CUDA BASED REAL TIME IMPLEMENTATION OF REGION COVARIANCE DESCRIPTORS

ALAN KOVARYANS MATRİSLERİNİN CUDA TABANLI GERÇEK ZAMANLI PARALEL HESAPLANMASI

MUHAMMET ALİ ASAN

Assist. Prof. Dr. ADNAN ÖZSOY
Supervisor

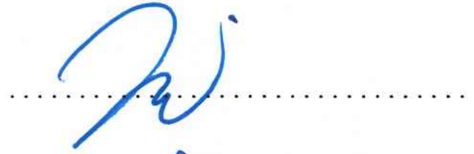
Submitted to Graduate School of Science and Engineering of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Engineering

September 2019

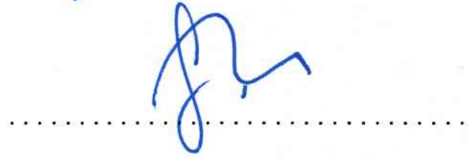
This work titled **"CUDA Based Real Time Implementation Of Region Covariance Descriptors"** by **Muhammet Ali ASAN** has been approved as a thesis for the Degree of **Master of Science** in **Computer Engineering** by the Examining Committee Members mentioned below.

Prof. Dr. Asım Egemen YILMAZ
Head

Assist. Prof. Dr. Adnan ÖZSOY
Supervisor



Assoc. Prof. Dr. Süleyman TOSUN
Member



Assist. Prof. Dr. Ayça TARHAN
Member

Assist. Prof. Dr. Gönenç ERCAN
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 School of Science and Engineering on ....../....../..........

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU
Director of the Institute of
Graduate School of Science and Engineering

”Hayatta en hakiki mürşit, ilimdir.”

Mustafa Kemal ATATÜRK

ETHICS

In this thesis study, prepared in accordance with the spelling rules of Institute of Graduate Studies in Science of Hacettepe University,

I declare that

- all the information and documents have been obtained in the base of the academic rules.
- all audio-visual and written information and results have been presented according to the rules of scientific ethics
- in case of using others works, related studies have been cited in accordance with the scientific standards
- all cited studies have been fully referenced
- I did not do any distortion in the data set
- and any part of this thesis has not been presented as another thesis study at this or any other university.

11/09/2019

MUHAMMET ALİ ASAN

YAYINLAMA VE FİKRİ MÜLKİYET HAKLARI BEYANI

Enstitü tarafından onaylanan lisansüstü tezimin / raporumun tamamını veya herhangi bir kısmını, basılı (kağıt) ve elektronik formatta arşivleme ve aşağıda verilen koşullarla kullanıma ama iznini Hacettepe Üniversitesine verdiğimi bildiririm. Bu izinle Üniversiteye verilen kullanım hakları dışındaki tüm fikri mülkiyet haklarım bende kalacak, tezimin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım hakları bana ait olacaktır.

Tezin kendi orijinal çalışmam olduğunu, başkalarının haklarını ihlal etmediğimi ve tezimin tek yetkili sahibi olduğumu beyan ve taahhüt ederim. Tezimde yer alan telif hakkı bulunan ve sahiplerinden yazılı izin alınarak kullanılması zorunlu metinlerin yazılı izin alınarak kullandığımı ve istenildiğinde suretlerini Üniversiteye teslim etmeyi taahhüt ederim.

Yükseköğretim Kurulu tarafından yayınlanan **" Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"** kapsamında tezim aşağıda belirtilen koşullar haricinde YÖK Ulusal Tez Merkezi / H. Ü. Kütüphaneleri Açık Erişim Sisteminde erişime açılır.

- o Enstitü / Fakülte yönetim kurulu kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren 2 yıl ertelenmiştir. [(1)]

- o Enstitü / Fakülte yönetim kurulunun gerekçeli kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren .... Ay ertelenmiştir. [(2)]

- o Tezimle ilgili gizlilik kararı verilmiştir. [(3)]

26/09/2019
Muhammet Ali ASAN

ABSTRACT

CUDA BASED REAL TIME IMPLEMENTATION OF REGION COVARIANCE DESCRIPTORS

Muhammet Ali ASAN
Master of Science, Computer Engineering Department
Supervisor: Assist. Prof. Dr. Adnan ÖZSOY

September 2019, 47 pages

Computation time of computer vision applications is critical for real-time applications such as video processing. While applied methods achieve real-time performance, it is also vital to allow processors to work on other tasks. In order to use in real time video processing tasks, an algorithm should achieve 30 frames per second. In computer vision, the input data mostly consists of two dimensions thus running time of most applications are proportional to the square of input size.

To allow a computer system to process images, the computer must be able to understand the image. Image descriptors play an important role to help computers with understanding of images. Extracting features from image, classification, recognition, comparison of images becomes possible by using these features with image descriptors. In this study, a parallel implementation of a robust image descriptor called Region Covariance Descriptor on GPU using CUDA is given. While the serial algorithm is not enough for real-time processing, applied parallel implementation achieves real-time performance.

In this study, parallel and asynchronous computation of region covariance descriptors on GPU both allows CPU to perform other tasks and achieves 30 images to be processed in a

second. Region Covariance Descriptor which is introduced in object recognition and image classification application at first makes it possible to use covariance of basic image features like color, gradients as a robust descriptor. This descriptor is also applied to several image filtering and object tracking problems as well. As region covariance descriptors do not live in Euclidean space, traditional vector distance measurements methods can not be used to compare them. Instead, a symmetric positive definite matrix comparison method which is based on generalized eigenvalue of two matrices is used to compare covariance matrix. But because this method is complex, porting the algorithm to GPU is difficult and execution time takes longer. After our reviews on existing methods, we have replaced traditional covariance matrix comparison method with a metric which is robust, easy and fast to compute on GPU. In this study analysis of these two distance metric calculation methods is also discussed.

One of the main contributions of this work is that in this study an existing CUDA based integral image computation on GPU is replaced by a novel approach which works faster and uses less memory than the existing method. We should note that the proposed approach does not leverage any development in hardware between two works.

Keywords: GPU, Parallel Computing, Object Classification, Image Descriptors, Object Tracking, CUDA

ÖZET

ALAN KOVARYANS MATRİSLERİNİN CUDA TABANLI GERÇEK ZAMANLI PARALEL HESAPLANMASI

Muhammet Ali ASAN
Yüksek Lisans, Bilgisayar Mühendisliği
Danışman: Dr. Ögretim Üyesi Adnan ÖZSOY
Eylül 2019, 47 sayfa

Bilgisayarlı görü uygulamalarında kullanılan yöntemlerin çalışma süreleri gerçek zamanlı uygulamalar için (Örnek:Video İşleme) kritiktir.Uygulanan yöntemlerin gerçek zamanlı performans gösterebilmesi ve başka görevlerin de yerine getirilebilmesi gerekmektedir.Bir algoritmanın saniyede 30 görüntü ve üzerinde uygulanabilmesi gerçek zamanlı video işleme uygulamaları için yeterlidir.Bilgisayarlı görü uygulamalarında işlenen veri en az 2 boyutlu olduğundan algoritma çalışma süresi çoğunlukla veri boyutunun karesi ile orantılı olarak değişmektedir.

İşlenen verinin üzerinde çalışılabilmesi için görüntüler bilgisayar tarafından anlamlandırılabilmelidir.Bu noktada görüntü tanımlayıcılar (image features) önemli rol oynar.Görüntü üzerindeki nitelikler çıkartılarak bilgisayar tarafından görüntülerin sınıflandırma,tanıma,karşılaştırma işlemlerinin yapılabilmesi mümkün olmaktadır.Bu tez çalışmasında başarısı kanıtlanmış bir görüntü tanımlayıcı olan fakat işlenen veriye bağlı olarak yavaş çalışma zamanı olan alan kovaryans tanımlayıcıların (region covariance descriptor) gerçek zamanlı uygulamalarda kullanılabilmesi için paralel olarak hesaplanması gerçekleştirilmiştir.

Çalışmada GPU üzerinde bir paralel uygulama amaçlanmıştır.Alan kovaryans tanımlayıcılarının hesaplanması ve karşılaştırılması GPU üzerinde paralel bir şekilde yapılarak hem işlemi saniyede 30 görüntü işlenebilecek çalışma süresine getirmek hem de asenkron olarak CPU da başka işlemler yapılabilmesi mümkün olmuştur. Nesne tanımlama ve görüntü sınıflandırma uygulaması olarak literatüre kazandırılan alan kovaryans matrisleri görüntü üzerindeki renk tonları,gradyanlar gibi basit özniteliklerin ortak değişimlerini tanımlayıcı olarak kullanarak güçlü tanımlayıcılar elde etmemize olanak tanımaktadır.Alan kovaryans matrisleri görüntü yumuşatma,nesne takibi problemlerinde de başarılı bir şekilde uygulanmıştır.Alan kovaryans matrisleri Öklid uzayında yer almadığından vektör uzaklık ölçme yöntemleri bu matrisleri karşılaştırmada kullanılamamaktadır. Kovaryans matrisleri simetrik pozitif tanımlı matrisler olduğundan iki matrisin ortak eigen değerlerinin hesaplanması gibi karmaşık hesaplamalar gerektiren bir simetrik pozitif tanımlı matris karşılaştırma yöntemi kullanılmaktadır. Bu işlem ise hesaplama açısından karmaşık olduğundan hem GPU da hesaplanması zor hem de çalışma süresi uzundur. Literatür araştırması sonucu, kovaryans matrisleri karşılaştırmak için hem GPU da hesaplanması mümkün hem de geleneksel karşılaştırma yönteminden daha hızlı çalışan bir yöntem uygulanmıştır.Çalışmada bahsi geçen hesaplama yöntemlerinin paralel uygulanabilirliği,GPU üzerinde çalışma sürelerine etkileri ve verimlilikleri tartışılmıştır.

Bu tez çalışmasının ana katkılarından biri de literatürde var olan GPU üzerinde CUDA tabanlı integral görüntü hesaplama yöntemine alternatif olarak daha hızlı calışan ve daha az hafıza kullanılan bir yöntemin önerilmesidir. Önerilen yöntem iki çalışma arasındaki donanımsal herhangi bir gelişmeye bağlı değildir.

Anahtar Kelimeler: GPU, Paralel Hesaplama, Nesne Sınfılandırma, Görüntü Tanımlayıcılar, Nesne Takibi, CUDA

ACKNOWLEDGEMENTS

First and foremost, I would like to sincerely thank to my supervisor Asst. Prof. Dr. Adnan Özsoy for his time, patience and for all his valuable guidance at every stage of my research along this long way. Without that support, It would have been impossible to write this thesis.

Besides, I would like to thank to my thesis committee members for reviewing this thesis and giving precious comments. Moreover, I am grateful to my all friends and colleagues for their help, good wishes and would like to give special thanks to Filiz Pektaş for her moral support.

Finally, I express my most sincere gratitude to my family. They have supported my decisions and taught me what unconditional love means.

CONTENTS

Page

ABSTRACT i
ÖZET iii
ACKNOWLEDGEMENTS v
CONTENTS vi
FIGURES viii
1. INTRODUCTION 1
2. RELATED WORK 3
3. BACKGROUND 6
3.1. GPU Computing & CUDA 6
3.2. CUDA Programming Model 8
3.3. CUDA Execution Model 12
3.4. CUDA Memory Model 19
3.5. Object Detection 26
3.6. Region Covariance Descriptor 27
4. REGION COVARIANCE DESCRIPTOR CUDA IMPLEMENTATION 31
4.1. Covariance Matrix and Distance Calculation 35
5. RESULTS & ANALYSIS 37
5.1. Testbed Configuration 37
5.2. Test Results 37
6. CONCLUSION 43
REFERENCES 44

FIGURES

Page

3.1 CPU and GPU Communication 7
3.2 CUDA Computing Architecture 8
3.3 CUDA Programming Structure 9
3.4 CUDA Memory Hierarchy 10
3.5 Thread and Block Overview 11
3.6 Thread Alignment for a Typical Kernel 12
3.7 Overview of SM 13
3.8 SM and Thread Organization 14
3.9 Thread Visualizations 15
3.10 Thread States Illustrated 16
3.11 Register Overview of GPU 18
3.12 Shared Memory Overview of GPU 18
3.13 Memory Model Overview 20
3.14 Shared Memory Structure 22
3.15 Parallel Access to Shared Memory 24
3.16 Serial or Broadcast Access to Shared Memory 24
3.17 Random Access to Shared Memory 24
3.18 Padded Memory Access 25
3.19 Object Detection Steps 27
3.20 Summation Using Integral Images 29

4.1 RCD CUDA Implementation 31
4.2 Dedicated block grids for each input image 33
4.3 Row-wise summation of pixels 34
4.4 Column-wise summation of pixels. 34

5.1 Timeline view of CUDA calls 37

5.2 Comparison of Integral Image Computation on CUDA 39
5.3 Covariance Matrix Distance Calculation Methods Runtime 40
5.4 Detecting Target Object in Different Scales 40

TABLES

5.1 Profiling Results of our CUDA kernels 38

5.2 Average Computation Times 42

1. INTRODUCTION

Developments in hardware have increases computation performance naturally but the introduction of computation on graphic processor cards has further improved performance especially for computationally intensive tasks. Ability to assign tasks on a small but huge number of cores makes it possible to process large inputs in acceptable run-times. Image processing applications are a very good fit to be computed on the graphics card as it requires a huge number of data, millions of pixels in the image, to be used as input to the same instruction. Thus GPU is a great accelerator for image processing applications.

In this study, we handled object detection problem and tried to achieve real performance by leveraging the parallel computation on GPU. Object detection problem requires feature extraction; characteristics of pixels or regions in the image, feature descriptors; which combines features to represent regions and comparison of these descriptors to look for the region which is the most similar to searched region. It's been a very hot topic and many different features and feature descriptors are proposed. Each of them has an advantage over others but sometimes these advantages come with a computational cost.

This study targets Region Covariance Descriptor (RCD) proposed in [1]. RCD is constructed by the covariance of selected image statistics in an image patch. Here various statistics can be used as a feature, such as image intensity, color, edge orientation and responses, derivatives, and responses of various filters. These statistics can simply be concatenated into a vector which is then used to compute the covariance matrix. The resulting covariance matrix, as the authors state, naturally fuses the correlated features. It is also reported to be low-dimensional, further easing the computational burden compared to other high-dimensional descriptors. Lastly, as long as any ordering information is not included, RCD facilitates certain rotation and scale invariance.

After thorough research, we present, to our best knowledge, the first real-time GPU implementation of a widely used feature descriptor named *region covariance*. The pertinent descriptor is effectively an enhanced version of the statistical measure *covariance*, which in this context is used for representing a region (patch) of an image. We present a novel technique for optimizing the region covariance descriptor for GPUs where we achieve better performance than CPU, especially to meet real-time processing requirements. Beyond this, our

need for a way to compute integrals of multiple images as many as possible led us to replace the existing parallel computation of integral images with a more efficient implementation.

There are some serial implementations of RCD available online for CPU, the most prominent of them being written in MATLAB [2]. There is also an unofficial OpenCV implementation written in C++ [3] as a submission to Google Summer of Code.

Main contributions of this work can be summarized as follows :

- First real-time implementation of RCD
- An improved CUDA based image integral computation
- An efficient CUDA implementation for covariance matrix comparison

The rest of this thesis is organized as follows. In Chapter 2. similar works to accelerate image feature computations are discussed. In Chapter 3. CUDA functionalities, advantages of GPU, elements of GPU, object detection problem and theoretical background of RCD is briefly explained. In Chapter 4. our implementation considerations of CUDA based RCD is explained. Chapter 5. shows our test configuration, test results, comparisons with existing algorithms and discuses results. Chapter 6. sum ups the work explained in this thesis and discusses further improvements to achieved results.

2. RELATED WORK

Computer Vision (CV) is a field which addresses the vast amount of problems, spanning object detection, classification, moving object detection, segmentation, action recognition and many others. These problems, owing to their complexity, are primarily addressed using machine learning, low-level image processing and more recently, deep and complex models such as deep neural networks and graphical models [4].

In addition to the attention it received from the academic community, CV has always been at the forefront of industrial attention as well. Its ability to automate various tasks have made it vital to the industry. License plate recognition, face recognition, action recognition and visual object tracking algorithms are imperative for surveillance tasks [5]; image segmentation and low-level image processing are especially important in a medical image processing context [6]; stereo vision, real-time mapping and localization of objects are extremely important algorithms for robotics [7].

Conventional CPU architectures tend to fall short of meeting the demands of CV applications, at the heart of which resides massive matrix manipulation operations. Instead, academy and industry turned to more parallel-friendly hardware as matrix manipulations at scale are open to parallelism. One such hardware is Graphical Processing Units (GPU) which are designed to perform massively parallel executions, rose to the occasion and claimed their place as the ”go-to” hardware platform for CV community [8].

Before deep models where end-to-end learning is facilitated with raw image inputs, CV applications had a rather complex pipeline consisting of multiple stages; feature selection, extraction and then matching of features via a distance metric (for certain tasks, an ML algorithm can be plugged in here too). As this pipeline requires many iterations to fine-tune feature to be selected (the learning steps alike), it is of vital importance to select a feature descriptor with a good performance.
By development in hardware technology, concurrent executions become affordable and attracted the interest of computer vision researchers. By their nature computer vision applications require the same algorithm to apply to multiple data, i.e pixels in the image thus require parallelism as GPUs provide.

In [9], [10], [11] and [12], SIFT which is a widely used feature for image processing, is implemented on GPU and real-time performance is achieved. In a very recent study, the authors

in [13] provide a heterogeneous implementation of the Scale Invariant Feature Transform (SIFT) based on a CPU+GPU based on CUDA. They divide the SIFT feature computation steps into logical and computational domains and then selectively pass on the strong logical components to the CPU and the computationally intensive steps to the GPU. Data is exchanged between the CPU and the GPU to enable smooth operation. They achieve real-time performance with significant reductions in time while maintaining comparable performance to the CPU only implementation of SIFT. In [14], authors implemented SURF and SIFT on GPU for fingerprint identification task. For an autonomous navigation problem authors of [15] make use of GPU based SIFT implementation. [16] provides real-time performance for the detection of image forgery. Their system is a combination of sequentially applying the features From Accelerated Segment Test (FAST) and the Fast Retina Key-point binary descriptor (FREAK) to determine features and then perform matching. However, they parallelize the process by splitting the image into parts and then running the same sequence of steps on each part before matching is carried out. They have been able to achieve real-time performance for the tests performed. Feature extraction for medical image processing is considered in [17] where the authors discuss a parallel implementation of Haralick features [18]. Haralick features represent textural information in images are based on the gray level co-occurrence matrix (GLCM) which is computationally intensive to compute and requires a large amount of memory. The authors ease GLCM computation by encoding the data (to save memory) to reduce zero values and utilize the symmetric properties to provide a quicker determination which is then implemented on a CUDA GPU using the available parallelism.

For tracking problem, in [19] both KLT tracker [20] and SIFT features are implemented on GPU. In [21] authors have proposed a novel image de-blurring technique for mobile devices and provide GPU implementation of their method showing the importance of GPUs role in computer vision tasks. There are more examples in the literature that can be listed that make use of GPU based implementations for real-time performance. As image features are the words to describe the image to the computer, they are a vital part of any computer vision application that requires the understanding of shapes and objects in the image to give it a meaning.

Region covariance descriptor (RCD) is first proposed in [1]. A seminal work at its time, it computes the covariance of selected image statistics in an image patch. Various statistics (features) can be used, such as image intensity, color, edge orientation and responses, derivatives, and responses of various filters. These statistics can simply be concatenated

into a matrix which is then used to compute the covariance matrix. The resulting covariance matrix, as the authors state, naturally fuses the correlated features. It is also reported to be low-dimensional, further easing the computational burden compared to other high-dimensional descriptors. Lastly, as long as any ordering information is not included, RCD facilitates certain rotation and scale invariance.

Owing to its plug-and-play nature where any statistical information can be added/removed, RCD has been used for numerous vision tasks. In the original paper, it has been tested for object detection and texture classification. Other usages have been presented for tracking [22], image smoothing [23], associating successive video frames [24] and object detection [25]. All cases have reported improved results at their time, which are still comparable to the current state-of-the-art.[26] has a further explanation of usage, feature selection, and common issues about region covariance descriptors.

A very recent study [27] has presented parallel implementation of region covariance in GPU where they implemented [23] and reported 13 times improvement in speed for whole processing time. Our work differs from [27] mainly in terms of real-time processing in object detection. We also used different distance metrics which is easy and efficient to implement in CUDA but they followed traditional covariance distance calculation steps. Even they reported significant improvement in the run-time of the application, it could have been improved further by using the method described in [28].

Considering the gap in GPU implementation of region covariance descriptor we have proposed an implementation by effectively implementing integral images on CUDA and making use of a more lightweight and easy to implement distance metric on GPU. Proposed algorithm offers a run time which makes it possible to use it in real time image&video processing applications.

3. BACKGROUND

3.1. GPU Computing & CUDA

With new improvements in computer hardware, the definition of high-performance computing changes accordingly. Although the improvements might introduce new capabilities, the use of microprocessors is always in the center of everything to accomplish complex tasks with high throughput and efficiency. Getting high performance is not only related to hardware implementations, but also includes software tools and parallel programming paradigms.

Nowadays making use of GPUs in general purpose parallel programming is popular. Using GPUs in addition to CPUs is referred as heterogeneous parallel programming which leads us to new ways of architectural designs in parallel computation. Despite architectures of CPU and GPU are similar, a core of GPU is rather different than core of a CPU. The latter is heavyweight with complex control logic so it is designed for optimizing sequential programs. GPU core is lightweight with simple arithmetic logic and designed for data-parallel tasks to increase the throughput of parallel programs.

From a programmer's perspective, the problem in facilitating the GPUs for parallel programming is mapping concurrent calculations into a computer and executing them simultaneously on multiple computing resources. But the subtle point here is that how to break the problem into small sub problems and solving them concurrently. It is not only possible by software paradigms but also requires hardware side to realize efficient computation. The hardware aspect of it is about computer architecture while the software aspect pertains to parallel programming. Computer architecture is responsible for allowing parallelism on the architectural level. On the other hand, parallel programming aims solving problem concurrently by benefiting from computation power of underlying architectural level. So underlying architecture needs to provide structure to support concurrent execution of multiple threads or processes.

Parallelism is realized in two ways which are task parallelism and data parallelism. Task parallelism means many independent functions or tasks executed largely in a concurrent way; therefore, task parallelism can be defined as executing functions on multiple cores. But data parallelism is processing different chunks of data at the same time over the same functionality.

General Purpose Graphics Processing Units (GPGPUs) programming is most appropriate to carry out data parallelism. Because GPUs have thousands of small cores, on data parallelism tasks this small cores can apply same instructions on data heterogeneously while CPUs have to iterate over all data, depending on data size GPUs will outperform CPUs.

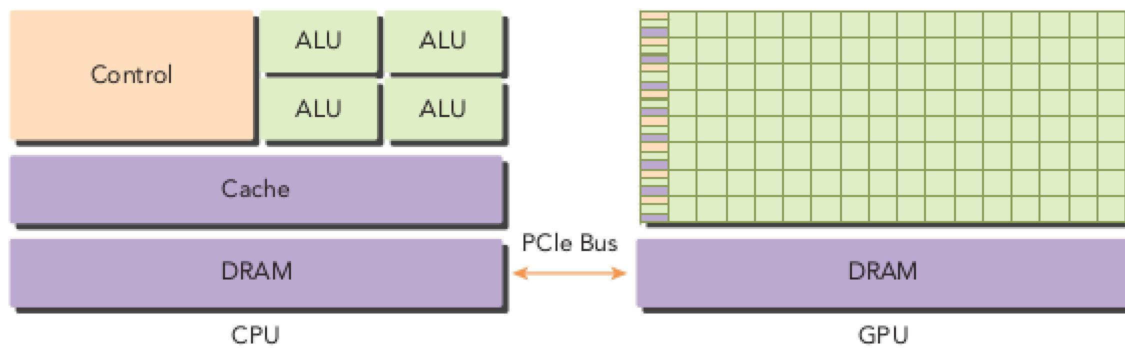


FIGURE 3.1: CPU and GPU Communication
Courtesy: Professional CUDA C Programming

GPU is not standalone computation tool but it is instead additional processor to CPU. So GPU must work together with CPU and the communication between two are provided by PCI Express bus. An example demonstration is given in Figure 3.1. CPU executes host code and is responsible for managing run time environment and providing data to GPU. GPU executes device code and is responsible for parallel data execution.

Leading graphics card manufacturer NVIDIA provides a software development kit called CUDA to allow programmers to make use of the aforementioned computational abilities of GPU devices. With abstraction provided by CUDA kit, data-parallel tasks can easily run on GPU. CUDA is a simple C library with utilities to allow programs to run functions concurrently on GPU cores. The execution order of a CUDA program is as follows:

1. Allocate device memory.
2. Copy data from host to device.
3. Invoke the CUDA kernel to perform computation.
4. Copy data back from device to host.
5. Destroy device memory.

3.2. CUDA Programming Model

The programming model is the bridge between the application and its implementation on hardware. How components of a program share information with each other, and how they work together coordinately is defined by the programming model. The programming model provides a logical view of computing architectures and it is involved in programming language and programming environment. This is also given schematically in Figure 3.2.

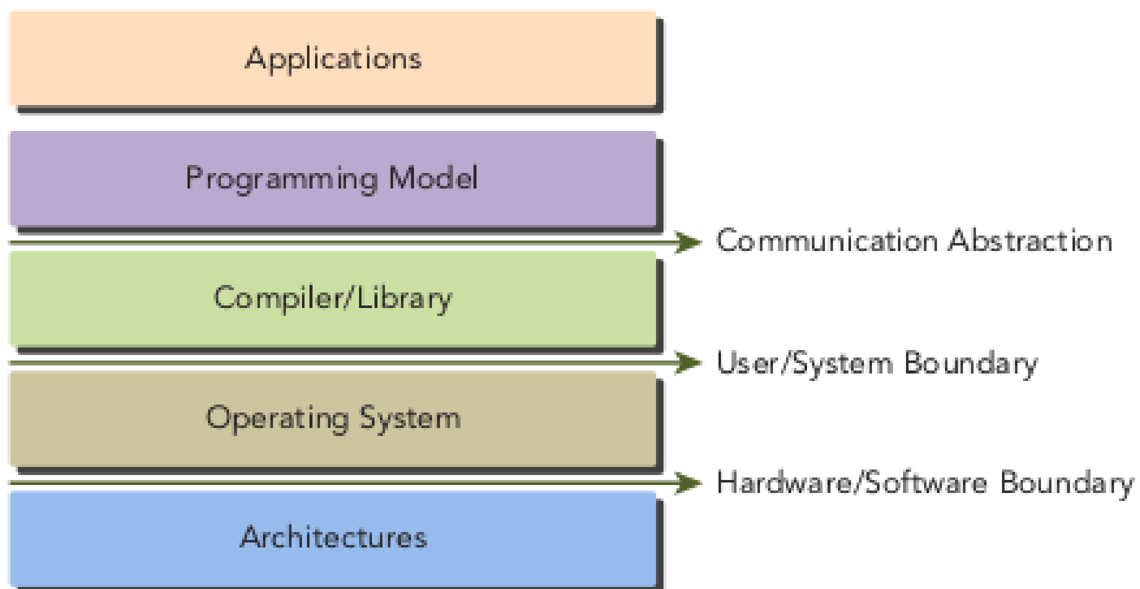


FIGURE 3.2: CUDA Computing Architecture
Courtesy: Professional CUDA C Programming

The programming abstraction specifies program and programming model which are realized via compiler and shared libraries. In CUDA, there are also two special features in addition to other programming models to make use of GPUs. These two features can be listed as :

- A way to organize threads on the GPU through a hierarchical structure
- A way to access memory on the GPU through a hierarchical structure

The essential component of CUDA programming is the kernel code that runs on GPU devices. Kernel code is not something different than sequential C program but CUDA is responsible for arranging written code to run on GPU threads concurrently. In the host code, the code that runs on CPU, programmers duty is to determine how the algorithm is mapped

to a GPU device. CUDA provides abstractions to work with GPU threads without knowing any detail of managing and creating GPU threads. An overview of the GPU structure and thread hierarchy is given in Figure 3.3.

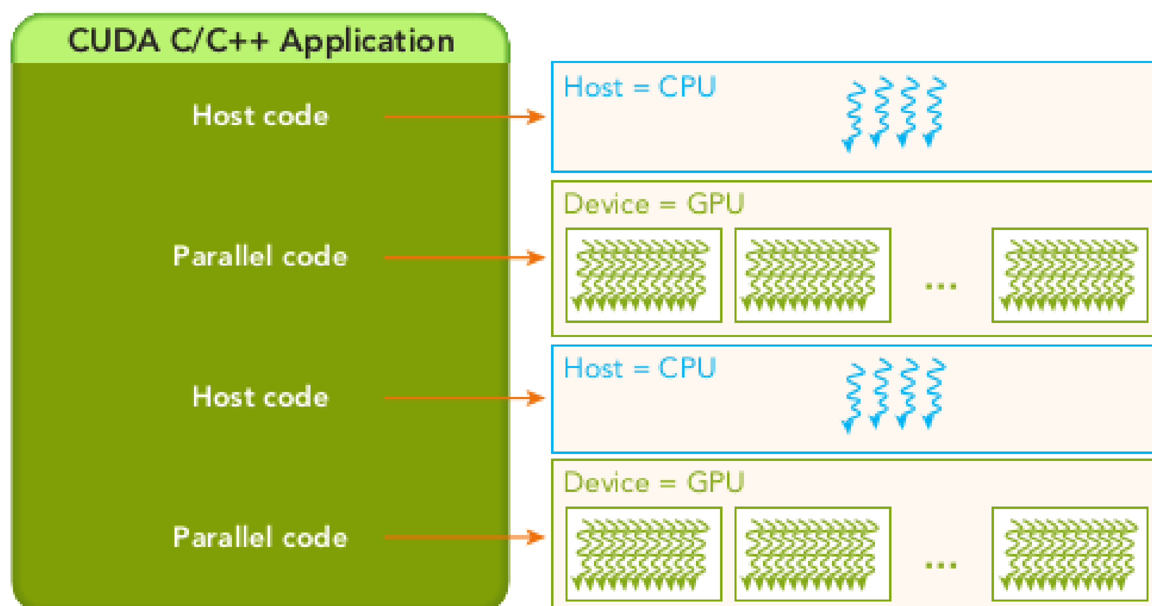


FIGURE 3.3: CUDA Programming Structure
Courtesy: Professional CUDA C Programming

CUDA codes work in the following manner. After executing kernel, code control is immediately returned to CPU so that it becomes ready to execute additional tasks. Therefore, it can be said that the CUDA programming model is primarily asynchronous. More technically, serial code which is written C is executed on host and parallel code written CUDA C is executed on the device where both kernel call and kernel execution is asynchronous.

Figure 3.4 shows CUDA programming model memory hierarchy. It is the abstraction of simplified memory hierarchy which shows two major parts: Global Memory and Shared Memory. Global Memory is like CPU system memory while shared memory is like CPU cache which can be directly controlled via CUDA.

After host invoked the kernel function, device takes over control and generates worker threads. One of the most vital parts of CUDA programming is to know the organization of these worker threads. CUDA provides thread hierarchy abstraction to allow to manage threads.

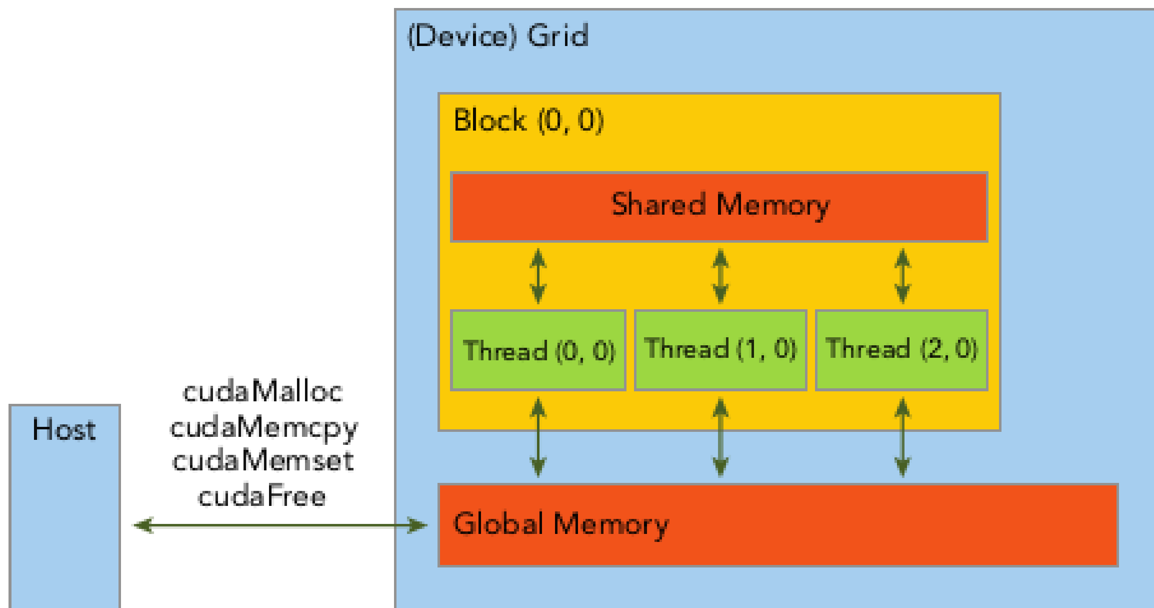


FIGURE 3.4: CUDA Memory Hierarchy
Courtesy: Professional CUDA C Programming

All threads generated by a kernel are called grid, which is organized as a 2D array of blocks [29]. Figure 3.5 is the representation of blocks of threads and grids of blocks.

When using threads, a programmer should be aware of the limitations of the hardware. Every selection of thread and block has a cost which is based on the hardware that is utilized. The size of grids and blocks depend on available registers and shared memory. Thread block can be constructed as 3D thread array. Threads inside same thread block cooperates with other threads either by local synchronization or shared memory while threads which are not in same block can not work together through shared memory. To differentiate threads, CUDA assigns unique ids to threads that is a combination of two unique coordinates:

- blockIdx (block index within a grid)
- threadIdx (thread index within a block)

Values above are pre-initialized and can be accessed in kernel function code to allow the developer to assign a different chunk of data to different threads.

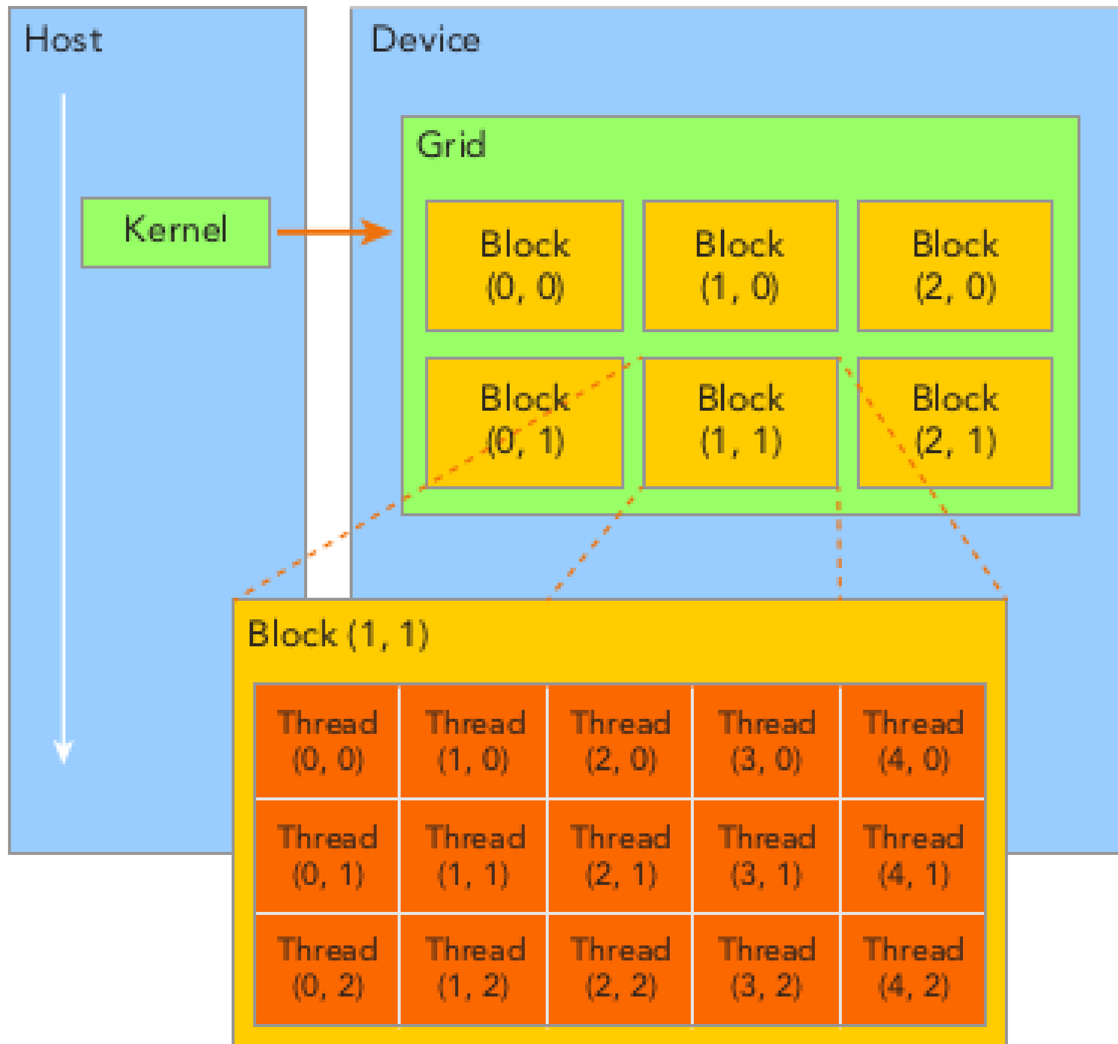


FIGURE 3.5: Thread and Block Overview
Courtesy: Professional CUDA C Programming

CUDA Kernel

A CUDA kernel call is similar to C function but runs with an extra configuration information given in as follows :

```
cuda_kernel <<<grid, block>>>(arguments);
```

This way developer can organize thread hierarchies by providing grid dimensions and the number of blocks inside them. For example, having 32 elements for calculation, a GPU can use different configurations for the kernel represented in Figure 3.6

```
cuda_kernel<<<4, 8>>>(arguments);
```

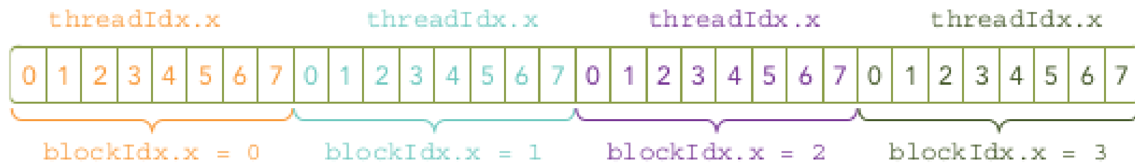


FIGURE 3.6: Thread Alignment for a Typical Kernel
Courtesy: Professional CUDA C Programming

Limitations of CUDA kernels

The hardware and API do not allow to infinitely broaden the computation and resources. It has some limitations due to the architecture and design considerations that determine how program should behave. Kernels are subject to some limitations : [29]:

- Access to device memory only
- Return type must be void
- Can not take variable number of arguments
- No static variables
- No function pointers

3.3. CUDA Execution Model

When efficient program writing is the topic of discussion, it is important to understand the structure of the computation platform. This understanding allows programmer to gain insight and manage the program flow according to the execution of the program on the hardware side. The base computation units for the GPU are Streaming Processors(SM). In layman terms, Streaming Processors resemble CPU cores where all of the resources are partitioned among them and the work is assigned to them. Each SM utilizes various components for the

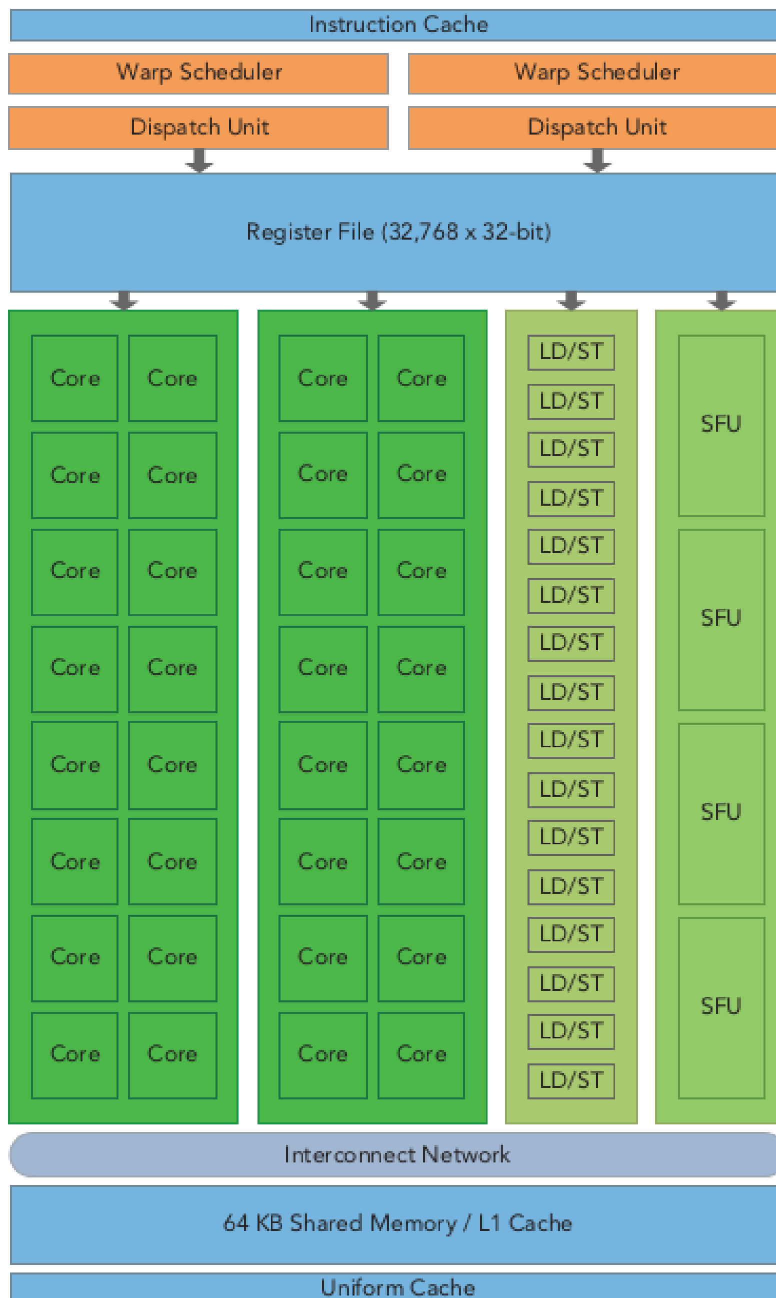


FIGURE 3.7: Overview of SM
Courtesy: Professional CUDA C Programming

parallel computation. Generally, GPU contains many SM arrays and it uses them to form hardware-level parallelism. Figure 3.7 shows the inner parts of the SM structure.

Each SM is built on multiple threads. That is, each SM can execute dozens of threads at the same time. SM can contains multiple *block*s, where blocks are the enveloping units for

the threads.Because there exists a lot of SMs on a single GPU, GPUs can create and execute thousands of threads asynchronously.

Kernels assigned to a specific SM can not change the SM it is assigned to at run time. When kernel function is invoked, threads in blocks are distributed among SM. If one block of threads is assigned to an SM, they can only run on that SM and has to stay there until all the execution finishes. SM and thread organization is given in Figure 3.8.

There is also the memory viewpoint of the SM. Shared memory and registers are two of the most important types of SM memory. Shared memory is distributed among blocks of threads. The threads inside the block can access the same shared memory. This allows threads to work jointly. Registers are distributed among threads and they are private to the assigned thread. The available size of shared memory and registers are special to SM and architecture dependent.

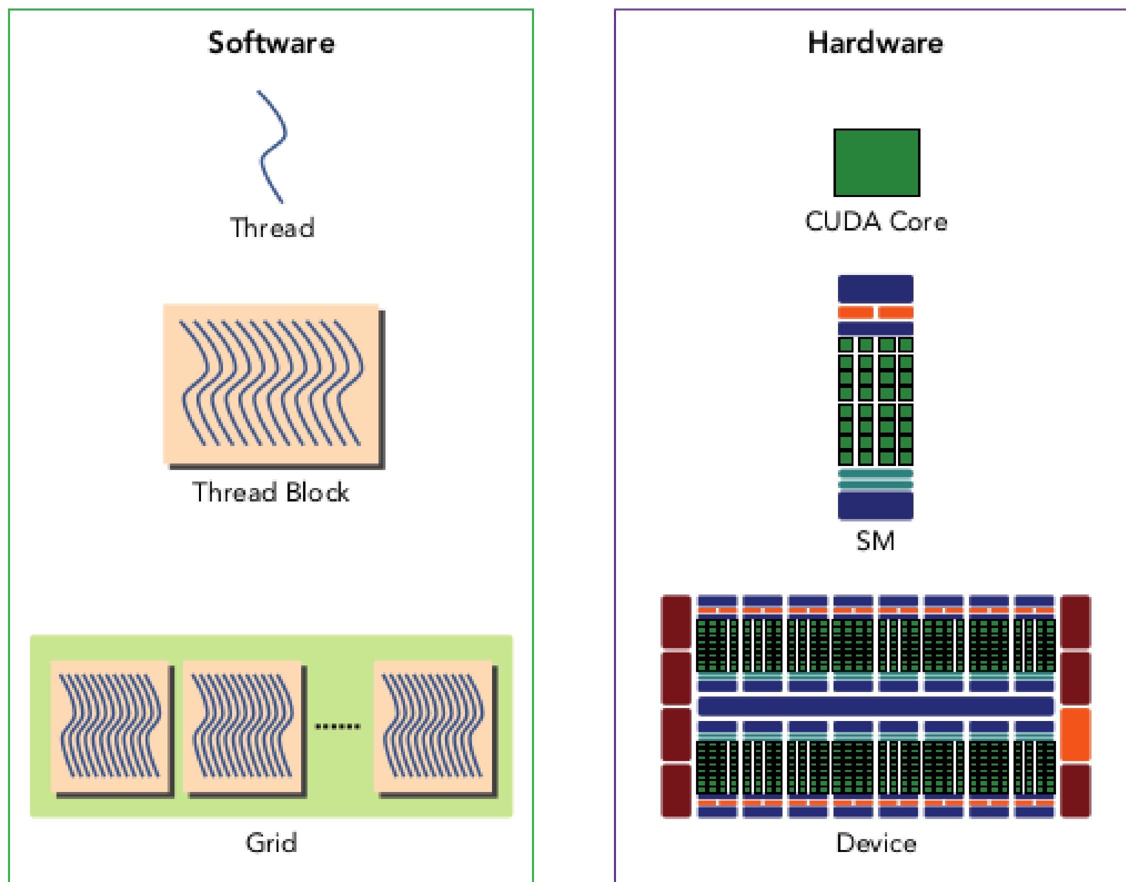


FIGURE 3.8: SM and Thread Organization
Courtesy: Professional CUDA C Programming

GPU parallelism is an example of Single Instruction Multiple Thread (SIMT) architecture. SM executes threads as groups of 32 threads. These threads executed at the same time by single SM are called *warps*. All threads in a warp executes same instruction. This is same as Single Instruction Multiple Data (SIMD) architecture commonly used in CPUs. The only difference is multiple data handled by different threads; that is, multiple data can be exposed to different instructions as different threads process them.

Warps

Each SM in CUDA contains many warps in it. Warps are essentially groups of threads, specifically 32, that execute the same instruction at the same clock cycle. Distribution of threads to warps is an important consideration for the performance and the efficiency of the program.

Threads are thought to be parallel execution units in every block in the logical point of view. They can be one, two or three-dimensional arrays. However, hardware handles the threads as if they are only one-dimensional vectors. Therefore, dividing the threads into warps is straightforward for SM. Figure 3.9 shows how threads are visualized from different views.

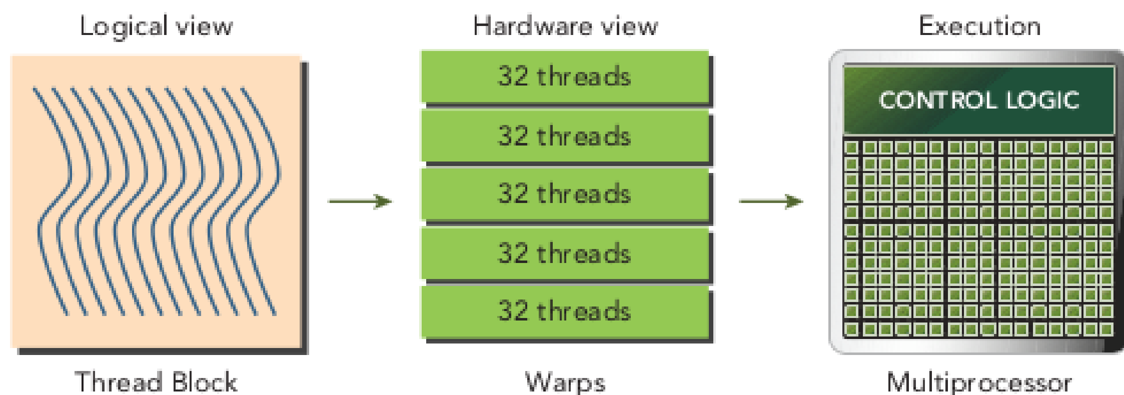


FIGURE 3.9: Thread Visualizations
Courtesy: Professional CUDA C Programming

Warp Divergence

Control flows are a vital part of every program. CPUs and GPUs also provide control flow constructors such as ”if-else” code blocks. However, GPU and CPU are different. CPU has

complex hardware to perform branch prediction. If the prediction is not correct, CPU may stall for several cycles and instruction pipeline is rearranged. However, there is such thing like branch prediction in GPUs thus all threads in a warp executes same instruction on the same cycle. This can be a problem when there is a branching statement in the code.

```
if (condition) {
    . . .
} else {
    . . .
}
```

When some of the threads in a warp need to execute ”if path” while the remaining need to execute ”else path” this leads to the warp divergence. When warp divergence occurs, warp executes each branch path, but it also disables threads that are not executing the path. That is why it can cause significantly degraded performance. Thread states for the execution timeline are given in Figure 3.10. Thread states show the divergence of the threads among if and else blocks given in green and pink colors. Threads in the same warp will have to wait for each thread in different branches, i.e. the if part and the else part. In figure 3.10 green representations would wait for the pink representations.

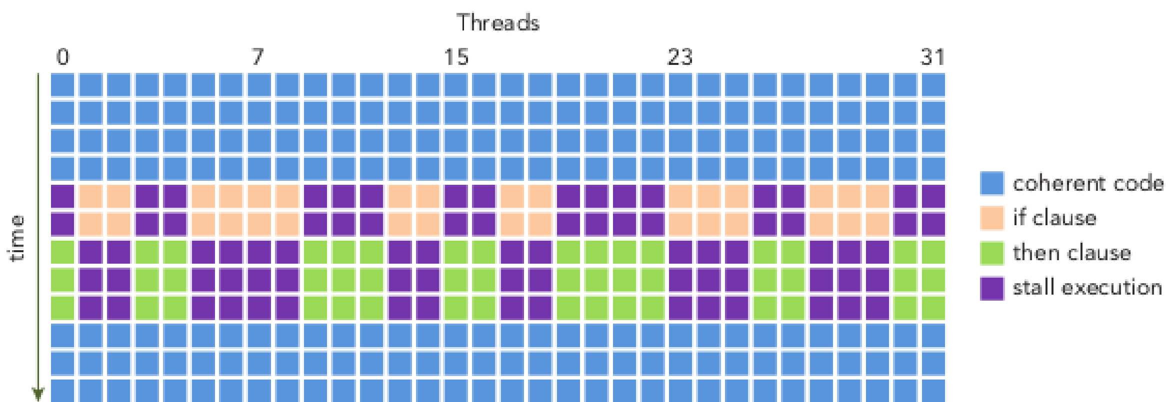


FIGURE 3.10: Thread States Illustrated
Courtesy: Professional CUDA C Programming

It is possible to avoid warp divergence by adjusting the data and threads in a way that they can be divided into warps with only a single branching occurs. This approach consists of sorting the data depending on the operations that will be done at the GPU. An example can be given as moving data which will be used in branch 1 to first X blocks whereas moving the remaining data to the last X blocks. Discretely dividing data can prevent warp divergence in this way.

In the following code the $cudaKernel1$ tries to assign the first branch to even-numbered threads and the remaining to the odd-numbered threads. This is very inefficient and nonsense way of using branches. Instead, the code $cudaKernel2$ shows a more natural and concise way of using threads. It adjusts the threads so that group of $warpSize$ threads executes same branch. With this configuration a better performance can be achieved.

```
__global__ void cudaKernel1(float *c) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    float a=0.0f;
    float b=0.0f;

    if (threadId % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }

    c[threadId] = a + b;
}
```

```
__global__ void cudaKernel2(float *c) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    float a=0.0f;
    float b=0.0f;

    if ((threadId / warpSize) % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }

    c[tid] = a + b;
}
```

Each SM contains 32-bit registers. These registers are partitioned among threads. Also a fixed amount of shared memory is partitioned among blocks. Number of blocks and active warps on a SM depends on availability of these resources.

Figure 3.11 shows when more thread consumes more registers, fewer warps can be placed on an SM. If the number of registers a kernel consumes is reduced, more warps will be processed simultaneously. Figure 3.12 illustrates that when a thread block consumes more shared memory, fewer thread blocks are processed simultaneously by an SM. If the amount of shared memory used by each thread block is reduced, more thread blocks can be processed simultaneously.

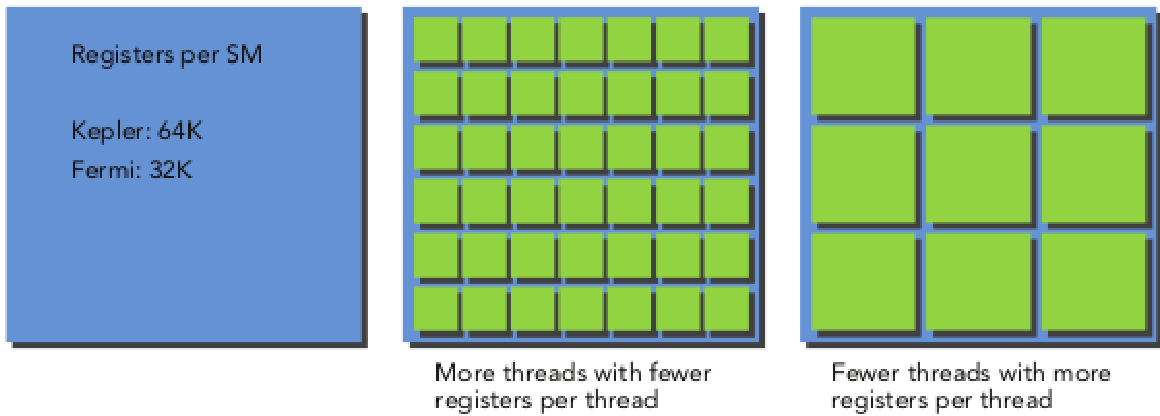


FIGURE 3.11: Register Overview of GPU
Courtesy: Professional CUDA C Programming

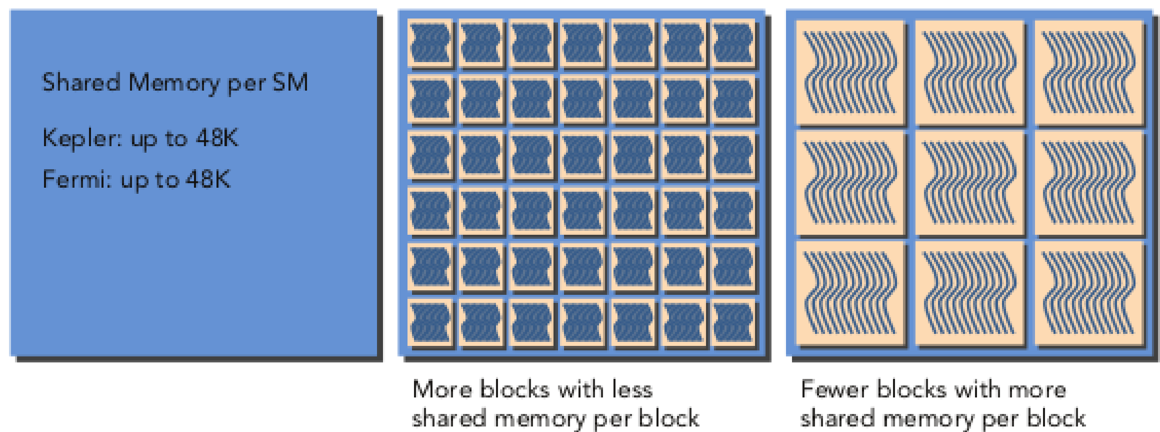


FIGURE 3.12: Shared Memory Overview of GPU
Courtesy: Professional CUDA C Programming

Occupancy

Instructions are executed sequentially within each CUDA core. When one warp stalls, the SM switches to executing other eligible warps. The ideal condition is for program to have optimal number of warps so that all device cores are occupied. Occupancy is calculated by dividing number of active warps to the maximum warp number, per SM.

```
occupancy = active warps / maximum warps
```

Loop Unrolling

Branches are big performance drawbacks for the threads as there is no branch predictor hardware on GPUs because of the way they are optimized. Loops and branches consume a lot of computational power. Therefore, any control flow instruction such as ”if, while, for, switch” results in degraded performance.

These performance issues can be compensated by compiler and programmer. Compiler sometimes handles the branches by optimizing if-else blocks or unrolling loops. This helps threads not to process the same loop condition check over and over again so that only the code inside the loop would be executed. This efficient calculation can be done by utilizing the branch prediction. Branch prediction is done in compile-time if the condition depends on the thread number and determinable at compile time. It is also possible to unroll loops manually by ”unroll” directive. When ”pragma unroll” macro is added just before the loop, it instructs the compiler to unroll the loop. This directive allows the user to partially or completely unroll the loops. An example code is given for the complete unroll in following code:

```
for (int i = 0; i < 5; i++) {
    a[i] = b[i] + c[i];
}
```

When pragma unroll macro is applied the code above, the compiled code becomes equivalent to following code.

```
a[i] = b[i] + c[i];
a[i] = b[i] + c[i];
a[i] = b[i] + c[i];
a[i] = b[i] + c[i];
a[i] = b[i] + c[i];
```

When iteration number in loop increases loop unrolling reduce performance drawbacks proportionally.

3.4. CUDA Memory Model

An important factor in application performance is access patterns to memory. Providing large data at high speed is not always possible. GPU contains different types of programmable

memory. Memory trade-off in GPU resembles the memory trade-off in CPU case. The larger the memory, it is slower to access to it; the smaller the memory, it is fast to access to it. The practical limitations made it impossible to produce a fast and big memory at the same time. Therefore, memory is divided into a hierarchical structure where the memory is demonstrated in Figure 3.13 and different memory types can be listed as :

- Registers
- Constant memory
- Texture memory
- Shared memory
- Global memory

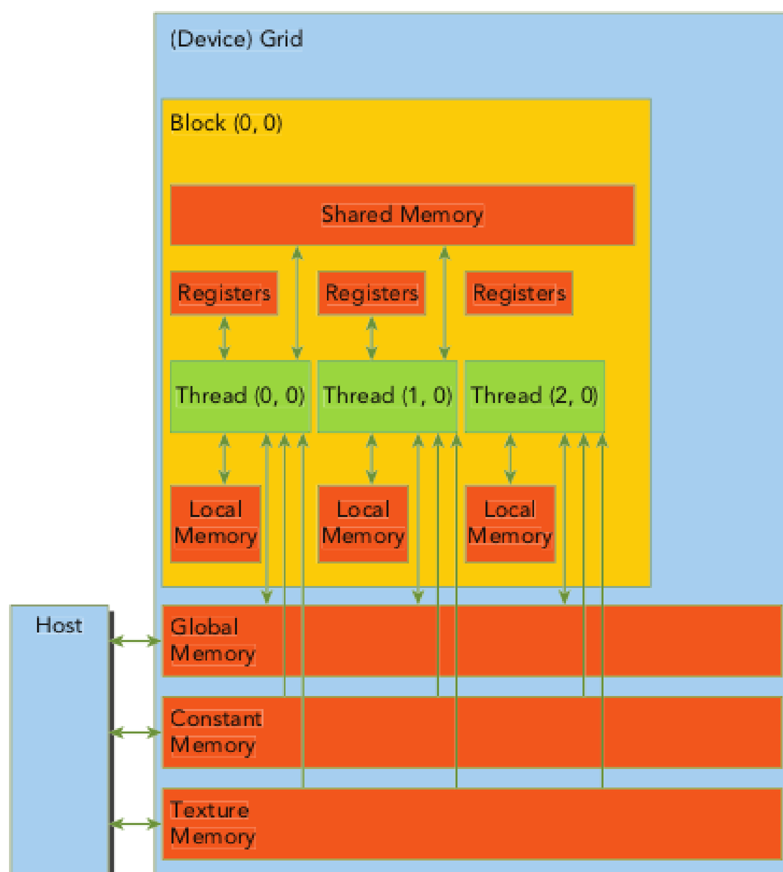


FIGURE 3.13: Memory Model Overview
Courtesy: Professional CUDA C Programming

The types of memory are explained in detail below. Each of them has different scopes and lifetime.

Registers

Registers offers the fastest memory access and are only owned by a single thread. Registers lifetime is determined by the kernel. Kernel exit implies termination of the variable in registers.

Local variables and local arrays are stored in registers with a single restriction. If the size is known in compile time, and the variable size can fit into the registers, then it is stored in registers. This is because register sizes are scarce resources that are unable to store big data due to the trade-off explained in the previous sections.[29]

Constant Memory

Constant memory is in device memory cached per SM. It is declared statically and accessible to all kernels without write permission as it is in the global scope. [29]

Texture Memory

Texture memory is a type of global memory which resides in the device memory and its cache process is read-only and it is cached per-SM. It is configured for 2D spatial locality

Shared Memory

Shared memory is among the most crucial parts of memory management. Since it is on-chip, low latency and high bandwidth are the typical properties of the shared memory. It is even possible to fetch the data within one or two clock cycles depending on the architecture. Compared to global memory, it is much faster, even 20 - 30 times in latency and 10 times in bandwidth. Shared memory can be thought as a pool between threads in the same block. In other words, it can be thought as a register for a thread block.

Since the operation of threads is asynchronous, it is important to synchronize the threads when multiple write operations will be executed. The synchronization is done by *syncthreads* function. Over utilization of the shared memory causes different warps to wait idly for others to synchronize for a long time. As shared memory partitioned among thread blocks of SM, it is also a limit for the number of active warps. Thus it limits the parallelization of the program as its size is not infinite and divided into threads, actually 64 KB per SM. As more shared memory is used by each kernel, smaller sizes of shared memory become available to each thread. Figure 3.14 shows the hardware overview of the memory structure.

Shared memory supports both dynamic and static allocation. Kernels can have dynamic allocation as well as static allocation as long as some conditions are satisfied. Firstly, the dynamic memory size should be determined before each kernel call so that the GPU can allocate the required size. This is done by passing a third argument to the kernel call inside the device code. Secondly, multidimensional are not allowed thus dynamic array is only supported if the array dimension is one.

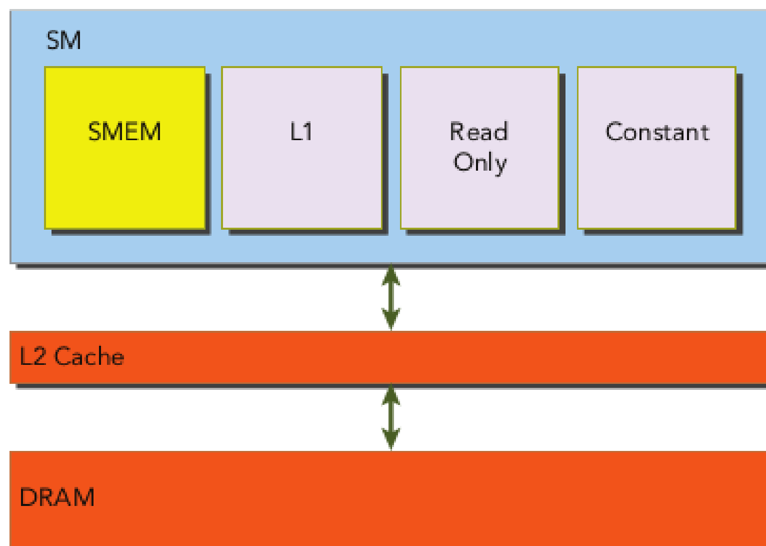


FIGURE 3.14: Shared Memory Structure
Courtesy: Professional CUDA C Programming

Shared memory is organized in banks. Banks are memory portions that are accessed concurrently, with a single attempt. Shared memory is divided into 32 equally sized portions. The number explains the single warp, explained in previous parts, has 32 threads. The fact that there are limited number of shared memory locations that can be accessed at the same

time makes it significant to adjust memory access so that there are no bank conflicts. Bank conflict is a type of conflict that occurs when multiple addresses request from the same bank. If this is the case, the request is not processed at that transaction and it is repeated. As a result, another transaction is required and performance degrades considerably.

When shared memory access is triggered, there are three possibilities :

- Parallel access
- Serial access
- Broadcast access

Parallel access is the case when memory transaction is done in parallel which means threads in a warp requests to separate banks, and the transaction is done in a single attempt. This is also called conflict-free transfer. This pattern implies some, if not all, transactions are processed in a single memory transaction.

Serial access is the case when different threads access the same bank but with different locations. In this way, only a single transaction can be done, and others should wait for the current transaction and the request is repeated.

Broadcast access is the case when different threads request the same bank and the same address. In this configuration, only a single attempt is made and this single operation is shared by all of the requests with no performance drawback. The only difference compared to parallel access becomes the low usage of the memory bandwidth as access to a single address requires a single transaction, but it results in poor bandwidth utilization.

Figure 3.15 shows the parallel case where different threads request different banks where the transaction is done in a single attempt. On the other hand, Figure 3.16 shows an irregular access pattern. In this case, there are two possibilities, either broadcast or serial access. If requested data are the same for all requests, only a single attempt is enough as broadcast access happens. However, if requested data are separate inside each bank, GPU needs to repeat the request and bank conflict occurs. In Figure3.17 random access to shared memory is shown. Despite there is no bank conflict, this type of access is still not efficient at least for spatial locality consideration.

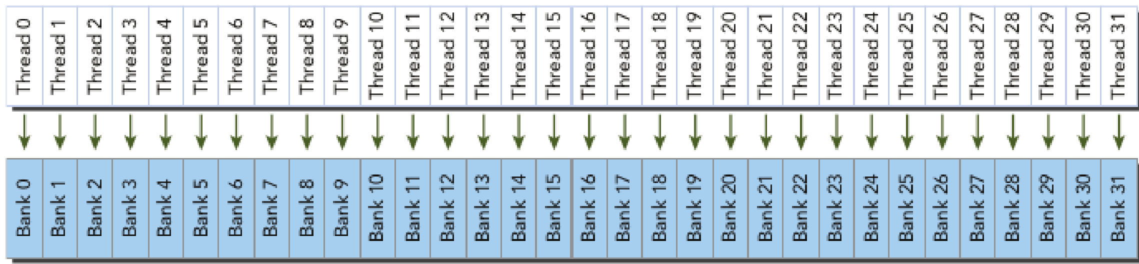


FIGURE 3.15: Parallel Access to Shared Memory

Courtesy: Professional CUDA C Programming

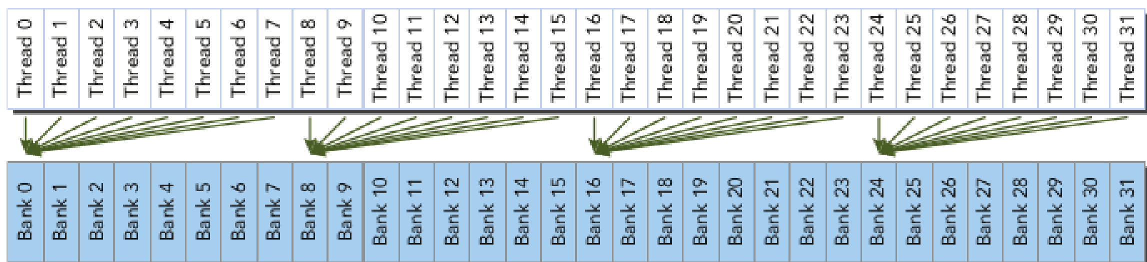


FIGURE 3.16: Serial or Broadcast Access to Shared Memory

Courtesy: Professional CUDA C Programming

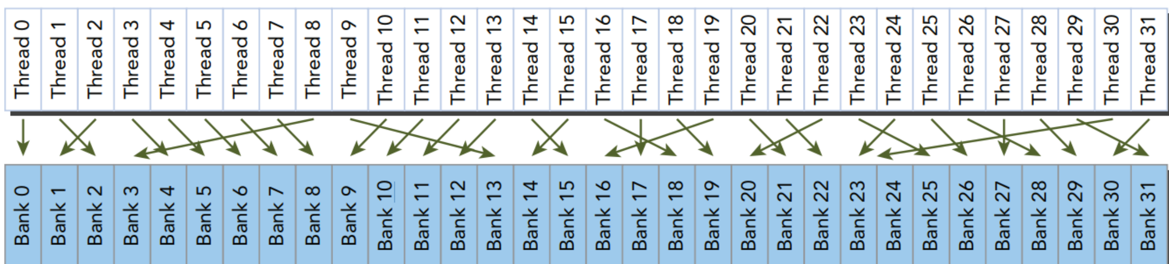


FIGURE 3.17: Random Access to Shared Memory

Courtesy: Professional CUDA C Programming

One solution to the serial access is memory padding. Padding is the process of shifting elements to separate data access patterns to different banks. It reduces total memory available because of the padded spaces, but it increases performance if used properly. Figure 3.18(left), shows a sample case where all threads in a warp try to access the same bank because of the same index. Since a bank can serve only one transaction per attempt, the other attempt is needed to be requested. This performance degradation can be solved with padding as shown in Figure 3.18(right) where shifting data prevented threads to access the same bank.

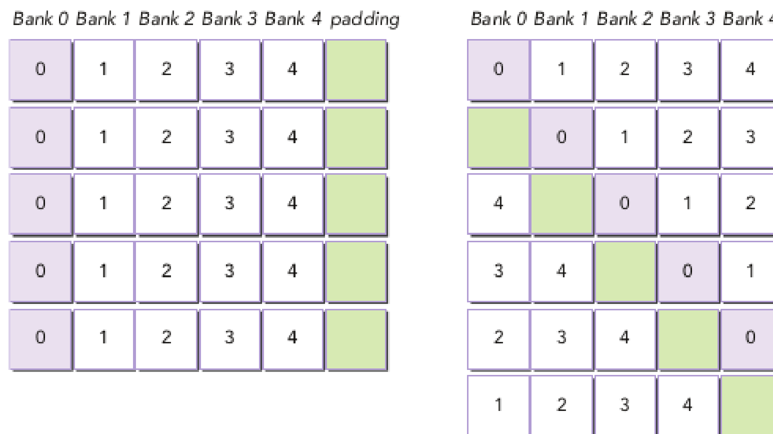


FIGURE 3.18: Padded Memory Access
Courtesy: Professional CUDA C Programming

Global Memory

Global memory is the largest memory kind in GPU. Also it is the most used memory type. The word global refers to its scope and lifetime. It is visible to every SM, and it can be declared as static or dynamic. To allocate space in global memory *cudaMalloc()* function is used.Because it is accessible by all threads, access to it must be synchronized to avoid different threads to modify the same portion of it while other threads are reading.

Data communication to global memory is done via 32-bytes or 128-bytes aligned transactions. It means that the first place to access is multiple of 32 bytes,128 bytes. Access to global memory is a critical factor for better optimization and is realized by satisfying the two following conditions :

1. Memory address distribution across the threads.
2. Alignment of memory addresses per transaction.

The number of requests required to deliver needed data is important because when unused bytes increase it reduces throughput efficiency.

Correct memory access significantly accelerates programs. Fetching and sending data from and to memory has a considerable impact on the performance of the program. From the view of warps, it is important to reach memory by considering two different ways: aligned

and coalesced memory accesses. Aligned memory access is about the start address of the requested memory partition. Only exact multiples of 32-byte and 128-byte addresses are used as the starting point of the read operation if it is an aligned memory access. Since memory can only provide these margins to the cache in a single attempt, it is much faster to read the data with a starting point that is multiple of 32 or 128 where the size depends on the cache. The data distribution is also important to fetch data in less number of attempts. For example in a 32-byte sized cache, if data between address 1 and 32(included) is requested, the memory should try to cache the data two times: one for the data 1-31 and one for the 32. Even though the total size is 32 bytes, speed is halved because of the doubled number of access.

Coalesced memory access is accessing consecutive bits on the memory. Since cache loads a specific size of data from the memory, loading consecutive bytes makes it faster to load the same data. While loading a small portion of data from memory determining which data to load applications follow the principle of locality.

Locality is the concept of accessing values which are in the same set rather than values that are randomly chosen. There are two types of locality:

- Temporal locality (locality in time)
- Spatial locality (locality in space)

Spatial locality is the assumption that the neighborhood of an address accessed recently is more likely to be accessed for the near future in the memory. Temporal locality is the principle that the variable that has been just accessed is more likely to be accessed again. Based on these principles, it is up to the programmer to design the algorithm considering the background of the hardware. Apart from locality programmer need to know different types of memory models which CUDA offers. Locality becomes meaningful when appropriate memory is used properly.

3.5. Object Detection

In the field of computer vision, object detection is the problem of finding appearance of objects in image or videos. Object to be detected is given as input to algorithm, then algorithm

tries to represent input object in a way that computer can differentiate it from other objects or background in images. Any information used to distinguish image regions is called feature. Any points used to extract feature from image is called key-points. Defining characteristics of all features together, image descriptors are created.

Object detection pipeline is depicted in Figure 3.19 Features extracted from key points are turned into meaningful descriptors. Finally, among the descriptor locations, that one which is the closest to target descriptor is said to be the location of the target in the image.

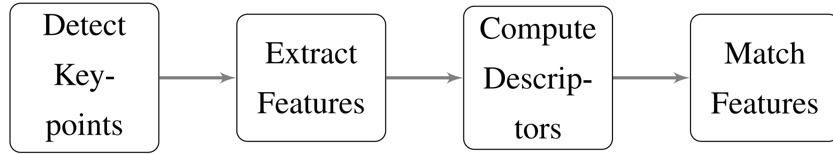


FIGURE 3.19: Object Detection Steps

3.6. Region Covariance Descriptor

Region covariance introduced in [1] is enough to fuse simple features to describe patches independent of rotation, pose. In RCD, the covariance of simple image statistics computed for a region of interest as a descriptor. Here covariance is used as a feature instead of the joint distribution of the image statistics, thus the dimension of features becomes much smaller.

Region R is represented by covariance

$$C_R = \frac{1}{N-1}\sum_{k=1}^{n}(Z_k - \mu)(Z_k - \mu)^T \tag{1}$$

where T denotes transpose, n is feature vector dimension (5 in our experiments), μ is the mean of features inside the patch, and Z_k represents n dimensional feature vector inside region of interest in *width*height* dimensional feature image F. F is constructed by using red, green, blue channels and first derivatives of the image and defined as Equation 2 in our experiments.

$$F(x,y) = \left[I_R(x,x) \;\; I_G(x,y) \;\; I_B(x,y) \;\; \frac{\delta I(x,y)}{\delta x} \;\; \frac{\delta I(x,y)}{\delta y} \right] \tag{2}$$

A regions covariance on the image has no information about the number of points inside the region. This makes covariance descriptor scale-invariant to some extent. Here it is important to note that the aforementioned rotation and pose invariance of this descriptor is up to the selected features. Using derivatives which are dependent on intensity makes this descriptor dependent to illumination. But on the other hand, using covariance as region descriptor allow us to form a complex descriptor using simple features.

Some optimization methods used to reduce computational cost makes it affordable to use for processing static images but they were not enough to use this descriptor in real time tasks. What is more, problems like object detection requires descriptors to be extracted from thousands of small windows on images.In [1] Equation 1 is rewritten as for two matrices:

$$C_R\left(i,j\right) = \frac{1}{n-1}\left[\sum_{k=1}^{n} z_k\left(i\right) z_k\left(j\right) - \frac{1}{n}\sum_{k=1}^{n} z_k\left(i\right)\sum_{k=1}^{n} z_k\left(j\right)\right] \tag{3}$$

Equation 3 gives a hint that once we calculate the sum of regions using each of z_i and each $z_i z_j$ pair, we do not need the calculate the sums again and again. Also, this will be re-work as most windows in sliding windows on the image will be overlapping except few pixels. To overcome this problem Integral Images is used. The following section explains how integral images makes it faster to calculate region co-variances in constant time.

Integral Images

Integral Images introduced in [30] is used to calculate region sum in a faster way. In integral image representation, a pixels value is the sum of all values inside the rectangle between top-left of image and pixel itself. Mathematically it is defined as Equation 4. Integral image representation of features allows any regional sum to be computed in $O(1)$ time.

$$IntegralImg(x', y') = \sum_{x<x', y<y'} I(x, y) \tag{4}$$

Once we calculate integral for each feature image, $z_i s$ and $z_i s z_j s$ in Equation 3, $\sum z_k\left(i\right) z_k\left(j\right)$ can be computed by $A + D - B - C$ as shown in Figure 3.20, for each feature image with constant time access.

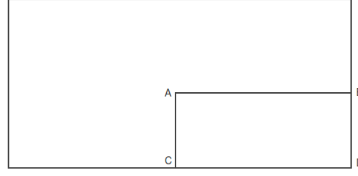


FIGURE 3.20: Summation Using Integral Images

Comparing Covariance Matrices

Comparing descriptors in image processing applications is as important as selecting good descriptor. Unlike many SIFT-like descriptors, covariance matrices do not lie on Euclidean space so it is not possible to use simple vector similarity measures in comparison of covariance matrices. Forstner Distance measure explained in [31] was state of the art at the time of work [1] and was used to compare covariance matrices. Metric is defined in Equation 5:

$$d\left(A,B\right)=\sqrt{\sum_{i=1}^{n}ln^{2}\lambda_{i}\left(A,B\right)} \quad (5)$$

In distance metric $\lambda_i\left(A,B\right)$ refers to Generalized Eigenvalue of two covariance matrices A and B, computed as

$$\lambda_i A x_i - B x_i = 0 \quad i = 1....n \quad (6)$$

and $x_i \neq 0$ are the generalized eigenvectors. With this distance measure, $d\left(A,B\right)$ is always positive except the case $A = B$ where the distance is zero.

Forstner Distance to compare covariance matrices requires even more processing as it contains a calculation of generalized eigenvalues of nxn dimensional matrices which are not straightforward to compute when $n > 3$. As an alternative to Forstner Distance, Jensen-Bregman LogDet Divergence is introduced in [28].

Jensen-Bregman LogDet Divergence distance has lots of advantages over Forstner Distance. First of all, it is lighter to compute than [31] and performs as accurate as it does. Beyond these, it is easy to implement Jensen-Bregman LogDet Divergence both on CPU and GPU.

Equation 7 formulates the distance metric.

$$d(A,B) = \log\left|\frac{A+B}{2}\right| - \frac{1}{2}\log|AB| \tag{7}$$

where $|A|$ denotes determinant of matrix A. One more advantage of using this metric is, the formula for comparison can be divided into parts which can be calculated independently from each other and reused. In the following sections, minor tricks to speed up this calculation are explained.

4. REGION COVARIANCE DESCRIPTOR CUDA IMPLEMENTATION

By its nature computer vision algorithms are a good fit for parallel programming as it requires matrix-based operations and applying the same instruction to all pixels of the image. Beyond its fast computation on the problems which require heavy computations on multiple inputs, computations on the GPU are done asynchronously.This allow CPU to run other tasks while algorithms are executed on GPU.

Considering the benefits of the GPU computation we implemented a CUDA version of an existing object detection method, which performs well in terms of accuracy but lacks enough computation speed to be used for real-time tasks. With this implementation, we aimed to achieve real-time computation under $30fps$ per frame on GPU while the CPU version takes more than $300ms$ per frame, which is lagged considerably behind the real-time performance. In our implementation, we tried to reduce data transfer between CPU and GPU. To this end, we are transferring input image data to GPU and applying all the steps of object detection in Figure 3.19 in GPU side. Only a few numbers of distances and their indexes are returned as a match candidate. In CPU host, minimum of these distances are found and its index is an index of the matching window.

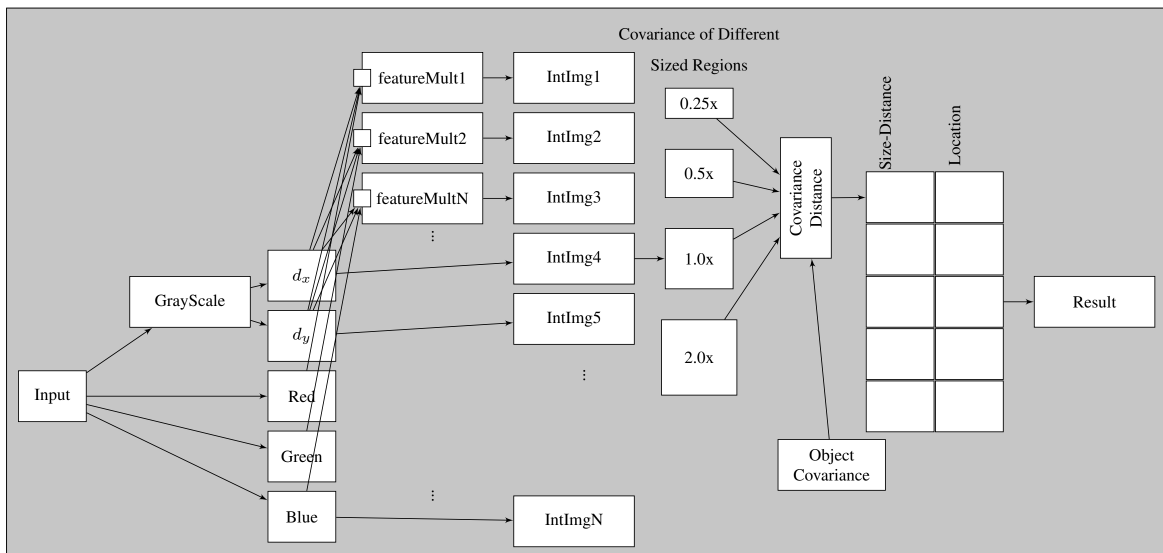


FIGURE 4.1: RCD CUDA Implementation

After copying three channel color image to the device we are generating a gray-scale image and extracting red, green and blue channel as a feature. Using gray image, first order image

derivatives are calculated using the filters shown in Equation 9. Then integral images are computed in a more efficient and faster way compared to existing work [32]. Lastly, covariances matrix of each region is computed. We used a sliding window approach where different scaled object windows are sliding by small step-sizes in two dimensions of the image. Equation 1 is applied to all small windows to generate the covariance matrix, a robust descriptor for the region. Getting distances of each window to target covariance calculated, it turns to a linear searching problem. The whole process is shown in Figure 4.1 The following sections highlight each kernel we have implemented.

Generating Feature Images on GPU

Feature image generation is a straightforward task to parallelize. We make sure the entire calculation is parallelized over threads. Gray image calculation is a simple multiplication. Each thread applies Equation 8 to all pixels of input image.

$$I' = I_r * 0.2627 + I_g * 0.6780 + I_b * 0.0593 \tag{8}$$

Getting gray-scale image calculated, convolving matrices I_x and I_y in Equation 9 with each pixel, vertical and horizontal edge feature images are calculated.

$$I_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} I_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{9}$$

Integral Image Calculation

NPP library[33] provides integral image computation for a single image. Equation 1 requires $n + (\frac{n^2+n}{2})$ integral image computation, which is not possible to be computed concurrently. Thus NPP is not efficient enough to be used for our problem. Instead we adopted the efficient integral image implementation proposed in [32]. In that study, authors implement the integral image calculation via two scans (parallel sum introduced by [34]) and transpose computation sequence. First, the scan kernel is used to calculate row-wise sums. This matrix is then transposed and the same scan operation is applied. This effectively means row-wise

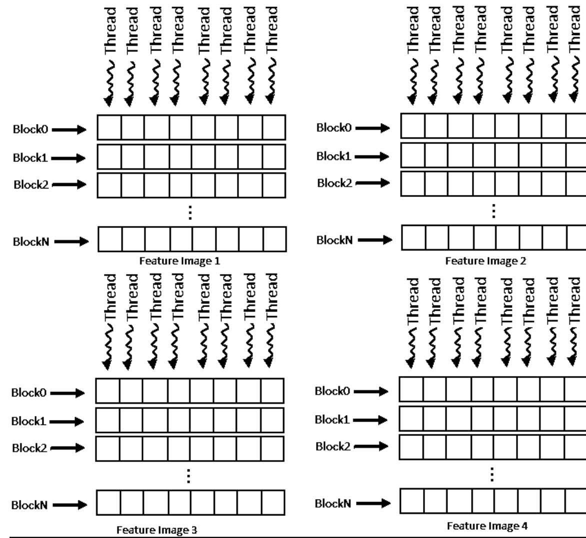


FIGURE 4.2: Dedicated block grids for each input image

and column-wise summations back to back, which is essentially the integral image. To get the integral of input feature image, two scan and two transpose operations are needed. Instead, we replaced $transpose - scan - transpose$ with our efficient implementation for multiple images. Their method requires additional memory for each input image to store intermediate results of transpose operation. Extra memory for each image, 20 images in our experiments, is not efficient at all. Also, we should note that this number increases with the number of features, n. Beyond this, their implementation runs parallel scan algorithm on both dimensions of input but their implementation assumes input image dimensions to be a power of two, thus input images of arbitrary size require extra work and overhead for GPU cores with their implementation.

In our implementation, we let block grids run on each of input image concurrently where each block grid has enough threads to apply $scan$ and $columnsum$ kernels to their input. Figure 4.2 depicts this process. In our row-wise summation, each thread block process each row of input and applies the parallel scan algorithm. Parallel scan is depicted in Figure 4.3 and column-wise summation is depicted in Figure 4.4. Each thread block processes each column of the input image and each grid of blocks runs on different input image concurrently. This type of data access is a good example of coalesced global memory access as threads accessing neighbor pixels to make use of spatial locality.

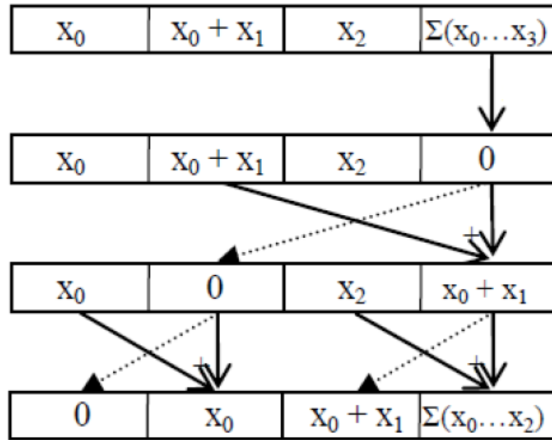


FIGURE 4.3: Row-wise summation of pixels

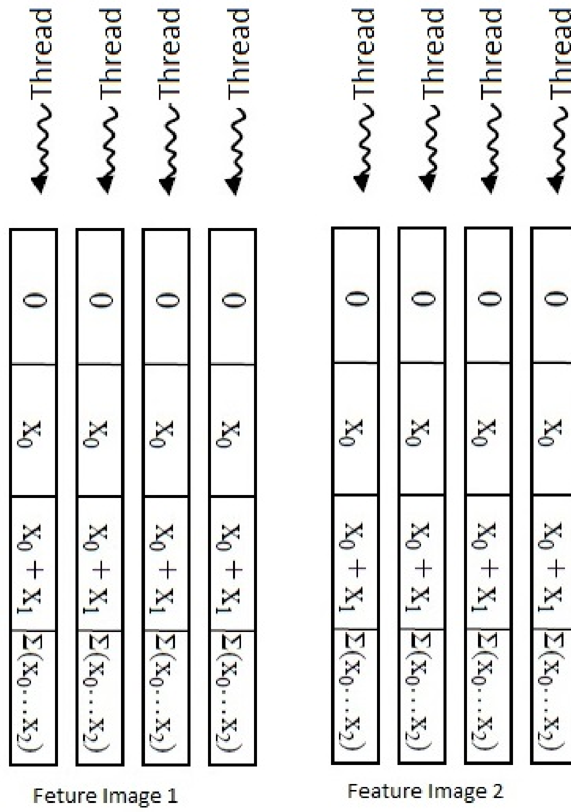


FIGURE 4.4: Column-wise summation of pixels.

4.1. Covariance Matrix and Distance Calculation

Having feature image integrals, which are separate gray-sclae image for each feature, computed $\sum z_k(i)\, z_k(j)$ in Equation 3 can be computed by $A + D - B - C$ in Figure 3.20 for each feature image with constant time access.

Covariance matrices are symmetric matrices. So calculating lower triangular values of the matrix then copying values to upper triangular indexes whole matrix is created in a memory efficient way. Each patch window of the image is processed by a block of threads where each thread calculate an element and its symmetric element in the upper triangle in n x n covariance matrix. In other words, each block does computation for one window on the image while each thread inside the block computes covariance of two features for that particular window.

As accessing to *shared memory* is quite faster than accessing to *global memory* and covariance matrix computation in each block requires a small amount of memory, covariance matrices are computed to shared memory. This covariance data in shared memory is lost as soon as a block using shared memory exit execution. In order to reduce global memory access and kernel call costs instead of storing covariance matrices in global memory, distance calculation in Equation 3 is applied to covariance matrices in shared memory. This way, only covariance distance and patch index of the patch are written to global memory. Thus data transferred with global memory is decreased by n x $n - 1$ for each of the thousands of covariance matrix. The target object in the input image may appear in different scales. In order to improve robustness against scale change, covariance matrices are calculated for different patch sizes.

To compute distance between object window covariance matrix and covariance matrix of window of image, Equation 7 is modified. $\frac{1}{2}\log|AB|$ requires targets covariance matrix to be multiplied by each of the window covariance matrices. Using the property of determinant $|AB| = |A||B|$ and property of $\log(AB) = \log(A) + \log(B)$ Equation 7 can be rewritten as:

$$d(A,B) = \log\left|\frac{A+B}{2}\right| - \frac{1}{2}\left(\log|A| + \log|B|\right) \tag{10}$$

Equation 10 has significant advantages over Equation 7.

- Matrix Multiplication is replaced by addition.
- $\frac{1}{2}\log|AB|$ can be calculated once and cached.

Decoupling descriptor from the equation, *log-determinant* of the target matrix can be calculated once and cached so this computation is not done again and again. This is especially useful for tasks which requires one-to-many matrix distance calculation problems. Video object detection problem is a good example for this as it requires comparison of object covariance matrix to covariance matrices of all object windows on all frames of the window. In other words, for a 1000 frames long video with 1000 window on each frame, $(1000*1000-1)$ distance comparison uses cached value.

Finding the Best Match on GPU

In the previous step, the distance between the object window and each window in an image at different scales are calculated. Finding the most similar window to object window is a basic minimum value search problem. Depending on the scales used (smaller scales requires a large number of small-sized target object windows) and input image dimensions, the number of windows can grow up to four thousand. Finding the minimum of such a large array is a straightforward task for CPU. But copying that much data from GPU to CPU is time-consuming. To reduce data transfer, we perform searching for the best match on the GPU side.

Searching for the minimum value efficiently is achieved by modifying the methods explained in [35] to include indexes of values. In our configuration, each thread block finds the minimum value of each 2048 element by performing a parallel reduction [35]. Minimum distance values and their global index from each block are then transferred to CPU to find the closest match. It is CPUs duty to find minimum value of all blocks.

5. RESULTS & ANALYSIS

5.1. Testbed Configuration

We aimed to achieve real-time performance with $30fps$ for a resolution of 2K (2048x1152) and 36 seconds long video (1136 frame) for 8 scales from 0.25 to 2. We ran our tests on our workstation computer with Intel(R) Xeon(R) Silver 4114 CPU 40x 2.20GHz CPU 128GB with NVIDIA Titan X GPU 12GB and Intel Core i5 7440HQ CPU 4x 2.80GHz 16GB and Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz GeForce GTX1050 4GB. To do a fair comparison, existing implementation is modified to work with the latest OpenCV version and is applied all aforementioned optimizations as well. On CPU side CPU Vectorisation is used to speed up computations. As OpenCV makes use of all possible optimizations on hardware our attempts to parallel or vectorized implementation does not help to achieve faster speed than OpenCV. Thus we ran our tests on CPU with vectorization enabled and vectorization disabled configurations.

As RCD is powerful to describe image regions, we decided to choose five features as input to the descriptor which was robust enough to track the target in a video. Thus all the remaining experiments are based on but not limited to 5x5 covariance matrices. It should be noted that other complex features can easily be integrated into our RCD implementation.

5.2. Test Results

In our experiments object detection with RCD CPU C++ implementation[3] is used as the baseline for the tests. It takes 350 ms ($3fps$) with non-vectorized code while it takes 180 ms ($5.5fps$) average with vectorized implementation. This shows that, even in the optimal case, with a very good CPU with 40 cores CPU parallelization will not be sufficient for real-time implementation of RCD while we achieved 37 fps with GPU implementation.

FIGURE 5.1: Timeline view of CUDA calls

TABLE 5.1: Profiling Results of our CUDA kernels

Time%	**Time**	**Avg**	**Min**	**Max**	**Name**
38.68%	6.6005ms	6.6005ms	6.6005ms	6.6005ms	$parallelScan\,()$
22.14%	3.7774ms	472.17us	896ns	3.6841ms	$[CUDAmemcpyHtoD]$
17.33%	2.9573ms	1.4786ms	212.83us	2.7445ms	$pointwiseMultiply\,()$
11.17%	1.9065ms	953.24us	199.49us	1.7070ms	$columnSum\,()$
3.43%	584.48us	292.24us	42.016us	542.47us	$generateFeatures\,()$
3.26%	556.45us	556.45us	556.45us	556.45us	$rowSum\,()$
2.52%	430.34us	53.792us	19.104us	152.42us	$covarianceDistance\,()$
1.42%	242.15us	121.07us	20.000us	222.15us	$calculateGreyImage\,()$
0.03%	4.8320us	2.4160us	2.0800us	2.7520us	$[CUDAmemcpyDtoH]$
0.03%	4.6400us	4.6400us	4.6400us	4.6400us	$findMinValAndIndex\,()$

Following the CUDA processing flow given in Figure 4.1, we implemented the cuRCD. Table 5.1 shows execution times of our kernel calls. *parallelScan* function which consumes the majority of the run-time computations as shown in timeline view of kernel calls in Figure 5.1 That kernel computes sum of 1152 rows which has 2048 elements for each of 20 images. This function makes use of the shared memory which has low latency but limited resource (96KB max). This limits the amount of data processed asynchronously because shared memory can only store limited data. As our target image is small, its row-based summation is not a good input to *parallelScan* function, thus *rowSum* method, simple accumulative sum of values, is used to calculate sum of rows of the object image. Comparing to the achieved run-time around 0.5ms for a 400x400 object image, parallel scan algorithm can achieve around 6ms on 2048x1152 images (20 images each and each one of them is approximately 15 times greater than 400x400 object image, generating 300 times more data) which is quite efficient.

pointwiseMultiply, columnSum, generateFeatures and *calculateGreyImage* functions are both called for object image and for 20 feature images of each video frame. The execution each these functions on 20 large input frames is efficient compared to execution on the single object image. We observe execution time on input frame image less than 15 times execution time on object image despite 300 times more data is processed (explained in the above paragraph).This is an indicator of the efficiency of processing multiple inputs at the same time. All remaining functions, including copying results back to CPU, are observed to run in a negligible amount of time compared to the aforementioned function calls.

In our experiments, we saw that our integral image calculation method performs better, significantly faster than [32], but also our implementation needs no extra memory. Figure 5.2

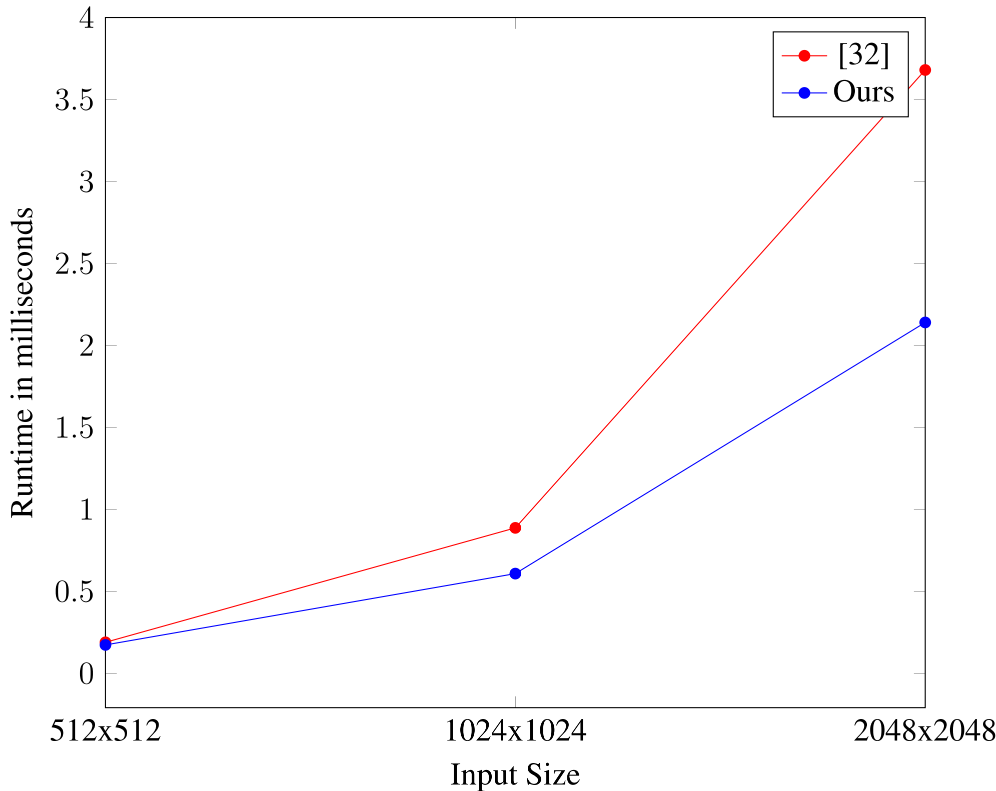


FIGURE 5.2: Comparison of Integral Image Computation on CUDA

shows average (of 10000 measure) computation times of single image integral with different input sizes on Nvidia GeForce GTX1050. Our integral image calculation method is faster than [32]. Here an important point is that our method doubles the number of images to be processed concurrently. This is a significant improvement for our problem as it requires computing integral of 20 images concurrently.

By computing the covariance matrix distances on GPU side we are both saving time and space. Instead of storing each covariance matrix we save only distance between region and target covariance matrices. As stated before, implementing traditional covariance matrix distance calculation on GPU is not a straightforward task and requires lots of efforts. We employed Jensen-Bregman Distance algorithm which is easier to implement on CUDA and faster than Förstner Distance as shown in Figure 5.3. We experimented on Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz with random positive definite symmetric dense matrices, imitating covariance matrices. In order to reduce variance we run 100000 experiments. The figure shows, the runtime of both methods in CPU with different covariance matrix sizes. It is obvious that even in $5x5$ covariance matrix we are using, run

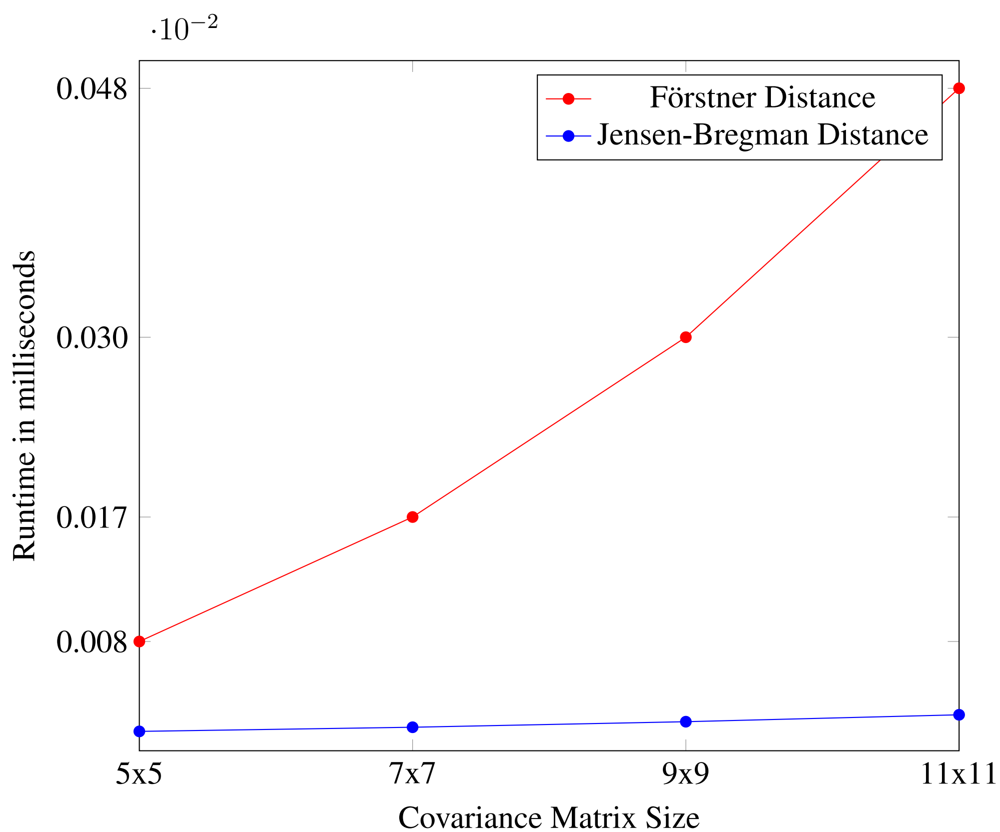


FIGURE 5.3: Covariance Matrix Distance Calculation Methods Runtime

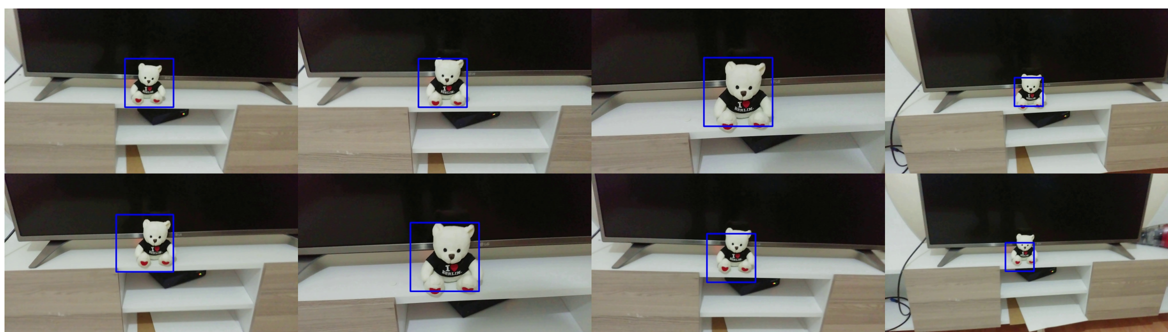


FIGURE 5.4: Detecting Target Object in Different Scales

time of Jensen-Bregman Distance is negligible compared to Förstner Distance. The exact mean values of measurements are $\mu = 0.00796$ $\sigma = 0.002153, \mu = 0.017348$ $\sigma = 0.0031835, \mu = 0.030931$ $\sigma = 0.0052793, \mu = 0.048878$ $\sigma = 0.0064542$ for Förstner Distance and $\mu = 0.0015812$ $\sigma = 0.00069955, \mu = 0.0018769$ $\sigma = 0.00059308, \mu = 0.0022343$ $\sigma = 0.00089925, \mu = 0.0027018$ $\sigma = 0.00086892$.

In the CUDA implementation with the same scale sizes, we had similar patch indexes detected as the target in the image. This proved that our GPU implementation does not corrupt any information in any step. Figure 5.4 shows detection on different scales.

Table 5.2 shows the numerical results of object tracking task on each configuration. Table shows average value of 1100 measurement. Standard deviation for distribution over measures are $29.49, 12.34, 1.27$ and 1.32 respectively. Our implementation that runs on GPU clearly achieves real-time performance in %99.4 of 10000 experiments $\left(\frac{\# \, runtime < 30ms}{experimentcount}\right)$. Despite the price of Titan X GPU is around 10 times more expensive than GTX 1050 GPU, average computation time is still closer to each other (slower but still real-time). This is also an indication that our implementation does not require expensive video cards to execute. On the other hand, vectorized implementation on a powerful 40 cores CPU system with enough RAM capacity can only achieve 15% of these speeds. It is also important to consider that achieved real-time performance runs asynchronously from CPU execution thus allows the host system to execute other tasks during computation on GPU.
Our implementation only leverages parallelism on concurrently processing different images and pixels.Modern GPUs provide *asynchronous copy and execution*, it is possible to concurrently copy input image, *next frame in the video*, asynchronously while kernels are executed on the device[36]. We investigated that copying three channels floating-point 2K image to GPU takes around 8ms and memory location occupied by input image is no longer required after generating features which is the very first step of RCD implementation. Thus, remaining steps after transferring data to devices can be run concurrently with data transfer of the next frame in the video.

TABLE 5.2: Average Computation Times

Implementation	**Computation Time**
Non-Vectorized RCD CPU 40x 2.20GHz	350 ms (3fps)
Vectorized RCD CPU 40x 2.20GHz	180 ms (5.5fps)
cuRCD on NVIDIA Titan X	27.168 ms (37fps)
cuRCD on GTX 1050	31.29 ms (32fps)

Making a fair comparison of our results to the [27], where they recently provided a CUDA implementation of the RCD algorithm as well, is not possible as their code is not available and also their first aim to filtering performance rather than speed.

6. CONCLUSION

In this study, we present CUDA implementation for region covariance descriptor which allow it to be used in real-time tasks. The proposed implementation, when applied to an object detection problem, can run in real-time with 2K videos. Our results showed that parallel implementation is achieving 6 times faster computation time comparing to the vectorized CPU implementation. No doubt that the distance metric proposed in [28] for comparing covariance matrix plays a crucial role. It works as accurately, it is less compute-intensive and can easily be parallelized and ported to CUDA. Applying light region covariance matrix distance computation to multiple small images frequently on cores with low computational power was vital to our implementation. Our other contribution is that we showed that integral image computation proposed in [32] can be improved by leveraging coalesced memory accesses and reducing the memory used by half. Also, we removed the transpose operation so that output can be handled without any transformation. Even it is stated that resulting transposed image can be read easily by replacing the row-column index on access for further processing, it is not efficient at all especially in GPU processing where memory access pattern is crucial. We reduced three kernel execution of their method for calculating column-wise summation with one kernel call thus saved from calling $20x2 = 40$ kernels for a single frame in the video. As shown in Figure 5.2, we achieved a faster integral computation while reducing memory usage by half.

Our proposed implementation can be counted as native base implementation and can be improved further either by optimizing each kernel execution separately like proposing better memory access optimizations or applying parallelism to steps of the algorithm in a different way. By choosing efficient metrics for covariance matrix computation on CUDA framework, asynchronous execution of steps to multiple inputs and proposing more optimal memory access patterns, our work presents a concrete ground on real-time RCD for future research.

REFERENCES

[1] Oncel Tuzel, Fatih Porikli, and Peter Meer. Region covariance: A fast descriptor for detection and classification. In *Proceedings of the 9th European Conference on Computer Vision - Volume Part II*, ECCV'06, pages 589–600. Springer-Verlag, Berlin, Heidelberg, **2006**. ISBN 3-540-33834-9, 978-3-540-33834-5. doi:10.1007/11744047_45.

[2] Region covariance descriptor matlab implementation, **2006**.

[3] Object tracking with rcd serial c++ implementation, **2010**.

[4] David A Forsyth and Jean Ponce. Computer vision: A modern approach. In ., pages 88–101. **2003**.

[5] Benjamin Coifman, David Beymer, Philip McLauchlan, and Jitendra Malik. A real-time computer vision system for vehicle tracking and traffic surveillance. In *Transportation Research Part C: Emerging Technologies, Vol 6 Number 4*, pages 271–288. **1998**.

[6] Rao KMM and Rao VDP. Medical image processing. In *Proceedings of Workshop on Medical Image Processing and Applications*, pages –. **2006**.

[7] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam). In *Robotics Automation Magazine Part II, Vol 13 Number 3*, pages 528–533. **2006**.

[8] Jürgen Schmidhuber. Deep learning in neural networks: An overview. In *Neural Networks Volume 61*, pages 85–117. **2015**.

[9] Hannes Fassolda and Jakub Rosner. A real-time gpu implementation of the sift algorithm for large-scale video analysis tasks. In *Real-Time Image and Video Processing*. **2015**.

[10] Blaine Rister, Guohui Wang, Michael Wu, and Joseph R. Cavallaro. A fast and efficient sift detector using the mobile gpu. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2674–2678. **2013**.

[11] Ahmed Mehrez, Ahmed A. Morgan, and Elsayed E. Hemayed. Speeding up spatiotemporal feature extraction using gpu. In *Journal of Real-Time Image Processing*, pages 1–29. **2018**.

[12] K Aniruddha Acharya, R Babu, and Sathish S. Vadhiyar. A real-time implementation of sift using gpu. In *Journal of Real-Time Image Processing*. **2014**.

[13] Jiashen Li and Yun Pan. Gpu-based parallel optimization for real-time scale-invariant feature transform in binocular visual registration. *Personal and Ubiquitous Computing*, 23, **2019**. doi:10.1007/s00779-019-01222-3.

[14] Ali Ismail Awad. Fingerprint local invariant feature extraction on gpu with cuda. In *Informatica (Slovenia) Vol 37*, pages 279–284. **2013**.

[15] Myung Hwan Tak and Young Hoon Joo. Localization and autonomous navigation using gpu-based sift and virtual force for mobile robots. In *The Transactions of The Korean Institute of Electrical Engineers Vol 65*, pages 1738–1745. **2016**.

[16] M Sridevi, S Aishwarya, Amedapu Nidheesha, and Divyansh Bokadia. *Parallel Image Forgery Detection Using FREAK Descriptor: Proceedings of ICTIS 2018, Volume 2*, pages 619–630. **2019**.

[17] Leonardo Rundo, Andrea Tangherloni, Simone Galimberti, Paolo Cazzaniga, Ramona Woitek, Evis Sala, Marco Nobile, and Giancarlo Mauri. *HaraliCU: GPU-Powered Haralick Feature Extraction on Medical Images Exploiting the Full Dynamics of Gray-Scale Levels*, pages 304–318. **2019**. ISBN 978-3-030-25635-7. doi:10.1007/978-3-030-25636-4_24.

[18] Robert Haralick, K Shanmugam, and Ih Dinstein. Textural features for image classification. *IEEE Trans Syst Man Cybern*, SMC-3:610–621, **1973**.

[19] Sudipta N. Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Feature tracking and matching in video using programmable graphics hardware. In *Machine Vision and Applications Vol 22*, pages 202–217. **2007**.

[20] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. In *International Journal of Computer Vision Vol 9*, pages 137–154. **1991**.

[21] Chih-Hsiang Chang and Nasser Kehtarnavaz. Computationally efficient image deblurring using low rank image approximation and its gpu implementation. In *Journal of Real-Time Image Processing Vol 12 Number 3*, pages 567–573. **2016**.

[22] Serdar Cakir, Tayfun Aytaç, Alper Yildirim, Soosan Beheshti, Ö Nezih Gerek, and A Enis Cetin. Salient point region covariance descriptor for target tracking. In *Optical Engineering Vol 52 Number 2*. **2013**.

[23] Levent Karacan, Erkut Erdem, and Aykut Erdem. Structure-preserving image smoothing via region covariances. In *ACM Transactions on Graphics (TOG) Vol 32 Number 6*. **2013**.

[24] Nicole M. Artner, Adrian Ion, and Walter G. Kropatsch. Multi-scale 2d tracking of articulated objects using hierarchical spring systems. In *Pattern Recognition, Volume 44*, pages 800–810. **2011**.

[25] Kevin Mader and Gil Reese. Using covariance matrices as feature descriptors for vehicle detection from a fixed camera. In *ArXiv*. **2012**.

[26] H. Faulkner, E. Shehu, Z. L. Szpak, W. Chojnacki, J. R. Tapamo, A. Dick, and A. van den Hengel. A study of the region covariance descriptor: Impact of feature selection and image transformations. In *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. **2015**.

[27] Xiaomin Yang, Lihua Jian, Wei Wu, Kai Liu, Binyu Yan, Zhili Zhou, and Jian Peng. Implementing real-time rcf-retinex image enhancement method using cuda. *J. Real-Time Image Process.*, 16(1):115–125, **2019**. ISSN 1861-8200. doi:10.1007/s11554-018-0803-y.

[28] Anoop Cherian, Suvrit Sra, Arindam Banerjee, and Nikolaos P Papanikolopoulos. Efficient similarity search for covariance matrices via the jensen-bregman logdet divergence. In *2011 International Conference on Computer Vision, ICCV 2011*, pages 2399–2406. **2011**.

[29] Ty McKercher John Cheng, Max Grossman. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, **2014**. ISBN 1118739329, 9781118739327.

[30] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I. **2001**. ISSN 1063-6919. doi:10.1109/CVPR.2001.990517.

[31] Wolfgang Förstner and Boudewijn Moonen. *A Metric for Covariance Matrices*, pages 299–309. Springer Berlin Heidelberg, Berlin, Heidelberg, **2003**. ISBN 978-3-662-05296-9. doi:10.1007/978-3-662-05296-9_31.

[32] Berkin Bilgic, Berthold KP Horn, and Ichiro Masaki. Efficient integral image computation on the gpu. In *Intelligent Vehicles Symposium (IV)*, pages 108–117. **2010**.

[33] Nvidia performance primitives, **2019**.

[34] Guy E. Blelloch. Prefix sums and their applications.

[35] Harris M. Parallel reduction. In *NVIDIA Developers Guide*. **2010**.

[36] Cuda c programming guide, **2012**.

HACETTEPE UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
THESIS/~~DISSERTATION~~ ORIGINALITY REPORT

HACETTEPE UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
TO THE DEPARTMENT OF COMPUTER ENGINEERING

Date: 27/09/2019

Thesis Title / Topic: CUDA BASED REAL TIME IMPLEMENTATION OF REGION COVARIANCE DESCRIPTORS

According to the originality report obtained by ~~myself~~/my thesis advisor by using the *Turnitin* plagiarism detection software and by applying the filtering options stated below on 26/09/2019 for the total of 47 pages including the a) Title Page, b) Introduction, c) Main Chapters, d) Conclusion sections of my thesis entitled as above, the similarity index of my thesis is 6 %.

Filtering options applied:

1. Bibliography/~~Works Cited~~ excluded
2. Quotes ~~excluded~~ / included
3. Match size up to 5 words excluded

I declare that I have carefully read Hacettepe University Graduate School of Science and Engineering Guidelines for Obtaining and Using Thesis Originality Reports; that according to the maximum similarity index values specified in the Guidelines, my thesis does not include any form of plagiarism; that in any future detection of possible infringement of the regulations I accept all legal responsibility; and that all the information I have provided is correct to the best of my knowledge.

I respectfully submit this for approval.

27.09.2019
Date and Signature

Name Surname:	Muhammet Ali ASAN
Student No:	N14223202
Department:	COMPUTER ENGINEERING
Program:	COMPUTER ENGINEERING
Status:	☒ Masters ☐ Ph.D. ☐ Integrated Ph.D.

ADVISOR APPROVAL

APPROVED.

Assistant Prof. Adnan ÖZSOY

(Title, Name Surname, Signature)

ÖZGEÇMİŞ

Adı Soyadı	: Muhammet Ali ASAN
Doğum yeri	: Şanlıurfa
Doğum tarihi	: 19.11.1990
Medeni hali	: Bekar
Yazışma adresi	: Alpaslan Mah.Yakakent Cad. No:81 Alacam/Samsun
Telefon	: 0531 833 5506
Elektronik posta adresi	: muhammetaliasan@outlook.com
Yabancı dili	: İngilizce

EĞİTİM DURUMU

Lisans	: Ankara Üniversitesi Elektrik-Elektronik Müh. (2014)

İş Tecrübesi

2019-	Pix4D GmbH
2018-2019	Luxoft GmbH
2015-2018	STM A.Ş.
2014-2015	Sim-Tek Simülasyon Teknolojileri