Optimizing Vision Benchmark for GPU

By

Dirgh Buch
[09MCE028]

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
AHMEDABAD-382481

May 2012

Optimizing Vision Benchmark for GPU

Major Project II

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

PREPARED BY :

Dirgh Buch

09MCE028

GUIDED BY :

Prof.Vibha Patel

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD

DECLARATION

I, **Dirgh Buch**, **09MCE028**, give undertaking that the Major Project entitled **"Optimizing Vision Benchmark for GPU"** submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in Institute of Technology of Nirma University, Ahmedabad, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Dirgh Buch

DEDICATION

This thesis is dedicated to **God Almighty**, who has been my driving force in all situations I faced during the project work.
I would also like to dedicate this thesis to my guide **Prof. Vibha Patel**, whose continuous guidance and suggestion kept me motivated and also to my family who are my moral support for the whole tenure.

CERTIFICATE

This is to certify that the Major Project I, entitled "Optimizing Vision Benchmark for GPU", submitted by Mr. Dirgh V. Buch [09MCE028], towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In many opinion, submitted work has reached a level required for being accepted for examination. The result embodied in this major project I, to the best of my knowledge,haven't been submitted to any other university or institution for award of any master degree.

Prof. Vibha Patel
Guide, Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. S.N. Pradhan
Professor,P.G. Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. Ketan Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

ABSTRACT

The San Diego Vision Benchmark Suite (SD-VBS), a suite of diverse vision applications drawn from the vision domain. From the assembly line to home entertainment systems, the need for efficient real-time computer vision systems is growing rapidly.NVIDIA has developed the CUDA (Compute Unified Device Architecture) which can be used to speedup computer vision application. CUDA enables software developers to access the GPU through standard programming languages such as 'C'. It also gives developers access to the GPU's virtual instruction set, onboard memory and the parallel computational elements. Taking advantage of parallel computation will significantly speedup applications. Here we explore the potential power of using CUDA and NVIDIA GPUs to speedup common computer vision algorithms along with algorithmic optimizations. Approaches to optimize few applications of SD-VBS on GPU are part of this thesis. To analyze simulation time of these applications inputs of different size are feeded.

ACKNOWLEDGEMENT

It gives me great pleasure in expressing thanks and profound gratitude to **Prof. Vibha Patel** ,Professor, Department of Computer Science and Engineering,Institute of Technology, Nirma University, Ahmedabad for her valuable guidance and continual encouragement throughout the Major Project II. I am heartily thankful to her for her time to time suggestion and clarity of the concepts of the topic that helped me a lot during my project work.

I like to give my special thanks to **Dr. S. N. Pradhan**, P.G. Coordinator, Department of Computer Science and Engineering,Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the project.

Without the blessings of God,these work would not have been so easy to carry out.
I am grateful to my family members who were always there to keep me morally motivated all the time.
I am equally grateful to all my friends who were doing in-house project.Without them this whole year would not have been so joyful and it really went by ease.

Dirgh Buch
[**09MCE028**]

Abbreviation Notation and Nomenclature

API	Application Program Interface
ALU	Arithmetic Logic Units
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUDA FFT	CUDA Fast Fourier Transform
CUBLAS	CUDA Basic Linear Algebra Subroutine
GFLOPS	Million Floating Point Instructions Per Second
GPGPU	General Purpose Graphics Processing Units
GPU	Graphics Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
NVCC	CUDA Compiler
PTX	Parallel Thread Execution
MP	Multiprocessors
SD-VBS .... San Diego Vision Benchmark SDK	Software Development Kit
SP	stream processors

Contents

Certificate **v**

Abstract **vi**

Acknowledgement **vii**

Abbreviation Notation and Nomenclature **viii**

Contents **viii**

List of Figures **xi**

List of Tables **xii**

1 Introduction **1**

1.1 Motivation of the Project 2

1.2 Objective & Scope 3

1.2.1 Objective 3

1.2.2 Scope Of The Project 3

2 Literature Survey and Important observations **4**

2.1 San Diego Vision Benchmark Suit(SD-VBS) 4

2.1.1 Feature Tracking 5

2.1.2 Scale invariant feature Tracking(SIFT) 6

2.1.3 Disparity Map 7

2.2 Introduction to NVIDIA CUDA 8

2.2.1 CUDA Memory Model 9

2.2.2 CUDA Execution Model 11

2.2.3 CUDA Programming Model 11

3 Implementation and Performance Analysis 13
3.1 Methodology 13
3.2 Feature Tracking 14
3.3 Scale Invariant Feature Transform(SIFT) 20
3.4 Disparity Map 23
3.5 Overall Analysis 24

4 Conclusion and future work 26
4.1 Conclusion 26
4.2 Future Work 26

References 27

A Installation of CUDA 29
A.1 CUDA 4.0 with Parallel Nsight 29
A.1.1 Installation 29
A.1.2 Creating and Executing Visual studio project 29
A.1.3 Debugging in Parallel Nsight 30
A.2 CUDA 2.3 in emulation mode 30
A.2.1 Creating and Executing Visual studio project 30

List of Figures

1.1 Floating point operations per second for the GPU and CPU(Source [8]) . . . 3

2.1 Applications in SD-VBS . . . 4
2.2 Applications in SD-VBS . . . 5
2.3 Applications in SD-VBS . . . 6
2.4 Applications in SD-VBS . . . 7
2.5 CUDA Software Stack . . . 8
2.6 CUDA Memory Model . . . 9
2.7 CUDA Programming/Execution Model. . . . 12

3.1 Feature Tracking hotspots . . . 14
3.2 Feature Tracking Phase wise Execution Time . . . 16
3.3 Blur function Execution time Behavior . . . 17
3.4 Feature Tracking Output . . . 18
3.5 Execution time for different Datasets of Tracking . . . 19
3.6 Smooth function Execution time Behavior . . . 20
3.7 SIFT Phase wise Execution Time . . . 22
3.8 Disparity Map Phase wise Execution Time . . . 23
3.9 Execution time for Different Datasets of Disparity Map . . . 24
3.10 Execution time for all three applications in SD-VBS . . . 25

List of Tables

2.1 Characteristics of CUDA memories 10

3.1 Performance Improvement in SD-VBS applications 24

Chapter 1

Introduction

Computer vision is the science that enables computer systems to extract information from an image or a sequence of images. The development of computer vision is essential for the advancement of a multitude of areas including medical, entertainment and security. Computer vision systems are useful for tasks such as industrial control, event detection, informational organization and object modeling. Other domains of computer vision systems also include motion analysis, image restoration and scene construction. With a wide variety of emerging applications, the demand for more advanced computer vision systems is quickly growing.

There are several limiting factors when developing accurate real-time computer vision systems. Most computer vision tasks require a great deal of mathematical computation. For many computer vision algorithms, the analysis of a single image can take anywhere from a few seconds to several hours to process. In short, computer vision algorithms require a large number of computations as well as an equally large number of memory values. Computer vision algorithms are applied to broad range of environments from home entertainment systems to the operation of unmanned aerial and ground vehicles. Each new generation of application increases the need for more computational resources. Traditionally software developers and even scientists have strictly relied on the increase of processor clock frequency as the primary method of gaining performance for the next generation of applied algorithms.[12].

Graphics Processing Units (GPUs) are commonly found on graphics cards and computer mother boards. These specialized processors have great potential for solving a number of problems. Unlike traditional Central Processing Units (CPUs), the GPU contains as many as several hundred mathematical computation cores. These cores have evolved recently from being able to perform simple graphics computations to fully capable processing engines. Due to the nature of many processing cores, GPUs can perform mathematical computations in a massively parallel manner.

Many applications and algorithms have the potential to take advantage of the parallel processing capabilities of GPU systems. In most cases, problems possessing data level parallelism are best suited for GPU execution. Data parallelism focuses on distributing the large amounts of data across different parallel computing cores. A problem appears data-parallel if each core can perform the same identical task on different pieces of distributed data. There are distinct ranges of data-parallel problems.

Small scale image processing that includes the parallel manipulation or analysis of pixels can be achieved with multiprocessor extensions such as Single Instruction, Multiple Data (SIMD). Larger problems of data-parallel computing can be solved with large scale distributed systems consisting of multiple independent computers that communicate through a computer network or network grid. Non-large-scale data-level parallel problems are ideal applications that can be optimized. In this categorization, modern computer vision applications fall into a unique domain since they are more complex than simple image processing tasks yet would not be described as large enough to require massive computing resources of a distributed system.

In this thesis we have carried out various optimizations possible on the applications from SD-VBS. We have shown our experimental results and our methodology to carry out the optimization.We have shown the performance improvement because of the use of GPU for parallelization.

1.1 Motivation of the Project

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. As we can see from the graph in figure 1.1 there is drastic difference in GFLOPs between the best CPU architecture and GTX 480 on which we have tested our implementation.It is the reason to chose GPU for parallelization instead of a multicore CPU.

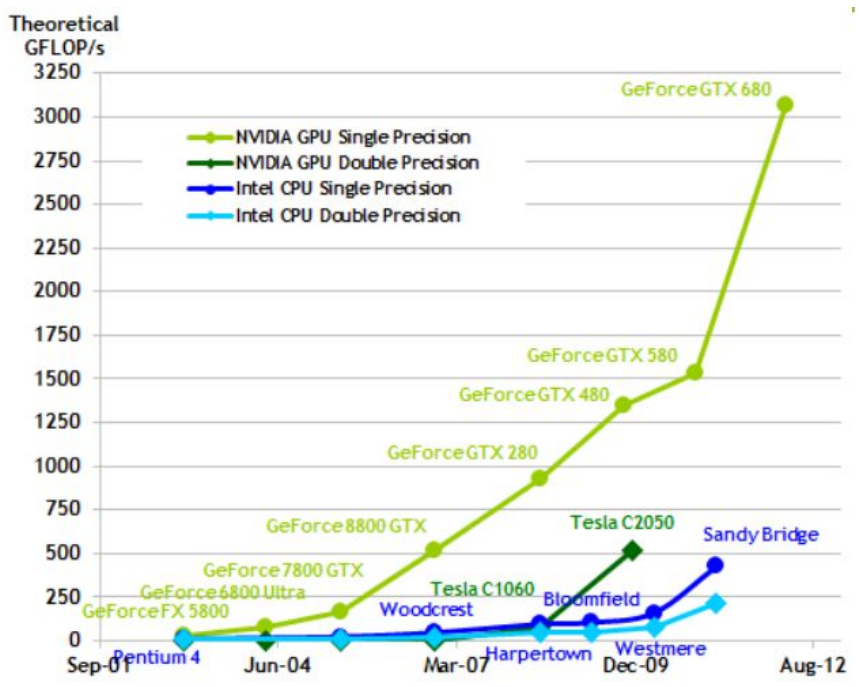


Figure 1.1: Floating point operations per second for the GPU and CPU(Source [8])

1.2 Objective & Scope

1.2.1 Objective

Objective of this project is to optimize San-Diego Vision Benchmark for GPU.

1.2.2 Scope Of The Project

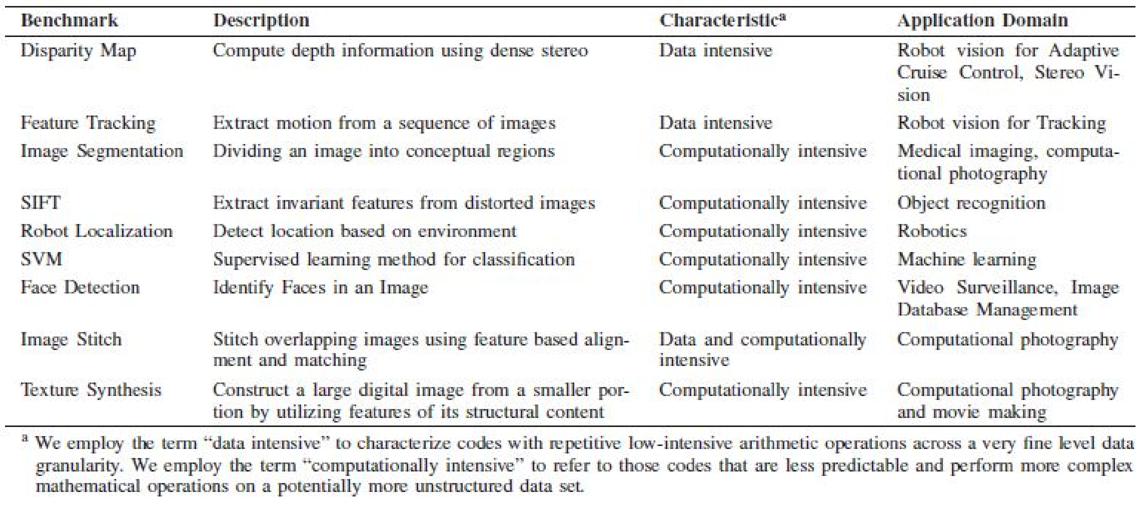
Optimization of Applications of SD-VBS benchmark Feature Tracking, SIFT and Disparity Map.

Chapter 2

Literature Survey and Important observations

2.1 San Diego Vision Benchmark Suit(SD-VBS)

The Department of Computer Science and Engineering at the University of California, San Diego has developed a vision benchmark suite known as "The San Diego Vision Benchmark Suite" (SD-VBS) [1]. The suite contains applications from the following representative areas: Image Processing and Formation, Image Analysis, Image Understanding, and Motion, Tracking and Stereo Vision. This suite contains nine representative computer vision applications shown in figure2.1 and each application contains a set of image inputs that vary in size.



Benchmark	Description	Characteristic[a]	Application Domain
Disparity Map	Compute depth information using dense stereo	Data intensive	Robot vision for Adaptive Cruise Control, Stereo Vision
Feature Tracking	Extract motion from a sequence of images	Data intensive	Robot vision for Tracking
Image Segmentation	Dividing an image into conceptual regions	Computationally intensive	Medical imaging, computational photography
SIFT	Extract invariant features from distorted images	Computationally intensive	Object recognition
Robot Localization	Detect location based on environment	Computationally intensive	Robotics
SVM	Supervised learning method for classification	Computationally intensive	Machine learning
Face Detection	Identify Faces in an Image	Computationally intensive	Video Surveillance, Image Database Management
Image Stitch	Stitch overlapping images using feature based alignment and matching	Data and computationally intensive	Computational photography
Texture Synthesis	Construct a large digital image from a smaller portion by utilizing features of its structural content	Computationally intensive	Computational photography and movie making

[a] We employ the term "data intensive" to characterize codes with repetitive low-intensive arithmetic operations across a very fine level data granularity. We employ the term "computationally intensive" to refer to those codes that are less predictable and perform more complex mathematical operations on a potentially more unstructured data set.

Figure 2.1: Applications in SD-VBS

2.1.1 Feature Tracking

Tracking is about extracting motion information from a sequence of images.Feature extraction and a linear solver that calculates the movement of features. Robotic Vision and Automotive domain for realtime vehicle are the areas where feature tracking is widely used. Kanade Lucas Tomasi (KLT) tracking algorithm is implemented in SD-VBS for feature tracking[1]. The algorithm comprises of three major computation phases: image processing, feature extraction and feature tracking shown in 2.2. The image processing phase works on operate on pixel level granularity which involves noise filtering, gradient image and image pyramid computations.

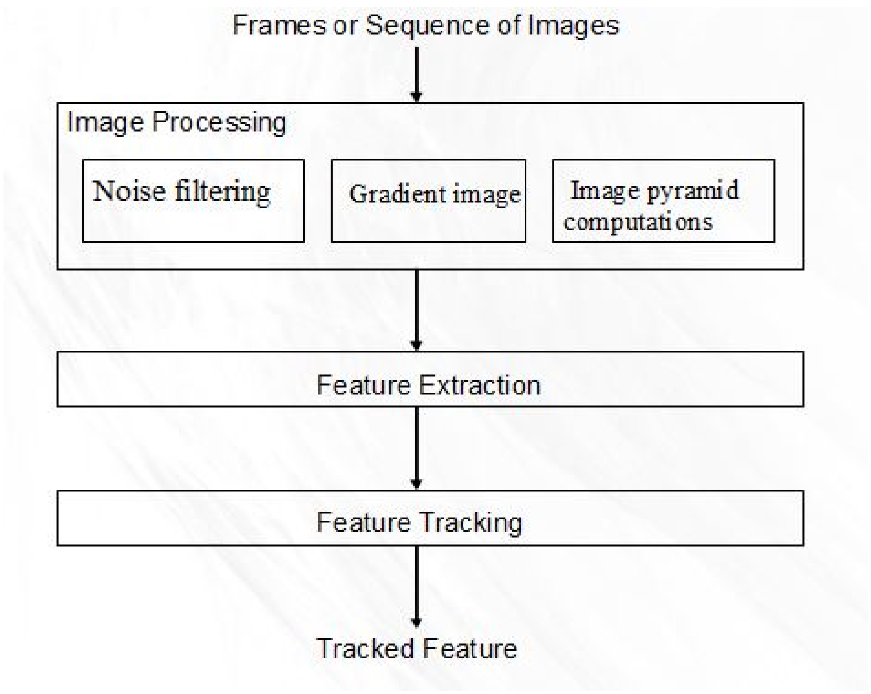


Figure 2.2: Applications in SD-VBS

The main part of algorithm, feature extraction and tracking, operates on coarse grained data, which is identified by the features[2] . The working set varies significantly across the major computation blocks. Image processing field is parallelization friendly because while it operates on the entire image, the operations are restricted to Multiply and Accumulate (MAC) making it a data intensive phase of the application. Complex matrix operations such as matrix inversion and motion vector estimation, makes feature extraction and tracking kernels computationally intensive and makes exploitation of innate parallelism more challenging.

2.1.2 Scale invariant feature Tracking(SIFT)

The Scale Invariant Feature Transform (SIFT) algorithm is used to detect and describe robust and highly distinctive features in images. These features are invariant to scaling, rotation and noise.SIFT finds wide applicability in domains such as object recognition, image stitching, 3D modeling, video tracking. An interesting and desired feature of SIFT descriptors is its robustness to partial occlusions, which makes it a pervasive algorithm for navigation and match moving applications.

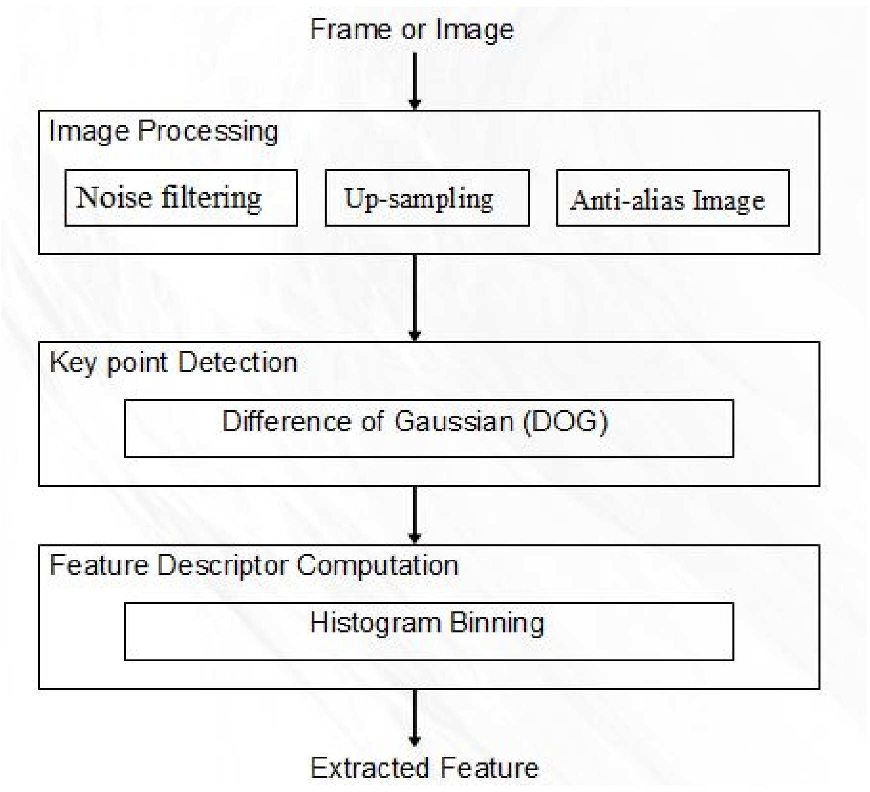


Figure 2.3: Applications in SD-VBS

SIFT implements David Lowes Scale Invariant Feature Transform algorithm in the SD-VBS. This algorithm computes features and their descriptors given a gray scale image. Key kernels of the algorithm are image processing, key point detection and feature descriptor computation shown in 2.3. The preprocessing stage of SIFT involves filtering operations in addition to a data compute intensive linear interpolation phase which includes upsampling to extract anti-alias image. The detection of key point phase involves creation and pruning of the Difference of Gaussian (DOG) Pyramids. Creation of the DOG is data intensive while feature extraction is computationally intensive. To assign orientations to feature points in the descriptor computation kernel, histogram binning is implemented. This phase is highly compute intensive. The image processing and DOG creation phase are characterized by regular pre-fetch friendly memory access pattern and predictable working set. Identification of key points and descriptor assignment is plagued by irregular memory pattern and intensive computations.[3].

2.1.3 Disparity Map

Disparity map algorithm is about calculating depth information.Given a pair of stereo images for a scene, taken from slightly different positions, it computes the depth information for objects jointly represented in the two pictures. The depth information is useful to decide the relative position of objects. Robot vision systems use Disparity Map extensively to compute the depth information, which is useful in applications such as cruise control, pedestrian tracking, and collision control. The implementation is based on Stereopsis, also known as Depth Perception[4]. From a stereo image pair, the disparity algorithm computes dense disparity.

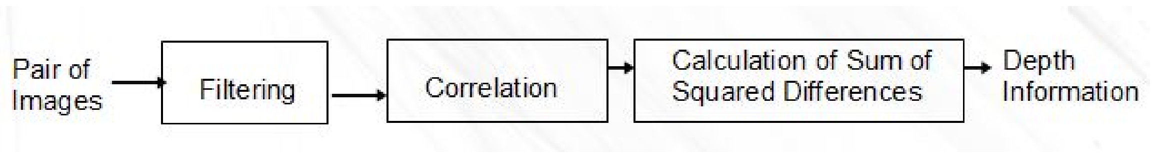


Figure 2.4: Applications in SD-VBS

Dense disparity operates on pixel level granularity unlike sparse disparity where depth information is computed on features of interest. filtering, correlation, calculation of sum of squared differences (SSD) and sorting are major kernels of Disparity map shown in 2.4. SD-VBS has the 2-D filtering operation implemented as two 1-D filters for better cache locality. Correlation and SSD are computed on every pixel across the image, making them expensive data intensive operations.
In conclusion, disparity is a parallelization-friendly algorithm whose performance is only limited by the ability to pull the data into the chip.

2.2 Introduction to NVIDIA CUDA

NVIDIA introduced CUDA (Compute Unified Device Architecture), a general purpose parallel computing architecture with a parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. [6].

NVIDIA CUDA SDK has been designed for running parallel computations on the device hardware: it consist of a compiler, host and device runtime libraries and a driver API. CUDA software stack is composed of several layers: a hardware driver (CUDA Driver), an API and its runtime (CUDA Runtime), two higher-level mathematical libraries (CUDA Libraries) of common usage as shown in figure 2.5.[7]

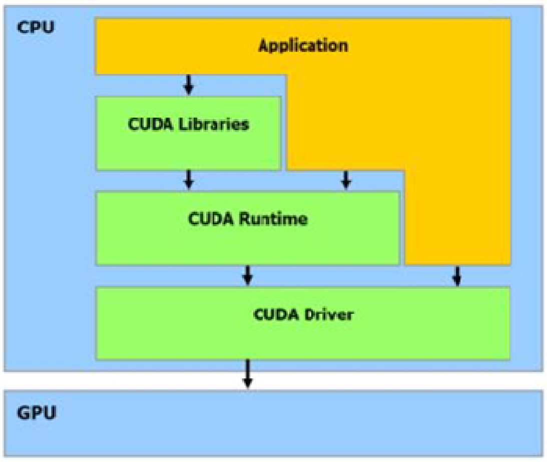


Figure 2.5: CUDA Software Stack

GPU performance is influenced by the architectural organization of the hardware platform. NVIDIA suggests that achieving the highest GPU occupancy and optimizing the use of the memory hierarchy are the two main factors behind GPU performance [8]. In fact, both of them are related since maximizing the occupancy can help to cover latency during global memory loads. Overall, they ensure the best performance even if some resources remain under utilized. Therefore, maximizing occupancy should be examined at a later stage in the compilation process, once data related issues have been properly addressed.

2.2.1 CUDA Memory Model

CPU and GPU have separate memory spaces. CPU memory known as host memory and GPU memory known as Device memory. CUDA offers different types of memories with different configuration. The local, global, constant and texture spaces are regions of device memory [6]. Each multiprocessor has following memory space as shown in figure 2.6

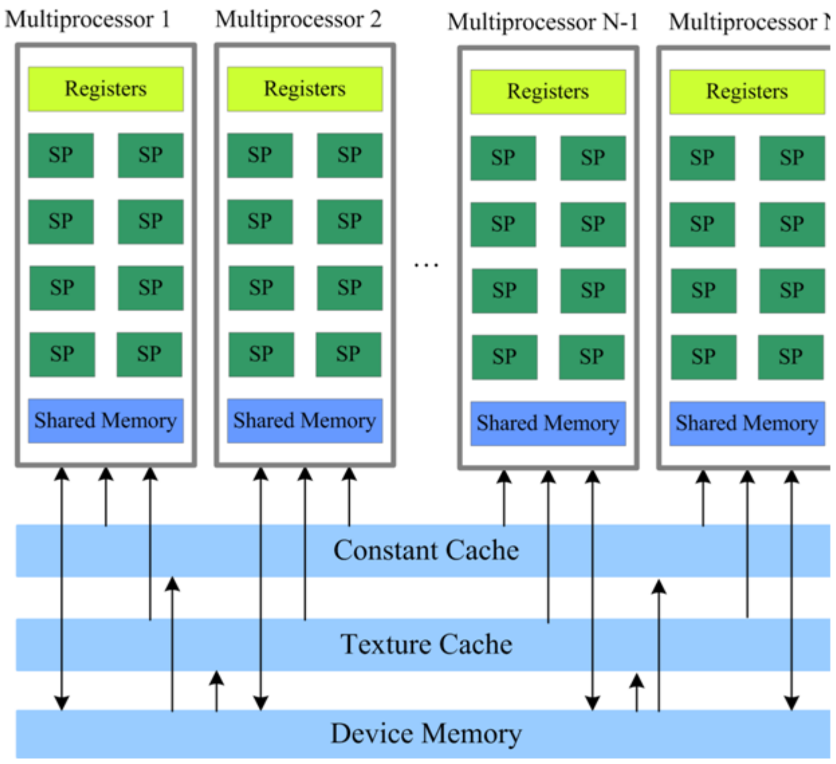


Figure 2.6: CUDA Memory Model

- **Local Memory:** It is small volume of memory, which can be accessed only by one streaming processor. The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses. Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are [6]:
  - Arrays for which it cannot determine that they are indexed with constant quantities,
  - Large structures or arrays that would consume too much register space,

Memory	Location	Cached	Access	Scope
Register	On-Chip	NO	Read/Write	One Thread
Local	Off-Chip	NO	Read/Write	One Thread
Global	Off-Chip	No	Read/Write	All Threads + Host
Constant	Off-Chip	YES	Read	All Threads + Host
Texture	Off-Chip	YES	Read	All Threads + Host

Table 2.1: Characteristics of CUDA memories

  - Any variable if the kernel uses more registers than available (this is also known as register spilling).

- **Global Memory:** It is the largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern solutions (and up to 4 GB in Tesla). It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles).
  Global memory resides in device memory and device memory is accessed via 32-,64-, or 128-byte memory transactions.

- **Shared Memory:** It is 16-KB memory shared between all streaming processors in a multiprocessor. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

- **Constant Memory:** It is a 64 KB, read only memory for all multiprocessors. It's cached by 8 KB for each multiprocessor. The constant memory space resides in device memory.

- **Texture Memory:** This memory space resides in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. not fetch latency.

2.2.2 CUDA Execution Model

CUDA Execution model consist of Grid, ThreadBlocks, and Threads.

- **Grid :** An entire grid is handled by a single GPU chip.
- **ThreadBlocks :** The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.
- **Threads :** Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.

2.2.3 CUDA Programming Model

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The GPU is viewed as a compute device : that is a coprocessor to the CPU(host), has its own DRAM(device Memory), Runs many threads in parallel [4]. Data parallel portion of application are executed on the device as kernels which run in parallel on many threads. Difference between GPU and CPU thread are:

- GPU threads are extremely lightweight and requires very little creation overhead.
- GPU needs 1000s of threads for full efficiency where as multicore CPU needs only a few.

A kernel is executed as a grid of thread blocks. A thread block is a batch of thread that can cooperate with each other by efficiently sharing data through shared memory, and synchronizing there execution for hazard free shared memory accesses. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by figure 2.7. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system. It must be possible to execute thread block in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling

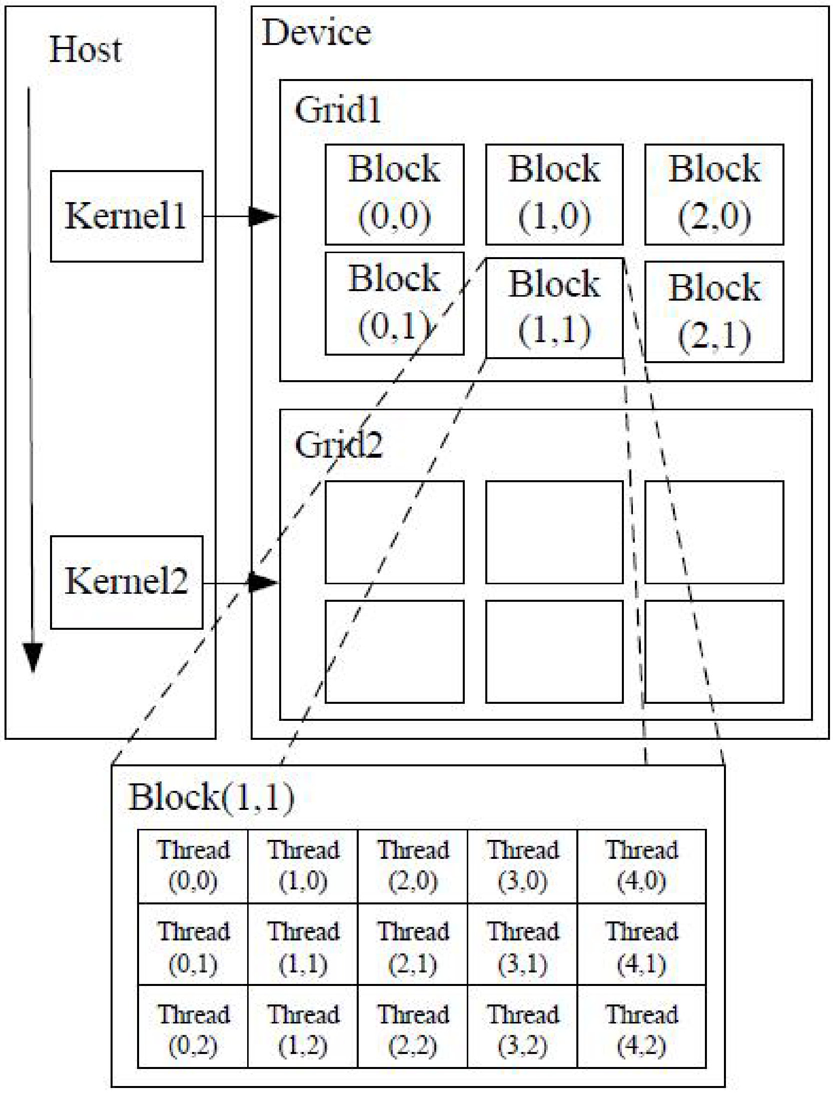


Figure 2.7: CUDA Programming/Execution Model.

programmers to write code that scales with the number of cores. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and thread synchronization is expected to be lightweight.

Chapter 3

Implementation and Performance Analysis

Here we have shown the results of parallelization of different applications in SD-VBS. We have also taken into consideration various problem size to compare the results. For comparing the result we have varied the size of problems.We have chosen following datasets for different size of problems, one with highest size, one with medium size and the one with least size.

- fullhd (1920 x 1080)
- vga (680 x 480)
- cif (352 x 288)

All the implementations are done in Visual Studio 2008 with CUDA 4.0. They are tested on hardware NVIDIA GeForce GPU GTX-480 with block size 32 x 32.

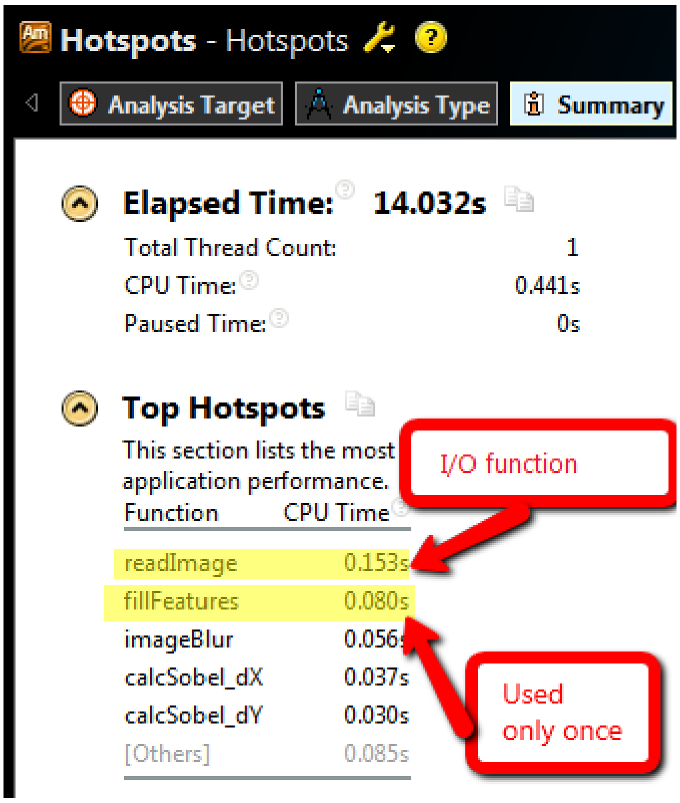
3.1 Methodology

- Execute the application code as Visual Studio project.
- Using Intel V-Tune Performance Amplifier identify the hotspots.
- Identify the modules which are frequently used in application by code inspection.
- Make CUDA kernel for the identified hotspot and frequently used modules.
- Compare the execution time for both sequential and parallel module.

- Look for the other possible optimizations.
- Compare final results.

3.2 Feature Tracking

Feature Tracking application is data intensive application[1].In this application we extract motion information from sequence of images. Figure 3.1 shows the result of V-Tune Performance Amplifier.



Elapsed Time: 14.032s

Total Thread Count:	1
CPU Time:	0.441s
Paused Time:	0s

Top Hotspots

This section lists the most application performance.

Function	CPU Time
readImage	0.153s
fillFeatures	0.080s
imageBlur	0.056s
calcSobel_dX	0.037s
calcSobel_dY	0.030s
[Others]	0.085s

Figure 3.1: Feature Tracking hotspots

As *imageRead()* is I/O operation it cannot be converted into parallel code and the *fillfeature()* function is used only once.From the figure 3.1 following are the major modules or kernels which can be parallelized.

- imageBlur
- calcSobelX
- calcSobelY

First we need to convert the hotspot into CUDA kernel.*calcSobelX()* has parallel portion that we can convert into CUDA code.Following code shows the parallel portion of function.

```
F2D* calcSobel_dX(F2D* imageIn)
{
\\ variable initialization
for(i=startRow; i<endRow; i++)
    {
        for(j=startCol; j<endCol; j++)
        {
        \\Sequantial Code
        }
    }
    \\ return
}
```

CUDA code for the same is shown below.

```
calcSobelXCUDA(F2D *imageIn_d,F2D *tempOut_d,F2D *kernel_1d,F2D *kernel_2d,F2D *
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
int idy = blockIdx.y * blockDim.y + threadIdx.y;
        \\ Code convert to work in parallel
}
function calculateSobelX(F2D* imageIn)
{
    \\ Device Memory allocation
    \\ Data trasfer from host to device
    \\ Block and thread creation
    calcSobelXCUDA<<<dimgrid,dimBlock >>>(imageIn_d,tempOut_d,kernel_1d,kernel_2
    \\ Data transfer from device to host
    \\ Release device memory
}
```

For the parallelization of the application following graph in figure 3.2 shows phase wise reduction in execution time for fullhd dataset.

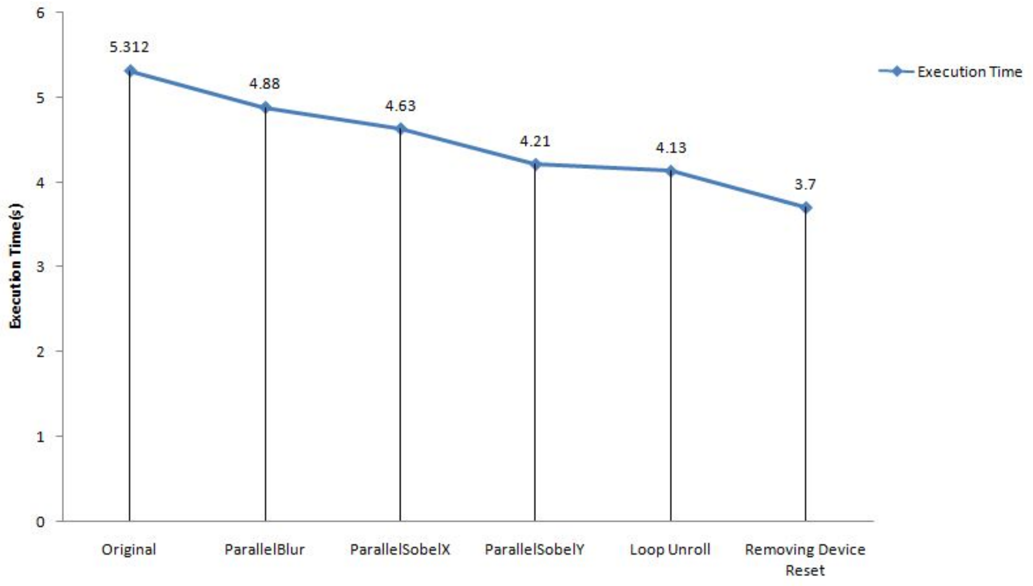


Figure 3.2: Feature Tracking Phase wise Execution Time

As we can see there is continuous performance improvement as we keep on parallelizing modules. Loop unrolling phase shows the performance improvement when we unroll steps in *for* loops for the faster execution.Initial stage the implementation was giving memory allocation failed error. To handle that error *cudaDeviceReset()* function was used. When the implementation was complete the code was tested by removing the function which resulted in quite performance improvement. To understand such behavior each kernel was separately measured in execution time. It was observed that the memory transfer overhead comes into picture, only for the first time the kernel is called. For the rest of the executions it shows lesser execution time. Figure 3.3 graph shows this behavior.

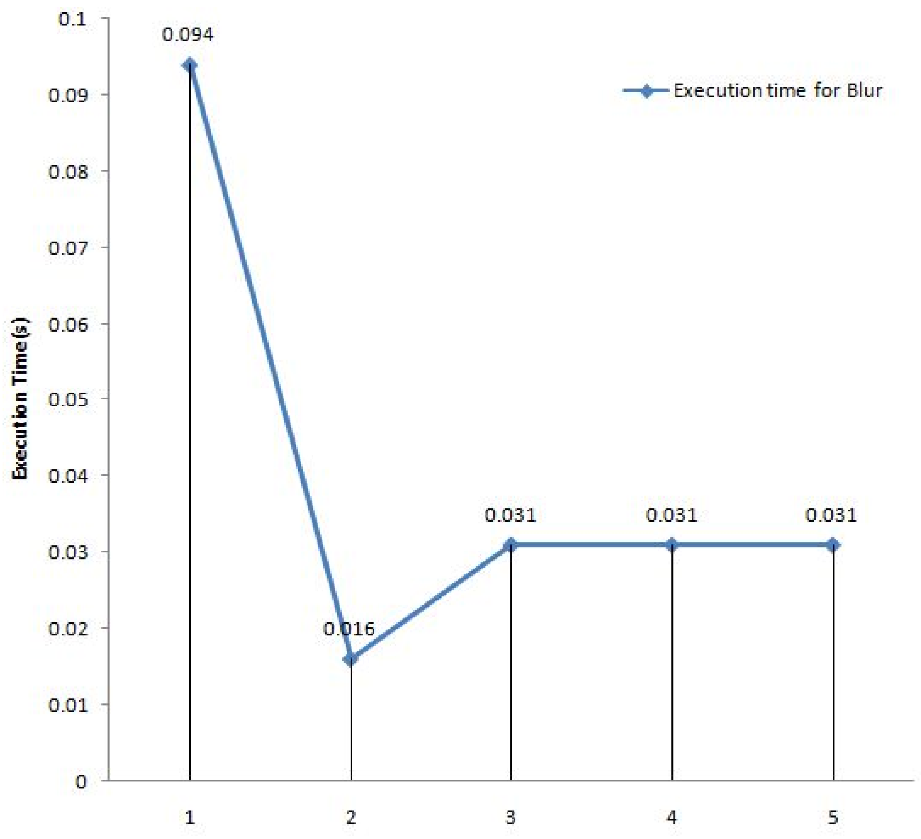
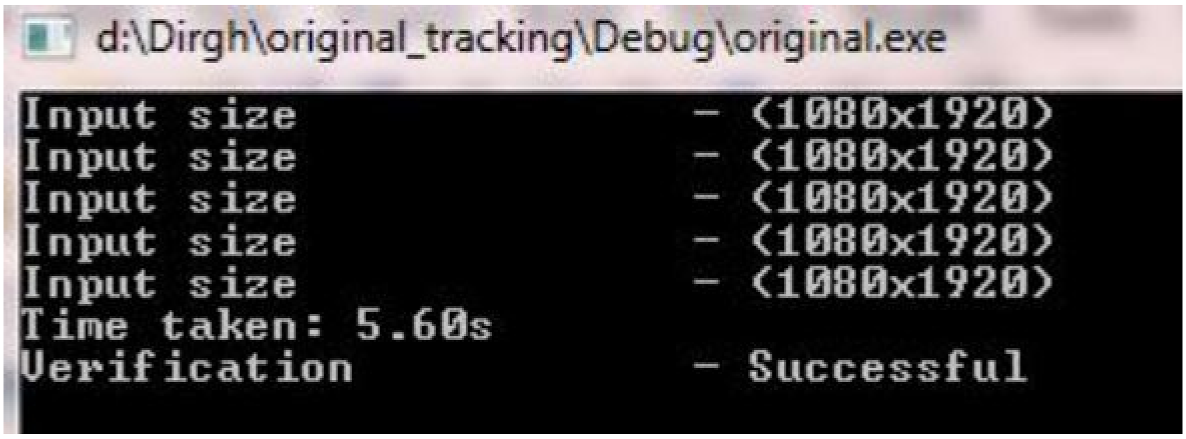


Figure 3.3: Blur function Execution time Behavior

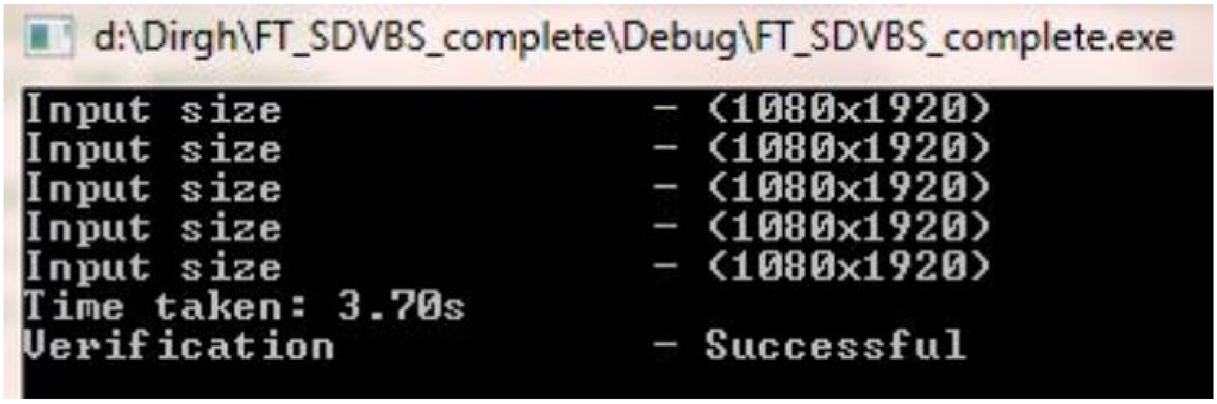
3.4a and 3.4b figures shows the complete execution of parallel and sequential version of algorithm.so from that we can calulate speed up as

```
Sequantial/parallel=5.60/3.70=1.513
```



```
d:\Dirgh\original_tracking\Debug\original.exe
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Time taken: 5.60s
Verification                    - Successful
```

(a) Sequantial Execution of Feature Tracking



```
d:\Dirgh\FT_SDVBS_complete\Debug\FT_SDVBS_complete.exe
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Input size                      - (1080x1920)
Time taken: 3.70s
Verification                    - Successful
```

(b) Parallel Execution of Feature Tracking

Figure 3.4: Feature Tracking Output

Apart from the *blur* function behavior there is another phenomenon regarding the size of the datasets. As we have discussed earlier we have taken three size of datasets cif,vga and fullhd. fullhd has the highest size. Our experiments shows that smaller the size CPU outperforms GPU because of the certain overheads associated with GPU execution. Graph in figure 3.5 shows the result of our experiments.

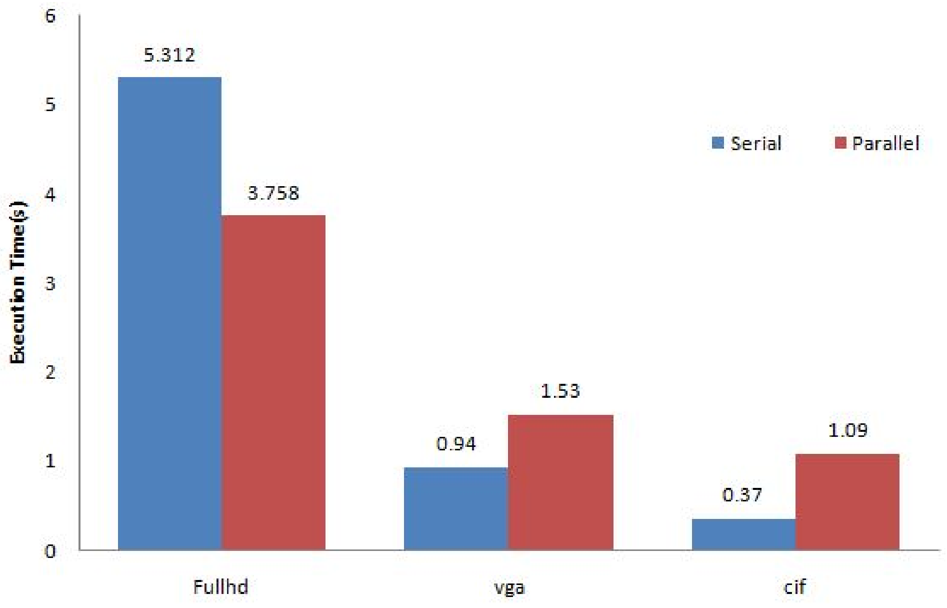


Figure 3.5: Execution time for different Datasets of Tracking

3.3 Scale Invariant Feature Transform(SIFT)

In the previous section we have discussed about Feature Tracking which is data intensive application means it is coded with repetitive low-intensive arithmetic operations across a very fine level data granularity. In this section we discuss about Scale Invariant Feature Transform(SIFT) which is, unlike feature tracking,compute intensive application. Means this has complex mathematical operations. In SIFT we extract invariant features from distorted images.

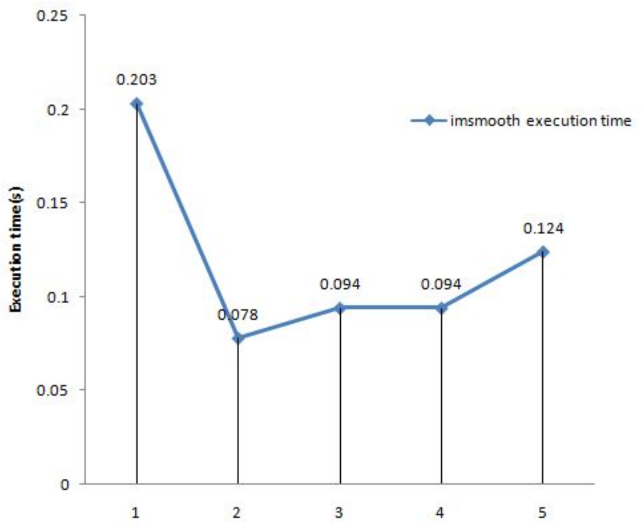


Figure 3.6: Smooth function Execution time Behavior

In SIFT by code inspection we found that *imsmooth()* is called numerous times. By parallelizing it we have significantly improved the performance. As discussed earlier the first execution takes time and rest of execution becomes faster.This behavior is explained in figure 3.6.

In the first phase of parallelization we converted the *imsmooth()* function to the CUDA kernel. This conversion gave us tremendous improvement in performance. While looking for further possible optimizations we found that the kernel can be split into two as the later portion of the kernel needed to wait long for the preceding portion to get its work done. However, the *synchThread()* was doing the job, we splited the kernel and that gave us further reduction in execution time.Following code we have shown how we have split the kernel into two.

```
 smoothCUDAKernel(F2D* buffer_d,float *temp_d,F2D* myarray_d,F2D* out_d,int W)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
int idy = blockIdx.y * blockDim.y + threadIdx.y;
    \\ first for loop
    synchronizeThreads();
    \\ Second for loop
}
function smoothImage(F2D* buffer,float *temp,F2D* myarray,F2D* out,int W){
 \\ Memory Allocation
     smoothCUDAKernel <<<dimgrid,dimBlock >>>(buffer_d,temp_d,myarray_d,out_d,W
 \\ return
 }
```

We split the kernel into two for two for loops in the kernel as shown in code above.

```
smoothCUDAKernelPart1(F2D* buffer_d,float *temp_d,F2D* myarray_d,int W)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
int idy = blockIdx.y * blockDim.y + threadIdx.y;
    \\ first for loop

}
smoothCUDAKernelPart2(F2D* buffer_d,float *temp_d,F2D* myarray_d,F2D* out_d,int
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
int idy = blockIdx.y * blockDim.y + threadIdx.y;

    \\ Second for loop
}
function smoothImage(F2D* buffer,float *temp,F2D* myarray,F2D* out,int W){
 \\ Memory Allocation
     smoothCUDAKernelPart1 <<<dimgrid,dimBlock >>>(buffer_d,temp_d,myarray_d,W)

     smoothCUDAKernelPart2 <<<dimgrid,dimBlock >>>(buffer_d,temp_d,myarray_d,ou
 \\ return
 }
```

Splitting the loop into two makes the execution marginally faster.Readings for the same are shown in figure 3.7.

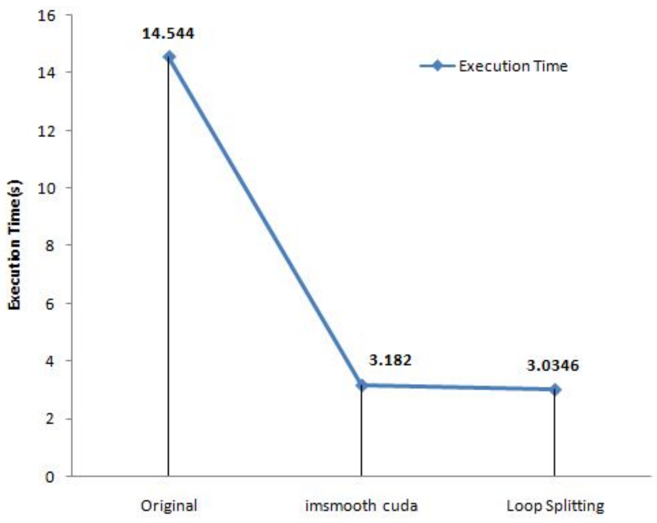


Figure 3.7: SIFT Phase wise Execution Time

3.4 Disparity Map

Disparity map is data intensive application.As we have discussed in previous section it finds the depth information from pair of images taken form different angle. We have identified two functions which we can convert into CUDA kernels.

- finalSAD
- findDisparity

findDisparity() function is the core function which calculates disparity. These functions do not have complex mathematical computation. Phase wise execution time reduction is shown in the figure 3.8 below Considering all the three problem

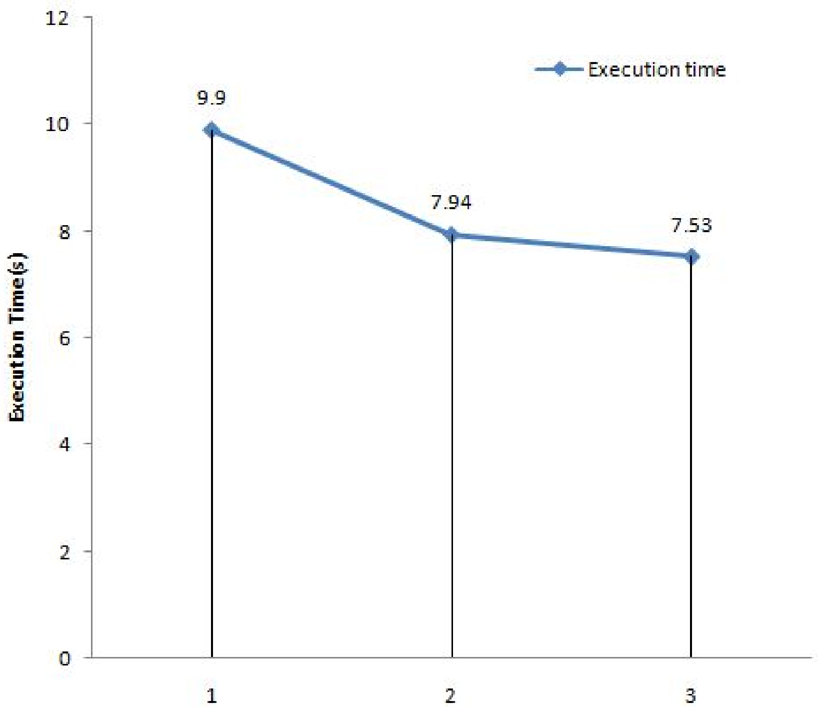


Figure 3.8: Disparity Map Phase wise Execution Time

size, optimization is best for the fullhd dataset. It gives the maximum speed up. Other dataset results in the higher execution time than the sequential one. Readings for the same are shown in figure 3.9 below.

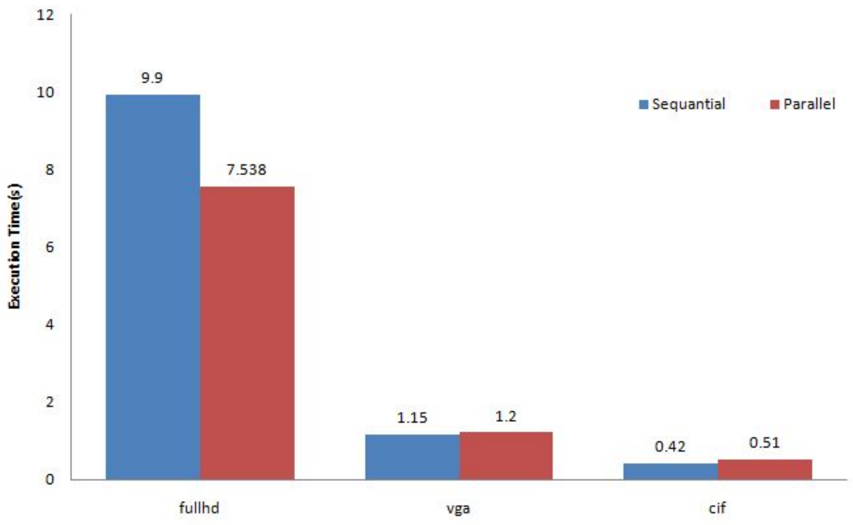


Figure 3.9: Execution time for Different Datasets of Disparity Map

3.5 Overall Analysis

From our experimental results we can calculate speed up for all of the three applications. Speedup for all three shown in table 3.1 SIFT applications results in

Application	Serial	Parallel	Speedup
Feature Tracking	5.312	3.785	1.413
SIFT	14.544	3.034	4.792
Disparity Map	9.9	7.538	1.313

Table 3.1: Performance Improvement in SD-VBS applications

maximum speedup while Feature tracking and Disparity map has nearly similar speedup. The reason behind that is *imsmooth()* function is executed numerous times which results in the lesser and lesser execution time. Results are shown graphically in figure 3.10.

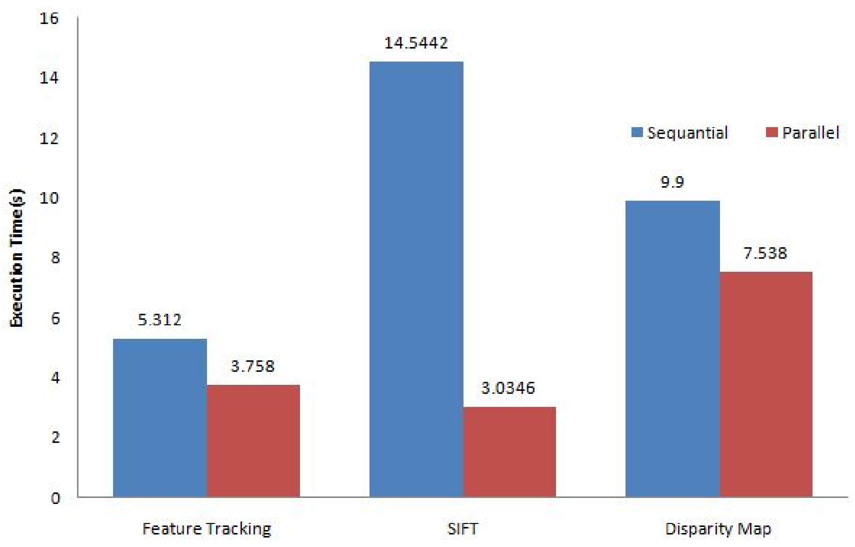


Figure 3.10: Execution time for all three applications in SD-VBS

Chapter 4

Conclusion and future work

4.1 Conclusion

The approaches suggested for optimization are described in this thesis. Its experimental results shows that there is significant performance improvement as expected. However, when problem is smaller CPU can beat GPU because of the overhead associated with the data transfer on GPU and kernel execution. Another usual phenomenon which has been observed is that the memory transfer overhead is found only for the first time when the kernel is called. For the rest of the executions there was no memory transfer overhead. Highest speedup is achieved for SIFT application compared to Feature Tracking and Disparity Map. Reason behind this is that the parallel function in SIFT is called number of times. More the number of times the parallel function is called higher will be the speedup.

4.2 Future Work

Our work can be used as a test suite to validate efficiency of automatic parallelization tools. The tools which convert the sequential code to a parallel one.

References

[1] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor, ”SD-VBS: The San Diego Vision Benchmark Suite”, IEEE Interational Symposium on Workload Characterization, October 2009.

[2] B. D. Lucas and T. Kanade, An Iterative Image Registration Technique with an Application to Stereo Vision, in Proceedings of Imaging Understanding Workshop, 1981.

[3] D. G. Lowe, Distinctive Image Features from Scale-Invariant Keypoints, in International Journal for Computer Vision, 2004.

[4] P. Chang, D. Hirvonen, T. Camus, and B. Southall, Stereo-Based Object Detection, Classification, and Quantitative Evaluation with Automotive Applications, in IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 2005.

[5] John Nickolls, Ian Buck & Michel Garland, NVIDIA, Kevin Skadron, University of verginia,”Scalable parallel Programming with CUDA”,March/April 2008, ACM QUEUE

[6] D. C. Clarissa Tacchella, ”Nvidia cuda compute unified device architecture”,nomatr - 707827 , 708250.

[7] D. L. N. Research,”nvidia gpu architecture and implications”, NVIDIA Corporation 2007.

[8] CUDA C Programming Guide version 4.

[9] http://http.developer.nvidia.com/ParallelNsight/2.1/Documentation/UserGuide/HTML/Parallel_Nsight_User_Guide.htm

[10] http://parallelnsight.nvidia.com/

[11] http://developer.nvidia.com/cuda-toolkit-40

[12] www-bcf.usc.edu/~jbarbic/multi-core-15213-sp07.ppt

Appendix A

Installation of CUDA

A.1 CUDA 4.0 with Parallel Nsight

A.1.1 Installation

1. Obtain CUDA toolkit from [11] and install it in the default directory
2. Obtain CUDA developer drivers from [11] for your operating system and install it
3. Obtain GPU Computing SDK from [11] for your operating system and install it.
4. Register for Parallel Nsight and download suitable Parallel Nsight for your operating system and machine.
5. For CUDA 3.2 install visual studio 8 with sp1 and for CUDA 4 install visual studio 2010.
6. For the list of minimum requirement for Parallel Nsight visit [9]

A.1.2 Creating and Executing Visual studio project

1. Click on new Project.
2. In Installed Templates - NVIDIA - CUDA - CUDA 4.0 Runtime will create new CUDA project ready to execute.
3. kernel.cu file will have sample program of CUDA which you can replace

4. Then you can run it like a simple visual studio project by pressing F5 or from menu.

A.1.3 Debugging in Parallel Nsight

1. Installing Parallel Nsight will give you Nsight menu in visual studio menu bar.
2. First you have to start Nsight Monitor.
3. then go to Nsight - start cuda debugging.

A.2 CUDA 2.3 in emulation mode

Installation of CUDA 2.3 is done the same way CUDA 4.0 is installed we just have to install visual studio 2008. and we are not going to install Parallel Nsight.

A.2.1 Creating and Executing Visual studio project

1. create Empty win32 project
2. do

   ```
   Configuration Properties >> Custom Build Step >> General:
   Command Line = $(CUDA_BIN_PATH)\nvcc.exe -ccbin
   $(VCInstallDir)bin -c -D_DEBUG -DWIN32 -D_CONSOLE -D_MBCS
   -Xcompiler /EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MT -I$(CUDA_INC_PATH)
   -I./ -o $(ConfigurationName)\example1.obj example1.cu
   ```

3. do

   ```
   Configuration Properties >> C/C++ >> Code Generation:
   Runtime Library = Multi-threaded (/MT)
   ```

4. do

   ```
   Configuration Properties >> Linker >> Input:
   Additional Dependencies = cudart.lib cutil32.lib
   ```

5. Your project is ready to execute and debug.