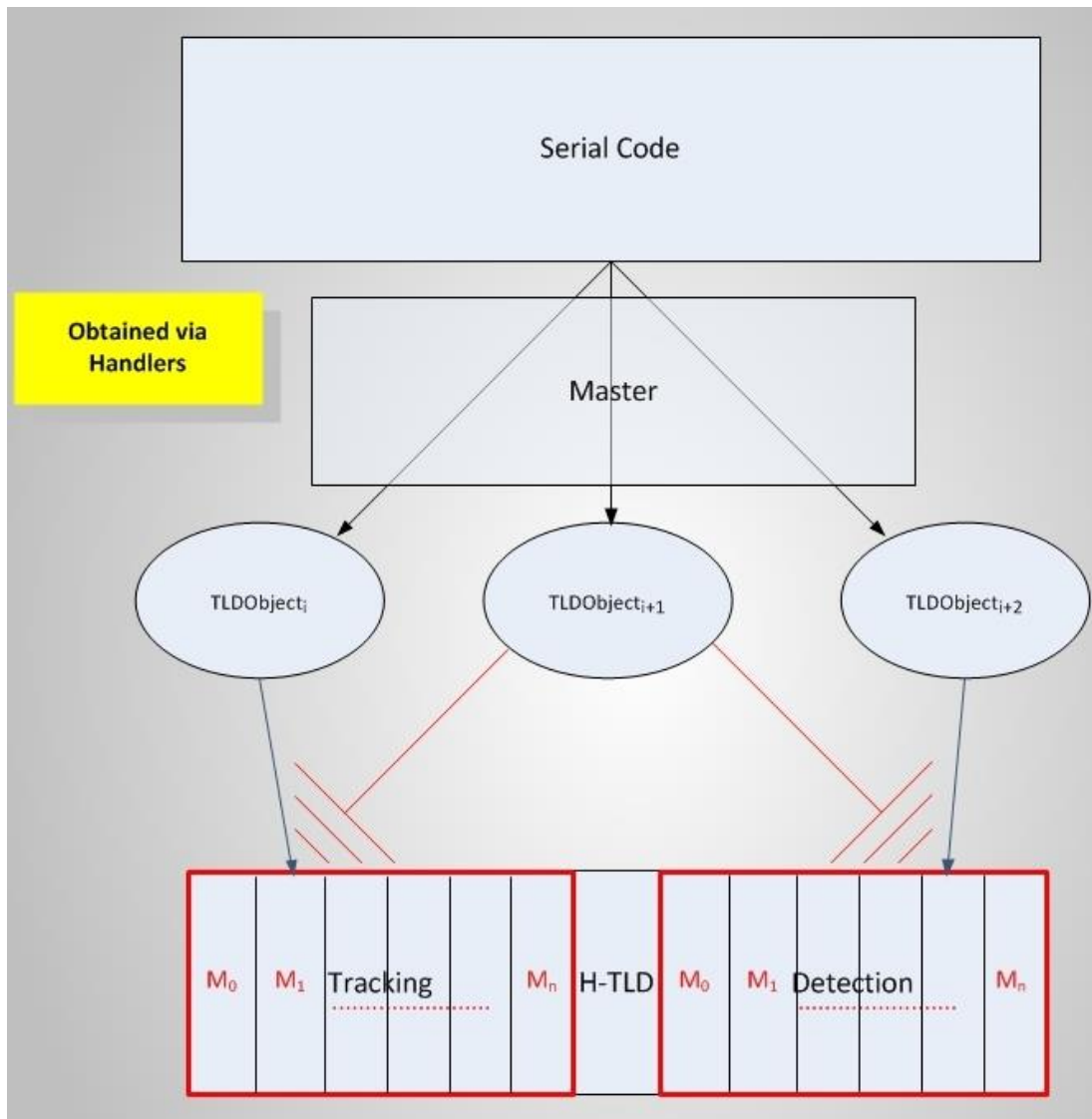


Figure 4-1 Communication with H-TLD via TLDOBJECTs

As can be seen in Figure 4-1, whenever one of the methods from any module is occupied by one of these TLDObjets, any other object's request, that demands the execution of the same method, should be suspended.

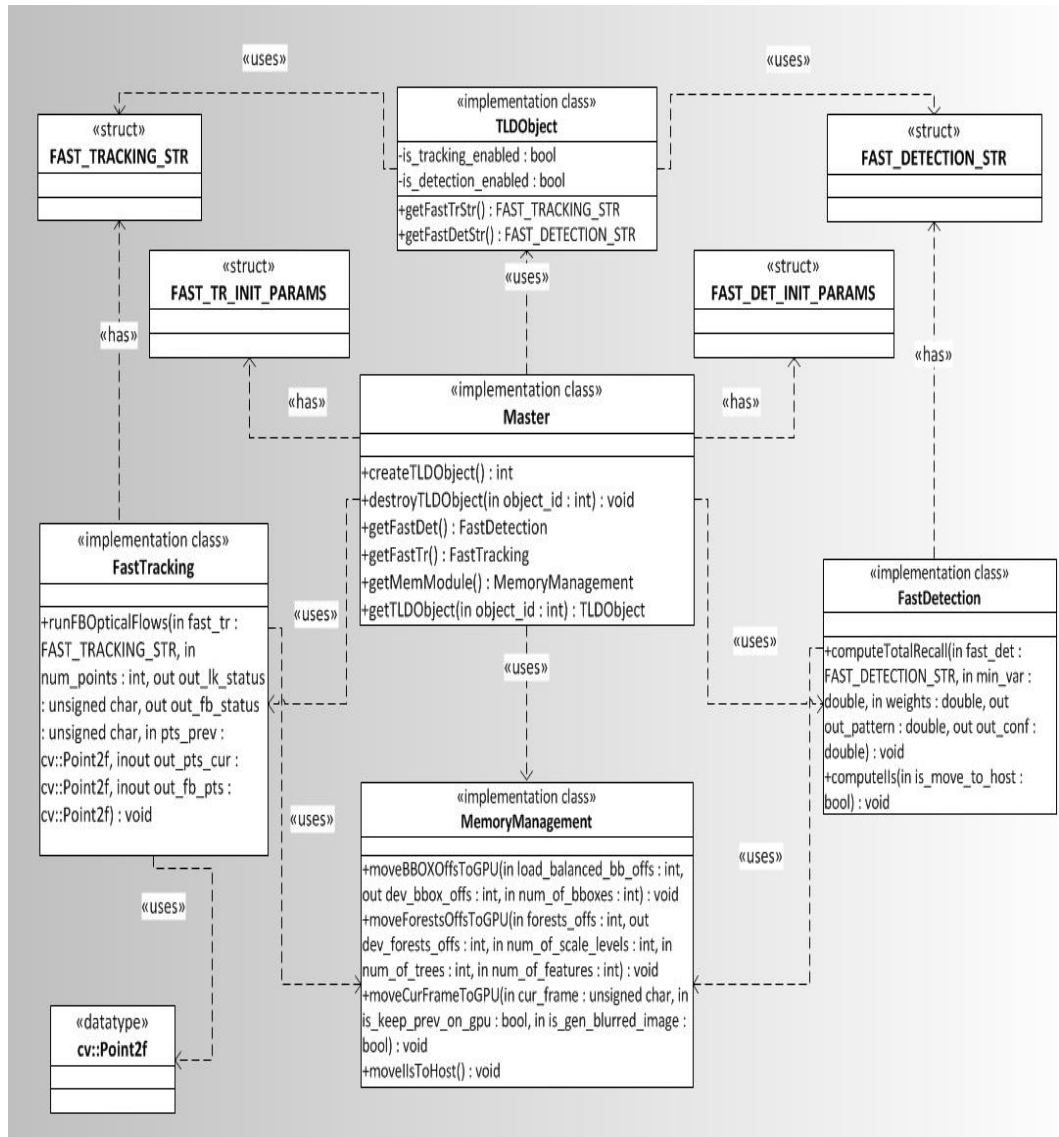


Figure 4-2 Class Diagram for H-TLD Objects and Structures

Figure 4-2 depicts the relationships among the key objects. There is no circular dependency between those objects (loosely coupled). In other words, when one of those classes is required to be changed (or extended) all the rest will not be affected. “has” relation shows that class/struct which is pointed by head of the arrow is owned

by class/struct attached to the tail of that same arrow and “uses” relation depicts that class/struct which is attached to the tail of the arrow has an instance of class/struct which is pointed by head of that same arrow. Since all classes/structs have their definitions and all necessary comments in their corresponding source codes; only a brief definition for each class/struct will be given in following items:

1. **TLDObject**: It is the key to communicate with the serial code. It has all state variables for an object under observation, and it stores all the information required by singleton H-TLD modules. Many of those variables are created at initialization time of the TLDObject as they remain intact until the disposal of that TLDObject; hence causing to eliminate overhead that would have been brought by allocation and de-allocation instructions. Each TLDObject is associated with a unique id called “handler” and it is the only way to acquire the TLDObject on demand in a serial context.
2. **MemoryManagement**: It has methods used within both serial code context (moving the current frame to the GPU, etc.); and the modules in H-TLD (such as converting BB offsets stored in the form of array-of-structures into structure of arrays for coalesced memory access of GPU). It manages all CPU/GPU data transfers; and conducts all other memory related operations.
3. **FastTracking**: It has methods which are executed in a heterogeneous way to reduce execution time of tracking module. For the time being, it only attempts to speed up forward/backward optical flow computations based on [12] using [19]. It has associated C++ structure called “FAST_TRACKING_STR” and it is created per TLDObject. It holds state variables for running tracking related operations.
4. **FastDetection**: It has methods which are executed in a heterogeneous way to reduce execution time of detection module. It has two methods for computing integral images via [1] and total recall computation which is explained in the next part of this chapter. It has associated C++ structure called “FAST_DETECTION_STR” and it is created per TLDObject. That structure holds state variables for running detection related operations.

5. **Master**: It is a container for having access to all API objects. When it is created for the first time, it creates the other H-TLD modules in accord with the parameters passed from serial code context. Access to any TLDOBJECT or H-TLD module (either FastTracking or FastDetection), should be done through this module. Another purpose of it is to create and destroy TLDOBJECTS on demand.
6. **Error Management**: This module is responsible for error handling. The class, named “HETLDError” is the type of all errors thrown by H-TLD. It inherits all basic fields, methods, constructors and destructors from its parent which is standard C++ library’s “runtime_error” class. All what clients should do is to surround API calls with try-catch blocks in order to capture and handle those errors. Table 4-1 describes its public fields for whenever an error is captured, clients will know what error has just occurred and in which module it did so.

<u>Public Field Name</u>	<u>Definition</u>
_module	It holds enumeration for the module that caused erroneous case.
_error_code	It stores the error code. Clients may check out “hetld_errors.hpp” file in order to figure out the exception (in that file each module has its own set of errors; so that API users may match the error code with the one they got at run-time).
what_arg	It is inherited from the parent class and clients may use it for displaying error messages in a user-friendly way.

Table 4-1 HETLDError Class’ Public Fields and Their Definitions

4.2 Implementation of H-TLD Detection Module

For the time being, H-TLD's detection module accelerates two critical parts of the original detection module from [2] (See Chapter 6 for speed-ups):

- **Total Recall Computation**: As mentioned in the previous chapters, detection module uses a sliding window methodology to detect whether the object under observation is still in the video or not and if it is, then to estimate object's location. Thus, for each BB; a confidence value must be assigned per frame. This operation has high computational load due to the high number of BBs for even lower resolution video streams (e.g. for a 470x310 resolution frame; ~30,000 BBs need to be scanned).
- **II Calculation**: Integral Image, which is also known as “summed area table”, is a data structure and algorithm for generating the sum of values in a rectangular subset of a grid (a video frame could be assumed as a grid). Equation (4-1) gives the formulation for forming an II out of a video frame. Suppose that there is a video frame I , then II is calculated as:

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y') \quad (4-1)$$

Where a pair of x and y specifies a particular location in the frame I .

Once II is calculated, in order to find the sum of all pixel values in any sub rectangular region within an II like the one in Figure 4-3 (Image was copied from [20]):

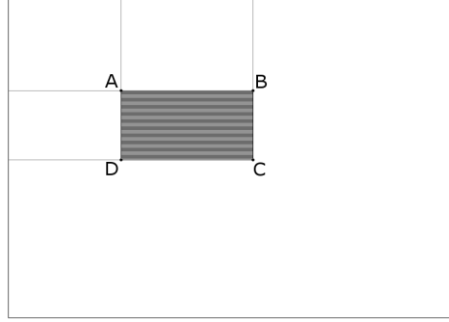


Figure 4-3 A subset of a Rectangular Region on an II

$$i(x,y) = I(C) + I(A) - I(B) - I(D) \quad (4-2)$$

As it can be seen from Equation (4-2), it is only adding and subtracting four array references.

Before Random Forest Index (RFI) calculation step (a subpart in total recall computation), an integral image and squared integral image must be calculated for the use in PV-computation to eliminate many of the BBs before they make their way to RFI calculation phase; hereby reducing total recall computation time. The reason why PV-Computation is the first step of the ensemble classifier is that it is far cheaper operation by comparison with the rest. Both integral images are calculated (right after current frame is moved to GPU by “MemoryManagement” module) by [1]’s “nppiSqrIntegral_8u32s_C1R” method. Results of this method and comparison with its serial counterpart could be found in Chapter 5.

- **Obtaining Blurred Image**: Original frame is not used in “RFI Calculation” phase when pixel comparisons are made; instead it is blurred by a “Gaussian Filter” for the sake of eliminating details. “MemoryManagement” module blurs that frame on-demand using a method from [19] and stores it on a separate location on GPU memory.

“Total Recall Computation” does not take place only on GPU; contrary to II-Calculation; but on both processing units to efficiently utilize both CPU and GPU. That computation includes 4 separate subparts executed in the order given below (however, all these steps are executed for number of asynchronous invocations and each invocation is independent of each other; hence none of the invocation has to wait for each other to be completed):

- PV-Computation (GPU Only),
- BB Stream Compaction (GPU Only),
- RFI Calculation (GPU Only),
- Confidence Calculation (CPU Only).

Each subpart’s implementation details will be discussed in separate subsections along with the reasons for choosing particular processing unit type (CPU or GPU). A general overview of “total recall computation” could be found in the pseudo-code shown below (Please, see [21] for what “single region” and “parallel region” come to the meaning of).

Algorithm Total Recall Computation

```
1: procedure COMPUTETOTALRECALL(numOfAsyncInv) ▷ Number of Asynchronous Calls  
                                         Calculated in Accordance with  
                                         Device Constraints : Threads  
                                         Per Block, Threads Per SM,  
                                         and Shared Memory Size in Order  
    ▷ Parallel Region Starts Here  
2: curThreadId ← omp_get_thread_num()  
3: Initialize Other OpenMP Private Fields for Each CPU Thread  
    ▷ Single Region "A" Starts Here  
4: for i ← 0, numOfAsyncInv do ▷ Commands in this Loop are Asynchronous  
    ▷ Groupi is a Group of Clusters Assigned  
    to be Processed in Asynchronous Call i  
5:    Issue PV – Computation for Clusters in Groupi to GPU  
6:    Issue Compacting BB Stream for Groupi to GPU  
7:    Issue Copying of Shift Amount within the BB Stream  
8:    for Each BB in Groupi from GPU to CPU  
9:  end for  
    ▷ Single Region "A" Ends Here  
10: for i ← 0, numOfAsyncInv do  
    ▷ Single Region "B" Starts Here  
11:    Wait Until Commands in Asynchronous Call i is Completed  
12:    Compute RFIs for Groupi  
13:    Copy RFIs for Groupi from GPU to CPU  
    ▷ Single Region "B" Ends Here  
14:    start ← Start BB Position for This Thread to Process  
15:    end ← End BB Position for This Thread to Process  
16:    k ← 0  
17:    for j ← start, end do  
18:      confVal ← Calculate Confidence Value for BBj  
19:      tempArray[curThreadId][k] ← confVal  
20:      k ← k + 1  
21:    end for  
22:  end for  
    ▷ Single Region "C" Starts Here  
23:  jobsDone ← 0  
    ▷ Here Working Thread is Spinning (Busy-Waiting)  
24:  while true do  
25:    if jobsDone == numOfCpuThreads then  
26:      break the Loop  
27:    else if Any Thread has Completed Confidence Value Calculation then  
28:      threadId ← threadId of Thread Which Completed Its Task  
29:    else  
30:      threadId ← -1  
31:    end if  
32:    if threadId ≠ -1 then  
33:      Copy Confidence Values from tempArray  
34:      of the Thread with Id threadId into outputArray  
35:      jobsDone ← jobsDone + 1  
36:    end if  
37:  end while  
    ▷ Single Region "C" Ends Here  
    ▷ Parallel Region Ends Here  
39:  return outputArray  
40: end procedure
```

Figure 4-4 Pseudo-Code for Total Recall Computation

4.2.1 PV-Computation (GPU Only)

Some concepts which are used to organize data in order to increase speed-up that will be referred to in the implementation part are detailed below.



Figure 4-5 Terms Used in PV-Computation

1. **Scale Level**: Open-TLD generates scanning windows (i.e. BB) via applying different scaling values (fractional number that equals to 1.2^k where k is in $[-(n-1)/2, (n-1)/2]$ as specified in [2] and where n is equal to the total number of scale levels) to width and height of the original object's BB, in order to prevent object from not being detected in the case which it is rescaled in some way. Just before it creates a group of BB that belong to certain scale level, it multiplies original width and height of the object's BB with scaling value to find width and height of BB at that certain scale level. In this thesis, scale levels are numbered from 0 to $(n-1)$.
2. **Scan Line**: Each BB is represented by its 4 corners. Since each BB's top and bottom border align with the horizontal axis (i.e. overlaps with a particular row of any II), each pair of corner locations (either at the top or bottom. e.g. top left and top right corner locations form a pair) of any BB lies within a single row of II. Any this type of row is called "scan line". Note that, **one scan line may form top and/or bottom borders of BBs from different scale levels (This property will result in reducing the number of scan lines that must be read into shared memory)**.

3. **Scan Line Pair**: A pair of scan lines encompasses top and bottom borders of a group of BB at a given scale level. That is to say, one scan line for top borders and another one for bottom ones of a particular group of BBs at a given scale level.
4. **Chunk**: is the smallest container unit for BBs from different scale levels. They are building blocks of clusters. Although each chunk is carefully designed to house BBs spanning a single scan-line pair from every scale level; since number of scan-line pairs decrease as scale level goes up and due to shared memory limit, chunks do not always have to have bounding boxes from all scale levels. While the number of BBs among chunks may vary, the chunks are designed in such a way to have approximately the same number of BBs. But one thing within a chunk is guaranteed to be consistent which is that a group of BBs spanning a scan-line pair cannot be impartible and should be located within a single chunk, no matter what
5. **Cluster**: Actual logical unit which is run in GPU thread blocks. Each thread block of CUDA enabled device is responsible for executing one cluster per invocation. Each cluster has almost (See subsection called “Load Balancing Concern”) equal number of chunks.

In the remainder of this chapter, firstly the design of kernel on GPU is discussed. Then, memory optimizations, more specifically shared memory and coalesced memory access to global memory concerns are mentioned. And finally, load balancing design and memory access patterns on GPU are addressed.

4.2.1.1 PV-Computing Kernel on GPU

This kernel is composed of three parts:

1. Read the scan-line pairs into the shared memory.
2. Compute patch variance (PV) for each BB either based on integral image or squared integral image.

3. Decide on whether BB's calculated PV passes specified threshold value (min PV) and assign 0 or -1 (depending on the outcome of PV-Computation) for below and above threshold value respectively.

Pseudo-code in Figure 4-6 shows implementation details of the kernel. Note that, since kernel code is run by each thread in a block; that pseudo-code should be treated as if it was a piece of serial code run by a single thread.

Although order of time complexity function (i.e. Big O) of any algorithm in [2] was not modified; functions themselves were done so in order to have them complied with parallel structures and parallel computation methodologies. Adding some “overhead instructions” (such as initializing instructions before a loop begins), “control instructions” (such as incrementing an index to advance any loop), and some other helper instructions in the main loop such as having access to some extra memory locations are some modifications that did not affect the order of time complexity function. In conclusion, “basic instructions” (instructions that are executed for as many times as the size of input) remained intact.

Time complexity function of PV-Computation is given in Equation (4-3).

$$g(n, m) = 2 \times (m + 2 \times n) \quad (4-3)$$

Where n is equal to the number of BBs and m is equal to the number of scan-lines to read.

For a given complexity function $f(n)$, $O(f(n))$ (i.e. Big O) is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

$$g(n) \leq c \times f(n) \quad (4-4)$$

If $c = 4$, $n \geq 1$ and $m \geq 1$; then Equation (4-5) satisfies Equation (4-4).

$$2 \times (m + 2 \times n) \leq 4 \times (m + n) \quad (4-5)$$

It means that the order of time complexity function is $O(n + m)$. Thus, PV-Computation is still running at linear time as does its serial counterpart (which is $O(n)$).

Algorithm PV-Computation

```

1: procedure COMPUTEPVONGPU(
    bboxOfss,           ▷ Offsets Related with BBs such as Its Top-Left Corner
    iis,                ▷ Reference to Both Integral and Squared Integral Images
    cumSumOfSL,         ▷ Cumulative Sum of # of Scan-Lines for All Clusters
    scanLineRows,       ▷ Contains Row Number for Each Scan-Line
    cumSumOfBBPCL,      ▷ Cumulative Sum of # of BBs for All Clusters
    minVar,             ▷ Threshold Value for Passing PV-Test
    bbVar)              ▷ Temporary Array for Storing Variance of Each BB
    ▷ Extract Information Which Belongs to this Block
2: slStIdx ← cumSumOfSL[blockIdx.x]           ▷ Offset to Where Threads of
                                                This Block will Start Read-
3: bbStIdx ← cumSumOfBBPCL[blockIdx.x]         ▷ Offset to Where Threads of
                                                This Block will Start Read-
                                                ing BB's Offset
4: slSize ← cumSumOfSL[blockIdx.x + 1] - slStIdx ▷ Num of Scan Lines That
                                                This Block Requires
5: bbSize ← cumSumOfBBPCL[blockIdx.x + 1] - bbStIdx ▷ Num of BBs That This
                                                Block is Going to Process
    ▷ Here Main Loop Begins
6: for i ← 0, 2 do
7:   if i == 0 then
8:     scanLines ← Read Integral Image into Shared Memory Collaboratively
9:   else
10:    scanLines ← Read Squared Integral Image into Shared Memory Collaboratively
11:   end if
12:   for j ← blockIdx.y * blockDim.x + blockIdx.x, do
13:     j < bbSize,
14:     j += blockDim.x * blockDim.y
15:     mx ← (scanLines[bboxOfss[br]] - scanLines[bboxOfss[tr]] -
16:           scanLines[bboxOfss[bl]] + scanLines[bboxOfss[tl]]) / bboxOfss[asw]
17:     if i == 0 then
18:       bbVar[bbStIdx + j] ← -(mx * mx)
19:     else
20:       bbVar[bbStIdx + j] ← bbVar[bbStIdx + j] + mx
21:     end if
22:   end for
23:   ▷ Writing PV Results to Global Memory
24:   for i ← blockIdx.y * blockDim.x + blockIdx.x, do
25:     i < bbSize,
26:     i += blockDim.x * blockDim.y
27:     if bbVar[bbStIdx + i] ≥ minVar then
28:       pvStatus[bbStIdx + i] ← 0
29:     else
30:       pvStatus[bbStIdx + i] ← -1
31:     end if
32:   end for
33:   return pvStatus
34: end procedure

```

Figure 4-6 Pseudo-Code for PV-Computation on GPU

There is a mapping between concepts in a multi-threaded environment and the ones in PV-Computation:

1. Each thread in a particular block is mapped to more than one scan lines (not necessarily; but it might happen as dimensions of the frame becomes larger and larger) located in any of the IIs (either normal or squared); at the time when II (either Squared or Plane II) pixels are read into shared memory collaboratively.
2. A cluster is mapped to a single block per kernel invocation and vice versa,
3. A BB's PV-Computation is mapped to a single thread associated with a particular block; but not vice versa (i.e. a thread may process more than one BB of a cluster that its block is assigned to).

Figure 4-7 depicts these relationships:

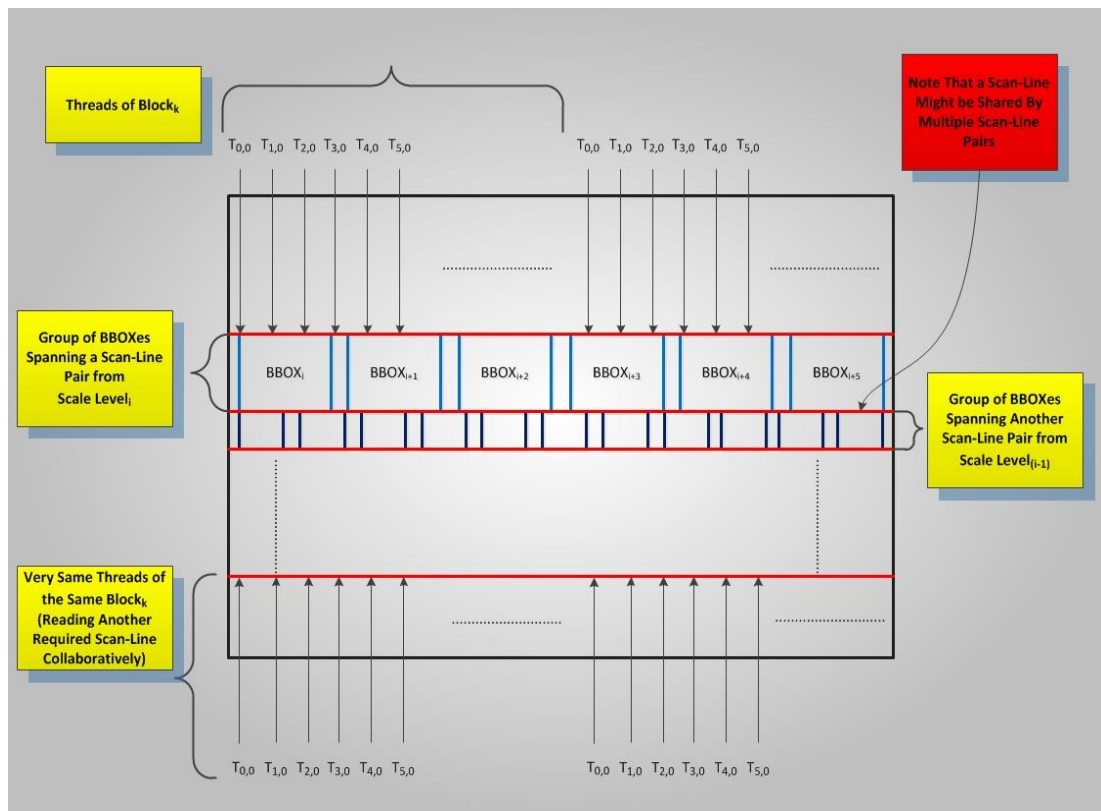


Figure 4-7 Mapping Multi-Threaded Concepts to PV-Computation Ones

4.2.1.2 Optimization of Memory Access

There are two main concerns regarding the use of shared memory:

1. Pixels located on a single scan-line are used by multiple threads in a block while PVs are being computed (in some cases, threads may use some pixel values for more than once); hence it would have resulted in multiple accesses to global memory for exactly the same memory locations. There are two cases in which this phenomena shows itself up.
 - As illustrated in Figure 4-7 BBs from scale level i and scale-level $i-1$ share the same scan-line which spans bottom and top borders of their associated scan line pairs respectively. Those BBs from different scale levels are more likely to be processed in the same CUDA block. This is explained in more detail in the subsection where load balancing and exploitation of spatial locality are described.
 - BBs are not that far away from each other and have gaps between them as shown in Figure 4-7, BBs drawn in between scan-line pairs are shifted 10% of their width to left; therefore resulting in multiple accesses to very same pixels by those BBs that are adjacent horizontally to each other. All in all, scan-lines required by threads of any block should be read into shared memory for accelerating speed of memory access.
2. As it can also be seen from pseudo-code; a PV for a BB is computed in two steps rather than in one (outer loop iterates for 2 times). The reason is the limited size of the shared memory of a GPU's streaming multiprocessor. As of yet, CUDA architecture 5.0 supports no more than 64KB of shared memory [22]. Thus, scan-lines for II and squared- II are read into the shared memory separately, so that more BBs could be processed per invocation within a single CUDA block. More scan-lines each block reads into shared memory, more BBs confined in between scan-line pairs are processed per single kernel invocation; i.e. more chunks a cluster may hold; less number of clusters are formed to be processed. As a result, it leads to fewer number of

kernel invocations for PV computation which is the key to reduce the potential overhead of multiple kernel invocations at this stage.

Only concern about global memory access is how BB offsets are read into thread registers. Those accesses must be coalesced; so that maximum bandwidth might be used per memory transaction (See Chapter 6 of [23] for more details about global memory chip design). Figure 4-8 depicts how array of BB offsets was organized on the host side:

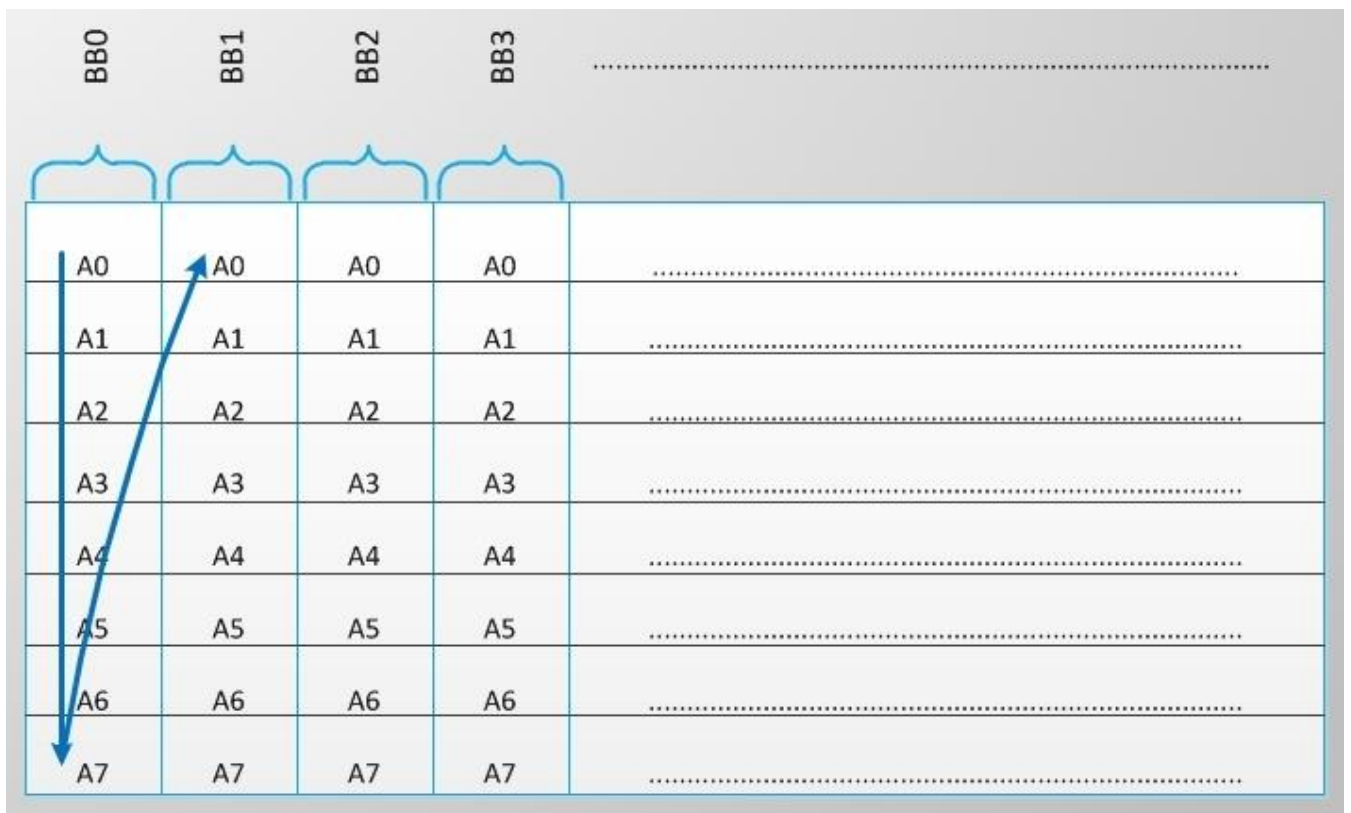


Figure 4-8 BB Organization on RAM in Serial Code Context

As the direction of arrows indicates, it is a row major memory access and all attributes of any BB's offset are placed sequentially. This pattern is called Array of Structures (AoS).

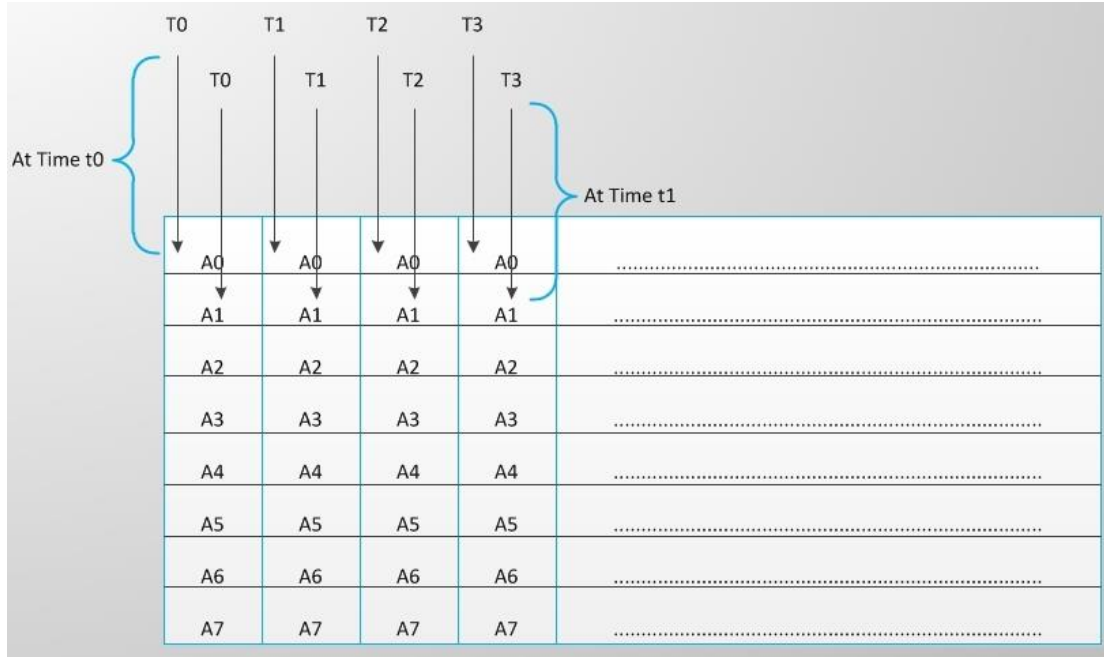


Figure 4-9 Access Pattern for BB Offs without Conversion on Device Memory

Using this pattern, memory accesses to global memory would be non-coalesced as illustrated in Figure 4-9. Coalesced access could be achieved by realignment of data stored on device's global memory. This is done by a kernel implemented for this purpose named "convertBBXOffsToSoA" in "MemoryManagement" module. This kernel groups each attribute type; in contiguous memory locations, converting the AoS pattern into SoA pattern. Mapping of AoS based indexing into SoA is illustrated in Figure 4-10.

Figure 4-11 shows new access pattern.

First off , current index to the AoS array is computed as follow:

$$indexToAoS = threadGlobalId + iterationNumber \times totalNumberOfThreads$$

Secondly, which attribute "the value read" refers to has to be found (i.e. attribute index for the current BB. Attribute indices starts with 0 and ends at $n - 1$; where n is

equal to the number of BB's offset attributes).

Suppose that, $(a \% b)$ operation is defined as $a - \text{floor}(a/b) \times b$.

$$\text{attrIndex} = \text{indexToAoS} - \text{floor}(\text{indexToAoS} * \text{oneOverNumOfAttrs}) \times \text{numOfAttrs}$$

“One over number of attributes” is kept around because division is relatively much more expensive than multiplication.

Thirdly, destination location for that attribute has to be calculated. Since number of attributes each attribute group has is equal to total number of BBs; the following shows that how destination location within SoA array is computed:

$$\text{indexToSoA} = \text{attrIndex} * \text{numberOfBBBoxes} + \text{floor}(\text{indexToAoS} \times \text{oneOverNumOfAttrs})$$

Then all has to be done is to move that attribute to its final destination:

$$\text{structureOfArrays}[\text{indexToSoA}] = \text{arrayOfStructures}[\text{indexToAoS}]$$

Figure 4-10 Conversion from AoS to SoA

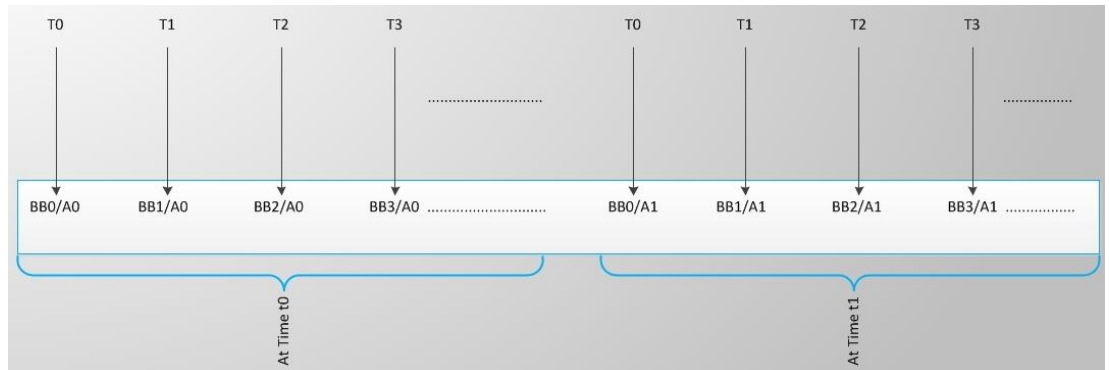


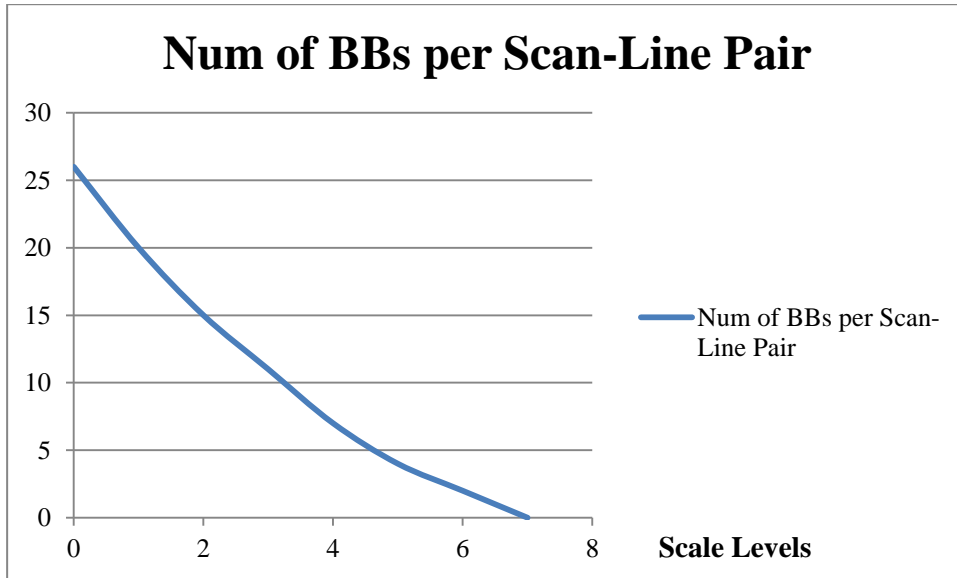
Figure 4-11 Accessing BB Offsets on GPU's Global Memory after Conversion

4.2.1.3 Load Balancing Concern

In PV-Computation, ideally, BBs from different scale levels should be grouped in such a way to ensure the clusters to have the same number of BBs to be processed.

Spatial locality between BBs must be exploited as well; so that scan-lines read into shared memory might be used by higher number of threads during PV-Computation process. Equation (4-6) and Figure 4-12 illustrate the decrease in the number of BBs in a scan-line pair with the increasing scale level. Note that BBs specified by a particular scan-line pair must be processed by a single CUDA block which means that those BBs are impartible among CUDA blocks per kernel invocation. In Equation (4-6), f is a function of scale level and gives the number of BBs a scan-line pair may hold at a certain scale level x :

$$f(x) = \frac{w_I - \alpha^x \times w_0}{\beta \times \alpha^x \times w_0} \quad (4-6)$$



Where: $w_I = 470\text{px}$, $\alpha = 1.2$, $\beta = 0.1$, $w_0 = 128\text{px}$, and $x(\text{scale level})$ varies from 0 to 7.

Figure 4-12 Relationship between Scan Line Pairs and Scale Levels

Where w_I is the width of video frame, α is base scale level (which is equal to 1.2 in Open TLD's [2] implementation, β is shifting factor (it is in the range of (0, 1] and

in Open TLD's [2] implementation equal to 0.1) and w_0 is the width of initial bounding box that defines the object under observation.

Graph in Figure 4-12 is visual depiction of the function f in Equation (4-6) (assuming all parameters are known; but scale level x). Figure 4-13 shows order in which BBs are stored on memory on the host side.

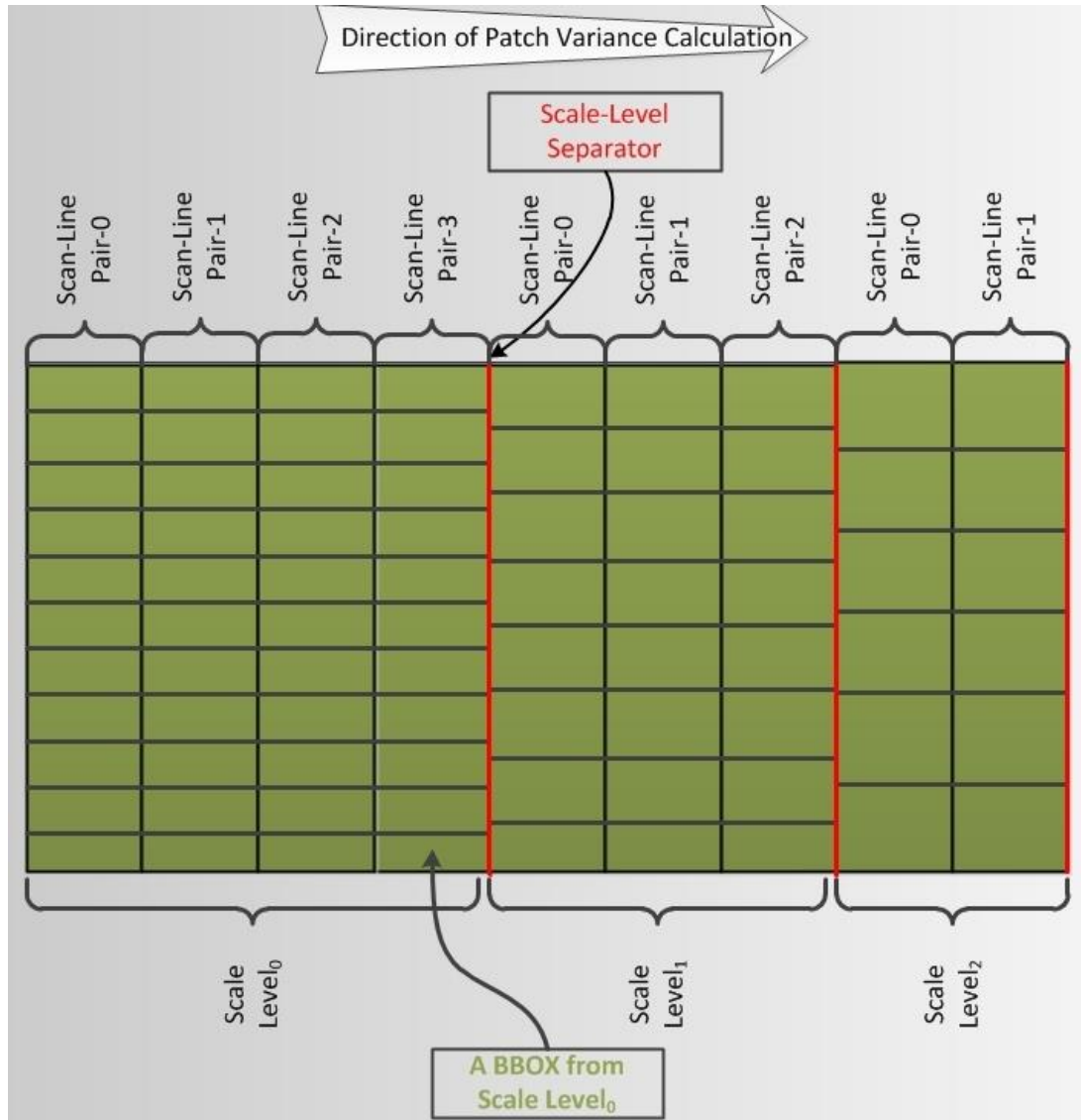


Figure 4-13 Processing of BBs without Loading Balancing

Processing of BBs in the order as they are processed on the host side prevents proper load balancing on GPU side. As direction of PV-Computation in Figure 4-13 indicates, each scan-line pair and the BBs located within it at a certain scale level is located contiguously in the memory. Thus, blocks of PV-Computing kernels would start processing fewer numbers of BBs compared to the earlier ones. This would cause unbalanced kernel loads where many processing units become idle after a while. To prevent such unbalanced operations, a pre-processing step as shown in Figure 4-14 takes place once at the time when TLDOBJECT is initialized.

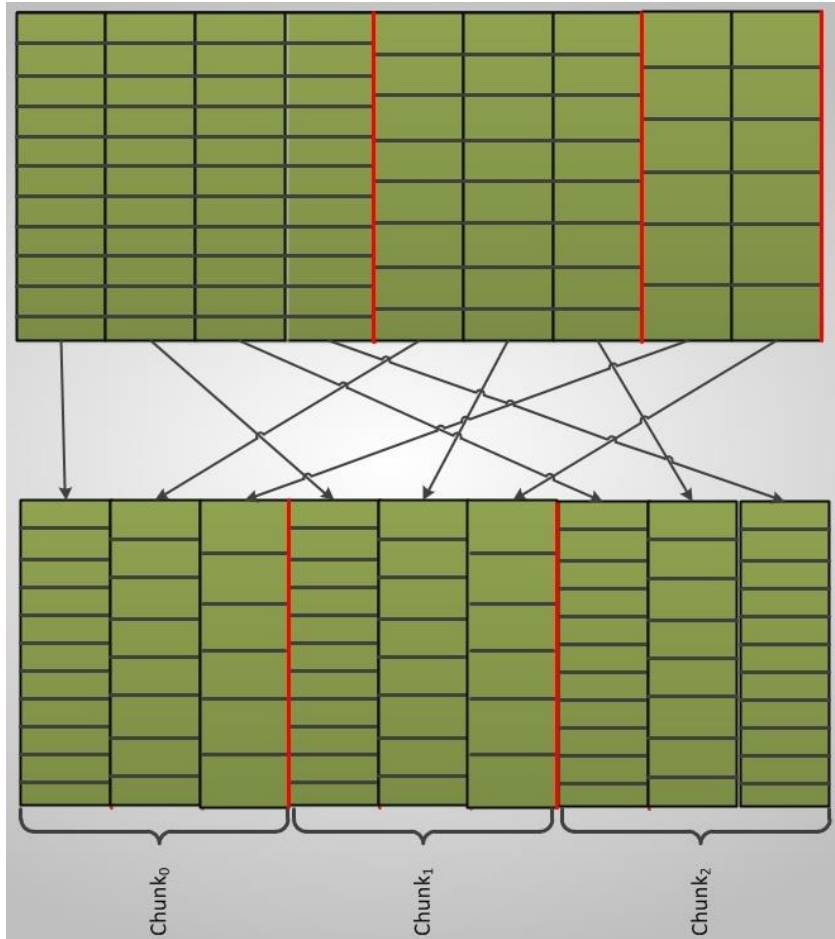


Figure 4-14 BB Ordering for Load Balancing

4.2.2 BB Stream Compaction (GPU Only)

We need to eliminate the BBs which fail the test and dispatch only the ones passed to the next stage to allow sequential access to remaining BBs in the following steps, and

copy only their RFIs back to host side after completion of “confidence index calculation” step (the next stage in where RFIs for only PV-Test succeeding BBs). Figure 4-15 shows the status of BB stream right after running PV-Computation.

BBOX ₀	BBOX ₁	BBOX ₂	BBOX ₃	BBOX ₄	BBOX ₅	BBOX ₆
0	-1	-1	0	-1	0	-1




Figure 4-15 BB Stream after Running PV-Computation

As it is seen Figure 4-15, in order to compact BB stream and group all succeeding BBs together, they must be shifted to the left. If a prefix sum is run with the output of PV-Computation shown in Figure 4-15; then result shown in Figure 4-16, which gives the shift amounts that next two steps (RFI Calculation and Confidence Calculation) require, is obtained (That prefix-sum is performed by the state of art library CUB [24]):

BBOX ₀	BBOX ₁	BBOX ₂	BBOX ₃	BBOX ₄	BBOX ₅	BBOX ₆
0	-1	-2	-2	-3	-3	-4




Figure 4-16 Left-Shift Amounts after Running “Prefix-Sum”

Equation (4-7) gives the formula that moves succeeding BBs to their correct locations within the BB stream for compaction. Where i equals to the i th BB’s index in the original BB stream and LSH_{AMOUNT} is the array which holds left shift amounts for zipping that BB stream.

By this way, after RFI calculation step, there is no need to compute and copy all RFIs for all BBs; but only of those that passed PV-Test. This step produces (memory operations between host and device which are very expensive. This can be seen from the results listed in Chapter 6). If all those l-shift values are negated via multiplying them with “-1”, one may use those r-shift values to find previous locations (the ones before compaction occurs) of those succeeding BBs by adding up to their current location after compaction.

$$\text{BBOX}(i)_{\text{newLocation}} = i + \text{LSH}_{\text{AMOUNT}(i)}, \quad (4-7)$$

4.2.3 RFI Calculation (GPU Only)

In this subpart, calculation of indices to confidence array (i.e. weights array) is described. In [2], it was explained that how random forest are formed and features of each tree in that forest are indexed. On GPU side, there are some memory accesses which cannot be coalesced at all. For instance, memory accesses to blurred image data can't be coalesced due to the nature of how comparison is made for a single feature on condition that this comparison is made by a single GPU thread (See Figure 4-17). Besides, a single thread conducts all feature comparisons for a single BB in this implementation.

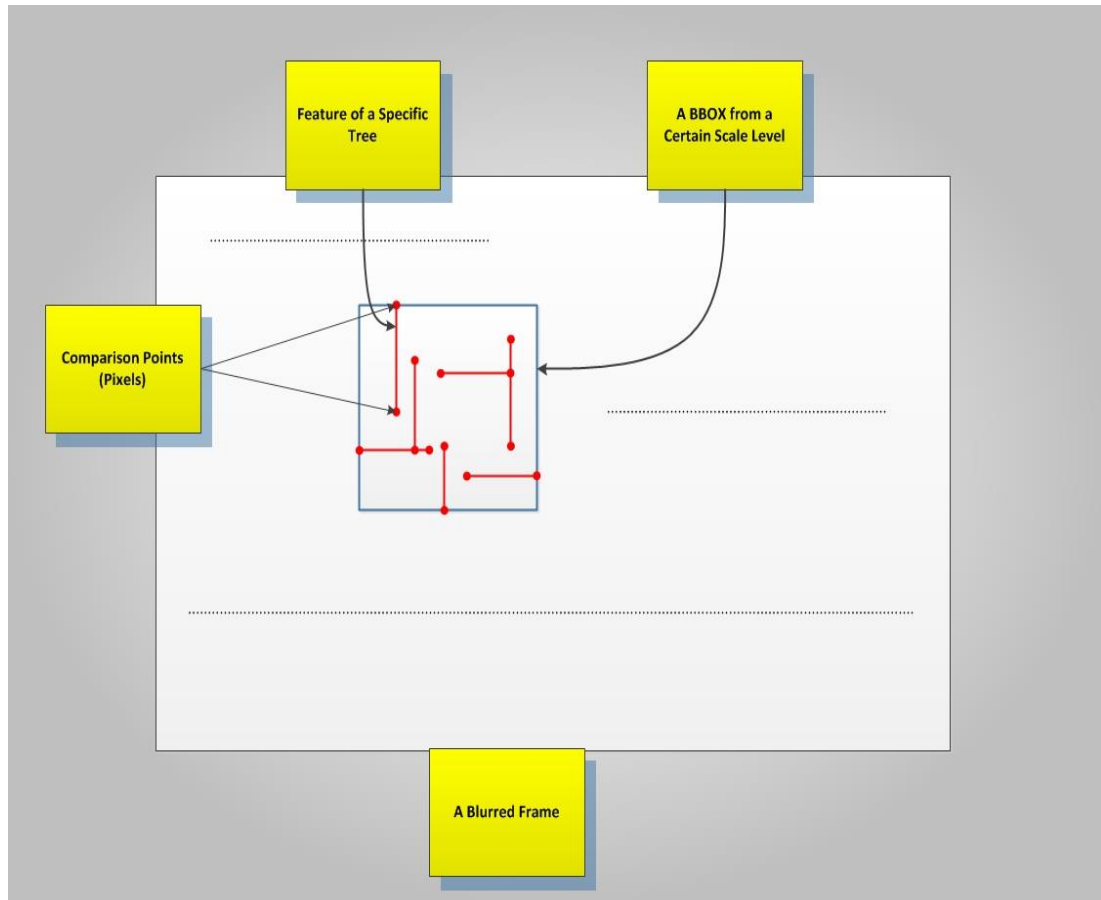


Figure 4-17 Feature of a Random Forest's Tree

Although pixels which are subjected to comparison are not located on contiguous memory locations, new architectures from CUDA team have been trying to leverage global memory transactions and furthermore, parallel computation power of today's CPUs is not a match to the one of GPUs provided that one has 4, 8 cores whereas the other one has hundreds of cores. That is to say, it may easily compensate time loss caused by that non-coalesced global memory access pattern. Pseudo-code in Figure 4-18 shows that how each individual GPU thread computes RFIs based on aforementioned pixel comparison.

Time complexity function of RFI calculation is given in Equation (4-8).

$$g(m, n, o) = m \times (4 + n \times (o + 3)) \quad (4-8)$$

Where m equals to the number of BBs, n equals to the number of trees and o equals to the number of features.

If $c = 8$, and $m, n, o \geq 0$, then the inequality in Equation (4-9) satisfies the inequality mentioned in Equation (4-4).

$$4m + mno + 3mn \leq 8mno \quad (4-9)$$

It means that the order of time complexity function is $O(mno)$. Thus, RFI-Calculation is still running at linear time as does its serial counterpart (which is $O(mno)$ again).

Another version of RFI Calculation was also implemented and tested; but failed in speed-up due to data loading time into L-Cache of a CPU core. It was a hybrid solution for RFI Calculation in which CPU helps GPU by calculating a fraction of RFIs based on a benchmark which is run before any TLDOBJECT is created. Results can be found in Chapter 5.

Algorithm Index Calculation for Random Forests

```

1: procedure COMPUTE_RFI_INDICES_ON_GPU(
    start,                                ▷ BB Start Index
    end,                                  ▷ BB End Index
    numOfBB,                              ▷ end-start
    blurredImage,                          ▷ Blurred Version of the Current Frame
    bboxOffs,                             ▷ Offsets Related with BBs such as Its Top-Left Corner
    absTLCorner,                           ▷ Absolute Location of Top-Left Corners for All BBs
    forestsOffs,                           ▷ Offsets to Each Frame for All Features
    bbSHAmount) ▷ Left-Shift Amount for Each BB in Order to Compact the BB Stream
    ▷ Load All Forest Offsets into Shared Memory
2: features ← Read all forest offsets into shared memory collaboratively
    ▷ Make Pixel Comparisons and Find Indices
    ▷ All Confidence Indices for a Single BB is Calculated
    by a Single Thread
3: for i ← start + blockIdx.x * blockDim.x + threadIdx.x, do
    i < end,
    i += gridDim.x * blockDim.x
4:   bbIdx ← bbSHAmount[i] + i                                ▷ Its Idx Before Stream Com-
                                                                paction
5:   featurePtr ← bboxOffs[bbIdx +                             ▷ totalNumOfBB is Stored in
                                                                Constant Memory
                                                                PTR_TO_FEATURES]
6:   tlCorner ← absTLCorner[bbIdx]
7:   bbIdx ← i - start                                         ▷ Its Idx After Stream Com-
                                                                paction
8:   for j ← 0, numOfForests do
9:     treeOff ← featurePtr + j * 2 * numOfFeatures ▷ numOfFeatures is Stored in
                                                                Constant Memory
10:    index ← 0
11:    for k ← 0, numOfFeatures do
12:      index ← index << 1
13:      if blurredImage[tlCorner + features[treeOff + 2 * k]] >      then
14:        blurredImage[tlCorner + features[treeOff + 2 * k + 1]]
15:        index ← index | 1
16:      end if
17:    end for
18:    confIndices[j * numOfBB + bbIdx] ← index                ▷ numOfBB equals to the
                                                                Number of Remaining BBs
                                                                after PV-Test
19:  end for
20:  end for
21:  return confIndices
22: end procedure

```

Figure 4-18 Pseudo-Code for RFI Calculation on GPU

4.2.4 Confidence Calculation (CPU Only)

The last step takes place only on host side (since weights for random forests are constantly updated by learning component on host side from time to time; those weights must have been moved to GPU each time right after they are updated which would be very costly operation) and is the simplest one among all those have been

mentioned so far except relocating BBs for which caller may receive confidence values for all BBs in order as it expects. Open-MP [21] is put into action for which all CPU cores could be exploited like it has been done so far on GPU side. Each group of BBs (number of BBs in any such group is exactly equal to the number of BBs processed in the corresponding PV-Computing kernel) is once again split into much more smaller chunks; so that each of those chunks that are equal in size could be processed by individual CPU cores concurrently. There is an important notion that should be addressed before mentioning how confidence values for BBs are calculated on CPU cores. As explained before in PV- Computation subsection, in order to balance the load that each kernel invocation would be responsible for; locations of all BBs on the memory are changed. On the other hand, caller that calls “computeTotalRecall” method was unaware of this load balancing operation; therefore “computeTotalRecall” method should return all BBs’ confidence values in order that is known to the client. However, it’ll lead to a “false cache-line sharing” via L-caches of CPU cores. It is shown in Figure 4-19.

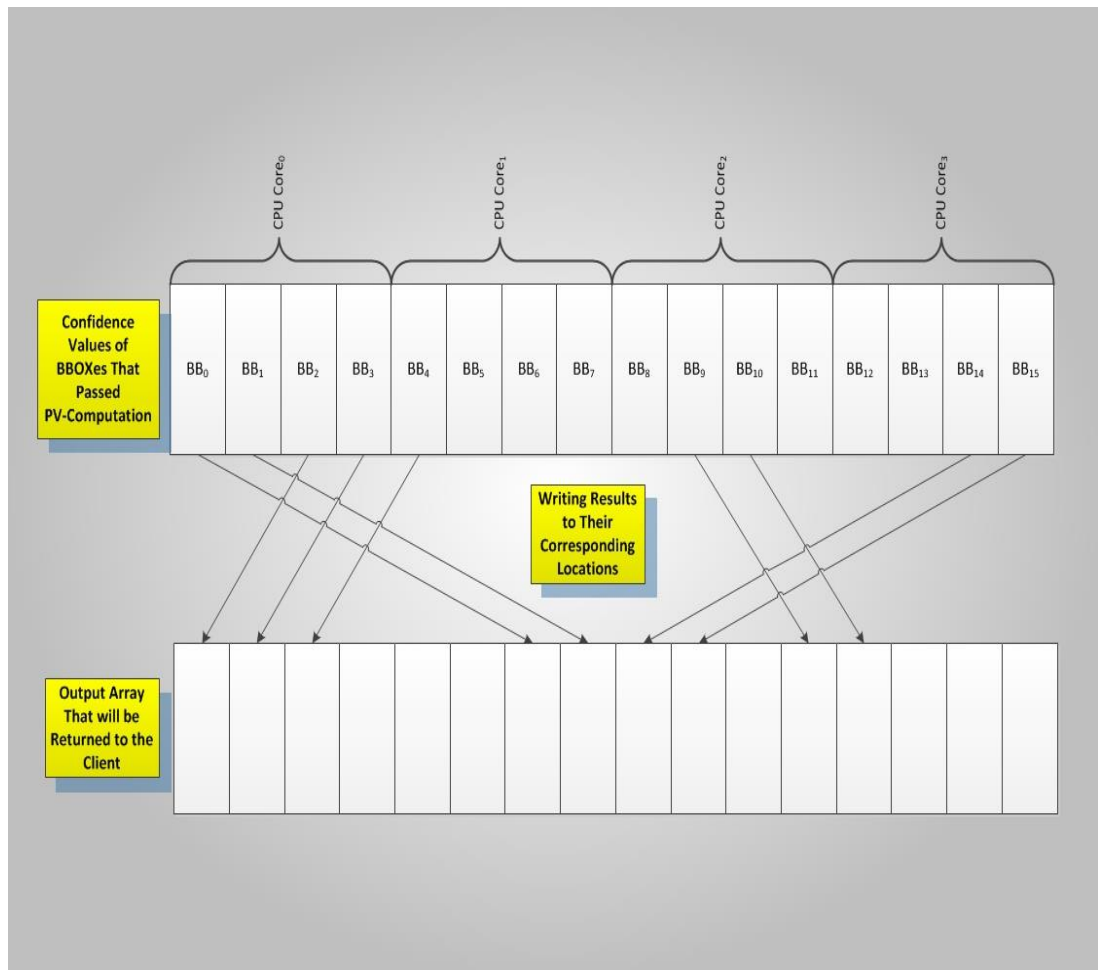


Figure 4-19 Confidence Values Written to Output Array

If data had been written in a fashion similar to the one shown in Figure 4-19, then cache-lines that are attached to different cores must have been flushed out to the RAM before any other core (which wants to execute a “write-transaction”) has access to any memory location nearby. It is easily overcome by creating separate temporary arrays for each CPU thread to write their corresponding results into; then only one CPU thread could be in charge of gathering those values located in the temporary arrays together on the final output array, which is supposed to be returned to caller. Pseudo-code in Figure 4-4 has already shown when it is done. After computing all confidence values for all those remaining BBs and writing those values into separate temporary arrays allocated for each CPU thread; they are copied

from those temporary arrays to the output array by a single CPU thread in such a way that caller may receive them in order which is known to it.

4.3 Implementation of H-TLD Tracking Module

In TLD [2] Median Flow Tracker (MFT) [12] is used.

MFT uses LK (Lucas-Kanade Optical Flow) [25] in order to generate a sparse motion-flow based on either some pre-determined points on the first frame (for initial flow generation) or some reliable points obtained from previous tracking results (for flow generation processes other than the first one). Suppose that there are two contiguous frames extracted from a video stream I_t, I_{t+1} . Before two-step filtering process (discussed in next two paragraphs), MFT generates a flow in the forward direction (from frame I_t to I_{t+1}); then it uses these predictions (i.e. next points on frame I_{t+1} for original points located on frame I_t) extracted from this flow to go back in the reverse direction (from frame I_{t+1} to I_t) for finding points called forward-backward points. Those next points (result of the first flow) and forward-backward points (result of the second flow) are sent to NCC (Normalized Cross Correlation) and FB (Forward-Backward) filters.

One of the two filtering methods used in [2] is NCC filter and its formula is given in Equation (4-10).

$$R(x_t, y_t) = \frac{\sum_{x', y'} (S_t'(x', y') \cdot S_{t+1}'(x', y'))}{\sqrt{\sum_{x', y'} S_t'(x', y')^2 \cdot \sum_{x', y'} S_{t+1}'(x', y')^2}} \quad (4-10)$$

Where S_t is the searching window (centered on the location (x_t, y_t) in I_t) which we want to correlate with the searching window S_{t+1} (centered on the location (x_{t+1}, y_{t+1}) in I_{t+1}). (x', y') is the location of a pixel in local coordinate system of any searching window. If the correlation between these two searching windows is greater than a

certain threshold; then this point is considered as it is passed through the NCC filter, and is sent to FB filter for further filtration. This filter is not costly due to processing few thousand points even on videos with high resolutions depending on the size of BB of object under observation. Thus, this operation was left as it is (its cost for different resolutions can be seen in Chapter 3).

Decision as to whether to further filter out a point based on FB filter is made by performing Euclidean distance calculation on that original point and its corresponding forward-backward point; and this operation is also not heavy; therefore it was left as it is too (Note that it was not mentioned in Chapter 3 due to its low computational cost).

Although filtering methods used in template matching such as Squared Sum of Differences (SSD), NCC, or any other produces satisfying results; FB (Forward-Backward) filter complements them in cases which they might fail (This is explained in Chapter 2 and 5 of [12], and out of this thesis' scope). To sum up, LK tracker must be run twice for finding more reliable points in tracking module of [2]. As a result of this; it requires more processing power than any traditional tracking algorithm.

There are 2 sequential steps performed by Open-CV's GPU module [19] in order to accomplish the LK sparse optical flow:

- **Building Pyramids**: This is done by reducing resolution (half in both directions), and running interpolation on images I_t^L and I_{t+1}^L at levels other than the base level (at where I_t and I_{t+1} are located). This operation is both sequential and parallel. Each image at a higher level requires the image at the level just one below its level; hence CPU iteratively calls "build pyramid" routine on both I_t and I_{t+1} (note that, pyramids are built for both images). On the other hand, it is parallel in the sense of down-scaling the image at a level of pixel; because a pixel at a higher level only depends on the image at one level below, but not on its neighborhood; therefore this interpolation method is executed on GPU for both images simultaneously. Each GPU thread is

responsible for finding the value of a single pixel at one higher level. In Table 4-2, there is a piece of code copied from GPU module of Open-CV [19] that shows how an image at a higher level is built up from the image at one level below.

```

for (int level = 1; level <= maxLevel; ++level)
{
    pyrDown(prevPyr_[level - 1], prevPyr_[level]);
    pyrDown(nextPyr_[level - 1], nextPyr_[level]);
}

```

Table 4-2 Building Pyramids for LK Tracker

Where “prevPyr_” is I_t and “nextPyr” is I_{t+1} ; and where method “pyrDown” runs interpolation on GPU asynchronously to generate images (frames) at higher levels.

- Predicting Next Points: According LK sparse optical flow, when the over-determined linear system given in Equation (4-11) is iteratively solved for a certain number of times, or until motion vector reaches a certain threshold (this method is called Newton-Raphson) for a given point q_i on I_t , it approximates this point’s next location on I_{t+1} .

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i)I_y(q_i) \\ \sum_i I_y(q_i)I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \times \begin{bmatrix} -\sum_i I_x(q_i)I_d(q_i) \\ -\sum_i I_y(q_i)I_d(q_i) \end{bmatrix} \quad (4-11)$$

Where V_x and V_y show components of motion vector, q_i is the pixel inside the searching window, I_d is equal to $(I_{t+1} - I_t)$, and I_x, I_y , are the partial derivatives of the image I_t with respect to position (x, y) evaluated at the point q_i .

Each GPU thread is responsible for finding the next point on I_{t+1} that corresponds to a particular pixel (point) on I_t by solving the system given in Equation (4-11) iteratively. If there are less points to track that cannot highly occupy GPU’s Stream Multiprocessors (SMs); then CPU implementation may have a chance to outrun the

performance of GPU version (this is exemplified in Chapter 5 in the sub section dedicated to tracking).

There have been many ongoing discussions and studies on how the LK tracker (or any other tracker) should be implemented on a GPU or on a hybrid CPU-GPU platform as was explained in Chapter 2. However, it is easy to switch to a new algorithm from the one presented by Open-CV's GPU module [19]; and integrate it to H-TLD for which it is an independent method in a separate module.

CHAPTER 5

RESULTS

In this section, results of each accelerated part are discussed and compared with its original implementation (i.e. sequential one). Speed-up of each individual fraction of code mentioned in Chapter 4 is detailed in this chapter in such a way that how much of total wall clock time elapses on processing unit and the rest of it does on data transfer between host and device. More particularly;

- In detection module, “Total Recall Computation” is the part that takes the longest time to compute among all others. In this section, each of its subparts is analyzed performance wise.
- Some state-of-art technologies such as [24] and [1] were incorporated into the implementation accordingly, in order to increase the total speed-up. The contribution of such libraries is analyzed.
- Speed-up obtained in the tracking module by use of Open-CV [19] is analyzed.
- The effect of video resolution on speed-up is discussed. Hybrid and sequential “Total Recall Computation” methods are run on 3 video with different resolutions (low, medium, and high) to depict this effect.
- Finally, NVIDIA’s Visual Profiler is used to display how overlapping kernel executions and data transfer between host and device help H-TLD in improving performance.

Each method under observation is run for 10 times (both sequential and hybrid methods); then average elapsed time is calculated and this result is reported. In case of using [21] to exploit all cores of CPU, the longest time interval elapses on one of those cores is used as the base in measuring the performance for that piece of code. Resolution of video frames for all test cases is equal to 480x270. The purpose is to

have exactly one asynchronous call to device (each time when a single frame is processed) so that elapsed time for each individual stage within a particular part of the implementation could be measured accurately. All methods (either sequential or hybrid) were tested under the same circumstances (unnecessary processes were terminated, services were closed, network connection was disconnected).

Test platform specs are given in Table 5-1.

OS	Windows 7 x64
CPU	Intel i7 4770K 3.5 GHz, 4 Physical Cores, Hyper Threading Factor is 2
GPU	Tesla K 40c, Compute Capability 3.5, 15 SMs, 192 Cores per SM, 2 Async Copy Engine, Hyper-Q Enabled
RAM	32 GB DDR3
Serial Computer Expansion Bus	PCIe 2.1
CUDA Toolkit	6.0
CUDA Driver Version	6.0
CUDA Run time Version	6.0
Open-CV Version	2.4.9
Open-MP Version	2.0

Table 5-1 System Specs of the Test Platform

On the host (CPU) side, the C code shown in Table 5-2 is used to measure the elapsed time as it is one of the most precise techniques on Windows platforms.

As for the device (GPU) side, CUDA events and streams are used to measure the performance as show in Table 5-3. This piece of code and explanation of it could be found in Chapter 3 of [26] (In CUDA, events are the best timers, for asynchronous calls sent to GPU, in particular).

```

void startCounter(__int64 *counter_start, double *pc_freq) {

    LARGE_INTEGER li;
    if(!QueryPerformanceFrequency(&li))
        std::cout<<"QueryPerformanceFrequency Failed!"<<std::endl;

    *pc_freq = ((double)li.QuadPart)/1000.0;

    QueryPerformanceCounter(&li);
    *counter_start = li.QuadPart;
}

double getCounter(__int64 *counter_start, double *pc_freq) {

    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return ((double)(li.QuadPart - (*counter_start))) / (*pc_freq);
}

```

Following shows how they are used in order to measure elapsed time accurately:

```

startCounter(&counter_start, &pc_freq);
//Piece of Code to Measure Execution Time
//Code to Test Code to Test Code to Test...
//Code to Test Code to Test Code to Test...
time = getCounter(&counter_start, &pc_freq);

```

Table 5-2 Method Used to Measure Time on Host Side

```

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, stream);
//Piece of Code to Measure Execution Time
//Code to Test Code to Test Code to Test...
//Code to Test Code to Test Code to Test...
cudaEventRecord(stop, stream);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);

```

Table 5-3 Method Used to Measure Time on Device Side

In Figure 5-1, a sample frame from the test video in where we tracked a bottle of water could be seen.



Figure 5-1 A Sample Screen-shot Captured from the Test Video

5.1 Detection

5.1.1 Integral Image (II) Computation on GPU

In detection module, IIs are used to compute PV of each BB. Since PV-Computation takes place on GPU; IIs must reside on GPU as well. Moreover, II-Computation fits well to GPU architecture. Thus, once the current frame is moved to GPU; two IIs (plain II and squared II) could be calculated. Finally, it would also save the time to move both IIs to device provided that IIs were calculated on CPU. NPPI Library [1] has a method which calculates both at once and was mentioned in the beginning of Chapter 4.

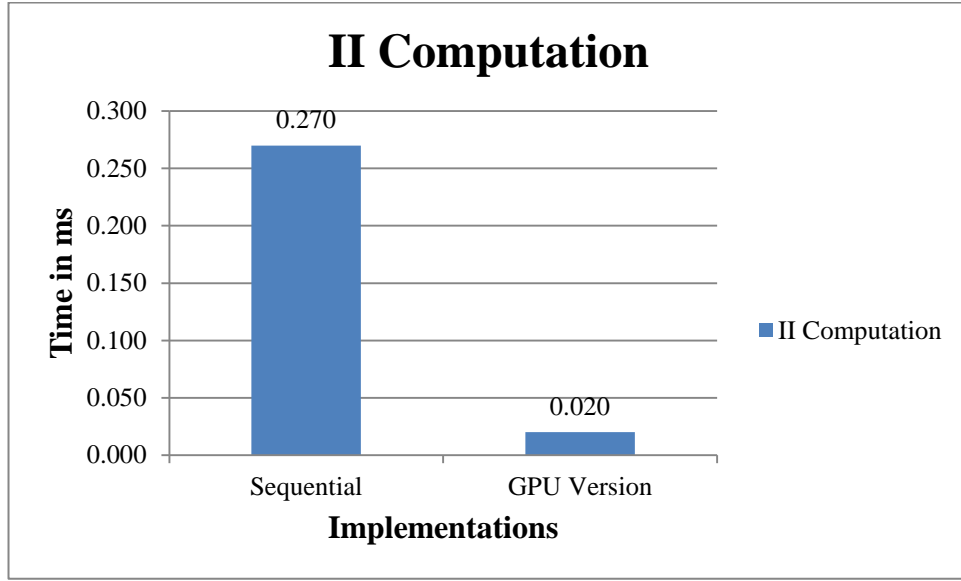


Figure 5-2 Average Time per Call for II-Computation

In Figure 5-2 average time per call for each method is displayed (transfer time for current frame is not included). NPPI's method is not hybrid. It is only run on device's processing units. Equation (5-1) shows speed-up.

$$(0.271/0.02) = 13.550X \quad (5-1)$$

Note that this method is called per frame; it has a significant effect on the overall performance.

5.1.2 Image Blurring (Open-CV Used)

A method from Open-CV [19] has been used to increase the speed of process of blurring the current frame. This blurred frame is required at the time when pixel comparisons (referred in Chapter 5 of [2]) are made. However, this comparison was not fair on sequential code's behalf due to running MATLAB's method to blur the current frame (After a process is created for MATLAB, MATLAB creates a virtual container, a JVM, in order to execute its m files). Yet the comparison of MATLAB's method used in [2] and the one from [19] which is used in H-TLD is shown in Figure 5-3 (transfer time for current frame is not included) and Equation (5-2).

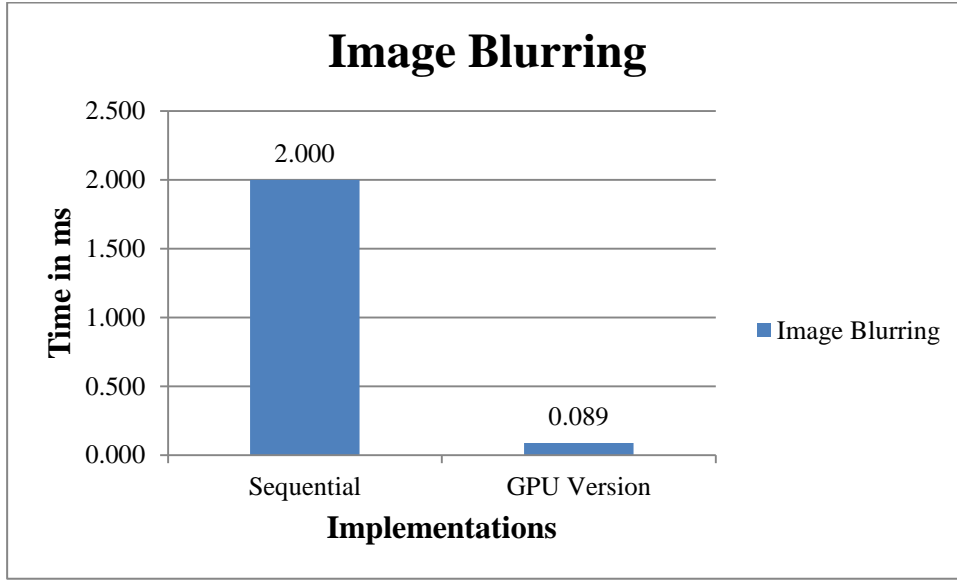


Figure 5-3 Average Time per Call for Image Blurring

$$(2/0.0894) = 22.371X \quad (5-2)$$

The time on MATLAB side was measured via “tic-toc” mechanism provided by the MATLAB platform.

5.1.3 Total Recall Computation on CPU and GPU Collaboratively

“Total Recall Computation” is a hybrid implementation; hence both CPU and GPU are used in various stages of it accordingly. In this sub section firstly, total improvement over serial code is given in order to highlight significance of our work and importance of use of heterogeneous computing. Then, each stage is observed separately to show the fact that how much time of it is spent on which processing unit and the rest of it is on data communication between those two different processing units.

In Figure 5-4 (all transfer time between host and device for hybrid implementation is included) and Equation (5-3), overall improvement could be seen.

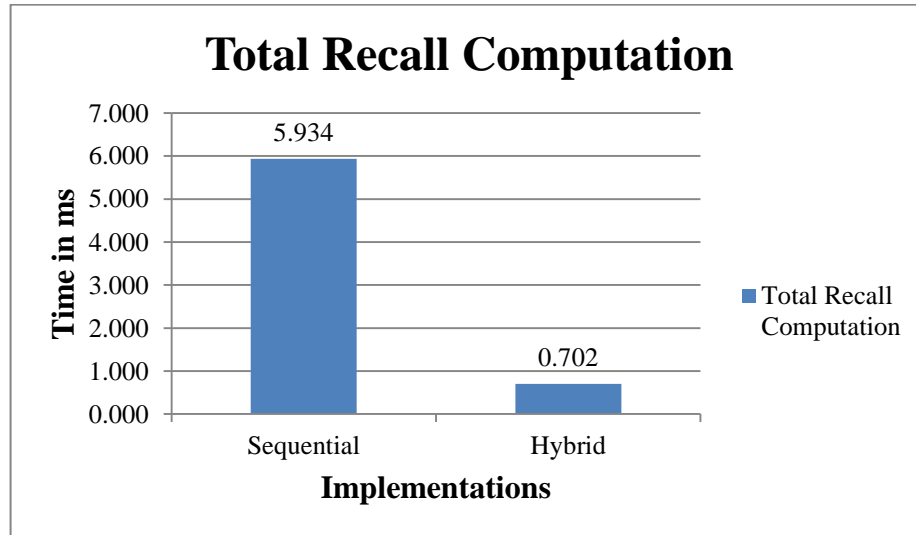


Figure 5-4 Average Time per Call for Total Recall Computation

$$\left(\frac{5.934}{0.702} \right) = 8.453X \quad (5-3)$$

As explained in Chapter 4, “Total Recall Computation (TRC)” is composed of several stages. In Table 5-4, each of those stages is analyzed individually. Note that, the total elapsed time when each part is measured separately and summed up (6.416 milliseconds); and when all steps are measured at once (5.934 milliseconds) are different; because there is an overhead that the timer itself creates.

Stage	Sequential (ms)	Hybrid (ms)
PV-Computation	0.170	0.0459 (GPU)
Stream-Compaction	-	0.0463 (GPU)
RFI Calculation	5.296	0.320 (GPU)
Confidence Calculation	Value 0.950	0.138 (CPU)
Other Operations	-	0.152(CPU & GPU)
Total	6.416	0.721

Table 5-4 Elapsed Time for Each Stage of TRC

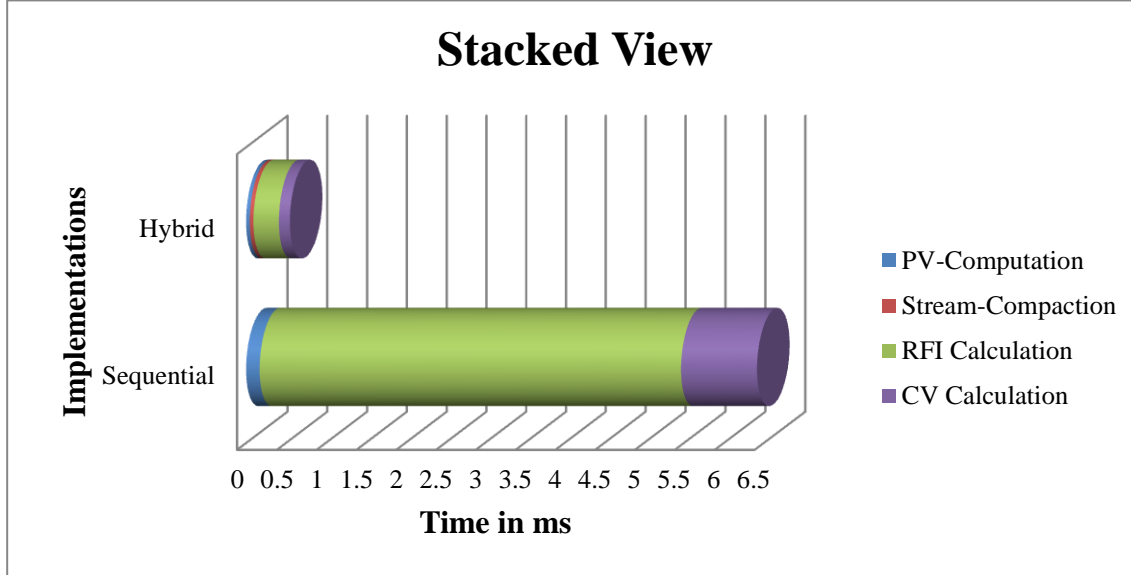


Figure 5-5 Stacked View for Elapsed Time of TRC

As it can be seen in Figure 5-5 much of the elapsed time is spent on “RFI Calculation” for both implementations. Since RFIs are required in “CV Calculation” phase; some data have to be transferred to host side per asynchronous kernel invocation at this stage (As it was shown in Figure 4-4, on line 9, RFI Calculation is repeated for number of asynchronous calls to PV-Computation; because at each loop cycle RFIs of BBs, that have passed the PV-Test, are found). Moreover, this elapsed time increases as the number of BBs (for instance, when the display resolution is increased) gets higher. This is formulated in Equation (5-4), and it is also equal to the size of data to move to host side in bytes.

$$f(x,y) = \text{sizeof(int)} * x * y \quad (5-4)$$

Where x is equal to the number of trees, and y is equal to the number of BBs that have passed PV-Test.

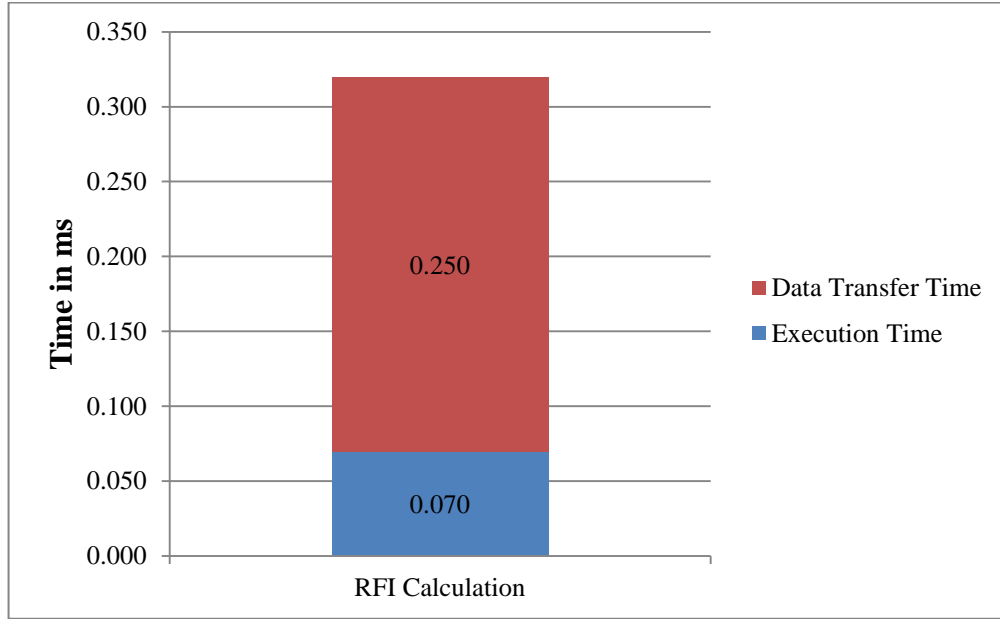


Figure 5-6 Data Transfer and Execution Time for RFI Calculation

As it is shown in Figure 5-6, data transfer takes ~78 % of “RFI Calculation”. Besides, this latency cannot be completely hidden; though there are some instructions which are executed at the time when RFI-Calculation takes place (they are not so computationally intensive or do many memory transactions that they might cancel it out). Since RFIs are used to calculate confidence values of BBs; all threads must be suspended until the transfer is completed. This issue is discussed in Chapter 6 and a persistent solution at hardware level for next generation CUDA-Enabled Devices is proposed.

5.2 Tracking Module

In tracking module only the optical flow calculation that is used in median flow tracker [12] to predict the next locations of good features (points) of the object under observation, was accelerated via GPU as was discussed in Chapter 4.

5.2.1 Optical Flow

LK-Tracker, which was implemented on GPU, is part of Open-CV [19]. It was shown that it is highly efficient only under the condition that the frame size is big enough to highly occupy the GPU cores [27]; otherwise CPU implementation (Open-

MP [21] is enabled) may outrun it. This phenomenon of massively parallel architectures applied to the case in which we track reliable points from previous frame to the next frame as well and is shown in Figure 5-7.

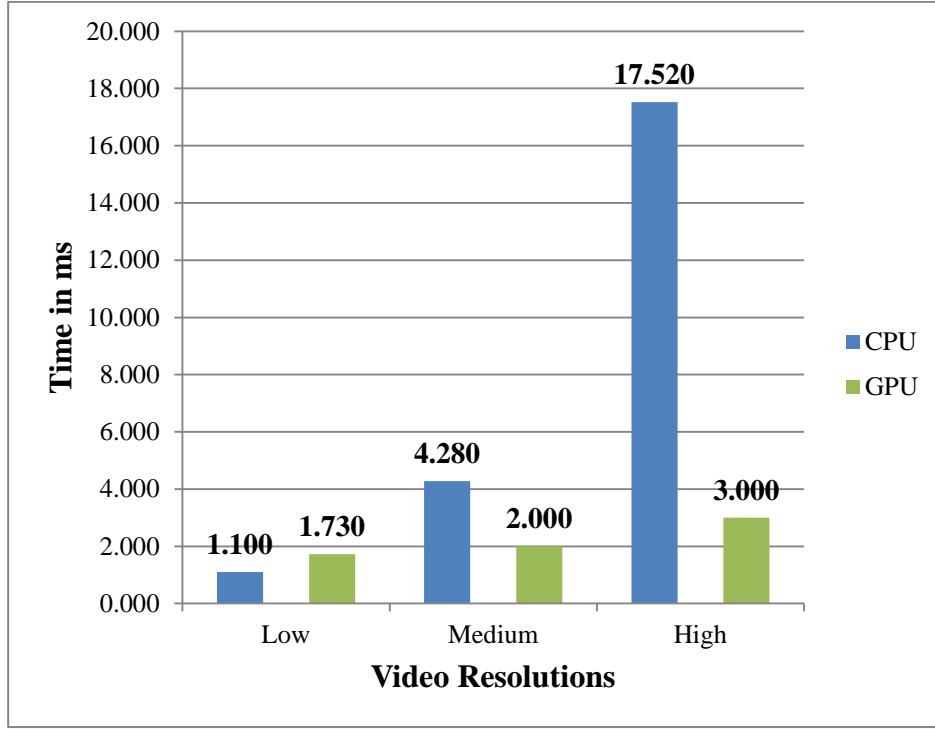


Figure 5-7 LK-Tracker GPU vs. CPU Implementations

Although GPU implementation is slower for videos that have low resolution, it is not significantly so bad and GPU becomes more advantageous for higher resolution videos.

$$(1.730/1.100) = 1.573X \quad (5-5)$$

$$(4.280/2.000) = 2.140X \quad (5-6)$$

$$(17.520/3.000) = 5.840X \quad (5-7)$$

As proved in Equations (5-5), (5-6), and (5-7) (speed-up of CPU for the low resolution video and of GPU for the medium and high resolution videos respectively), there is no need to add an additional complexity to the algorithm for checking size of the frame against the performance of the platform on which the application runs; then deciding on either optical flow should be executed on CPU or on GPU. In conclusion, H-TLD only runs its GPU version regardless of frame size and quality of the hardware.

5.3 Effect of Different Display Resolutions on the Performance

All tests so far were conducted on the video stream with relatively low display resolution. However, today's video capturing devices are able to capture increasingly higher resolutions. On the other hand, with the increasing video resolutions, the detection module should scan a much higher number of BBs, an II and a blurred image with larger size should be calculated, the tracking module should track more points when it transitions from one frame to another, etc. which in turn, requires more computation power preventing real-time operation. Thus, a great deal of processing power is required to run real time tracking applications with multiple objects being observed simultaneously.

In this sub section, the aim is to demonstrate the higher speed-up results obtained ineffective as the display resolution increases. There are 3 different resolutions called low, medium, and high in our test scenario. Resolutions are:

- 480x270: Low (1X),
- 960x540: Medium (4X),
- 1920x1080: High (16X).

This effect on detection module's TRC task (in where slow-down is the most immense and to which we focused on improving its execution time in Chapter 4) is shown in Figure 5-8.

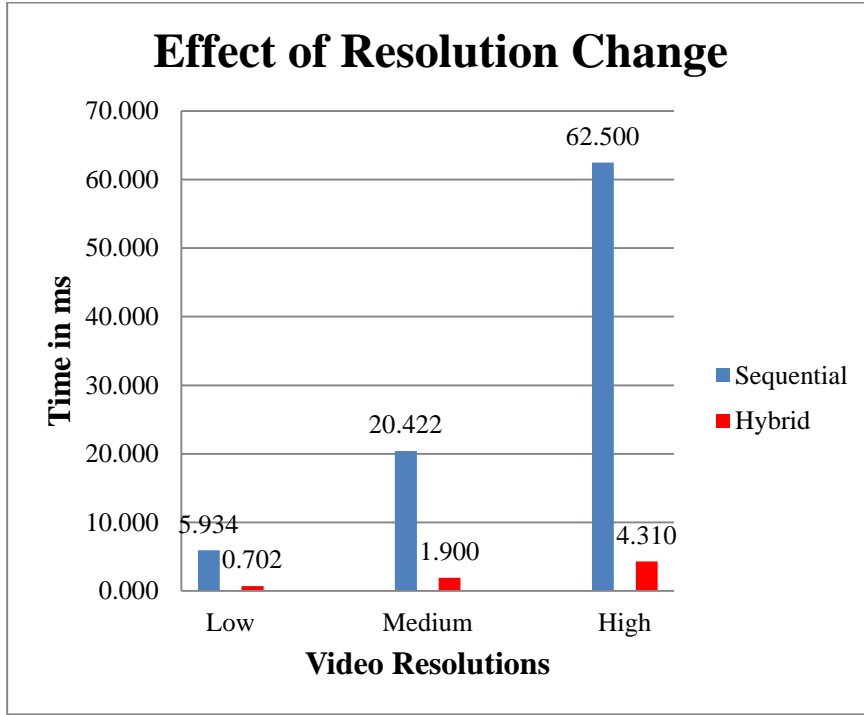


Figure 5-8 Effect of Resolution Change on TRC

It can be observed in Figure 5-8, that the gap between sequential and hybrid implementations get bigger as display resolution increases.

$$\left(\frac{5.934}{0.702} \right) = 8.453X \quad (5-8)$$

Gain for Low Resolution Video

$$\left(\frac{20.422}{1.900} \right) = 10.748X \quad (5-9)$$

Gain for Medium Resolution Video

$$\left(\frac{62.500}{4.310} \right) = 14.501X \quad (5-10)$$

Gain for High Resolution Video

Gains shown in Equations (5-8), (5-9), and (5-10) proves our hypothesis which claims that H-TLD will be faster as the resolution increases. Important reason for this significant improvement is that the occupancy of GPU cores is boosted due to having more pixels to be processed, and overlapping behavior of multiple kernels as

discussed in subsection 5.4. Even if there had been no overlapped kernel invocation, there would have been an increase in occupancy; but not as much as it is shown in this thesis.

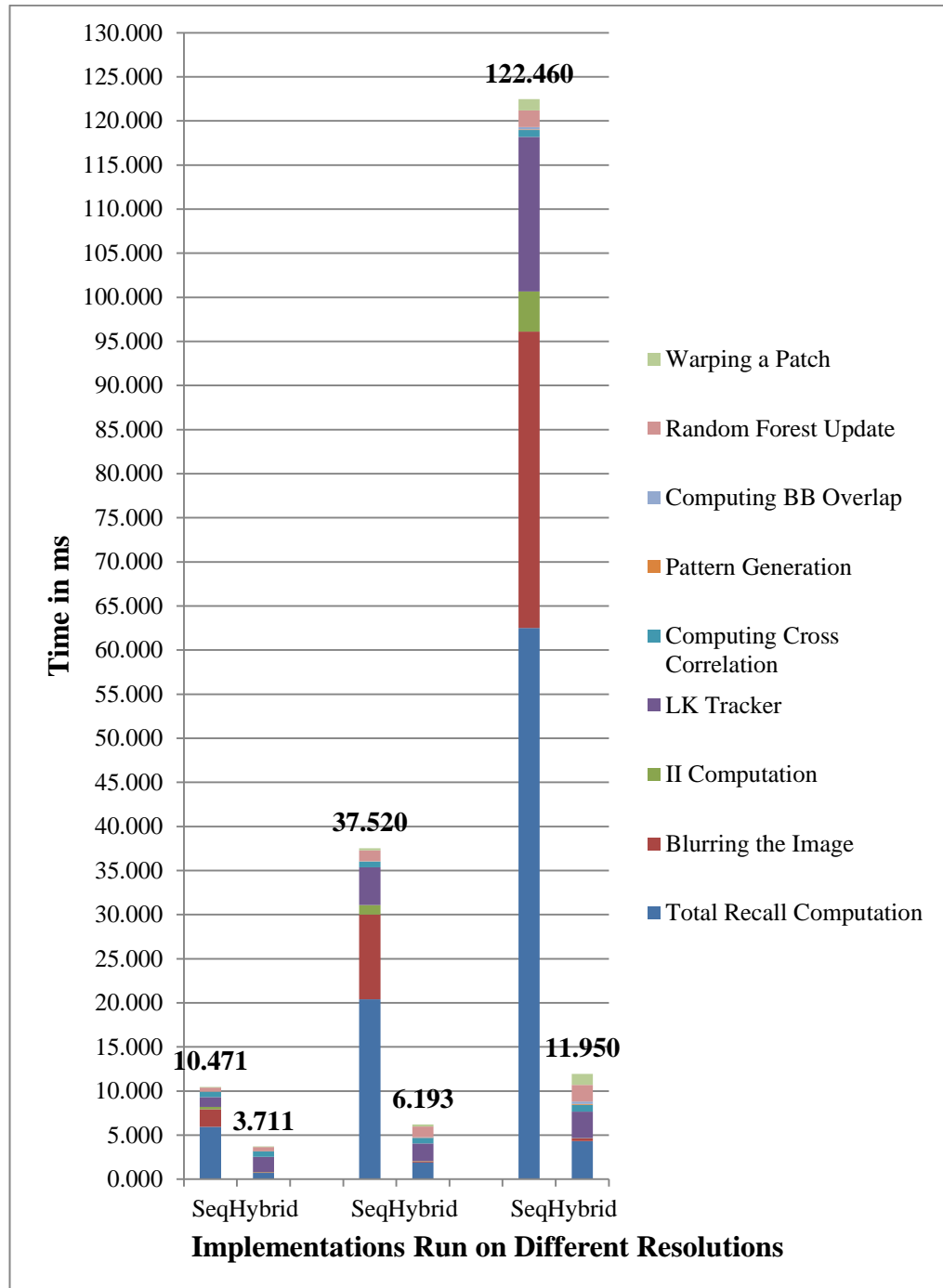


Figure 5-9 Total Gain of H-TLD

In Figure 5-9, overall speed-up including all tasks mentioned in Chapter 3 could be seen. It was formed by stacking up elapsed time of each individual task; because that is the only option in this thesis until the whole platform is moved to the native side as is explained in “Discussions” part of this section. As is the case with TRC, total speed-up increases with the increasing resolution. As for the highest resolution, speed-up reaches to a level at which hybrid implementation is ~10.248 times faster than the sequential one.

5.4 NVIDIA’s Visual Profiler & Overlapping Data Transfer and Kernel Executions

The NVIDIA Visual Profiler [28] is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. It displays the whole timeline of an application’s CPU and GPU activities. It does not only help developers on GPU side; but on CPU side as well (like displaying for how much time the display driver keeps the instruction that should be run on GPU in the queue, before it flushes the instruction out, etc.). It also allows developers to monitor any process; in turn they can specify the path to the executable file, and visualize its activity (For instance, in this sub section MATLAB’s JVM process is monitored to capture all GPU related activities; rather than running an entirely native process). Finally, in case of any unexpected behavior, it is easy to figure out what went wrong. In Figure 5-10, TimeLine-View (it is one of many different types of views of the profiler) is shown. It displays all operations taking place in a CUDA Context separately such as “MemCpy”. As seen in that figure, none of the vector operation overlaps with any other or with any memory transaction (one starts right after another one is completed).

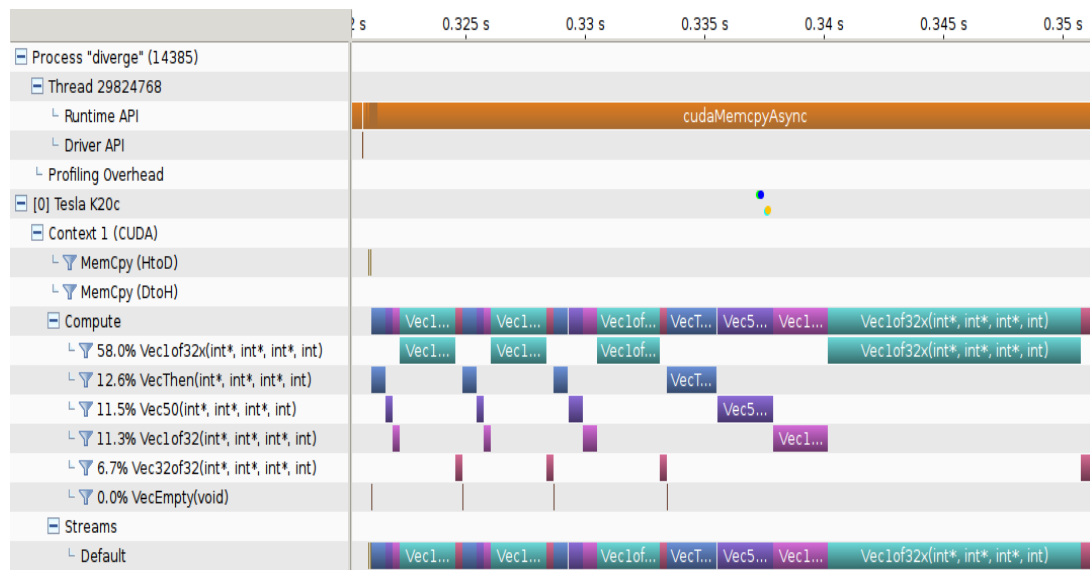


Figure 5-10 Timeline View of Visual Profiler

In Figure 5-11, a fraction of profiling TRC via NVIDIA Visual Profiler is shown. There are three important observations that could be inferred from Figure 5-11:

- Copying the data back to the host and RFI Calculation were overlapped.
- RFI Calculation takes long enough to hide the total latency for copying BB shift amounts which were computed in stream-compaction phase mentioned in Chapter 4.
- Copying RFIs back to host takes a long time.
- If occupation of SMs are low at any time when two or more independent kernels are invoked, it is also be possible to observe some kernel invocations to be overlapped with each other (requires compute capability higher than or equal to 2.x). In Figure 5-12 this behavior is proved to be correct for the case in which PV-Computation and Stream-Compaction take place.
- By this way; in the case occupancy of SMs is low, it is possible to boost up GPU cores in order to overcome the issue of idly stayed cores. Thus, it results in accelerating the overall execution time of the application. The key to overlap multiple independent kernel invocations is to cluster all independent calls for an individual kernel (e.g. computePVOnGPU kernel) in all streams

and issue them to GPU rather than issuing them to GPU in such a way that they are interleaved with other calls in all those streams.

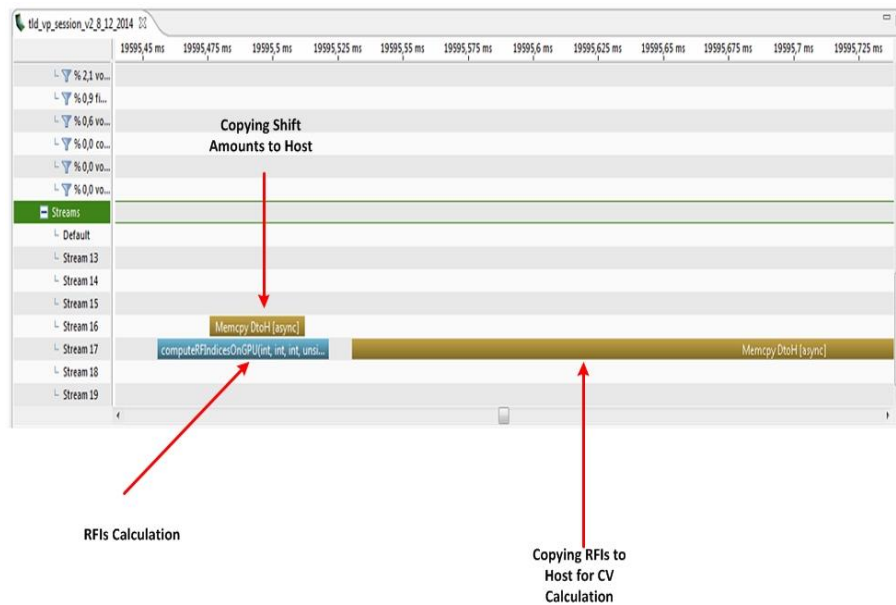


Figure 5-11 Visual Profiler Timeline View for TRC

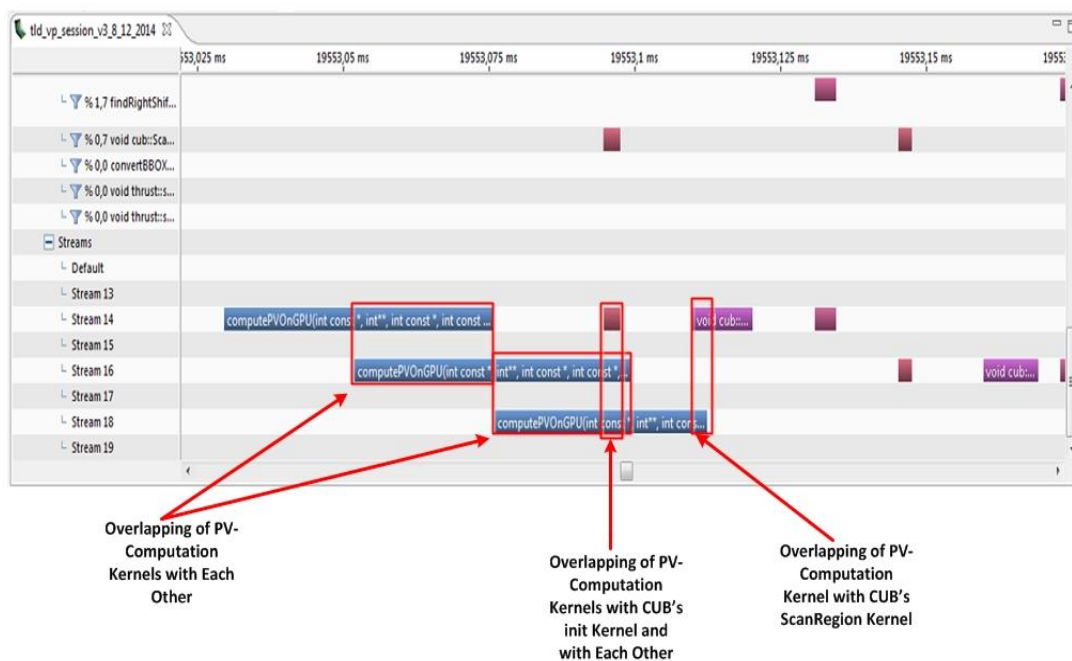


Figure 5-12 Timeline That Shows Overlapping Behavior of Multiple Kernel Invocations

```

for(int i = 0; i<num_of_async_calls; i++) {
    computePVOnGPU(i);
    doStreamCompaction(i);
    copyDataAsync(i);
}

```

Table 5-5 Wrong Call Order for GPU to Overlap Kernel Invocations

```

for(int i = 0; i<num_of_async_calls; i++)
    computePVOnGPU(i);
for(int i = 0; i<num_of_async_calls; i++)
    doStreamCompaction(i);

for(int i = 0; i<num_of_async_calls; i++)
    copyDataAsync(i);

```

Table 5-6 Correct Call Order for GPU to Overlap Kernel Invocations

In Table 5-5 and Table 5-6 this simple; but powerful technique which was exploited in this thesis' implementation to improve execution time of [2] is shown.

5.5 Discussions

In this sub section, difficulties we encountered during the implementation and how we solved them are given; so that the future developers will not make same mistakes or will have to deal with the same problems.

Today's compilers are smart enough to remove unused piece of codes; therefore developers might be tricked into the idea that their codes for which they will test the performance on are present in the source code; whereas their machine-level instruction equivalents might actually not be in the executable file. A fraction of code taken from [2]'s detection module on native side and shown in Figure 5-13 is a good example to this case.

```

16 while(1) {
17     // Get index of bbox
18     I = (int) floor(pState);
19     pState += pStep;
20     if (pState >= nBBOX) { break; }
21     bboxvar = bbox_var_offset(IIMG,IIMG2,BBOX+I*BBOX_STEP);
22     if(bboxvar >= minVar) {
23         //         cnf         = 0;
24         //         double *tPatt = patt + nTREES*I;
25         for (int i = 0; i < nTREES; i++) {
26             //Start Measuring Time Right Here...
27             Utilities::startCounter();
28             idx = measure_tree_offset(blur,I,i);
29             //Total Time to Compute Total Recall...
30             total_time4 += Utilities::getCounter();
31             //         tPatt[i] = idx;
32             //         cnf         += WEIGHT[i][idx];
33         }
34         //         conf[I] = cnf;
35     }
36 }
37 count4++;

```

Figure 5-13 Tricky Code for a Compiler

Suppose that “RFI Calculation” (“measure_tree_offset” method in Figure 5-13 does exactly this calculation) for sequential code is desired to be measured and no other instruction is to interfere with this execution. Logically, Commenting out the lines 31, 32, and 34 could make sense. However when the result is obtained after application is run and ~0 MS (note that, it does not matter whether video with high or low display resolution was used in this experiment) is written out in the console, it might be confusing for the developer (because the elapsed time to compute RFIs on GPU without data transfer time is included was equal to ~0.07 MS). Since the output of this method is never used by any following instruction; the compiler automatically removes it in order to optimize the code. This is called “dead code removal”. Although many advanced IDEs (Integrated Development Environment) such as Eclipse, Visual Studio 2010+, warn developers about the dead code at development

time; some others like MATLAB's code editor does not warn. Thus, extra caution must be taken before the compiler is run.

Secondly, the test platform was a combination of implementations that is; it had been implemented in MATLAB's script language and C/C++. Since MATLAB creates the process which our application lives in, that script is the client who calls C/C++ routines via "mex". "mex" is a built-in interface between MATLAB and subroutines written in C/C++ or Fortran. It acts like a bridge between C/C++ native executable code and MATLAB script code. This plugin allows MATLAB users to leverage any library written in C/C++. It may also compile C/C++ code by the compiler and linker tools provided by the OS which MATLAB runs on. However, this cross platform routine calls bring extra overhead to the application. Moreover, modules in [2] are separate entities, which means that their native instructions are executed as if they were living in individual processes; because there is no state-sharing notion (data exchange) between different executable "mex" files. Thus, when H-TLD creates its master module in detection module, it is not possible to share the reference of that C++ object with tracking module and vice versa. During the test phase, each module was tested one by one. That is to say, when there was an instance of master module in detection module; another instance of it in tracking module was commented out in order to come up with accurate results. This issue is addressed in Chapter 6.

Finally, all developers should be aware of the Windows Display Driver Model (WDDM) software on Windows Vista and successor platforms as well as other corresponding software on UNIX and other platforms; because it might be very difficult to realize why commands to GPU are not flushed immediately; but waiting in a queue instead. It could be seen when it is checked out on activity view of [29]. WDDM is a replacement for the Windows XP display driver model and is aimed at enabling better performance graphics and new graphics functionality. Display drivers in Windows Vista and later can choose to either adhere to this model or to Windows 2000 Display Driver Model (XDDM). With the removal of XDDM from Windows 8, however, WDDM became the only option. The problem is that WDDM decides on

criticality of a command and may issue CUDA commands to GPU later than they were expected to be done. Thus, if WDDM is to be skipped in order to see the actual result of any CUDA-based implementation; Tesla Compute Cluster (TCC) driver is the only option. The defect of this driver is that it is only compatible with very few GPUs. These kinds of GPUs have no VGA output and they are called “computing boards” rather than GPUs. They are specialized for only GP-GPU programming.

$$\sigma = 1/nf_v \times \sum_i^{nf_v} \text{dist}(cph_i, cpo_i) \quad (5-11)$$

Where nf_v is equal to the number of frames on which the object is visible in both implementations, cph_i and cpo_i are the center points of object’s BB on the i^{th} frame for hybrid and original implementations respectively, and σ is the deviation of center point of the object’s BB defined by the hybrid implementation from the one defined by the original implementation.

Implementation	Visible →	Obscured →	Visible →	Obscured	σ
Original	0-253	254-301	302-461	462-463	15.682px
Hybrid	0-254	255-299	300-461	462-463	

Table 5-7 Object Detection Sequence against the Medium Size Video

In Equation (5-11) and Table 5-7, how much close detections done by both implementations to each other are shown. In other words, how seriously causes such as floating-point precision of mathematical calculations, extra data type conversions for moving data from one memory location to another, etc. affected the accuracy of tracking, resulting in divergence from the center point of the object’s BB detected by the original implementation. Note that, our initial purpose was not to improve the quality of tracking via modification of the original TLD algorithm (it is TLD

algorithm's concern); hence we did not compare our results with the ground truth values; rather we checked our own fast implementation against the original sequential implementation in order to find whether the implementation differences had caused any abnormality in tracking. As shown in Table 5-7, the sequence is as follow: object is initially visible, then obscured by another object, then visible once again, and finally became hidden for a few numbers of frames. The object's size is equal to $\sim 47.790 \times 129.353$ px on the average throughout the whole sequence of frames; and deviation (σ) shown in Table 5-7 is equal to 15.682px. The reasons for this are that the blurring image method used in H-TLD is different than the one implemented in [2], and the optical flow method used in H-TLD produces slightly different reliable points. Moreover, when object (water bottle) is rotated BBs are not reliable.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this section, future works and expectations, and tasks that have been accomplished in this thesis are mentioned.

As seen in Figure 5-5, ~45% of TRC's time is spent on RFI Calculation; and moreover ~80% of the RFI Calculation's time (see Figure 5-6) is lost in moving those RFIs to host side for CV-Calculation. If this data transfer had been removed from this equation, speed-up given in Equation (6-1) would have been achieved instead of the one we obtained in Equation (5-3).

$$\left(5.934 / (0.702 - 0.25)\right) = 13.128X \quad (6-1)$$

In a GPU Tec. conference (mentioned in [30]), NVIDIA announced the successor of Maxwell GPUs. This new GPU family will be named as "Pascal". As written in [30], a new type of memory called "stacked DRAM or 3D memory" (along with many other new features) will be introduced. By the aid of this new memory model, GPUs will achieve terabytes of bandwidth (several times greater than what Maxwell family has). This new technique will allow GPUs to gain access to data residing on RAM almost as fast as CPUs can do. As a result of this, H-TLD's bottleneck at where data transfer cannot be hidden will be overcome.

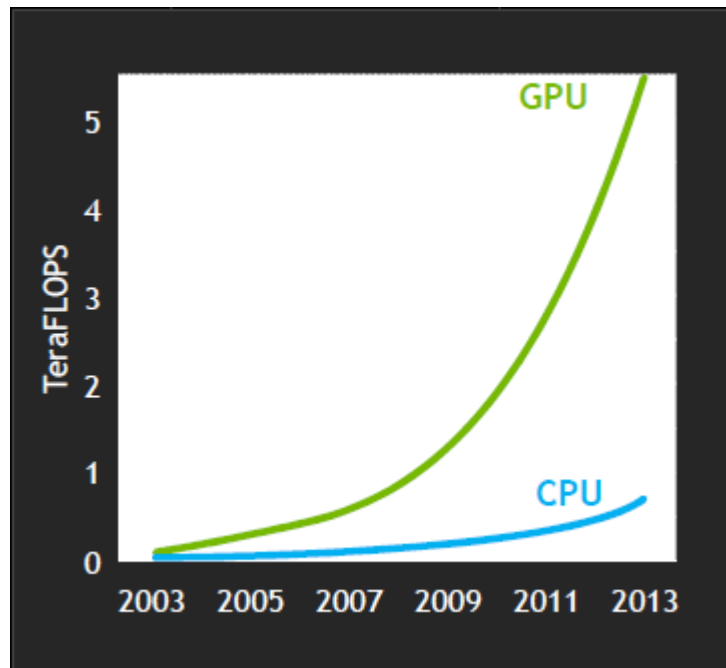


Figure 6-1 Change in Computation Power of CPU vs. GPU over Years

In Figure 6-1, in 2013 GTC (GPU Technology Conference); how computation power of GPU has excessively increased in time compared to the one of CPU. Note that, this processing power might only be harnessed provided that your algorithm and amount of data cause GPU to be highly occupied. Since the quality of videos are getting higher; eventually TLD will require more processing power to run at higher or at least exact same quality as it used to do now. Besides, this trend also proves that the speed-gain of H-TLD will have increased; even though no improvement is made to it.

In order to find out the total speed-up (it was mentioned in the discussion part of Chapter 5); all sequential code written in MATLAB's script language should be moved to the native side coded in C/C++ language. By this way, one instance of master module could be created and its reference could be shared across all modules. Besides, it will help the application in eliminating all those unnecessary overhead

created by MATLAB's JVM container and MEX's cross platform calls. Since our purpose in this thesis is to prove the speed-up against the sequential code written on native side (and is called by MEX engine), we were able to measure the performance without any interference of MATLAB's script language (only performance metrics for blurred image given in Chapter 5 is an exception to this). It is even difficult to differentiate the speed-up by the naked eye due to running tests on this mix platform; because videos with higher display resolutions cause so much speed loss, owing to the bidirectional data transfer between these native and script codes, that it suppresses the visual perception of observer.

Another future task is to figure out the reason why the quality of tracking is reduced as the display resolution increases. For video from our experiment dataset with the resolution of 1920x1080, Open TLD's tracking trajectory is broken for some time (for both sequential and hybrid implementations) and it restarts to track object. There are some assumptions we made such as "Since a fast movement of object from one frame to another may cause tracker to fail due to the nature of optical flow technique called "Pyramidal Lucas-Kanade" [15], hence for videos with higher resolutions, more pyramidal levels might be required in order to detect such fast changes". Consequently, a series of experiments should be conducted to figure it out.

As for the last future task, TLD [2] is not sensitive enough to morphological changes occurred in object under observation. For instance, whenever object is rotated around the axis other than the z-axis that stretches back and forth in the direction of camera's depth, it fails to learn variety of new appearances of the object. Assuming that the camera may capture enough information (frames) for such transformations, a computationally expensive; but effective learning algorithm that will be implemented on a hybrid platform, may be integrated to TLD to make it realize such changes in object's shape.

In Chapter 1 some goals were defined. It was proven that they can be achieved after serial code is fully moved to the C-based native side. The purpose of TLDObject was to enable the application to track multiple objects; so that the serial code that will be

written in C/C++ can exploit this fact. All results, given in Chapter 5, show that many per frame operations which require heavy processing power (see Chapter 3) were accelerated. As a consequence of these improvements;

- New hybrid algorithm developed in this thesis was tested with different resolutions; it was shown that it could be used with videos which have higher display resolutions (see sub section 5.1.3 in Chapter 5),
- Our TLDOject notion will provide application developers using H-TLD with the tracking of multiple objects within a single video stream,
- The time that was bought by H-TLD could be used to tune the configuration parameters of Open TLD; hence it will result in better tracking quality.

REFERENCES

- [1] NVIDIA Performance Primitives. [Online]. <https://developer.nvidia.com/npp>
- [2] Z. Kalal, K. Mikolajczyk, and J. Matas, "Tracking-Learning-Detection," *IEEE Transactions On Pattern Analysis and Machine Intelligence*, vol. 34, no. 7, pp. 1409-1422, July 2012.
- [3] S. Rennich. (2011) On Demand GPU Tech. Conf. [Online]. <http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf>
- [4] G. Nebehay, "Robust Object Tracking Based on Tracking-Learning-Detection," Technische Universität Wien, May 2012.
- [5] M. Isard and A. Blake, "CONDENSATION—Conditional Density Propagation for Visual Tracking," *International Journal of Computer Vision*, vol. 29, no. 1, pp. 5-28, 1998.
- [6] D. Comaniciu, V. Ramesh, and P. Meer, "Real-time tracking of non-rigid objects using mean shift," in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 2000, pp. 142-149.
- [7] V. Lepetit and P. Fua, "Monocular Model-Based 3D Tracking of Rigid Objects: A Survey," *Foundations and Trends in Computer Graphics and Vision*, vol. 1, no. 1, pp. 1-89, 2005.
- [8] M. Ozuysal, P. Fua, and V. Lepetit, "Fast Keypoint Recognition in Ten Lines of Code," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, June 2007, pp. 1-8.
- [9] R. T. Collins, Y. Liu, and M. Leordeanu, "Online Selection of Discriminative Tracking Features," *IEEE Transactions on Pattern Analysis and Machine*

- Intelligence*, vol. 27, no. 10, pp. 1631-1643, October 2005.
- [10] O. Chapelle, B. Schölkopf, and A. Zien, *Semi-Supervised Learning*., 2006.
 - [11] Z. Kalal, J. Matas, and K. Mikolajczyk, "P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, June 2010, pp. 49-56.
 - [12] Z. Kalal, K. Mikolajczyk, and J. Matas, "Forward-Backward Error: Automatic Detection of Tracking Failures," in *Pattern Recognition (ICPR), 2010 20th International Conference on*, 2010, pp. 2756-2759.
 - [13] E. Ringaby, "Optical Flow Computation on Compute Unified Device Architecture," 2008.
 - [14] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-Time Dense and Accurate Parallel Optical Flow using CUDA," 2009.
 - [15] J. Bouguet, "Pyramidal Implementation of the Lucas Kanade Feature Tracker," Interl Corporation,.
 - [16] P. Guler, D. Emeksiz, A. Temizel, M. Teke, and T. Taskaya Temizel, "Real-time Multi-Camera Video Analytics System on GPU," *Journal of Real Time Image Processing*, March 2013.
 - [17] S.A. Mahmoudi, M. Kierzynka, P. Manneback, and K. Kurowski, "Real-time Motion Tracking Using Optical Flow on Multiple GPUs," *Bulletin of the Polish Academy of Sciences*, vol. 62, no. 1, pp. 139-150, 2014.
 - [18] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *4th Alvey Vision Conf.*, 1988, pp. 147-151.
 - [19] OpenCV Community. OpenCV Documentation. [Online].

<http://docs.opencv.org/modules/gpu/doc/introduction.html>

[20] Anonymous. Wikipedia. [Online].

http://en.wikipedia.org/wiki/Summed_area_table

[21] Open MP Org. Open MP. [Online]. <http://openmp.org/wp/>

[22] NVIDIA. CUDA Compute Capabilities. [Online].

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

[23] D. B. Kirk and W. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach.*, 2012.

[24] NVIDIA CUB. [Online]. <http://nvlabs.github.io/cub/>

[25] T. Kanade and B. D. Lucas, "An iterative image registration technique with an Application to Stereo Vision," in *Proc 7th Intl Joint Conf on Artificial Intelligence*, 1981, pp. 674-679.

[26] CUDA Toolkit Documentation. [Online]. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz39mQ0rc3K>

[27] NVIDIA. (2014) CUDA Best Practices Guide. [Online].

<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#occupancy>

[28] NVIDIA Visual Profiler. [Online]. <http://docs.nvidia.com/cuda/profiler-users-guide/index.html#axzz39tFYHyEs>

[29] NVIDIA Nsight. [Online]. <http://www.nvidia.com/object/nsight.html>

[30] NVIDIA Blog. [Online]. <http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/>