# PARALLELIZATION ANALYSIS OF ECO TRACKING ALGORITHM ON GPUS

# ECO İZLEME ALGORİTMASININ GPU'LARDA PARALELLEŞTİRME ANALİZİ

**UĞUR TAYGAN**

**ASST. PROF. DR ADNAN ÖZSOY**

**Supervisor**

Submitted to

Graduate School of Science and Engineering of Hacettepe University

as a Partial Fulfillment to the Requirements

for the Award of the Degree of Master of Science

in Computer Engineering

2020

This work titled **“PARALLELIZATION ANALYSIS OF ECO TRACKING ALGORITHM ON GPUS”** by **UĞUR TAYGAN** has been approved as a thesis for the Degree of **MASTER OF SCIENCE** in **COMPUTER ENGINEERING** by the Examining Committee Members mentioned below.

Prof. Dr. Mehmet Önder EFE

Head ..……

Asst. Prof. Dr. Adnan ÖZSOY

Supervisor ..……

Prof. Dr. Süleyman TOSUN

Member ...……..

Prof. Dr. Alptekin TEMİZEL

Member ……..

Assoc. Prof. Dr. Ahmet Burak CAN

Member …….……

This thesis has been approved as a thesis for the Degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING** by Board of Directors of the Institute of Graduate Studies in Science and Engineering on ..... / ..... / .....

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU

Director of the Institute of Graduate
School of Science and Engineering

# ABSTRACT

## PARALLELIZATION ANALYSIS OF ECO TRACKING ALGORITHM ON GPUS

**Uğur TAYGAN**

**Master of Science, Computer Engineering Department**

**Supervisor: Asst. Prof. Dr. Adnan ÖZSOY**

**June 2020, 48 pages**

Object tracking is a very popular area in image processing. Its popularity comes from the variety of its application areas. It is used for security and surveillance, autonomous vehicles, human-machine interaction, traffic control and so on. Due to its application areas, an object tracking algorithm is usually expected to be fast. On the other hand, an object tracking algorithm should be accurate and robust and this usually increase the amount of calculations to be done. The nature of the many image processing applications are suitable for parallel programming. Since, GPUs consist of large number cores, they are widely used in image processing and object tracking applications. In this thesis, we analyze an object tracking algorithm for its suitability of parallelism. We detected the time-consuming parts of the algorithm by using profiling tool. Each part of the algorithm is handled separately and implemented on GPU. Additionally, we have worked on the chances of optimization by using GPU capabilities. We compared our methods with the original parts of CPU based approach by testing them on five datasets.

**Keywords:** Object tracking; image processing; computer vision; GPU, parallel computing

# ÖZET

## ECO İZLEME ALGORİTMASININ GPU'LARDA PARALELLEŞTİRME ANALİZİ

**Uğur TAYGAN**

**Yüksek Lisans, Bilgisayar Mühendisliği Bölümü**

**Tez Danışmanı: Dr. Öğr. Üyesi Adnan ÖZSOY**

**Haziran 2020, 48 sayfa**

Nesne izleme, görüntü işlemede çok popüler bir alandır. Popülerliği, uygulama alanlarının çeşitliliğinden kaynaklanmaktadır. Güvenlik ve gözetim sistemleri, otonom araçlar, insan-makine etkileşimi, trafik kontrolü gibi alanlarda kullanılmaktadır. Uygulama alanları nedeniyle, bir nesne takibi algoritmasının hızlı olması beklenmektedir. Öte yandan, bir nesne izleme algoritması doğru ve güvenilir olmalıdır ve bu durum genellikle yapılacak hesaplama miktarını artırır. Birçok görüntü işleme uygulamasının doğası parallel programlamaya uygundur. GPU'lar çok sayıda çekirdek içerdiği için görüntü işleme ve nesne izleme uygulamalarında yaygın olarak kullanılırlar. Bu tezde, bir nesne izleme algoritmasını paralelliğe uygunluğu açısından analiz edilmiştir. Bir profilleme aracı kullanarak algoritmanın zaman alan kısımları belirlenmiştir. Algoritmanın belirlenen her bir parçası ayrı ayrı ele alınarak GPU'da gerçeklenmiştir. Ayrıca, GPU yeteneklerini kullanarak optimizasyon şansı üzerinde çalışılmıştır. Yöntemlerimizi beş veri kümesi üzerinde test ederek orijinal CPU tabanlı yaklaşımın ilgili parçaları ile karşılaştırdık.

**Keywords:** Nesne takibi; görüntü işleme; bilgisayarlı görü; GPU, paralel hesaplama

# ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor Asst. Prof. Dr. Adnan Özsoy for his endless patience, valuable guidance and advices throughout my research.

I would like to also thank to my thesis committee members; Prof. Dr. Mehmet Önder Efe, Prof. Dr. Süleyman Tosun, Prof. Dr. Alptekin Temizel, Assoc. Prof. Dr. Ahmet Burak Can for taking their time to review and providing insightful comments.

I am very grateful to my teammates at ASELSAN for providing me flexible work environment.

I want to express my appreciation to my family and my friends for their support.

Last but not least, I want to express my gratitude to Kübra Temiz for encouraging me throughout my study and being there for me all the time.

# TABLE OF CONTENTS

ABSTRACT ..... i

ÖZET ..... i

ACKNOWLEDGEMENTS ..... iii

TABLE OF CONTENTS ..... iv

LIST OF FIGURES ..... vi

LIST OF TABLES ..... viii

ABBREVIATIONS ..... ix

1. INTRODUCTION ..... 1

1.1. Overview ..... 1
1.2. Motivation ..... 2
1.3. Organization of the Thesis ..... 3

2. BACKGROUND ..... 4

2.1. Preliminaries ..... 4
2.2. DCF (Discriminative Correlation Filters) ..... 5
2.3. GPU Basics ..... 7

3. RELATED WORK ..... 10

3.1. Adaptive Spatially-regularized Correlation Filters (ASRCF) [15] ..... 10
3.2. A Robust Parallel Object Tracking Method for Illumination Variations (MRAT) [16] ..... 12
3.3. Parallel Tracking and Verifying (PTAV) [21] ..... 13
3.4. Efficient Convolution Operators for Tracking (ECO) [2] ..... 14

4. ANALYSIS OF THE ECO TRACKING ALGORITHM ..... 18

4.1. Datasets ..... 18
4.2. Setup ..... 19
4.3. Tracking Performance ..... 19

4.4. Benchmark Results ..... 20
4.5. Analysis using MATLAB Profiler ..... 23
4.6. Analysis using a Profiler ..... 26
5. IMPLEMENTATION ON GPU ..... 28
5.1. Sequential Approach ..... 29
5.2. Using `cudaMemcpyAsync` with Pinned Host Memory ..... 30
5.3. Zero-Copy Memory ..... 34
5.4. Using `cudaMallocManaged` with Unified Memory ..... 34
6. RESULTS ..... 37
6.1. Results obtained with Setup-1 ..... 38
6.2. Results obtained with Setup-2 ..... 40
7. CONCLUSION ..... 44
8. REFERENCES ..... 45
CURRICULUM VITAE ..... 48

# LIST OF FIGURES

Figure 1 The illustration of Fermi architecture and its SM [12]..... 8
Figure 2 Memory hierarchy on a NVIDIA GPU [13] ..... 9
Figure 3 The tracking framework of location and scale CF models in ASRCF [15]. ..... 11
Figure 4 The comparison of (b) the MRAT algorithm with (a) the base algorithm [16] ..... 13
Figure 5 Illustration of the PTAV framework. The tracking and verifying processes are carried on asynchronously in two parallel threads [21]. ..... 14
Figure 6 The visualization of learning framework of C-COT [8] ..... 15
Figure 7 The visualization of all 512 learned filters in the last convolutional layer of C-COT [2] ..... 16
Figure 8 Visualization of the remaining filters after eliminating the ones with negligible energy [2] ..... 17
Figure 9 The visualization of how Intersection over Union is calculated ..... 20
Figure 10 Ground-truth and prediction bounding boxes ..... 21
Figure 11 The Success Plot of ECO Tracker via using IoU ..... 21
Figure 12 The Precision Plot of ECO Tracker ..... 22
Figure 13 MATLAB Profiler output for data processing on CPU ..... 23
Figure 14 Indication of how often the methods are called and how much processing time they use ..... 27
Figure 15 Call graph showing dependency of algorithm components on each other ..... 27
Figure 16 The illustration of the distribution of the matrix elements over threads ..... 30
Figure 17 Pageable Data Transfer to GPU ..... 31
Figure 18 Using pinned host memory for the allocation and transfer ..... 32
Figure 19 The operations on the NULL stream ..... 33
Figure 20 The amount of concurrency ..... 33
Figure 21 Single memory space with UM ..... 35
Figure 22 CPU to GPU code transformation with Unified Memory ..... 37

Figure 23 The execution time of only one method on Setup-1. (a) the sequential approach with the pageable host memory, (b) the sequential approach with the pinned host memory, (c) the asynchronous approach with 3 streams, (d) with no-memcpy data pointers. ..... 39
Figure 24 Profiler output for unified memory (a) without using cudaMemPrefetchAsync, (b) with prefetching memory before kernel launch ..... 42
Figure 25 The execution time of only one method on Setup-2. (a) the sequential approach with the pageable host memory, (b) the sequential approach with the pinned host memory, (c) the asynchronous approach with 3 streams, (d) with no-memcpy data pointers, (e) using managed memory..... 42

# LIST OF TABLES

Table 1 Hardware and software configuration of the experimental setups ..... 19
Table 2 Caller functions of MTIMESX..... 24
Table 3 Execution time of the bottleneck functions on CPU and GPU ..... 25
Table 4 Speed-up accomplished by using vectorized data..... 26
Table 5 Pseudo-code of element-wise multiplication of two k x n matrices ..... 28
Table 6 Pseudo-code of element-wise division of two k x n matrices ..... 28
Table 7 The number of calls to the methods..... 29
Table 8 Pseudo-code of the element-wise multiplication kernel on GPU ..... 29
Table 9 Pseudo-code of the element-wise division kernel on GPU ..... 29
Table 10 The memory management functions used for asynchronous memory transfers ..... 32
Table 11 The utilized stream management functions ..... 33
Table 12 The functions used for managed memory with Unified Memory ..... 36
Table 13 The event management functions used for measurements ..... 38
Table 14 The total time spent in CPU and GPU kernels, and achieved speedup on Setup-1..... 38
Table 15 The measured running times and achieved speed-ups over the pageable memory on Setup-1 with respect to the serial approach..... 40
Table 16 The total time spent in CPU and GPU kernels, and achieved speedup on Setup-2..... 41
Table 17 The achieved speed-ups on Setup-2 with respect to the serial approach with pageable memory transfers ..... 41

# ABBREVIATIONS

| | |
|---|---|
| DCF | Discriminative Correlation Filter |
| VOT | Visual Object Tracking |
| ECO | Efficient Convolution Operators for Tracking |
| C-COT | Continuous Convolution Operator Tracker |
| UVA | Unified Virtual Addressing |
| CUDA | Compute Unified Device Architecture |
| UM | Unified Memory |

# 1. INTRODUCTION

## 1.1. Overview

The world among us is full of objects. We can group these objects into two categories with respect to their motion status; stationary and moving objects.

Every day, from the time we wake up and to the time we go to sleep, we are taking images from the world with our eyes and processing it with our brain. The outputs of this process includes detection, classification and tracking of the objects we see. We are doing this process every day in our lives. As a result, object detection, classification and object tracking became the center of interest in computer vision.

While the object detection and classification is focused on extracting information about the objects in the images, object tracking is focused on finding the position of the interested objects in every image sequence. In a broader manner, object detection is the process of understanding what objects are in the image; object tracking is the process of finding the same objects we saw in the previous image. Object tracking aims to predict the position and the trajectory of an object, whose only initial state is given, in an image sequence.

Object tracking has many practical applications such as security and surveillance, autonomous vehicles, human-machine interaction, traffic control.

This is such an important and well-famous are in the computer vision so that, there are competitions held for introducing new tracker algorithms. One of the famous competitions is VOT Challenge [1] which is held every year. People from all around the world are developing their own tracker algorithms and challenging each other to have the best tracker algorithm. The criterias used in benchmarking and performance evaluation of the tracker algorithms are accuracy and robustness [1]. The primary measure in VOT is the expected average overlap (EAO) – a principled combination of accuracy and robustness. Speed of the algorithms is not a main concern of the challenge. Because, the speed varies on hardware and implementation setup. There is a sub-challenge in VOT that evaluates the tracker algorithm only on their speed but it is not taken into account

for overall score. But, in many applications, a tracker is expected to have fast execution while still having high accuracy and robustness.

## 1.2. Motivation

The purpose of this study is to accelerate an accurate and robust tracker algorithm to succeed a higher speed while maintaining its reliability on tracking. Main key feature on doing this will be the use of GPU.

In this manner, a tracker algorithm named Efficient Convolution Operators for Tracking (ECO) [2], which is one of the best trackers in VOT Challenge 2017 [1], is selected. The main reason of selecting ECO is its reliable performance. It is a very important aspect in reliability required application areas such as, defense industry and autonomous vehicles. These areas also need applications to perform in adequate speed.

In recent years, the use of GPGPUs have become popular in areas like computer vision [3] and big data problems [4]. GPGPUs use the advantage of having many processor units. They are slower in clock speed when compared to common CPUs but can handle and execute instructions on many threads simultaneously. Nowadays, while a general CPU usually has 8 or 16 cores, a brand new NVidia GPU has more than 3000 cores. Even the GPU cores are usually slower than CPU cores, the computational power of GPUs are much greater due to the enormous number of cores.

Object detection and tracking algorithms are good candidate to be implemented on GPUs because they usually have parallelizable nature which can be divided into many threads.

Our main contributions in this thesis:

- We obtain a benchmark result of an object tracking algorithm on a database it has not been tested before.
- A deeper analysis on the object tracking algorithm is done by profiling its structure. This analysis is used for deciding on which parts of the algorithm

can be migrated to GPU for a faster execution. Eventually, GPU implementations of these parts are done.

- We improve the performance of our implementations with memory and kernel management tools of CUDA.

## 1.3. Organization of the Thesis

This thesis is structured as follows:

Chapter 2 provides a background information about the object tracking algorithms and GPUs.

In Chapter 3, the structure of the selected object tracking algorithm is explained.

Chapter 4 provides benchmark results, analysis of the algorithm with a profiler and investigates the methods which are eligible for GPU parallelization.

In Chapter 5, we discuss GPU parallelization steps and how the implementation carried out.

In Chapter 6, the evaluation results of the GPU implementation is presented and discussed.

Finally, Chapter 7 provides a conclusion by describing the lessons learned and brief summary about the work done.

# 2. BACKGROUND

## 2.1. Preliminaries

Correlation is a statistical measure of the relationship between two variables. It is commonly used in statistics, economics and signal processing applications. The correlation between two variables is measured with the correlation coefficient. It can take values between -1 and 1. The very simple way of calculating the correlation coefficient of $x$ and $y$ is:

$$r_{xy} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2(y_i - \bar{y})^2}}$$

$r_{xy}$:the correlation coefficient which gives the linear relationship between $x$ and $y$

$\bar{x}$:the mean of the values of $x$

$\bar{y}$:the mean of the values of $y$

If $r_{xy}$ is close to zero, it means that there is no correlation between $x$ and $y$.

If $r_{xy}$ is 1, there is a perfect positive correlation. So, the values of $x$ and $y$ increases or decreases at the same time with time same rate.

If $r_{xy}$ is -1, there is a perfect negative correlation. It means that while $x$ increases, $y$ decreases with time same rate or vice versa.

From this point of view, a correlation filter is a type of a filter constructed using the correlation of the input and the desired output. In object detection, the idea is to filter the input with this correlation filter so that, the output only includes the charecteristics of the object in focus.

The convolution of the signal $f_1(t)$ with another signal $f_2(t)$ is:

$$f_1(t) * f_2(t) = \int_{-\infty}^{\infty} f_1(\tau) f_1(t - \tau) d\tau \qquad (2.1)$$

The Fourier Transform is excessively used in signal processing. It allows you to look into the frequency components of a signal.

The Fourier Transform ($F$) of a signal $f$ is given by a complex integral:

$$\mathcal{F}\{f(t)\} = F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

where $\omega$ is the angular frequency, $j$ is the complex variable and $e^{-j\omega t} = \cos\omega t - j\sin\omega t$ gives the frequency components.

Performing the Fourier Transform on the both sides of the Equation 2.1, the convolution equation becomes:

$$\mathcal{F}\{f_1(t) * f_2(t)\} = \mathcal{F}\{f_1(t)\} \cdot \mathcal{F}\{f_2(t)\}$$

This equation is named as the Convolution Theorem. It states that the Fourier Transform of a convolution of two signals is the pointwise multiplication of their Fourier transforms.

The correlation of two signals can be expressed as:

$$f_1(t) \otimes f_2(t) = \int_{-\infty}^{\infty} f_1(\tau)f_1(t+\tau)d\tau \quad (2.2)$$

In the Fourier domain, the Equation 2.2 becomes:

$$\mathcal{F}\{f_1(t) \otimes f_2(t)\} = \mathcal{F}\{f_1(t)\} \cdot \mathcal{F}^*\{f_2(t)\}$$

In $\mathcal{F}^*\{f_2(t)\}$, * denotes that it is the complex conjugate of $\mathcal{F}\{f_2(t)\}$.

### 2.2. DCF (Discriminative Correlation Filters)

The number of object tracking algorithms which uses Correlation Filters has increased over these years [5-9]. They have shown remarkable performance on benchmarks and many state-of-the-art algorithms uses correlation filters.

The first use of Discriminative Correlation Filters in object detection goes back to the early 80's [10]. But became very popular with the recent works, starting with MOSSE tracker published by Bolme et al. in 2010 [5].

DCFs are usually trained online with the early samples from the image sequences. The increasing number samples cost high computational time and some of them may become unnecessary if the images contains very same

characteristics. So, the filter output is not changed and only causing loss of precious time. On the other hand, low number of samples causes performance issues. Today, the aim of the tracking algorithms, using DCFs, is to maximize the benefits gained from the training of the filters.

In filter based trackers, the first image is usually used for training the filter. So, the location of the target is given to the filter. After the filter is initialized with the first image, object tracking and training of the filter is carried out together. The resulting filter is applied to the image on the next image sequences. The location of the highest correlation achieved is new location of the target in the image. In every frame, the filter updated with the new correlation result.

We can categorize the tracking methods into two groups according to the size of the video input and way of handling the detection failures. DCF based methods can be labeled as long-term trackers. Long-term tracking methods usually include online learning of the appearance of the object and they are expected to recover when the object disappeared or cannot be detected in the image. But, short-term tracking is focused only the precision of the detection and once the target is lost they don't aim to recover and continue tracking. Short-term trackers are usually focused on working well for up to 1000 frames. On the other hand, long-term trackers are expected to be successful for more than 1000 frames and to recover if the object is lost in some of the frames. From this point of view, we can say that short-term trackers can be well fitted for surveillance purposes, while the-long term trackers are good if we want to follow the object constantly.

The computation of correlation matrices is easier and faster in Fourier domain rather than in time domain. Because, the calculation of the correlations consists of both summations and multiplications which make it harder to process. On the other hand, the calculation in Fourier domain includes only multiplications and eventually, it will be faster and straightforward.

As it is mentioned earlier, MOSSE [5] is the first tracker in which discriminative correlation filter is used successfully. The speed and robustness of the algorithm was so good that made it one of the state-of-the-art tracking algorithm at that

time. After that, many tracking algorithms merged from MOSSE and discriminative correlation filters.

MOSSE tracker uses DCF to predict the location of the object in each image sequence and it updates the filter in every frame. The correlation filter in MOSSE is obtained by minimizing the sum of squeared errors. Denoting the images by $f_1 \dots f_t$ and the filter output by $g_1 \dots g_t$, the filter function can be represented by:

$$\varepsilon = \sum_{k=1}^{t} \|h_t * f_k - g_k\|^2 \quad (2.3)$$

The desired correlation filter ($h_t$) at time $t$ is obtained by minimization of the Equation 2.3.

According to the Convolution Theorem, the correlation between $f$ and $h$ is the point-wise multiplication of them in the Fourier domain:

$$G = F \cdot H^*$$

where $F$ denotes Fast Fourier Transform (FFT) of the images $f$ and $H$ denotes the FFT of the correlation filter $h$. $G$ is the desired and the training output of the filter. So, the minimization problem of the MOSSE tracker algorithm can be expressed in Fourier domain as follows:

$$\min_{H^*} \sum_{k} |F_k \cdot H^* - G_k|^2$$

## 2.3. GPU Basics

The main use of GPUs has been to create and manipulate images for display purposes. The history of GPUs goes back to 1990's. They weren't fully programmable and the intended use of the early GPU hardware was only for graphics. The trend in GPU architecture is drawn toward a CPU-like programmable design. The first General Purpose GPU (GPGPU), named Fermi GPU, was released in early 2010 [11]. The illustration of Fermi architecture and its Streaming Microprocessor (SM) is given in Figure 1 [12].

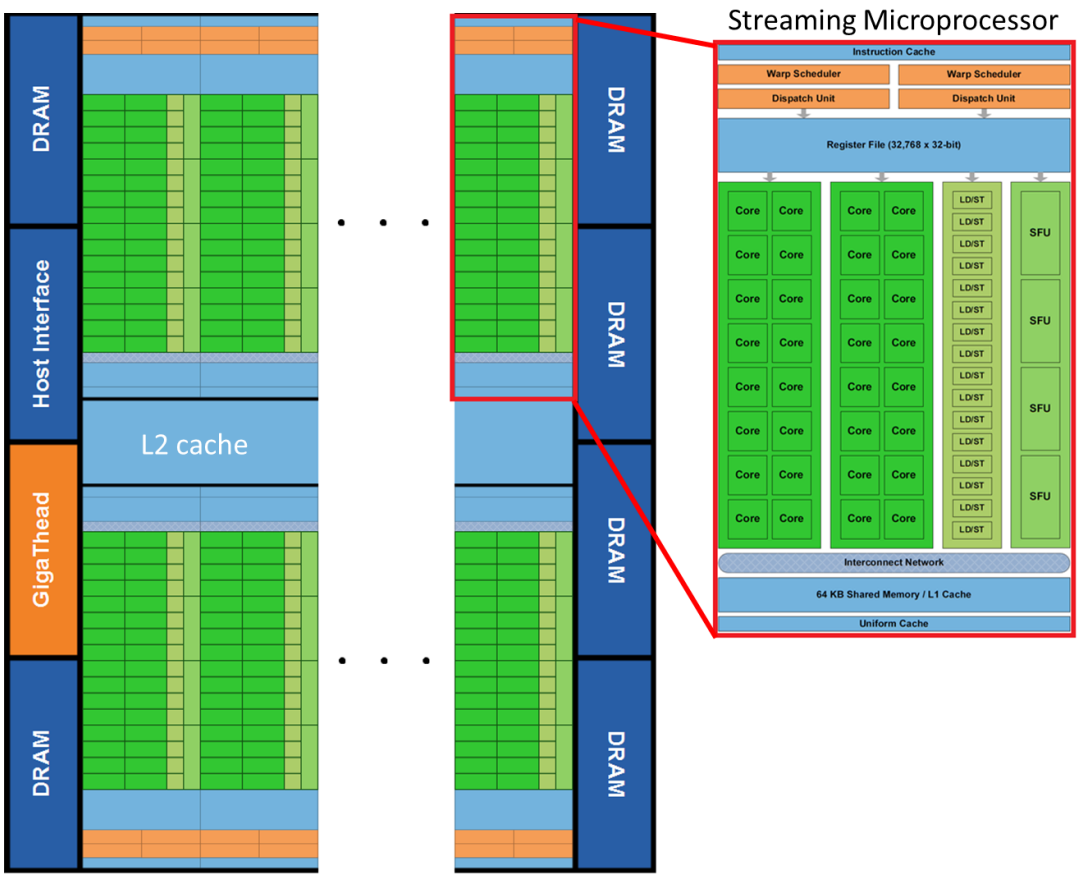


Figure 1 The illustration of Fermi architecture and its SM [12]

Fermi GPU brought new features such as: unified address space, concurrent, dual warp schedulers, and kernel execution [12]. It had a total of 512 cores located in 16 SMs. Today, the number of cores in GPUs exceeds 3000 with high bandwidth and larger memories.

GPUs use the Single-Instruction, Multiple-Thread (SIMT) execution model, introduced by NVIDIA in 2006, where multiple threads execute concurrently using a single instruction. Each SM can execute multiple SIMT groups. For example, a Fermi GPU which has 512 cores, can execute four 128-threaded SIMT groups simultaneously.

NVIDIA introduced a parallel programming model called CUDA which is used to execute programs written with different programming languages on NVIDIA GPUs.

Within a CUDA program, a kernel is called to be executed in a thread block. Each thread has its own ID, registers, program counter for executing an instance of the kernel. A thread block includes concurrently executing threads and a shared memory. A grid is a group of thread blocks that runs the same kernel and connected to a global memory. The hierarchy of the execution and the memory model is given in Figure 2 [13].

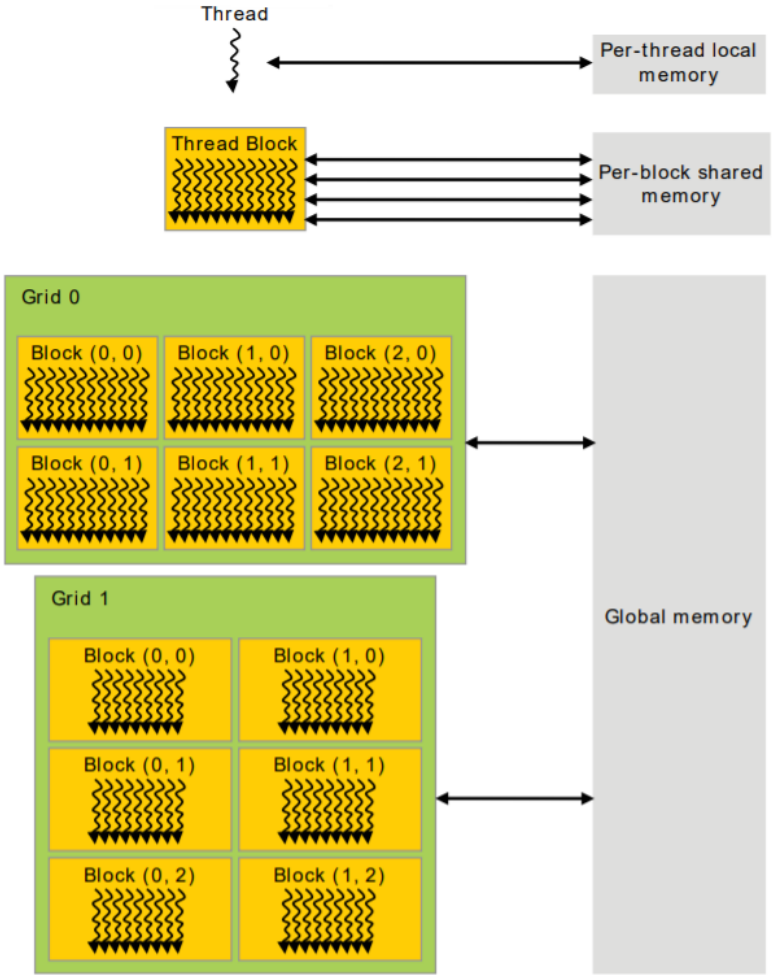


Figure 2 Memory hierarchy on a NVIDIA GPU [13]

# 3. RELATED WORK

In this chapter, some of the state-of-the-art DCF-based tracking algorithms will be presented.

After the Convolutional Neural Networks (CNNs) showed great results in the ImageNet competition [14] in 2012, they became popular instrument to be used in object detection and tracking. With the support of CNNs, DCF-based object tracking methods have shown extraordinary results on tracking benchmarks.

The feature maps of early DCF approaches were limited to a single-resolution. This means that all of the feature channels should have the same spatial resolution. However, this limits the use of multiple convolutional layers of varying spatial resolutions. Over the years, multi-channel feature maps are started to be used in DCF framework and the DCF based methods started to provide better results.

## 3.1. Adaptive Spatially-regularized Correlation Filters (ASRCF) [15]

ASRCF focuses on two major problems in CF-based methods [15]. First, the sampling process is prone to suffer from training with a poor samples from the image at boundaries of the target. The repetition of the training process with these poorly maintained feature causes the tracker failing eventutally. The main cause is the use of pre-defined and fixed spatial constraints on filter coefficients. Second, the constant extraction of the features on every sample for scale estimation and localization purposes results in too much computational burden and eventually the speed of the tracker degrades.

The objective function in ASRCF is given as follows:

$$E(\mathbf{H}, \mathbf{w}) = \frac{1}{2}\left\| \mathbf{y} - \sum_{k=1}^{K} \mathbf{x}_k * (\mathbf{P}^{\mathbf{T}}\, \mathbf{h}_k) \right\|_2^2 + \frac{\lambda_1}{2} \sum_{k=1}^{K} \|\mathbf{w} \odot \mathbf{h}_k\|_2^2 + \frac{\lambda_2}{2} \|\mathbf{w} - \mathbf{w}^r\|_2^2 \qquad (3.1)$$

where $\mathbf{x}$ and $\mathbf{h}$ are the vectorized image and the filter respectively. $K$ is the number of feature channels and y is the ground-truth response. $\mathbf{H}$ is the filter response and $\mathbf{w}$ is the weight of the filter channels.

As in a general CF, the tracking takes process with the filters learning from the minimization of this objective function in Equation 3.1.

The location of the target is determined with the localization function:

$$\mathbf{r} = \sum_{k=1}^{K} \mathbf{x}_{\boldsymbol{k}} \odot \mathbf{g}_k$$

The localization is done based on the maximizing the response of the filter on the image sequences.

The tracking framework of location and scale CF models of ASRCF is given Figure 3 [15].

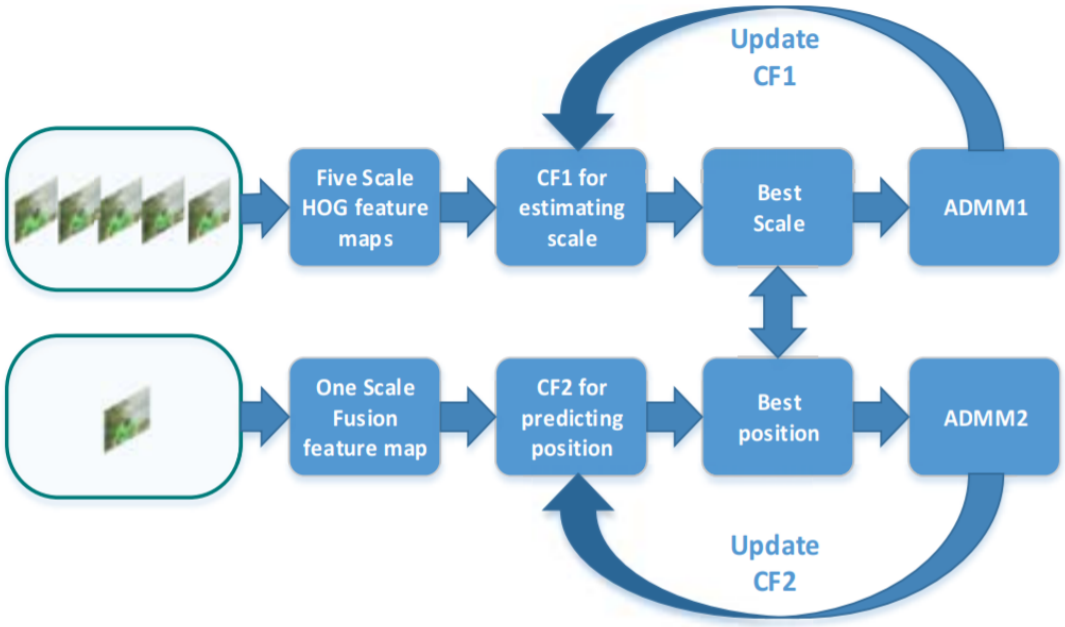


Figure 3 The tracking framework of location and scale CF models in ASRCF [15].

### 3.2. A Robust Parallel Object Tracking Method for Illumination Variations (MRAT) [16]

Liu, Shuai, et. al. [16] proposed a method (MRAT) performs detection on multiple regions with the use of an alternate template based on parallel computing.

The CF-based trackers starts searching the object from the last known location in the previous frame. The search area is usually kept small in order to achieve better speed. But this can result in loss of the target. MRAT enlarges the search area to improve the tracking robustness and deals with the speed degradation with the help of parallel computing.

The tracking process is prone to fail with large illumination changes. There are several methods to handle light changes. One approach is to gather as much information as possible from the frames like Histogram equalization [17]. The other approach focuses on light invariant features like edge features [18, 19]. Another method focuses on creating a prediction model to generate possible images of targets in different lighting conditions [20].

MRAT splits the image into 9 sub-regions. One of the region includes the search area of the correlation filter. The confidence score is calculated in the search area of each frame in order to locate the target. Under intense illumination changes, the confidence scores will be lower eventually and it is likely to lose the track of the target. In order to lower the effects of the illumination changes, MRAT also carries out detection on the other 8 regions when the original confidence score is lower then a threshold value. If the confidence score of a region is higher than the threshold value, it is selected as the new position of the target and the model is updated with the new position.

MRAT algorithm perform the detection on other 8 sub-regions with the parallel computing support. The illustiration of the MRAT algorithm is given in Figure 4 [16].

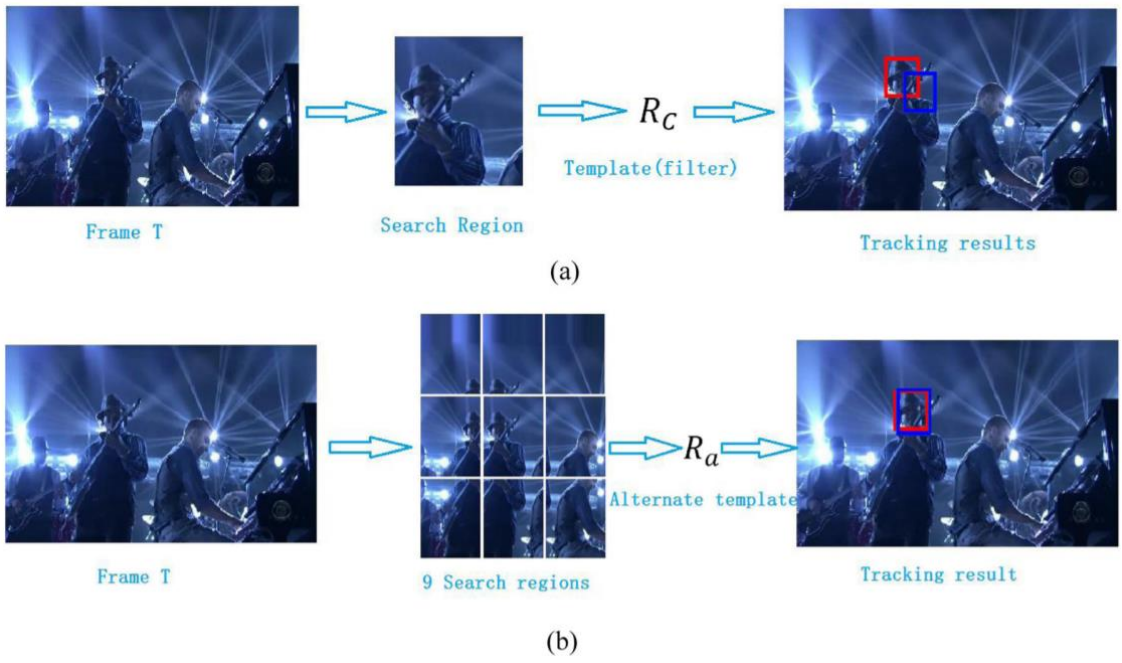


Figure 4 The comparison of (b) the MRAT algorithm with (a) the base algorithm [16]

### 3.3. Parallel Tracking and Verifying (PTAV) [21]

Heng Fan and Haibin Ling proposed a novel Parallel Tracking and Verifying [21] framework for achieve the accuracy and speed burden in an object tracking algorithm.

Increasing the accuracy of an algorithm usually results in speed degradation. In their work, the tracking task split into two parallel components working on two separate threads. One of the components is a base tracker T and the other component is a verifier V.

A tracking algorithm usually works fine for easy and slowly changing scenes. But tracker usually struggles to cope with dramatic changes in object appearance. These cases usually need much more computational process and analysis not to lose the target. In their approach, they named these cases as the verification. They are needed on some occasions and not on every frame. A simple tracker locates the target on every frame and the verification process is carried out for the detection of the tracking failure and correction of the result. The PTAV framework is illustrated in Figure 5 [21].

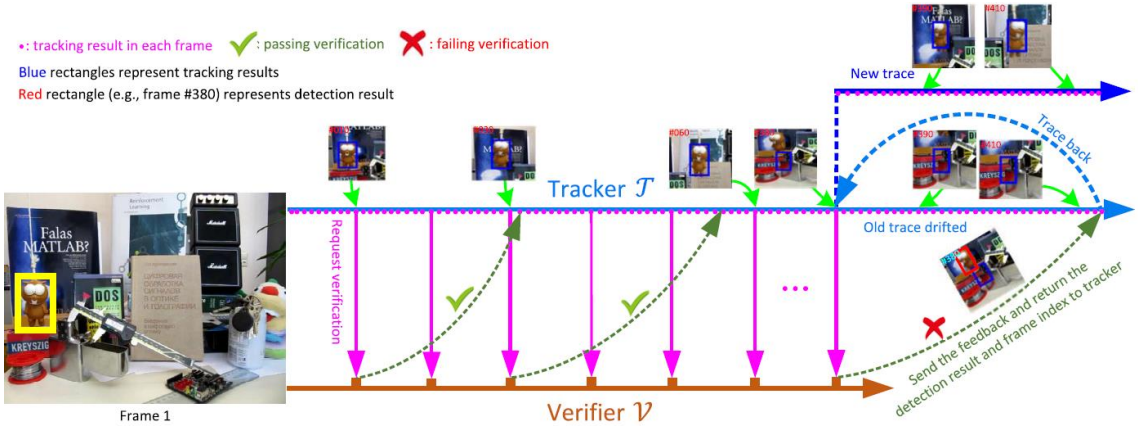


Figure 5 Illustration of the PTAV framework. The tracking and verifying processes are carried on asynchronously in two parallel threads [21].

PTAV framework includes a tracking solution by combining a correlation filter-based algorithm (Staple [22]) and a deep learning-based tracker (Siamese network [23]). The Staple (Sum of Template and Pixel-wise Learners) combines two complementary factors and learns a model. The model is based on a correlation filter which uses HOG features and a colour histogram. Two branched CNNs are used in the Siamese network. The VGGNet [24] architecture is used for CNNs.

PTAV frameworks combines the speed of a tracking algorithm with the robustness of a detection algorithm. Since the detection is not likely to happen in real-time, it is not carried out on every frame. While the tracker T is working on every frame, a verification request is sent to verifier V on every 10 frames. The result of the verification process is sent back to T and if a failure occurs tracker T continues with the information received from verifier V.

### 3.4. Efficient Convolution Operators for Tracking (ECO) [2]

Danelljan et al. proposed a method for learning a convolutional operator. Their learning formulation generates a continuous confidence map by use of convolution filters. Figure 6 provides a visualization of their continuous convolution operator which integrates multi-resolution deep feature maps [8]. (a) in Figure 6 is showing the feature map which includes the RGB input image

together with the convolutional layers of a pretrained deep network. The next column (b) is the visualization of the learned convolution filters. The convolution output of the each layer is given in the third column (c) and their combination into the final confidence function (d) provides the position of the object.

C-COT improved the mean overlap score of the state-of-the-art tracker from 77,3% to 82,4% on OTB-2015 [25], Temple-Color [26] and VOT2015 [27] datasets. It has been ranked as the best tracker in VOT2016 Challenge [28].

Each training sample $x_j$ has feature channels $x_j^1, \dots, x_j^D$. The sample count of each feature channel is denoted as $N_d$ thus, the sample space can be expressed as $\mathbb{X} = \mathbb{R}^{N_1} \times \dots \times \mathbb{R}^{N_D}$ [8].

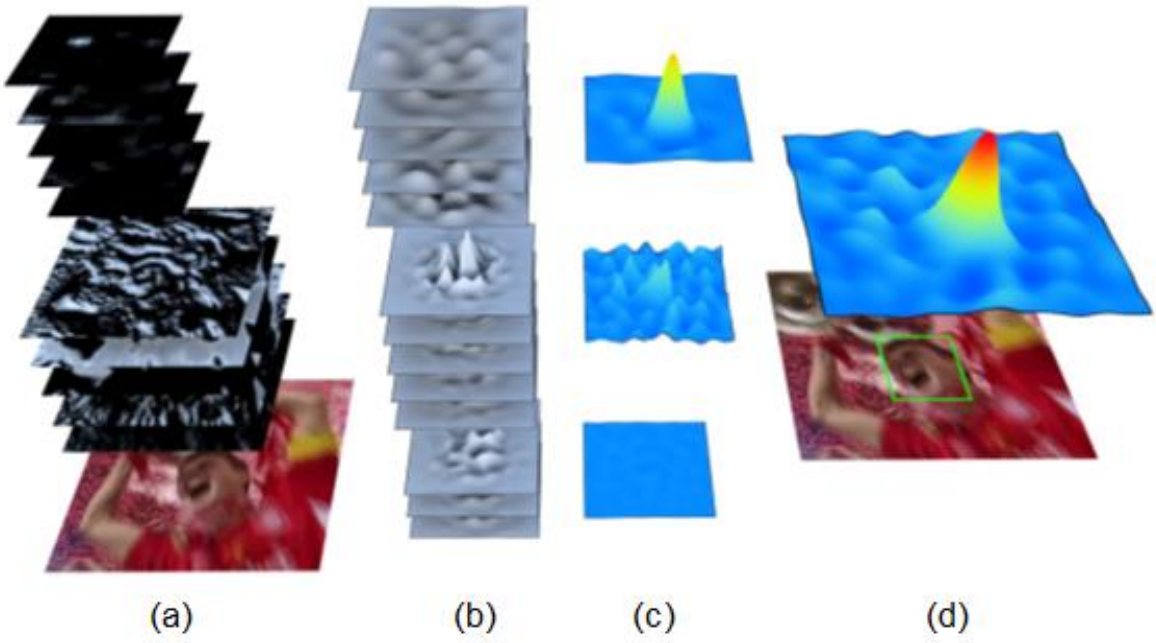


Figure 6 The visualization of learning framework of C-COT [8]

In order to transfer each feature map $d$ to the continuous spatial domain $t \in [0, T)$, an interpolation operator $J_d : \mathbb{R}^{N_d} \rightarrow L^2(T)$ is introduced [8]:

$$J_d\{x^d\}(t) = \sum_{n=0}^{N_d-1} x^d[n] b_d \left(t - \frac{T}{N_d} n\right)$$

The $L^2(T)$ space is considered to have complex functions periodic with $T > 0$.

The aim is to predict confidence scores for each layer $S_f\{x\}(t)$ with trained the convolution filters $f = (f_1 \dots f_D)$:

$$S_f\{x\}(t) = f * J\{x\} = \sum_{d=1}^{D} f^d * J_d\{x^d\} \tag{3.2}$$

In ECO, the filter $f$ in the formulation (3.2) is replaced with an alternative filter which reduces the number of model parameters and avoids over-fitting problem. The problem in C-COT was that many of the learned filters don't have enough energy to provide diversity as it can be seen in Figure 7 [2]. This causes that an important portion of the computatinal power is spent on these unnecessary filters. ECO has introduced a new convolutional operator that removes the filters that don't have much energy form the related convolutional layers.

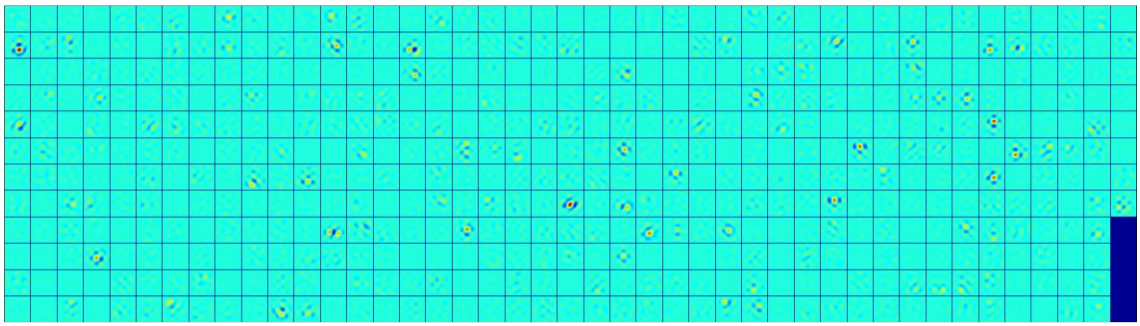

Figure 7 The visualization of all 512 learned filters in the last convolutional layer of C-COT [2]

The reduction is done by modifying the convolution operator given in (3.2). The obtained factorized convolution operator becomes:

$$S_{Pf}\{x\}(t) = Pf * J\{x\} = \sum_{c,d} p_{d,c} f^c * J_d\{x^d\} = f * P^{\mathrm{T}} J\{x\} \tag{3.3}$$

$P$ is a coefficient matrix for each of the filters of each feature layer. It is a $D \times C$ matrix where $D$ is the number of feature channels and the $C$ is the number of filters that have the sufficient energy. The resulting matrix multiplication is visualized in

Figure 8 [2].

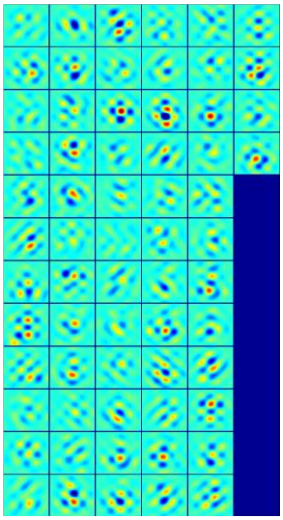

Figure 8 Visualization of the remaining filters after eliminating the ones with negligible energy [2][

In Equation (3.3), $J\{x\}$ is multiplied by the matrix $P^{\mathrm{T}}$ which results in a $C$-dimensional vector. This feature map is convolved with the desired filter $f$.

After the interpolation carried out, the filter is trained by minimizing the following expression:

$$E(f) = \sum_{j=1}^{m} \alpha_j \left\| S_f\{x_j\} - y_j \right\|^2 + \sum_{d=1}^{D} \|\omega f^d\|^2$$

Here $\alpha$ is the weight of the each sample and $\omega$ is the penalty function.

$$E(f, P) = \left\| \hat{z}^T P \hat{f} - \hat{y} \right\|^2 + \sum_{c=1}^{C} \left\| \widehat{\omega} * \hat{f}^c \right\|^2 + \lambda \|P\|_F^2$$

# 4. ANALYSIS OF THE ECO TRACKING ALGORITHM

We have discussed some of the state-of-the-art DCF-based tracking algorithms in Section 3. In this section, we will be analyzing, profiling and splitting an DCF-based algorithm into sub-parts for the investigation of parallelizable features. For this purposes, we chose to conduct the investigation on the ECO tracking algorithm. Since, DCF-based methods are based on the training of the filter, these analysis can be similarly applied to the algorithms other than ECO.

## 4.1. Datasets

The experiments is carried out on five datasets: VOT2017 [1], VOT2019 [29], OTB-100 [30], TLP [31] and UAV123 [32].

VOT2017 dataset has 60 image sequences and all of them has 30 fps rate over different resolutions. The total number of frames is 21.345. Since the ECO tracker is introduced in VOT2017 challenge, it is tested on VOT2017 dataset for better comparison on the benchmark result of the 2017 challenge [1].

VOT2019 Challenge [29] was held recently in late October 2019, the benchmark results and the dataset has been made available. Even though some of the image sequences are common with VOT2017 dataset, it will be beneficial to run the experiment also on VOT2019. The 2019 dataset has the same number of image sequences and frame rates as 2017 database. But the total frame count is now 19.760

OTB-100 [30] dataset has 100 image sequences with the total of 59083 frames. The images have varying resolutions and the total size is 2,61 GB.

TLP [31] is a long video dataset which is more suitable for Long-Term Object tracking. It consists of 50 HD videos over 676.431 frames. It is also modified for Short-Term Object Tracking as it has first 600 frames from each of the image sequences. The modified version is named as tinyTLP. In the experiments, tinyTLP is used.

UAV123 dataset is released by Image and Video Understanding Lab. at King Abdullah University of Science and Technology [32]. It consists of sequences captured from UAVs and has 123 sequences with more than 110.000 frames.

Each frame in all of the datasets are annotated by a rectangular bounding box.

### 4.2. Setup

The experiments were run on two setups which have the following specifications as given in the Table 1.

Table 1 Hardware and software configuration of the experimental setups

| | Setup-1 | Setup-2 |
|---|---|---|
| CPU | Intel Core i7-7500U (4 cores) | Intel Xeon Silver 4114 (40 cores) |
| Memory (RAM) | 16 GB | 126 GB |
| GPU | NVIDIA GeForce GT940MX (384 cores) | NVIDIA GeForce GTX 1080 Ti (3584 cores) |
| Memory (GPU) | 2 GB | 12 GB |
| Operating System | Ubuntu 16.04 | Ubuntu 18.04 |
| CUDA Version | 8.0 | 10.0 |
| OpenCV Version | 3.4.4 | 3.4.4 |

### 4.3. Tracking Performance

The performance analysis will be done using Intersection over Union (IoU) scores. IoU is calculated as dividing the intersection area of the ground-truth bounding box and the resulting bounding box generated from the tracker by the total area covered by the composition of these bounding boxes as visualized in Figure 9.

The formulation of IoU can be expressed as follows:

$$IoU = \frac{Intersection\ over\ Overlap}{Intersection\ over\ Union} \tag{4.1}$$

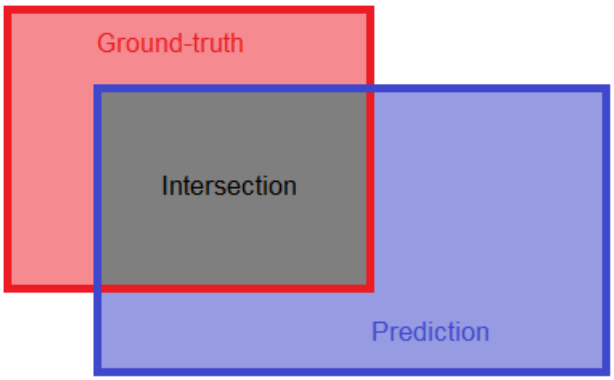


Figure 9 The visualization of how Intersection over Union is calculated

Precision and Success curves are another popular performance measurement criteria's. Success curve is plotted over an overlap threshold. Precision Plot is used to measure what percentage of the center error is within the center error threshold. The center error threshold is the maximum acceptable distance in pixels with the center of bounding box and the center of the ground-truth. The popular threshold used in object tracking benchmarks is 20. The threshold value of 20 is redundant of the object and the image size.

### 4.4. Benchmark Results

IoU scores are calculated for every frame. The ground-truth and the estimated bounding boxes also drawn on the current frame for display purposes which can be seen in Figure 10.

The Success Plot is given in Figure 11. The IoU values which is larger than the overlap threshold means that the target is successfully tracked in that frame. The Succes Rate starts to fall down dramatically as the overlap threshold is increased beyond 0,5. This is also a popular threshold level for IoU.

The precision score is computed by a center error threshold between 5 and 50 pixels. The resulting Precision Plot is given in Figure 12.

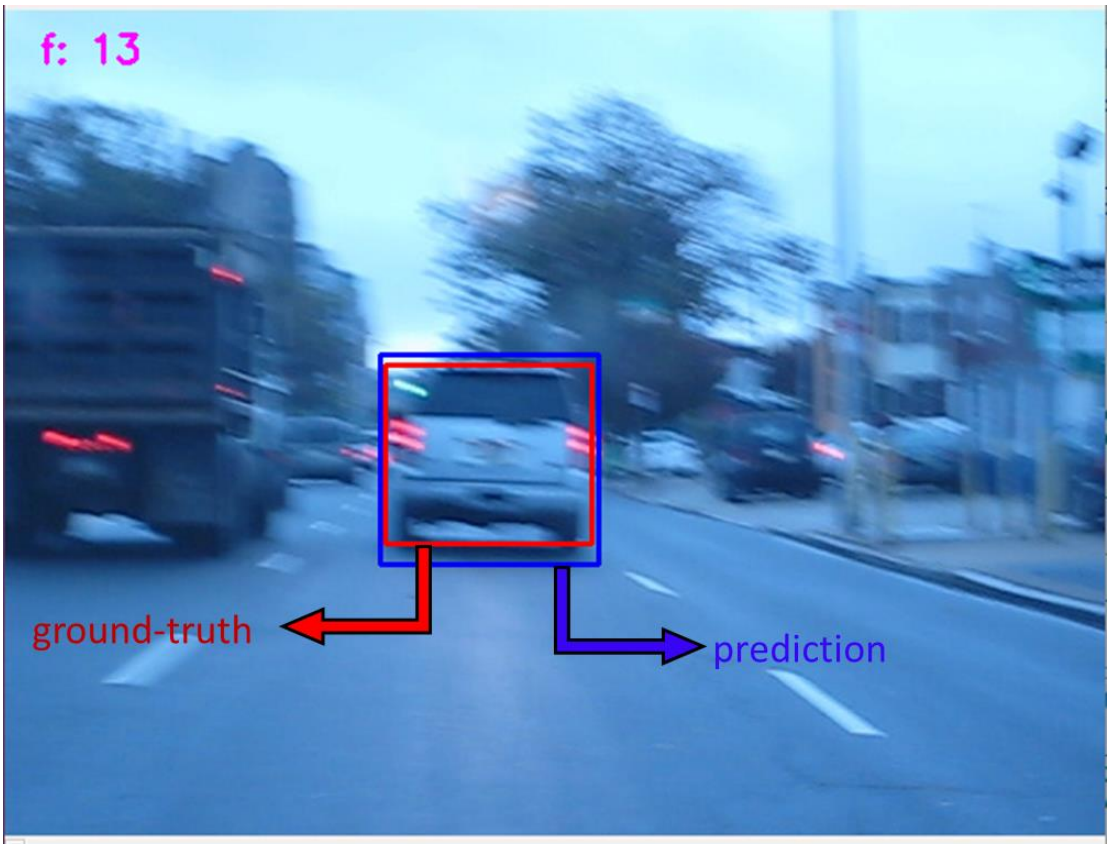


Figure 10 Ground-truth and prediction bounding boxes

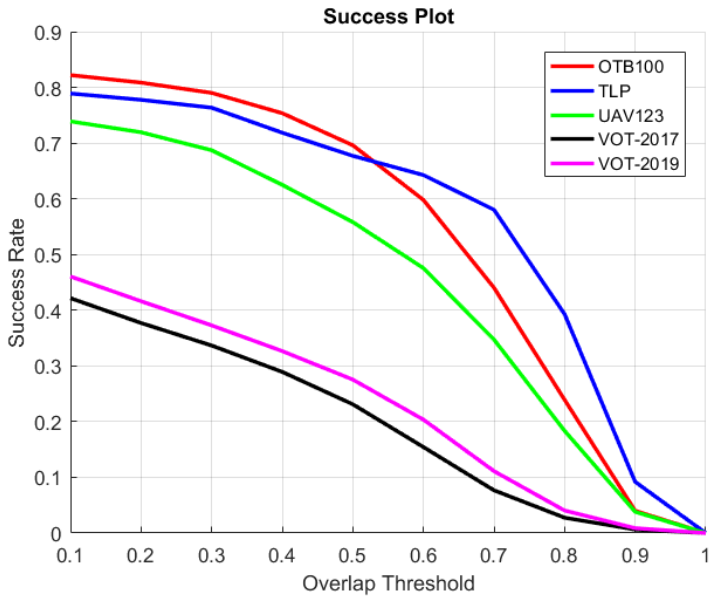


Figure 11 The Success Plot of ECO Tracker via using IoU

When examining these results, it is necessary to consider the characteristics of the data sets. The source of motion in image sequences can occur in two ways: one is the movement of the target and the other is the movement of the image recorder. The OTB100 data set usually contains image sequences where the image recorder is stabilized and the source of the motion is the target. Although both sources are active in the image sequences in the TLP and UAV123 datasets, the motion of the image recorder is less severe. However, in the VOT-2017 and VOT-2019 datasets, the intensity of both the target's and the image recorder's movements is generally very high.

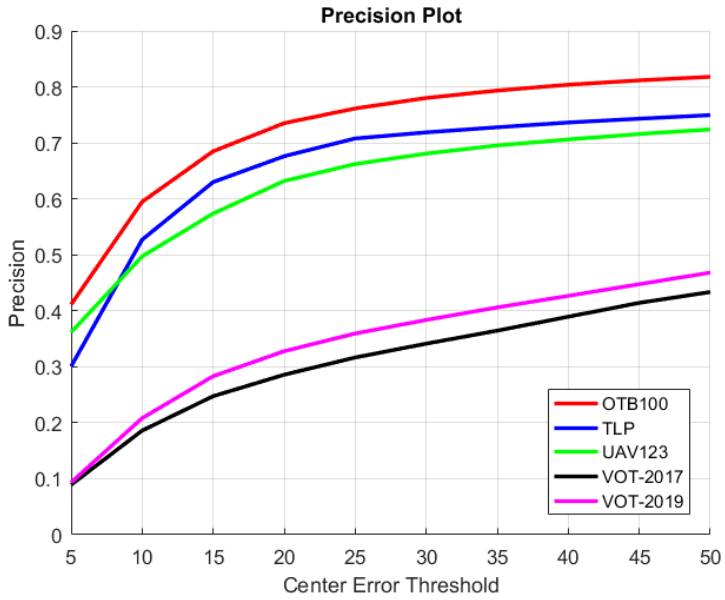


Figure 12 The Precision Plot of ECO Tracker

If we first examine the Success Plot in Figure 11, we see that the algorithm's performance is inversely proportional to the motion intensity of the target and the image recorder. In the Precision Plot in Figure 12, it is seen that the algorithm's performance in the TLP dataset approximates its performance in the VOT datasets. Since the resolution of the image frames in the TLP dataset is higher than in other datasets and the target sizes are generally large, the center error thresholds remain small. This led to the downward movement of the precision graph.

### 4.5. Analysis using MATLAB Profiler

**MATLAB Implementation on Single-core CPU**

The ECO tracking algorithm was originally implemented using MATLAB with GPU support. Thus, we conduct an analysis on the MATLAB implementation with MATLAB profiler.

The MATLAB implementation is profiled over one image sequence which contains 742 frames. The total time is 183.26 seconds.

Profile Summary

| Function Name | Calls | Total Time | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| mtimesx (MEX-file) | 40811 | 34.016 s | 34.016 s | |
| find_gram_vector | 742 | 18.530 s | 18.530 s | |
| channels\private\gradientMex (MEX-file) | 7412 | 16.169 s | 16.169 s | |
| table_lookup | 742 | 14.780 s | 12.320 s | |
| average_feature_region | 742 | 19.030 s | 10.198 s | |
| integralVecImage | 742 | 8.832 s | 8.832 s | |
| tracker | 1 | 177.419 s | 8.611 s | |
| get_fhog | 742 | 24.565 s | 8.104 s | |
| ...hape(x,[],1,1,size(x,4)).^2,1)+eps))) | 1484 | 7.501 s | 7.501 s | |

Figure 13 MATLAB Profiler output for data processing on CPU

The methods covering the most of the execution time are investigated and the ones which are eligible for parallelization are determined.

**MTIMESX [33]**: The most time spend in the mtimesx function. MTIMESX is a fast general purpose matrix and scalar multiply routine [33] and it is used for the element-wise multiplication of matrices. It has 6 calling functions. The measurements on the functions that use MTIMESX are given in Table 2.

The methods except lhs_operation have negligible execution time. On every call to **lhs_operation**, multiplication of around 10.000.000 complex numbers is carried on. This is a naturally parallel problem. The matrix sizes in the other calling functions are very small and they have negligible compute-time.

Table 2 Caller functions of MTIMESX

| Caller name | Calls | Time Spent | Time/Call |
|---|---|---|---|
| project_sample | 1486 | 0,881 | 0,0005929 |
| train_joint | 20 | 0,009 | 0,0004500 |
| lhs_operation_joint | 1280 | 0,201 | 0,0001570 |
| optimize_scores | 34827 | 0,498 | 0,0000143 |
| train_filter | 246 | 0,216 | 0,0008780 |
| lhs_operation | 2952 | 32,211 | 0,0109116 |

**find_gram_vector** routine, two matrices with around 4.000.000 elements are multiplied and this provides a good chance of parallelization. In **average_feature_region** and **integralVecImage** are the other bottleneck functions which includes matrix multiplications and summations in their basis.

### MATLAB Implementation with GPU Support

MATLAB has Parallel Computing Toolbox that includes many of the standart MATLAB built-in function to be executed in GPU. This minimizes the need of a new kernel implementation. MATLAB lets you declare GPU arrays by using the type `gpuArray`. It is also possible to transfer an array A from CPU memory to GPU memory via `A_gpu = gpuArray(A)`. On the other hand, a GPU array B can be transfered to CPU memory using `gather` function.

With the support of its built-in functions and easy memory allocation on GPU, a MATLAB code which is written for running on CPU can be also run on GPU by only migrating the variables from CPU memory to GPU memory. No change or minor changes may be needed for the methods themself.

### Overcoming the Bottlenecks

`bsxfun` is a MATLAB function that is used for applying element-wise operation to two arrays. `bsxfun(fun,A,B)` applies the element-wise binary operation specified by the function handle `fun` to arrays A and B. `fun` is replaced by

`@times` for the multiplication operation. For GPU arrays, `mtimesx` function is replaced by `bsxfun`.

The GPU counterparts of these methods have provided a better performance as expected. The execution times are given in Table 3.

Table 3 Execution time of the bottleneck functions on CPU and GPU

| **Description** | **CPU Time** | **GPU Time** |
|---|---|---|
| lhs_operation | 32,211 | 10,622 |
| find_gram_vector | 17,447 | 2,493 |
| average_feature_region | 9,981 | 2,892 |
| integralVecImage | 8,832 | 4,589 |
| **Total** | **68,471** | **20,596** |

With the GPU usage, the average profile time is decreased to 134,72 seconds from 183,26 seconds providing a 26,49% speed-up. The average profile times are obtained over 100 runs.

Further speed-up can be achieved changing the structure of the arrays. In the original implementation, the input arrays to the previously mentioned processes are 2D, 3D or sometimes 4D arrays. The vectorization of these array can provide better performance on calculations. Addition to the original approach, we changed the allocation of the arrays so that they are aligned in a vector array.

For example, the element-wise multiplication of two 4D arrays with the size of 50x30x125x63, took more time than two 1-D vector array with the same amount of elements. The average times spent for the multiplication are given in the following Table 4. The vectorization of the input array shortened the execution time by 21,76%.

Table 4 Speed-up accomplished by using vectorized data

| Array dimensions | CPU Time | GPU Time |
|---|---|---|
| 4-D arrays | 0,012951 | 0,000043362 |
| 1-D arrays | 0,013005 | 0,000033928 |

### Effects on Accuracy

We also investigated the differences between the results obtained from MATLAB implementations with CPU and GPU.

If it is not handled explicitly, it is likely to come across slight differences between the multiplications and summations of the floating points carried out in CPU and GPU because of the rounding modes and order of the accessing elements [34].

The maximum difference between the CPU and GPU calculations is measured as 0,2% which affects the tracking accuracy by a maximum of 0,4%.

## 4.6. Analysis using a Profiler

As previously described in Section 3.4, the algorithm includes a large amount of matrix products and convolution processes. An analysis was carried out to understand how much processing power these processes require and affect the speed of operation.

Valgrind [35] tool was selected for analysis. The Valgrind tool is a very successful tool in memory management, fault finding and processor profiling. The algorithm was run on the 3.734 image sequence with the Valgrind analysis tool. The visualization of the analysis outputs is given in Figure 14 and Figure 15.

The algorithm proceeds predominantly in two independent branches as can be seen in Figure 15. One of them is the part about the training of the filter and the other is the part where the feature maps of the image sequences are drawn. Approximately 74,15% of the time spent in the filter training section covers matrix multiplications and convolution. It is observed that approximately 30,28% of the entire running time of the algorithm consists of extracting HOG and ColorNames

property maps. FFT and matrix manipulations are frequently used during these operations. As it is known, these processes can be good candidates for GPU parallelization.

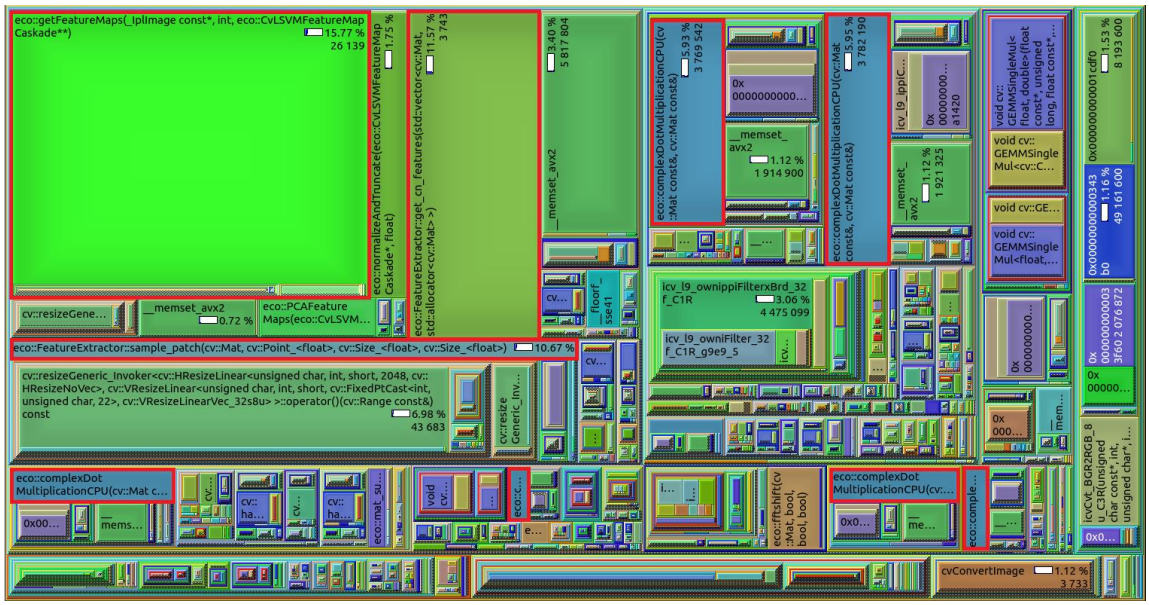


Figure 14 Indication of how often the methods are called and how much processing time they use

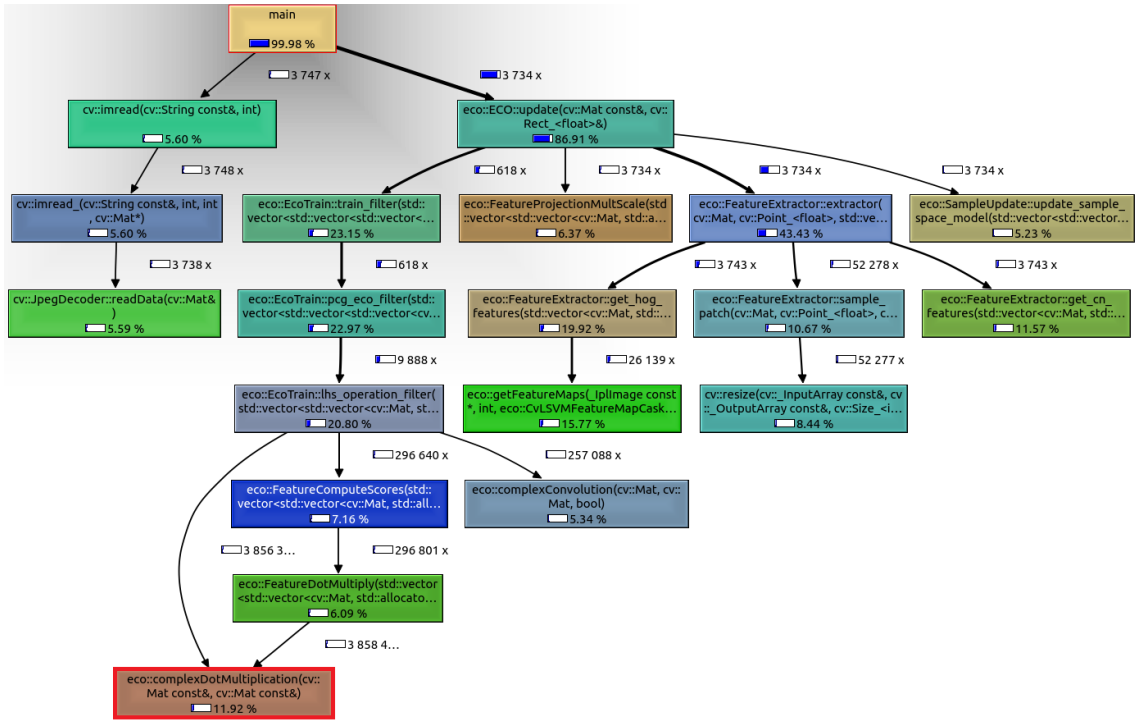


Figure 15 Call graph showing dependency of algorithm components on each other

# 5. IMPLEMENTATION ON GPU

As mentioned in the previous section, the ECO tracking algorithm excessively includes element-wise multiplication and division of the complex matrices. The initialization and the training of the filter, the extraction of the feature from the images, the convolution of the filter and the features, the computation of the energy include matrix multiplications and all using the same method whose algorithm is given in Table 5.

Table 5 Pseudo-code of element-wise multiplication of two k x n matrices

```
1:  procedure complexDotMultiplication(A, B)
2:      k = A.rows
3:      n = A.cols
4:      let C be a new k x n matrix
5:      for i = 1 : k
6:          for j = 1 : n
7:              C_ij = A_ij · B_ij
8:          end for
9:      end for
10:     return C
11: end procedure
```

The ECO algorithm also includes the element-wise division of the matrices as given in Table 6.

Table 6 Pseudo-code of element-wise division of two k x n matrices

```
1:  procedure complexDotDivision(A, B)
2:      k = A.rows
3:      n = A.cols
4:      let C be a new k x n matrix
5:      for i = 1 : k
6:          for j = 1 : n
7:              C_ij = A_ij / B_ij
8:          end for
9:      end for
10:     return C
11: end procedure
```

These methods are used for the multiplication and the division of the matrices whose sizes are changing between 50x50 and 200x200. At first glance, these sizes are very small to use a GPU for the multiplication. But a deeper look into

the ECO algorithm revealed that this method continuously called for each frame and these small matrices are parts of a much larger matrix. The total number of elements of the larger matrices are changing between 200.000 and 2.000.000 depending on the image size. The multiplication of these matrices are very good candidates to be parallelized with the use of a GPU.

The number of calls to the original methods and the GPU counterpart developed for this work are given in Table 7.

Table 7 The number of calls to the methods

| **Procedure** | **# of calls (in CPU)** | **# of calls (in GPU)** |
|---|---|---|
| Multiplication | 417.439.222 | 8.653.021 |
| Division | 9.425.588 | 725.045 |

### 5.1. Sequential Approach

Firstly, a sequential approach is followed for the parallelization of the methods. So, the elements of the matrices are copied to the GPU and they are evenly distributed over the threads. The pseode-code for the sequential approach is given in Table 8 and Table 9.

Table 8 Pseudo-code of the element-wise multiplication kernel on GPU

*$k = A.rows$, $n = A.cols$, $N = k * n$*
*let C be a new (k x n) length array*

```
1: procedure complexDotMultiplicationGPU(A, B, N)
2:         idx = blockIdx.x * blockDim.x + threadIdx.x
3:         if idx < N
4:                 C_idx = A_idx · B_idx
5:         end if
6: end procedure
```

Table 9 Pseudo-code of the element-wise division kernel on GPU

*$k = A.rows$*, n= *$A.cols$*, *$N = k * n$*
*let C be a new (k x n) length array*

```
1: procedure complexDotDivisionGPU(A, B, N)
2:         idx = blockIdx.x * blockDim.x + threadIdx.x
3:         if idx < N
4:                 C_idx = A_idx / B_idx
5:         end if
6: end procedure
```

In the sequential approach, the related kernel is called after the matrices are copied to the device memory as whole. The distribution of the matrix elements and the calculation process is visualised for the multiplication in the Figure 16 below.

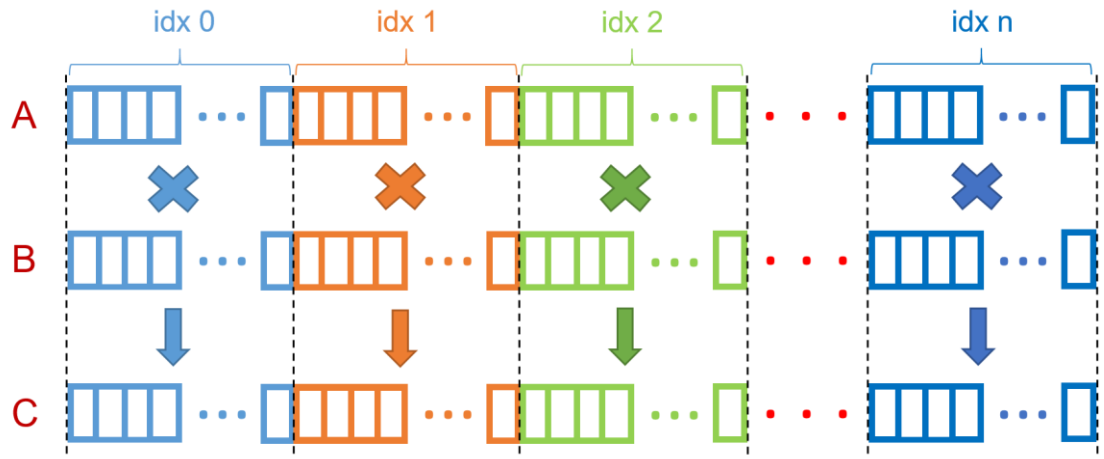


Figure 16 The illustration of the distribution of the matrix elements over threads

The organization of the threads is based on one-dimensional blocks and one-dimensional grids. This approach only uses the advantage of large number of cores on GPU to shorten the computation time. But this might not be the only advantage of using a GPU. Further improvements can be achieved by using the asynchronous memory transfers instead of the synchronous transfers.

### 5.2. Using `cudaMemcpyAsync` with Pinned Host Memory

Up to this point, the advantage of multi-threaded parallelism is used in order to achieve faster execution time. But, this is not the only way of shortening the computation time. Concurrency is the ability to run a number of tasks at the same time. Overlapping two actions is the most common optimization in asynchronous programming.

In CUDA, the kernels are asynchronous by default. `cudaMemcpyAsync` can be used in order to overlap two memory transfers in different directions. `cudaMemcpyAsync` is also asychronous with the kernels. So, it is possible to run

a CUDA kernel, `cudaMemcpyAsync` and do some other operations on the CPU and on the other GPU streams at the same time.

`cudaMemcpyAsync` requires Pinned (Page-locked) Host Memory allocation. The host memory allocation with `malloc` is pageable by default and it is managed by the operating system. While transfering data on the pageable host memory to GPU memory, the CUDA driver first copies the data to a temporarily allocated pinned memory, and then the transfer to device memory is carried out after that, as illustrated in Figure 17.

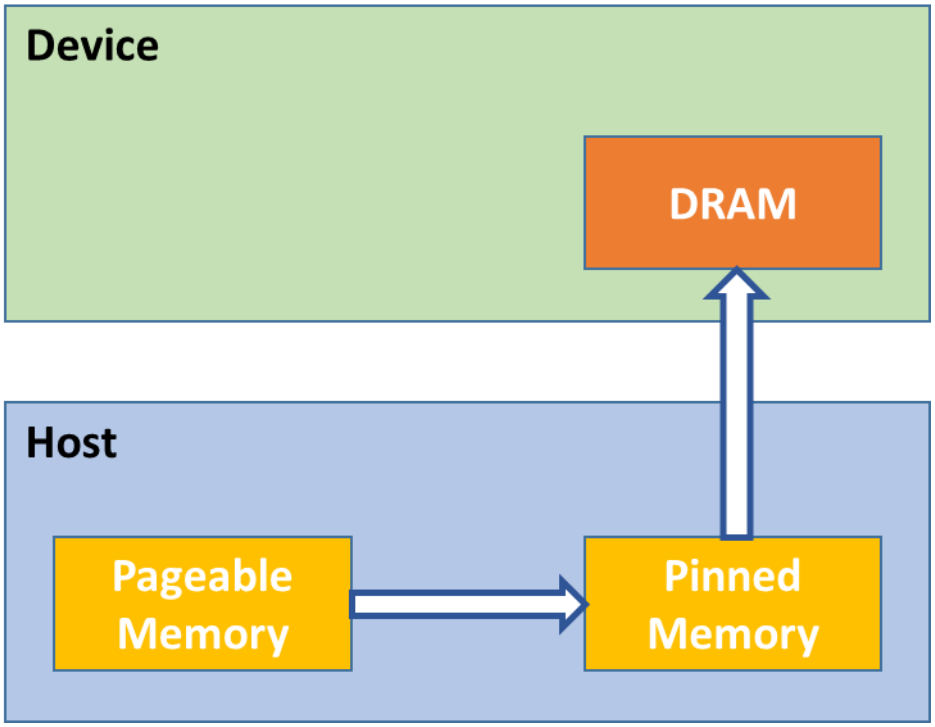


Figure 17 Pageable Data Transfer to GPU

The creation of another copy of the data can be very time consuming especially with large data and this will cause slower memory transfer between the host and the device. The CUDA Runtime API provides functions to allocate the data directly using the pinned host memory, as illustrated in Figure 18.

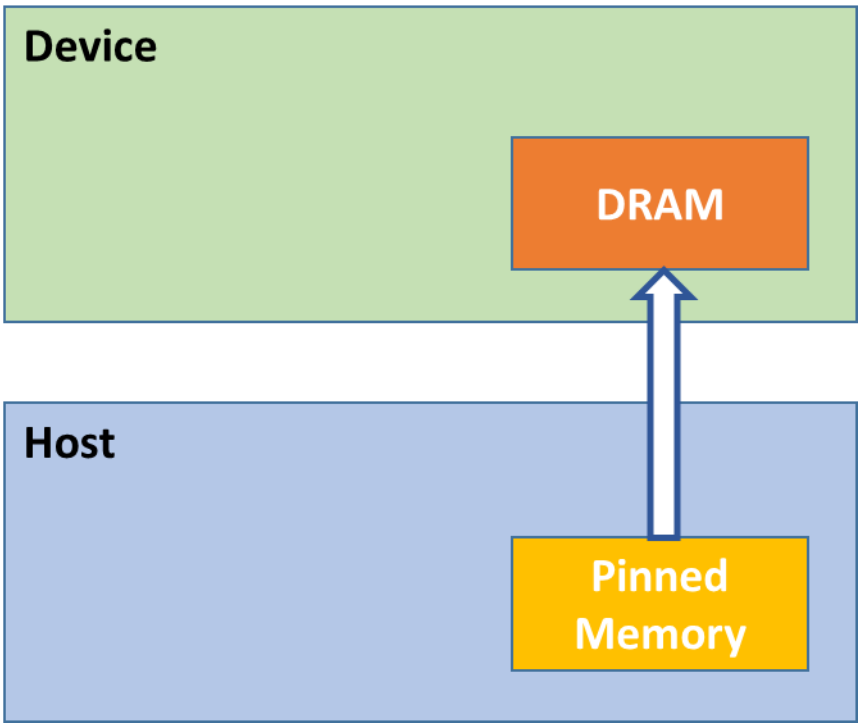


Figure 18 Using pinned host memory for the allocation and transfer

The use of the pinned host memory provides lower latency and increased bandwidth for both synchronous and asynchronous memory transfers. The excessive use of pinned memory will degrade host performance, since the available amount of pageable memory to the host decreases. The list of the memory management functions used in this work and their descriptions are listed in Table 10 [13].

Table 10 The memory management functions used for asynchronous memory transfers

| Function | Description |
|---|---|
| `cudaMallocHost(void** ptr, size_t size)` | Allocates pinned memory on the host |
| `cudaFreeHost(void* ptr)` | Frees pinned host memory |

CUDA provides concurrency by the execution of asynchronous commands in streams. CUDA devices have a default stream, usually referred as `stream 0` or NULL stream, which is synchrous with all streams and its operations cannot overlap other streams. If it is not explicitly specified, all operations run on the default stream. Figure 19 shows that how the NULL stream handles the

operations in a simple multiplication program running on GPU. Kernel is only launched after the completion of the transfer of matrix A and B to the GPU memory and the resulting matrix C can be copied to CPU after the kernel completes its execution.


Figure 19 The operations on the NULL stream

CUDA streams let us to overlap these actions in order to achieve concurrency. In order to create and handle the streams, the following stream management functions in Table 11 are used provided in CUDA Runtime API [13]. `cudaStream_t` is used to declare streams as a variable.

Table 11 The utilized stream management functions

| Function | Description |
|---|---|
| `cudaStreamCreate(cudaStream_t *stream)` | Create an asynchronous stream |
| `cudaStreamDestroy(cudaStream_t stream)` | Destroys and cleans up an asynchronous stream |
| `cudaStreamSynchronize(cudaStream_t stream)` | Waits for stream tasks to complete |

In this method, the matrix multiplication and division workload is distributed over the streams to maintain a 3-way concurrency which is illustrated in Figure 20.

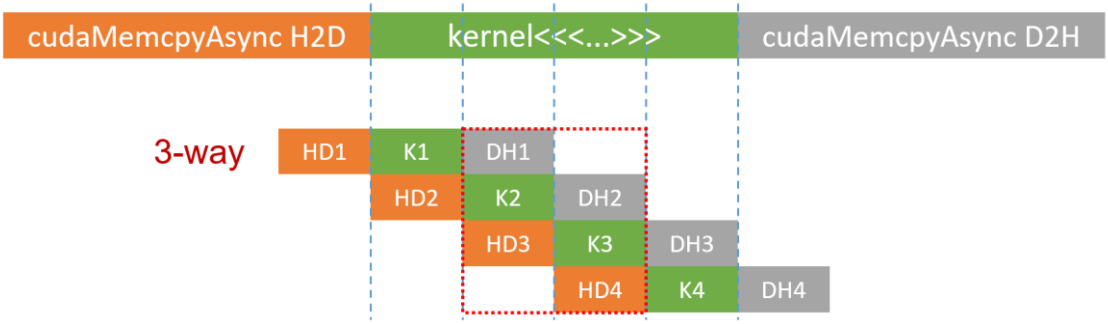


Figure 20 The amount of concurrency

### 5.3. Zero-Copy Memory

Normally, the host and device cannot directly access each other's variable. Zero-copy memory is one exception to this behaviour. Zero-copy memory is can be accessed by both the CPU and GPU.

The host memory allocation must be pinned and it is mapped into the GPU address space. `cudaHostAlloc(void** ptr, size_t size, unsigned int flags)` is used for allocation but with `cudaHostAllocMapped` flag [13]. `cudaMallocHost` is given `cudaHostAllocDefault` as flag, it emulates `cudaMallocHost` function which is used for pinned memory allocation for asynchronous memory transfers. The host memory can be accessed via `cudaHostGetDevicePointer` which passes back a device pointer *ptr*. This pointer can be directly passed to the kernel without any memory copy operation. But the memory accesses must be synchronized. Both the host and device should not access or modify the same memory space at the same. This will result in unpredictable behaviour.

If the device memory is insufficient, it can be used to laverage from the host memory. Since the host memory is accessed over PCI-Express, the latency is much worse than global memory.

The excessive use of pinned memory will degrade host performance in both asynchronous data transfer and zero-copy memory.

### 5.4. Using `cudaMallocManaged` with Unified Memory

Eventhough GPUs have very fast memories, the transfer speed of the data to the GPU is always be a limitation to get the best out of GPU performance. In GPU applications, it is desired that the data to be as close to the GPU as possible.

The explicit memory copies are the traditional way of transfering data from the CPU to the GPU or vice versa. Alltough this approach usually provides the best GPU performance, it needs to give much attention to the handling of the data access to get the most out of it.

In 2011, with CUDA 4.0, a special addressing type called Unified Virtual Addressing (UVA) is introduced. Before UVA is introduced, the pointers refering to the host and device memory have to be managed individually like it is described previously. With UVA, the pinned host memory has identical pointers for the host and device. As a result, it can be passed directly to a kernel. With UVA, it is not necessary to assign a device pointer to the pinned memory space like it is done with `cudaHostGetDevicePointer` for the zero-copy. The rest of the procedure is identical with the zero-copy memory.

A new support called Unified Memory was introduced with CUDA 6.0, in order to make the memory management even easier. A managed memory pool can be created with Unified Memory which is reachable from both the host and device with the same pointer. The data is automatically migrated to which processor needs access to the data. As a result, it improves performance and locality. On the other hand, UAV does not migrate data automatically like Unified Memory. UM lets the host and device to share a single virtual address space, as shown in Figure 21.

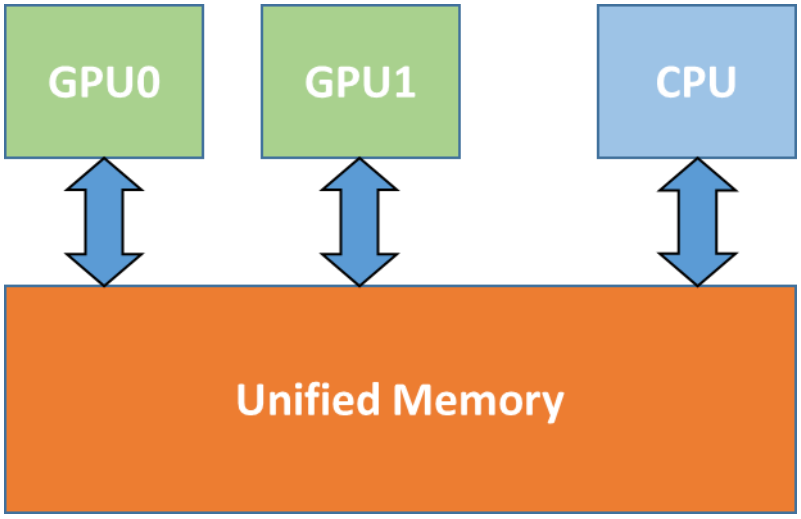


Figure 21 Single memory space with UM

Unified Memory provides a single pointer for the data like zero-copy memory. But zero-copy memory is located in host memory thus, the data can be accessed only over the PCIe bus which is prone to high-latency.

The functions provided by CUDA Runtime API [13] in order to allocate and free managed memory is given in the Table 12.

Table 12 The functions used for managed memory with Unified Memory

| Function | Description |
|---|---|
| `cudaMallocManaged(void** ptr, size_t size, unsigned int flags=0)` | Allocates memory that will be automatically managed by the Unified Memory system |
| `cudaFree(void* ptr)` | Frees managed memory |

But, Unified Memory is handled differently in our setups, due to the architectural differences. The Setup-1 has a GPU based on Maxwell architecture. In Maxwell architecture, all managed memory used by the CPU for writing has to be synchronized with the device before a kernel launch. So, the size of the Unified Memory is limited to physical memory in GPU.

The Setup-2 has GTX 1080Ti GPU which is based on Pascal architecture. With the Pascal GPUs, NVIDIA introduced a hardware support to extend Unified Memory management [36]. In Maxwell architecture, the migration of the data from CPU to GPU is normally handled by CUDA runtime. With this hardware support, Pascal GPUs gained 49-bit virtual addressing ability which is large enough to cover the virtual address space of the device and the host. Thus, the UM is no longer limited to device memory size, the full memory size of the system becomes accessible.

Another feature comes with the hardware support is page faulting. This means that CUDA driver will not need synchronize the managed memory before any kernel launch. If the managed memory space accessed by any processor is not available yet, we will have a page fault. This page fault will trigger an automatic migration of the data, as a result, the need of a data synchronization is removed.

The parallel programming usually starts with serial coding for CPU and ends up with porting the CPU code to the GPU. We start with the allocation on GPU, adding a copy feature to the code and replacing the multiplication function. In

order to use the pinned memory, the allocation of the data is needed to be changed. The use of zero-copy removed the extra data allocation and the copy requirement. But we still needed a copy for transferring the results to the CPU. Lastly, the Unified Memory let us to remove the transfer requirement for the results. The last code was very similar to the CPU code. An example for the transformation of the code from base CPU code to the GPU code with Unified Memory is given in Figure 22.

```
//N: length of the data
//pointers to data
float *dataA, *dataB, *result;
//allocate space data
dataA = (float *)malloc(N);
dataB = (float *)malloc(N);
result = (float *)malloc(N);

    .   //operations that initialize
    .   //dataA and dataB
    .

multiplyCPU(dataA, dataB, result, N);

free(dataA);
free(dataB);

    .   //operations that use result
    .
    .

free(result);
```

```
//N: length of the data
//pointers to data
float *dataA, *dataB, *result;
//allocate space data
cudaMallocManaged(&dataA, N);
cudaMallocManaged(&dataB, N);
cudaMallocManaged(&result, N);

    .   //operations that initialize
    .   //dataA and dataB
    .

multiplyGPU<<< ... >>>(dataA, dataB, result, N);
cudaDeviceSychronize();
cudaFree(dataA);
cudaFree(dataB);

    .   //operations that use result
    .
    .

cudaFree(result);
```

Figure 22 CPU to GPU code transformation with Unified Memory

# 6. RESULTS

In this section, the results obtained from the original CPU based method and the implemented GPU based methods will be discussed. Before providing the results, it is needed to be explained how the measurements are made.

The time spent in CPU methods are measured with the help of OpenCV libraries. OpenCV has a `cv::TickMeter` class to measure the passing time. This class has public member functions `start()`, `stop()` in order to start and stop the timer as expected. The passed time between these two functions can be gathered with `getTimeSec()` member function.

CUDA Runtime API provides event management functions in order to measure the timings in GPU. The functions that are used in this work given in Table 13. `cudaEvent_t` is used for the declaration of events.

In order to examine memory transfers, kernel execution and streams for optimization, NVIDIA Visual Profiler [37] is used. It can detect performance bottlenecks and provide suggestions for improvements.

Table 13 The event management functions used for measurements

| Function | Description |
|---|---|
| `cudaEventCreate(cudaEvent_t* event)` | Creates an event object |
| `cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)` | Records an event |
| `cudaEventSynchronize(cudaEvent_t event)` | Waits for an event to complete |
| `cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t end)` | Computes the elapsed time between events |
| `cudaEventDestroy(cudaEvent_t event)` | Destroys an event object |

The maximum difference between the CPU and GPU calculations is measured as 0,2% which affects the tracking accuracy by a maximum of 0,3%. The average calculation error is 0,04% and having negligible effect on the overall accuracy.

### 6.1. Results obtained with Setup-1

As stated in the Section 4.2, the Setup-1 has Intel Core i7-7500U CPU @2.7GHz, 16GB of RAM and NVIDIA GeForce 940MX with 384 cuda cores and 2GB memory. The operating system is Ubuntu 16.04 with CUDA 8.0 and OpenCV 3.4.4 installed. The algorithm has run on five datasets and they include 243.299 frames in total. The total time spent on the multiplication and the division processes is given in Table 14. According to these measurements, the calculations in GPU are two times faster than the original CPU based methods.

Table 14 The total time spent in CPU and GPU kernels, and achieved speedup on Setup-1

| CPU Kernel (in sec) | GPU Kernel (in sec) | Speed-up |
|---|---|---|
| 10005,87 | 1633,01 | ~6,13x |

The time consumed in memory transfers and kernels are analized using NVIDIA Profiling Tool. The profiler output for a single call to the multiplication method with around 2.000.000 elements for each matrix is given in Figure 23.

The sequential approach is eligible to use both pageable host memory and pinned host memory with the synchronous memory transfers. The sequential approach with the pinned host memory is 6,64% faster than with the pageable host memory. The use of asynchronous CUDA memory transfers and streams for 3-way concurrency provided a 5,11% improvement over the synchronous approach with the pinned host memory. Using the streams, the time spent in kernel is overlapped with the memory transfers and has no effect on the total processing time as can be seen in Figure 23(c). Since 940Mx has one copy engine, the host to device and the device to host data transfers can not overlap.

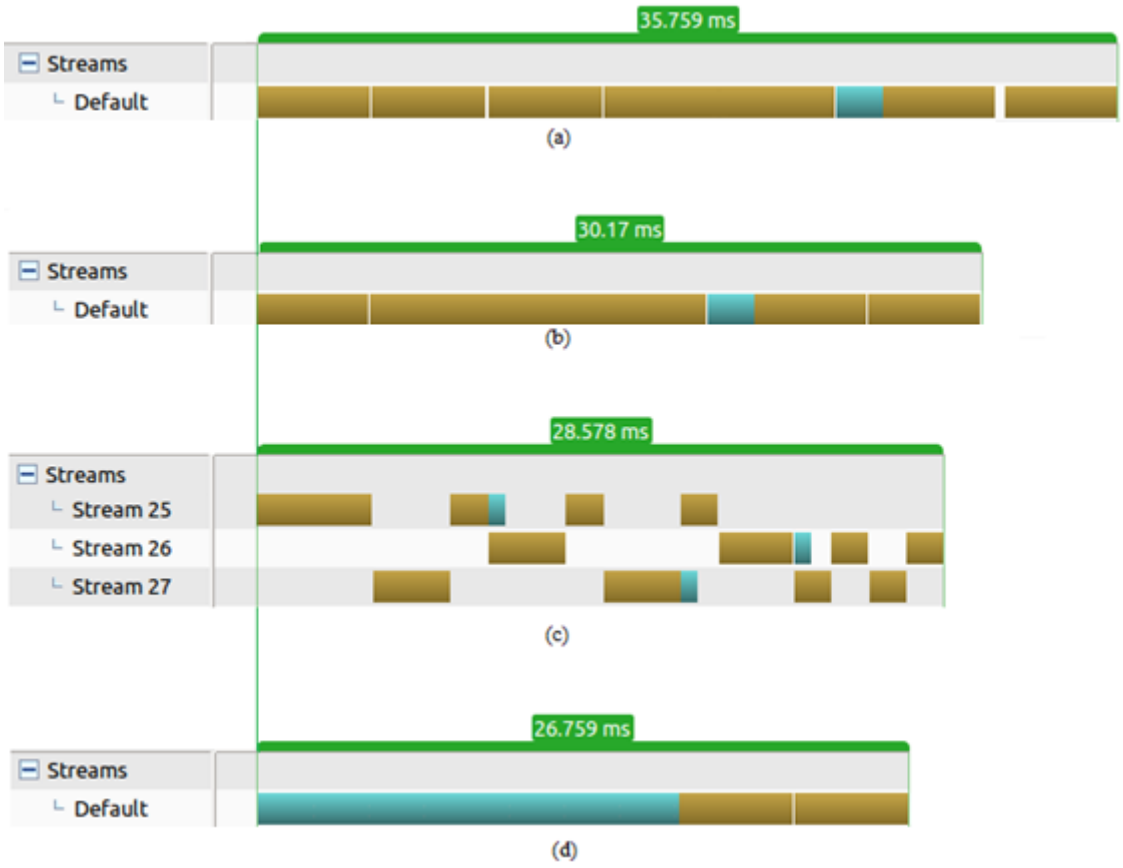


Figure 23 The execution time of only one method on Setup-1. (a) the sequential approach with the pageable host memory, (b) the sequential approach with the pinned host memory, (c) the asynchronous approach with 3 streams, (d) with no-memcpy data pointers.

On the other hand, this is not the ideal gain from the 3-way concurrency since it is possible to provide up to 3x speed-up. This is because the complexity of the calculations in kernels is not high enough to fully overlap with the time spent on the asynchronous memory transfers.

The profiler output in Figure 23(d) shows that there is no time spent in copying the data to the GPU but the time spent in kernel is increased dramatically due to the lack of bandwidth. Eventhough the kernel execution time increased, the total process is sped up by 21,75% with respect to the synchronous memory transfer with pinned host memory.

In GPU applications, the memory transfers have been a greater problem for achieving better performance results. This is why we have used five different method in order to retrieve data on GPU. The primitive sequential approach with pageable memory is selected as base method for comparison. The speed-ups obtained on Setup-1 are summarized in Table 15.

Table 15 The measured running times and achieved speed-ups over the pageable memory on Setup-1 with respect to the serial approach

| | **Pageable Memory** | **Pinned Memory cudaMemcpy** | **Pinned Memory cudaMemcpyAsync** | **Zero-copy memory** | **Managed memory** |
|---|---|---|---|---|---|
| Time(sec) | 25648,33 | 23945,48 | 22721,52 | 19825,56 | 27570,44 |
| Speed-up | 1x | ~1,07x | ~1,13x | ~1,29x | ~0,93x |

## 6.2. Results obtained with Setup-2

The Setup-2 is a workstation which is built for applications like big data processing, image processing, data mining, etc. It has 40 Intel Xeon Silver 4114 cores, 128 GB of RAM and two NVIDIA GTX1080Ti GPUs based on the NVIDIA Pascal architecture. Each of the GPUs has 3584 CUDA cores, memory speed up-to 11Gbps with 352-bit memory interface width and the memory bandwidth 484 GB/sec referred to spec-sheet. The maximum clock rate is 1620 MHz and memory clock rate 5505 MHz which are 36% and 175% more than 940MX of Setup-1 respectively. The operating system is Ubuntu 18.04 with CUDA 10.0 and OpenCV 3.4.4 installed.

Running the ECO algorithm on Setup-2 provided the kernel execution times as given in Table 16. As expected, sharing the workload between the threads led to 95,76% faster execution time over the CPU execution.

Table 16 The total time spent in CPU and GPU kernels, achieved speedup on Setup-2

| CPU Kernel (in sec) | GPU Kernel (in sec) | speedup |
|---|---|---|
| 5999,41 | 254,23 | ~23,60x |

Using the pinned host memory for the sequential approach instead of the pageable host memory provided 71,12% faster execution time. The asynchronous memory transfers with 3 streams were 76,30% faster than the base GPU method. The zero-copy memory usage also improved the execution time by 58,85% but not better than the other methods using pinned memory for explicit memory transfers. Finally, the use of the managed memory was only 14,72% faster than the base method with pageable memory. The achieved speed-ups over the base GPU approach are summarized in Table 17.

Table 17 The achieved speed-ups on Setup-2 with respect to the serial approach with pageable memory transfers

| | Pageable Memory | Pinned Memory cudaMemcpy | Pinned Memory cudaMemcpyAsync | Zero-copy memory | Managed memory |
|---|---|---|---|---|---|
| Time(sec) | 13932,69 | 4023,37 | 3302,08 | 5733,84 | 11881,83 |
| Speed-up | 1x | ~3,46x | ~4,22x | ~2,43x | ~1,17 |

At this point, we need to focus on the managed memory usage. At the first steps of the implementation with managed memory, the performance on GTX 1080Ti was much worse than the pageable memory. As described in Section 5.4, when a kernel try to access the memory, a page fault occurs and it triggers the migration of the data to GPU as demanded. But this causes migration time to interfere with the kernel execution and stalls it. CUDA provides a management function for prefetch the data on GPU side. In order to prefetch the data after the CPU initializes it, `cudaMemPrefetchAsync` is used before the kernel launch. The comparison of the cases with the profiler can be seen in Figure 24. Prefetching the data shortened the execution time by around 64.22%.

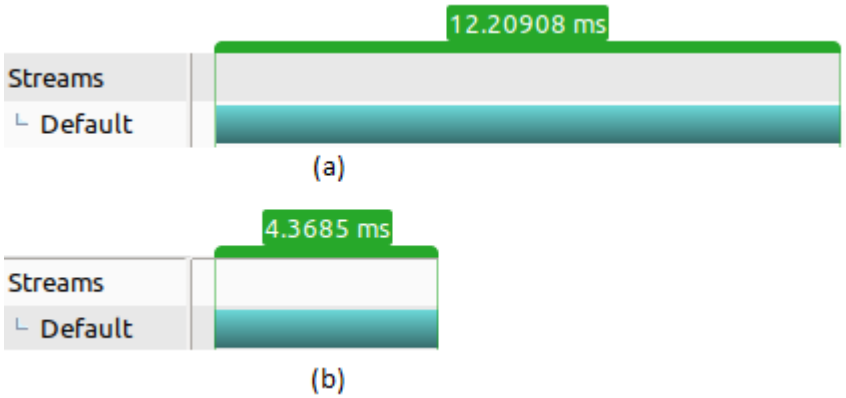


Figure 24 Profiler output for unified memory (a) without using cudaMemPrefetchAsync, (b) with prefetching memory before kernel launch

In order to better see the performance of the methods, the profiler output is given in Figure 25.

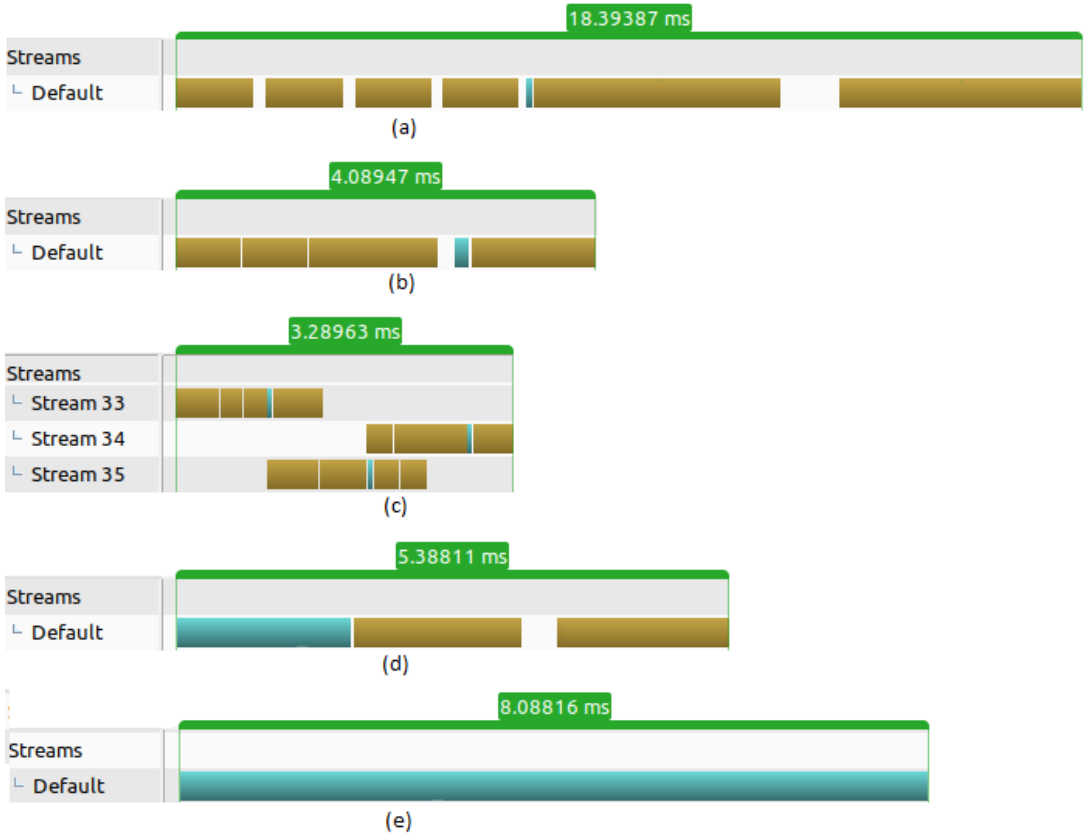


Figure 25 The execution time of only one method on Setup-2. (a) the sequential approach with the pageable host memory, (b) the sequential approach with the pinned host memory, (c) the asynchronous approach with 3 streams, (d) with no-memcpy data pointers, (e) using managed memory

As we can see in Figure 25(c), the input data to the stream kernels and results back to CPU are copied at the same time. GTX 1080Ti has two separate copy engines for the transfer data from host to device and from device to host.

# 7. CONCLUSION

We have obtained the benchmark results of ECO tracking algorithm on five datasets which have different characteristics. We have analyzed the algorithm and investigated its suitability for GPU parallelism with the help of a profiling tool.

After we have decided on candidate methods, we implemented GPU code for them starting from very naïve approach. The code implementation is done with NVIDIA's CUDA platform. CUDA provides C/C++ extension and APIs for programming and managing GPUs.

We managed to speedup its execution time with respect to the original implementation. After the first approach, we worked on the optimization of our implementation. We first started with the memory management. Page-locked (pinned) memory, zero-copy memory and Unified Memory are popularly used for increase GPU performance. We investigated their use and applied them to our implementation.

Later, we focused on another popular GPU parallelism concept called streams. We have introduced further improvements on the performance of the algorithm. We managed to achieve a 76,30% performance increase in execution time with respect to the original implementation.

We saw that the speed is not only related to the processing power of a processor. Since, we are always dealing with data, it is also important that how you reach it and affects the performance directly. Our work showed that traditional explicit memory copies may still provide better for performance but the gap is smaller than it used to be. The explicit memory transfers are subject to the programmers control and must be carefully handled. With the use of Unified Memory, one can transform a CPU code to GPU code faster. If you C/C++ code are using malloc for the memory allocation, all you need to do replace it with CUDA method then you are ready to go. This is very useful for fast prototyping and maintenance of a software.

# 8. REFERENCES


[1] M. Kristan *et al.*, "The visual object tracking vot2017 challenge results," in *Proceedings of the IEEE International Conference on Computer Vision Workshops (ICCVW)*, 2017, pp. 1949-1972, doi: 10.1109/ICCVW.2017.230.

[2] M. Danelljan, G. Bhat, F. Shahbaz Khan, and M. Felsberg, "Eco: Efficient convolution operators for tracking," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6931-6938, doi: 10.1109/CVPR.2017.733.

[3] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," *Integration,* vol. 69, pp. 309-320, 2019, doi: 10.1016/j.vlsi.2019.07.005.

[4] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Information Fusion,* vol. 42, pp. 146-157, 2018, doi: 10.1016/j.inffus.2017.10.006.

[5] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, "Visual object tracking using adaptive correlation filters," in *2010 IEEE computer society conference on computer vision and pattern recognition*, 2010, pp. 2544-2550, doi: 10.1109/CVPR.2010.5539960.

[6] Z. Chen, Z. Hong, and D. Tao, "An experimental survey on correlation filter-based tracking," *arXiv preprint arXiv:1509.05520,* 2015.

[7] M. Danelljan, G. Häger, F. Khan, and M. Felsberg, "Accurate scale estimation for robust visual tracking," in *British Machine Vision Conference, Nottingham, September 1-5, 2014*: BMVA Press.

[8] M. Danelljan, A. Robinson, F. S. Khan, and M. Felsberg, "Beyond correlation filters: Learning continuous convolution operators for visual tracking," in *European conference on computer vision*, 2016, pp. 472-488, doi: 10.1007/978-3-319-46454-1_29.

[9] A. Lukezic, T. Vojir, L. ˇCehovin Zajc, J. Matas, and M. Kristan, "Discriminative correlation filter with channel and spatial reliability," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6309-6318, doi: 10.1007/s11263-017-1061-3.

[10] C. F. Hester and D. Casasent, "Multivariant technique for multiclass pattern recognition," *Applied Optics,* vol. 19, pp. 1758-1761, 1980, doi: 10.1364/AO.19.001758.

[11] C. McClanahan, "History and evolution of gpu architecture," *A Survey Paper,* vol. 9, 2010.

[12] "Fermi Whitepaper." NVIDIA Corporation. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed May 2020).

[13] "CUDA C++ Programming Guide." NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed May 2020).

[14] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. Fei-Fei, "Imagenet large scale visual recognition competition 2012 (ilsvrc2012)," *See net. org/challenges/LSVRC,* p. 41, 2012, doi: 10.1007/s11263-015-0816-y.

[15] K. Dai, D. Wang, H. Lu, C. Sun, and J. Li, "Visual tracking via adaptive spatially-regularized correlation filters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4670-4679.

[16] S. Liu, G. Liu, and H. Zhou, "A robust parallel object tracking method for illumination variations," *Mobile Networks and Applications,* vol. 24, no. 1, pp. 5-17, 2019.

[17] X. Sun, P. L. Rosin, R. R. Martin, and F. C. Langbein, "Bas-relief generation using adaptive histogram equalization," *IEEE transactions on visualization and computer graphics,* vol. 15, no. 4, pp. 642-653, 2009.

[18] L.-H. Chen, Y.-H. Yang, C.-S. Chen, and M.-Y. Cheng, "Illumination invariant feature extraction based on natural images statistics—Taking face images as an example," in *CVPR 2011*, 2011: IEEE, pp. 681-688.

[19] L. Li, W. Huang, I. Y.-H. Gu, and Q. Tian, "Statistical modeling of complex backgrounds for foreground object detection," *IEEE Transactions on Image Processing,* vol. 13, no. 11, pp. 1459-1472, 2004.

[20] G. Silveira and E. Malis, "Real-time visual tracking under arbitrary illumination changes," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, 2007: IEEE, pp. 1-6.

[21] H. Fan and H. Ling, "Parallel tracking and verifying: A framework for real-time and high accuracy visual tracking," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 5486-5494.

[22] L. Bertinetto, J. Valmadre, S. Golodetz, O. Miksik, and P. H. Torr, "Staple: Complementary learners for real-time tracking," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1401-1409.

[23] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005, vol. 1: IEEE, pp. 539-546.

[24] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556,* 2014.

[25] Y. Wu, J. Lim, and M.-H. Yang, "Object tracking benchmark," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 37, no. 9, pp. 1834-1848, 2015.

[26] P. Liang, E. Blasch, and H. Ling, "Encoding color information for visual tracking: Algorithms and benchmark," *IEEE Transactions on Image Processing,* vol. 24, no. 12, pp. 5630-5644, 2015.

[27] M. Kristan *et al.*, "The visual object tracking vot2015 challenge results," in *Proceedings of the IEEE international conference on computer vision workshops*, 2015, pp. 1-23.

[28] M. Kristan *et al.*, "The Visual Object Tracking VOT2016 Challenge Results," Cham, 2016: Springer International Publishing, in Computer Vision – ECCV 2016 Workshops, pp. 777-823, doi: 10.1007/978-3-319-48881-3_54.

[29] M. Kristan *et al.*, "The seventh visual object tracking vot2019 challenge results," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2019, pp. 0-0.

[30] "Visual Tracker Benchmark." http://www.visual-tracking.net (accessed May 2020).

[31] A. Moudgil and V. Gandhi, "Long-term visual object tracking benchmark," in *Asian Conference on Computer Vision*, 2018: Springer, pp. 629-645.

[32] M. Mueller, N. Smith, and B. Ghanem, "A benchmark and simulator for uav tracking," in *European conference on computer vision*, 2016: Springer, pp. 445-461, doi: 10.1007/978-3-319-46448-0_27.

[33] J. Tursa. "Fast Matrix Multiply with Multi-Dimensional Support." https://www.mathworks.com/matlabcentral/fileexchange/25977-mtimesx-fast-matrix-multiply-with-multi-dimensional-support (accessed May 2020).

[34] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," *rn (A+ B),* vol. 21, no. 1, pp. 18749-19424, 2011.

[35] "Valgrind." https://valgrind.org/ (accessed May 2020).

[36] "Pascal Whitepaper." NVIDIA Corporation. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (accessed May 2020).

[37] "NVIDIA Visual Profiler." NVIDIA Corporation. https://developer.nvidia.com/nvidia-visual-profiler (accessed May 2020).