Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA

Effective techniques to process complex image data in real time using GPUs

By Bhaumik Vaidya

Packt›
www.packt.com

Table of Contents

Preface

1. Introduction to CUDA and Getting Started with CUDA
    - Introduction Technical Requirements
    - Introduction to CUDA
        - Need of Parallel Processing
        - Introduction to GPU architecture and need of CUDA
        - CUDA Architecture
    - Applications of CUDA
    - CUDA Development Environment
        - CUDA supported GPU
        - NVIDIA graphics card driver
        - Standard C compiler CUDA Development Kit
    - Installation of CUDA toolkit on all Operating Systems
        - Windows
        - Linux
        - Mac
    - Basic print Hello, CUDA! program in CUDA C
        - Steps on Windows
        - Steps on Ubuntu
    - Summary
    - Questions
2. Parallel programming using CUDA C
    - Introduction
    - Technical Requirements
    - CUDA program structure
        - Two-variable addition program in CUDA C
        - A Kernel Call
        - Configuration of kernel parameters CUDA API functions
        - Passing parameters to CUDA functions
            - Passing parameter by value
            - Passing parameter by reference
    - Thread execution on a device
    - Accessing GPU device properties from CUDA programs
        - General Device Properties:
        - Memory Related Properties
        - Thread Related Properties
    - Vector Operation in CUDA
        - Two-vector addition Program
        - Comparison of throughput between CPU and GPU code
        - Elementwise squaring of Vector in CUDA

Parallel Communication Patterns
Map
Gather
Scatter
Stencil
Transpose
Summary
Questions
3. Threads, Synchronization and Memory
Introduction Technical
Requirements Threads
Memory Architecture
Global Memory
Local Memory and Registers
Cache Memory
Thread Synchronization
Shared Memory
Atomic Operations
Constant Memory
Texture Memory
Dot Product and Matrix Multiplication Example
Dot Product
Matrix Multiplication
Summary
Questions
4. Advanced concepts in CUDA
Introduction
Technical Requirements
Performance measurement of CUDA Programs
CUDA Events
Nvidia Visual Profiler
Error handling in CUDA
Error handling from within the code
Debugging tools
Performance improvement of CUDA Programs
Using an optimum number of blocks and threads
Maximizing Arithmetic Efficiency
Using Coalesce or strided memory access
Avoiding Thread Divergence
Using Page-locked Host Memory
CUDA Streams
Using Multiple CUDA streams
Acceleration of sorting algorithms using CUDA
Enumeration or Rank Sort Algorithm Image
Processing using CUDA
Histogram Calculation on GPU using CUDA
Summary

Questions
5. Getting started with OpenCV with CUDA support
Introduction
Technical Requirements
Introduction to Image Processing and Computer Vision
Introduction to OpenCV
Installation of OpenCV with CUDA support
Installation of OpenCV on Windows
Using pre-built binaries
Building libraries from Source
Installation of OpenCV with CUDA support on Linux
Working with Images in OpenCV
Image representation inside OpenCV
Read and Display an Image
Read and Display Color Image
Creating Images using OpenCV
Drawing shapes on the blank image
Draw a Line
Draw a rectangle
Draw a circle
Draw an ellipse
Writing text on Image
Saving Image to a file Working with videos in OpenCV
Working with video stored in Computer
Working with videos from webcam
Saving video to a disk
Basic Computer Vision Applications using OpenCV CUDA module
Introduction to OpenCV CUDA module
 Arithmetic and Logical Operations on images
Addition of Two images
Subtracting Two images
Image Blending Image Inversion
Changing Color-Space of an Image
Image Thresholding
Performance comparison of OpenCV applications with and without CUDA support
Summary
Questions
6. Basic computer vision Operations using OpenCV and CUDA
Introduction
Technical Requirements
Accessing Individual Pixel Intensities of an Image
Histogram calculation and Equalization in OpenCV
Histogram Equalization
Grayscale Images
Color Image
Geometric Transformation on Images

        - Image Resizing
        - Image Translation and Rotation
    - Filtering Operation on Images
        - Convolution Operation on Image
        - Low Pass filtering on Image
            - Averaging Filter
            - Gaussian Filter
            - Median Filtering
        - High Pass Filtering on Image
            - Sobel Filter
            - Scharr Filter
            - Laplacian Filter
    - Morphological operations on Images
    - Summary
    - Questions
7. Object detection and tracking using OpenCV and CUDA
8. Introduction to Jetson Tx1 development board and installing OpenCV on Jetson TX1
9. Deploying computer vision applications on Jetson TX1
10. Getting started with PyCUDA
11. Working with PyCUDA
12. Basic Computer vision application using PyCUDA

Preface

CHAPTER 1, Introduction to CUDA and Getting Started with CUDA will introduce you to CUDA architecture and how it has redefined parallel processing capabilities of GPUs.

CHAPTER 2, Parallel programming using CUDA C will teach you to write programs using CUDA for GPUs.

CHAPTER 3, Threads,Synchronization and Memory will teach you how threads are called from CUDA programs and how multiple threads communicate with each other. It describes how multiple threads are synchronized when they work in parallel.

CHAPTER 4, Advanced concepts in CUDA teaches how CUDA stream works and how to use it to accelerate applications. It describes how different searching and sorting algorithms can be accelerated using CUDA.

CHAPTER 5, Getting started with OpenCV with CUDA support revises the concepts of using OpenCV in C++. This chapter describes installation of OpenCV library with CUDA support in all Operating systems.

CHAPTER 6, Basic computer vision Operations using OpenCV and CUDA teaches the reader to write basic computer vision operations using OpenCV.

CHAPTER 7, Object detection and tracking using OpenCV and CUDA teaches the reader to accelerate some real life computer vision application using OpenCV and CUDA.

CHAPTER 8, Introduction to Jetson Tx1 development board and installing OpenCV on Jetson TX1 introduces Jetson TX1 embedded platform and how it can be used to accelerate and deploy computer vision applications. It describes installation of OpenCV for Tegra on Jetson TX1 with Jetpack.

CHAPTER 9:Deploying computer vision applications on Jetson TX1 describes deploying computer vision applications on jetson Tx1.

CHAPTER 10:Getting started with PyCUDA introduces PyCUDA which is a python library for GPU acceleration. It describes installation procedure on all operating systems. It also teaches how to write and run basic program in PyCUDA.

CHAPTER 11:Working with PyCUDA teaches reader to write advance programs using PyCUDA.

CHAPTER 12:Basic Computer vision application using PyCUDA teaches development and acceleration of basic computer vision applications using PyCUDA. It describes color space conversion operation, histogram calculation and different filtering operations as example of computer vision applications.

Introduction to CUDA and Getting Started with CUDA

Introduction

This chapter gives a brief introduction to CUDA architecture and how it has redefined parallel processing capabilities of GPUs. The application of CUDA architecture in real life scenarios is shown. This chapter will serve as a starting guide for software developers who want to accelerate their application by using General purpose GPUs and CUDA. The chapter describes development environments used for CUDA application development and how CUDA toolkit can be installed on all operating systems. It covers how a basic code can be developed using CUDA C and executed on windows and Ubuntu operating systems.

The following topics will be covered in this chapter:

- Introduction to CUDA
- Applications of CUDA
- CUDA development environments
- Installation of CUDA toolkit on windows, linux and Mac OS
- Development of simple code using CUDA C

Technical Requirements

This chapter requires familiarity with the basic C or C++ programming language. All the code used in this chapter can be downloaded from following github link: https://github.com/bhaumik2450/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA /Chapter1. The code can be executed on any operating system though it is only tested on Windows 10 and Ubuntu 16.04.

Introduction to CUDA

Compute Unified Device Architecture (**CUDA**) is a very popular parallel computing platform and programming model developed by NVIDIA. It is only supported on NVIDIA GPUs. OpenCL is used to write parallel code for other types of GPUs like AMD and intel but it is more complex than CUDA. CUDA allows creating massively parallel applications running on GPUs with simple programming APIs. Software developers using C and C++ can accelerate their software application and leverage the power of GPUs by using CUDA C or C++. Programs written in CUDA are similar to programs written in simple C/C++ with addition of keywords needed to exploit parallelism of GPUs. CUDA allows programmer to specify which part of CUDA code will execute on CPU and which part will execute on GPU.

Next topic describes the need of parallel computing and how CUDA architecture can leverage the power of GPU in detail.

Need of Parallel Processing

In recent years, consumers have been demanding more and more functionalities on a single hand held device. So there is a need for packaging more and more transistors on a small area that can work fast and consume minimum power. We need a fast processor that can carry out multiple tasks with high clock speed, small area and minimum power consumption. Over many decades, transistor sizing has seen a gradual decrease resulting in the possibility of more and more transistors being packed on a single chip. This has resulted in a constant rise of the clock speed. However, this situation has changed in the last few years with clock speed being more or less constant. So what is the reason for this? Have transistors stopped getting smaller? The answer is no. The main reason behind clock speed being constant is high power dissipation with high clock rate. Small transistors packed in small area and working at high speed will dissipate large power, and hence it is very difficult to keep the processor cool. As clock speed is getting saturated in terms of development, we needed a new computing paradigm to increase performance of processors. Let us understand this concept by taking a small real life example.

Suppose you are told to dig a very big hole in small amount of time. You will have the following three options to complete this work in time:

1. You can dig faster
2. You can buy more productive shovel
3. You can hire more diggers who can help you in completing this work

If we can draw a parallel between this example and computing paradigm, then first option is similar to having faster clock. Second option is similar to having more transistors that can do more work per clock cycle. But as we have discussed in the previous paragraph, power constrains have put limitations on these two steps. Third option is similar to having many smaller and simpler processors that can carry out tasks in parallel. Graphics processing Unit (GPU) follows this computing paradigm. Instead of having big powerful processor that can do complex jobs, it has many small and simple processors that can get work done in parallel. The details of GPU architecture is explained in next section.

Introduction to GPU architecture and need of CUDA

GeForce 256 was a first GPU developed by NVIDIA in 1999. Initially GPUs were only used for rendering high end graphics on monitors. They were only used for pixel computations. Later on people realized that if GPUs can do pixel computations then it will also be able to do other mathematical calculations. Nowadays, GPUs are used in many applications other than those rendering graphics. These kinds of GPUs are called general purpose GPUs (GPGPUs).

Next question to come to your mind is what is the difference between hardware architecture of CPU and GPU that allows GPU to carry out parallel computation. CPU has a complex control hardware and less data computation hardware. Complex control hardware gives CPU flexibility in performance and simple programming interface but it is expensive in terms of power. On the other hand, GPU has simple control hardware and more hardware for data computation that gives it ability for parallel computation. This structure of GPU makes it more power efficient. The disadvantage is that it has more restrictive programming model. In the early days of GPU computing, graphics API like OpenGL and DirectX were the only way to interact with GPUs. This was a complex task for normal programmers who were not familiar with OpenGL or DirectX. This lead to the development of CUDA programming architecture, which provided easy and efficient way of interaction with GPUs. More details about CUDA architecture are given in the next section.

Normally, performance of any hardware architecture is measured in terms of latency and throughput. Latency is time taken to complete a given task while throughput is amount of task completed in a given time. These are not contradictory concepts. More often than not improving one improves the others. In a way, most hardware architectures are designed to improve either latency or throughput. Take an example of you standing in a queue of a post office. Your goal is to complete your work in less amount of time, so you want to improve latency, while an employee sitting at a post office window wants to entertain more and more customers in a day. So, the employee's goal is to increase the throughput. Improving one will lead to an improvement in the other in this case, but the way both sides look at this improvement is different.

In the same way, normal sequential CPUs are designed to optimize latency, while GPUs are designed to optimize throughput. CPUs are designed to execute all instructions in minimum time, while GPUs are designed to execute more instructions in a given time. This design concept of GPU makes it very useful in image processing and computer vision applications, which we are targeting in this book because we don't mind a delay in processing of a single pixel. What we want is that more pixels should be processed in a given time, which can be done on GPU.

So to summarize, parallel computing is the need of the hour if we want to increase computational performance at same clock speed and power requirement. GPUs provide this capability by having lots of simple computational units working in parallel. Now to interact with GPU and take advantage of it's parallel computing capabilities, we need a simple parallel programming architecture which is provided by CUDA.

CUDA Architecture

This section covers basic hardware modifications done in GPU CUDA architecture and what is the general structure of software programs developed using CUDA. We will not discuss about syntax of CUDA program just yet but we will cover steps to develop the code. The section will also cover some basic terminology that will be followed throughout this book.

CUDA architecture includes several new components specifically designed for general purpose computations in GPUs, which were not present in earlier architectures. It includes unified shedder pipeline that allows all Arithmetic and logical Units (ALUs) present on a GPU chip to be marshalled by a single CUDA program. The ALUs are also designed to comply with IEEE floating point single and double precision standards so that it can be used in general purpose applications. The instruction set is also tailored at general purpose computation and not specific to pixel computations. It also allows arbitrary read and write access to memory. These features make CUDA GPU architecture very useful in general purpose applications.

All GPUs have many parallel processing units called as cores. On hardware side, these cores are divided into streaming processors and streaming multiprocessors (SMs). GPU has a grid of these streaming multiprocessors. On software sides, a CUDA program is executed as series of multiple threads running in parallel. Each thread is executed on a different core. The GPU can be viewed as a combination of many blocks, and each block can execute many threads. Each block is bound to a different SM on GPU. How mapping is done between a block and SM is not known to a CUDA programmer but it is known and done by a scheduler. The threads from same block can communicate with each other. GPU has a hierarchical memory structure that deals with communication between threads inside one block and multiple blocks. This will be dealt in detail in the upcoming chapters.

As a programmer, you will be curious to know what will be the programming model in CUDA and how will the code understand whether it should be executed on CPU or GPU. For this book, we will assume that we have a computing platform comprising a CPU and a GPU. We will call CPU and its memory as host and GPU and its memory as device. A CUDA code contains the code for both, the host and the device. The host code is executed on CPU by normal C/C++ compiler, and the device code is executed on GPU by GPU compiler. Host code calls the device code by something called as 'kernel' call. It will launch many threads in parallel on a device. The count of how many threads to be launched on device will be provided by the programmer.

Now, you might ask how is this device code different than a normal C code. The answer is that it is similar to normal sequential C code. It is just that this code is executed on more number of cores in parallel. However, for this code to work, it needs data on device memory. So before launching threads, the host copies data from host memory to device memory. Thread works on data from device memory and stores the result on device memory. Finally, these data are copied back on host memory for further processing. To summarize, steps to develop CUDA C program are as follows:

- Allocate memory for data in host and device memory.
- Copy data from host memory to device memory.

- launch kernel by specifying degree of parallelism.
- After all threads are finished, copy data back from device memory to host memory.
- Free up all memory used on host and device.

Applications of CUDA

CUDA has seen unprecedented growth in the last decade. It is being used in wide variety of applications in various domains. It has transformed research in multiple fields. In this section, we will look at some of these domains and how CUDA is accelerating growth in that domain:

- **Computer Vision applications**

  Computer vision and image processing algorithms are computationally intensive. With more and more cameras capturing images at high definition, there is a need to process these large images in real time. With CUDA acceleration of these algorithms, applications like image segmentation, object detection and classification can achieve real-time frame rate performance of more than 30 frames per second. CUDA and GPU allows faster training of deep neural networks and other deep learning algorithms; this has transformed research in computer vision. NVIDIA is developing several hardware platforms like Jetson TX1, Jetson TX2, and Jetson TK1 which can accelerate computer vision applications. NVIDIA drive platform is also one of the platforms that is made for autonomous drive applications.

- **Medical Imaging**

  Medical imaging field is seeing widespread use of GPUs and CUDA in reconstruction and processing of MRI images and computed tomography (CT) images. It has drastically reduced processing time for these images. Nowadays, there are several devices that are shipped with GPUs, and several libraries are available to process these images with CUDA acceleration.

- **Financial computing**

  There is a need for better data analytics at a lower cost in all financial firms, and this will help in informed decision making. It includes complex risk calculation and initial and lifetime margin calculation which have to be done in real time. GPUs help financial firms to do these kinds of analytics in real time without adding too much cost overhead.

- **Life Science, bioinformatics and computational Chemistry**

  Simulating DNA genes, sequencing, protein docking are computationally intensive tasks that need high computation resource. GPUs help in this kind of analysis and simulation. GPUs can run common molecular dynamics, quantum chemistry and protein docking applications more than 5 times faster than normal CPUs.

- **Weather research and forecasting**

  Several weather prediction applications, ocean modeling techniques and tsunami prediction techniques utilize GPU and CUDA for faster computation and simulations

compared to CPUs.

- **Electronics Design Automation (EDA)**

  Due to increasing complexity in VLSI technology and semiconductor fabrication process, the performance of EDA tools is lagging behind in this technological progress. It leads to incomplete simulations and missed functional bugs. Therefore, EDA industry has been seeking faster simulation solutions. GPU and CUDA acceleration is helping this industry to speed up computationally intensive EDA simulations including functional simulation, placement and routing, Signal Integrity & Electromagnetics, SPICE circuit simulation etc.

- **Government and Defence**

  GPU and CUDA acceleration is also widely used by government and military. The aerospace, defense and intelligence industries are taking advantage of CUDA acceleration in converting large amount of data in to actionable information.

CUDA Development Environment

To start developing application using CUDA, you will need to setup development environment for it. There are some prerequisites for setting up development environment for CUDA. These include the following:

- CUDA supported GPU
- NVIDIA graphics card driver
- Standard C compiler
- CUDA development kit

How to check for this prerequisites and install them is discussed one by one in the following sub-section.

CUDA supported GPU

As discussed earlier, CUDA architecture is only supported on NVIDIA GPUs. It is not supported on other GPUs like AMD and intel. Almost all GPUs developed by NVIDIA in the last decade support CUDA architecture and can be used to develop and execute CUDA applications. The detailed list of CUDA supported GPUs can be found on NVIDIA website: https://developer.nvidia.com/cuda-gpus. If you can find your GPU featuring in this list, you will be able to run CUDA applications on your PC. If you don't know which GPU is there on your PC, then you can find it by following these steps:

On windows:

- In start menu, type device manager and press enter.
- In device manager, expand display adaptors.

  In that you will find the name of your NVIDIA GPU.

On Linux:

- Open Terminal
- Run `sudo lshw -C video`

  This will list information regarding your graphics card, usually including its make and model.

On Mac:

- Go to Apple Menu | About this Mac | More info
- Select Graphics/Displays under Contents list

  In that you will find name of your NVIDIA GPU.

If you have CUDA enabled GPU, then you are good to proceed to the next step.

NVIDIA graphics card driver

If you want to communicate with NVIDIA GPU hardware, then you will need a system software for it. NVIDIA provides device driver to communicate with GPU hardware. If NVIDIA graphics card is properly installed, then these drivers are installed automatically with it on your PC. Still, it is good practice to check for driver updates periodically from the NVIDIA website: http://www.nvidia.in/Download/index.aspx?lang=en-in. You can select your graphics card and operating system for driver download from this link.

Standard C compiler

Whenever you are running a CUDA application, it will need two compilers: one for GPU code and other for CPU code. The compiler for GPU code will come with installation of CUDA toolkit, which will be discussed in next section. You also need to install standard C compiler for executing CPU code. There are different C compilers based on operating systems:

On Windows:

For all Microsoft windows editions, it is recommended to use Microsoft visual studio C compiler. It comes with Microsoft visual studio and can be downloaded from its official website: https://www.visualstudio.com/downloads/

The express edition for commercial applications needs to be purchased but you can use community editions for free in non-commercial applications. For running the CUDA application, install Microsoft visual studio with Microsoft visual studio C compiler selected. Different CUDA version supports different visual studio editions so you can refer NVIDIA CUDA website for visual studio version support.

On Linux:

Mostly, all Linux distributions come with standard GNU C Complier (gcc), and hence it can be used to compile CPU code for CUDA applications.

On Mac:

On Mac operating system, you can install gcc compliler by downloading and installing Xcode for Mac OS. It is freely available and can be downloaded from Apple's website: https://developer.apple.com/xcode/

CUDA Development Kit

CUDA needs GPU compiler for compiling GPU code. This compiler comes with CUDA development toolkit. If you have NVIDIA GPU with latest driver update and installed standard C compiler for your operating system, you are good to proceed for the final step of installing CUDA development toolkit. Step by step guide for installing CUDA toolkit is discussed in the next section.

Installation of CUDA toolkit on all Operating Systems

This section covers instructions to install CUDA on all supported platforms. It also describes steps to verify installation. While installing CUDA, you can chose between network installer and offline local installer. Network installer has less initial download size but it needs internet connection while installation. Local offline installer has more initial download size. The steps discussed in the book are for local installation. CUDA toolkit can be downloaded for Windows, Linux and MAC OS for both 32 bit and 64 bit architecture from the following link: https://developer.nvidia.com/cuda-downloads.

After downloading installer, refer to the following steps for your particular operating system.

Windows

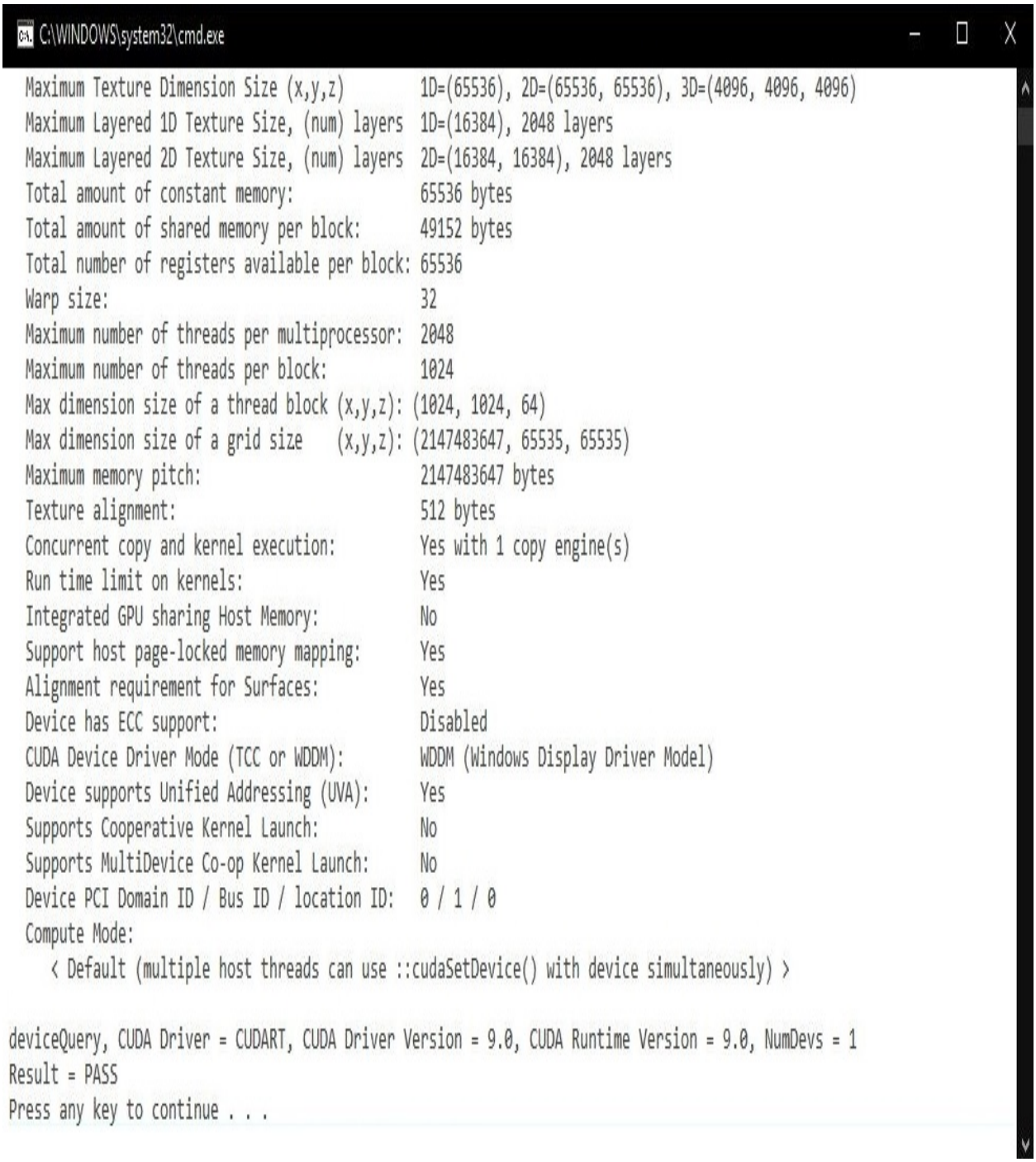
This section covers steps to install CUDA on Windows OS, which are as follows:

- Double click on installer. It will ask to select folder in which temporary installation files will be extracted. Select the folder of your choice. It is recommended to keep the default.
- Then, installer will check for system compatibility. If system is compatible, you can follow the on-screen prompt to install CUDA. You can chose between express installation (default) and custom installation. Custom installation allows to choose which features of CUDA to install. It is recommended to select express default installation.
- Installer will also install CUDA sample programs and CUDA visual studio integration.

Please make sure you have visual studio installed before running this installer

To confirm that the installation is successful, the following aspects should be ensured:

- All the CUDA samples will be located at C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0 if you have chosen default path for installation.
- To check installation, you can run any project.
- We are using devicequery project located at `C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\1_Utilities\deviceQuery`.
- Double click on `*.sln` file of your visual studio edition. It will open this project in visual studio.
- Then you can click on local windows debugger in visual studio. If the build is successful and following output is displayed, then installation is complete:


C:\WINDOWS\system32\cmd.exe
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
CUDA Device Driver Mode (TCC or WDDM): WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA): Yes
Supports Cooperative Kernel Launch: No
Supports MultiDevice Co-op Kernel Launch: No
Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
Press any key to continue . . .

Linux

This section covers the steps to install CUDA on Linux distributions. In this section, installation of CUDA in Ubuntu, which is popular Linux distribution, is discussed using distribution-specific packages or using apt-get command (which is specific to Ubuntu).

Steps to install CUDA using `*.deb` installer downloaded from CUDA website are as follows:

- Open terminal and run dpkg command which is used to install packages in debian based systems:

```
sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
```

- Install CUDA public GPG key using the following command:

```
sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub
```

- Then update apt repository cache using the following command:

```
sudo apt-get update
```

- Then you can install CUDA using the following command:

```
sudo apt-get install cuda
```

- Include CUDA installation path in PATH environment variable using the following command:

If you have not installed CUDA at default locations, you need to change the path to point at your installation location.

```
export PATH=/usr/local/cuda-9.1/bin${PATH:+:${PATH}}
```

- Set LD_LIBRARY_PATH environment variable:

```
export LD_LIBRARY_PATH=/usr/local/cuda-9.1/lib64\
                         ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

OR
You can also install CUDA toolkit by using apt-get package manager available with Ubuntu OS. You can run following command in the terminal.

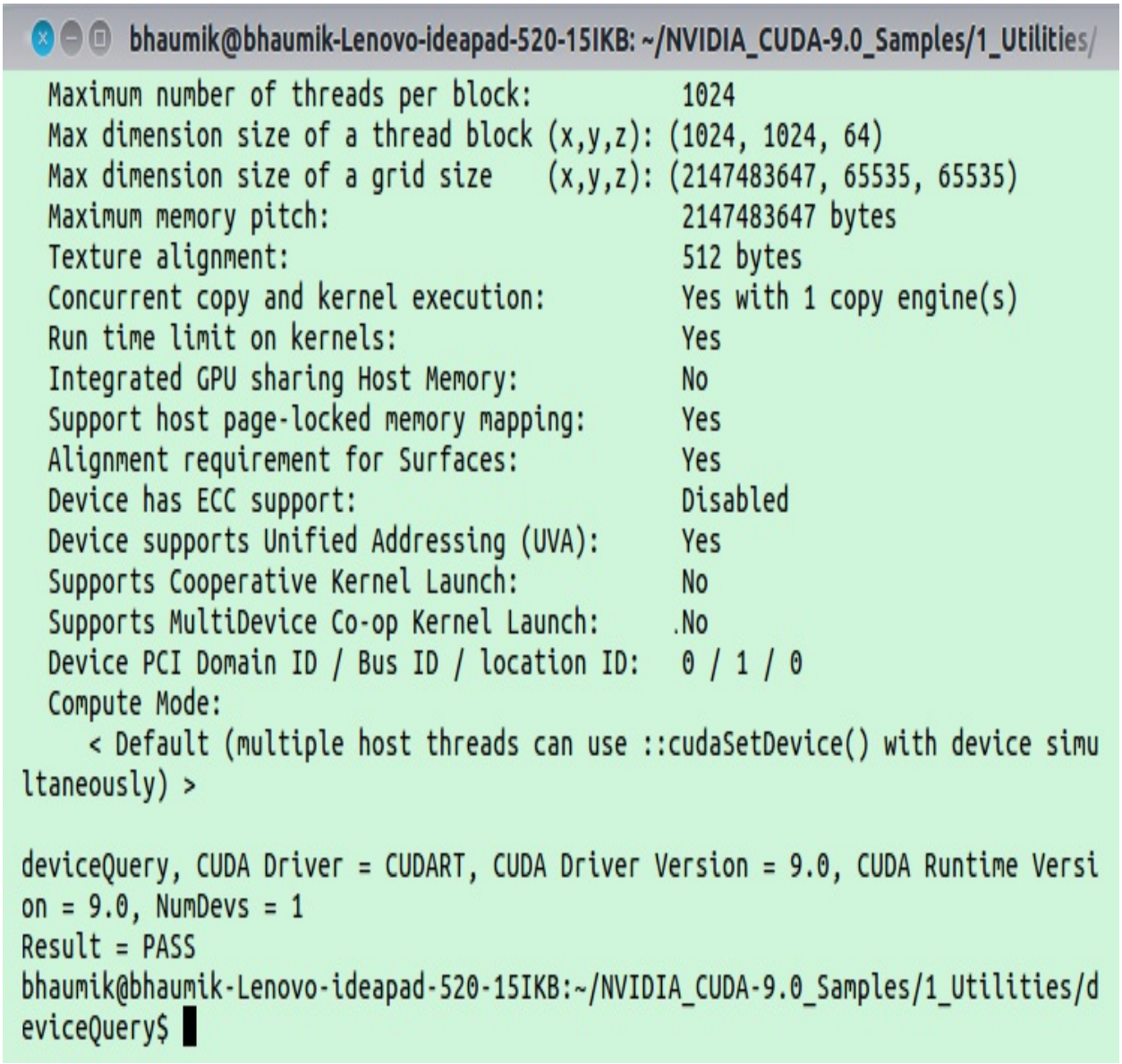
```
sudo apt-get install nvidia-cuda-toolkit
```

To check if CUDA GPU compiler has been installed or not you can run `nvcc -v` command from command prompt. It calls the gcc compiler for C code and the NVIDIA PTX compiler for the CUDA code.

You can install nvidia nsight eclipse plugin which will give GUI Integrated Developement Environment for executing CUDA programs using the following command:

```
sudo apt install nvidia-nsight
```

After installation, you can run deviceQuery project located at `~/NVIDIA_CUDA-9.1_Samples`. If the CUDA toolkit is installed and configured correctly, the output for deviceQuery should look similar to the following:



Mac

This section covers steps to install CUDA on Mac OS. It needs *.dmg installer downloaded from CUDA website. The steps to install after downloading the installer are as follows:

- Launch the installer and follow on screen prompt to complete the installation. It will install all prerequisites, CUDA toolkit and CUDA samples.
- Then, You need to set environment variables to point at CUDA installation using the following commands:

If you have not installed CUDA at default locations, you need to change the path to point at your installation location

```
export PATH=/Developer/NVIDIA/CUDA-9.1/bin${PATH:+:${PATH}}
export DYLD_LIBRARY_PATH=/Developer/NVIDIA/CUDA-9.1/lib\
                  ${DYLD_LIBRARY_PATH:+:${DYLD_LIBRARY_PATH}}
```

- Run the script: `cuda-install-samples-9.1.sh`It will install CUDA samples with write permissions.

- After compilation, You can go to `bin/x86_64/darwin/release` and run deviceQuery project. If the CUDA toolkit is installed and configured correctly, it will display your GPU device properties.

Basic print Hello, CUDA! program in CUDA C

In this section we will start learning CUDA programming by writing a very basic program using CUDA C. We will start by writing `Hello, CUDA!` program in CUDA C and execute it. Before going into the details of code, one thing that you should recall is that host code is executed by standard C compiler and device code is executed by NVIDIA GPU compiler. NVIDIA tool feeds host code to standard C compiler like visual studio for windows and gcc compiler for ubuntu and MAC OS for execution. It is also important to note that GPU compiler can run CUDA code without any device code. All CUDA code must be saved with `*.cu extension`.

Following is the code for `Hello, CUDA!`:

```
#include <iostream>
 __global__ void myfirstkernel(void) {
 }
int main(void) {
 myfirstkernel << <1, 1 >> >();
 printf("Hello, CUDA!\n");
 return 0;
 }
```

If you look closely at the code, it will look very similar to that of the simple `Hello, CUDA!` program written in C for CPU execution. The function of this code is also similar. It just prints `Hello, CUDA!` on terminal. So one question that should come in your mind is how this code is different and where is the role of CUDA C in this code. The answer to these questions can be given by closely looking at the code. It has two main differences compared to code written in simple C :

- An empty function called `myfirstkernel` with `__global__` prefix
- Call `myfirstkernel` function with `<< <1,1> >>`

`__global__` is a qualifier added by CUDA C to standard C. It tells the compiler that function definition that follows this qualifier should be complied to run on device rather than host. So in previous code, `myfirstkernel` will run on device instead of host, though in this code it is empty.

Now where will main function run? The nvcc compiler will feed this function to host C compiler as it is not decorated by global keyword, and hence main function will run on host.

Second difference in the code is the call to the empty function `myfirstkernel` with some angular brackets and numeric values. This is a CUDA C trick to call device code from host code. It is called as 'kernel' call. The detail of kernel call will be explained in later chapters. The values inside angular brackets indicate arguments we want to pass from host to device at run time. Basically, it indicates the number of blocks and the number of threads that will run in parallel on the device. So in this code `<< <1,1> >>` indicates that `myfirstkernel` will run on 1 block and 1 thread/block on device. Though this is not optimal use of device resources, it is a good starting point to understand the difference between code executed on host and code executed on device.

Again, to revisit and revise `Hello, CUDA!` code, `myfirstkernel` function will run on device with one

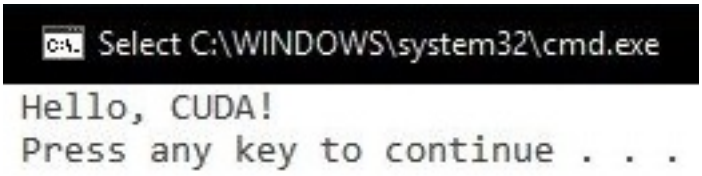
block and one thread/block. It will be launched from host code inside main function by a method called kernel launch.

After writing code, how you will execute this code and see the output? Next section describes the steps to write and execute `Hello, CUDA!` code on Windows and Ubuntu.

Steps on Windows

This section describes the steps to create and execute basic CUDA C program on Windows using Visual studio. The steps are as follows:

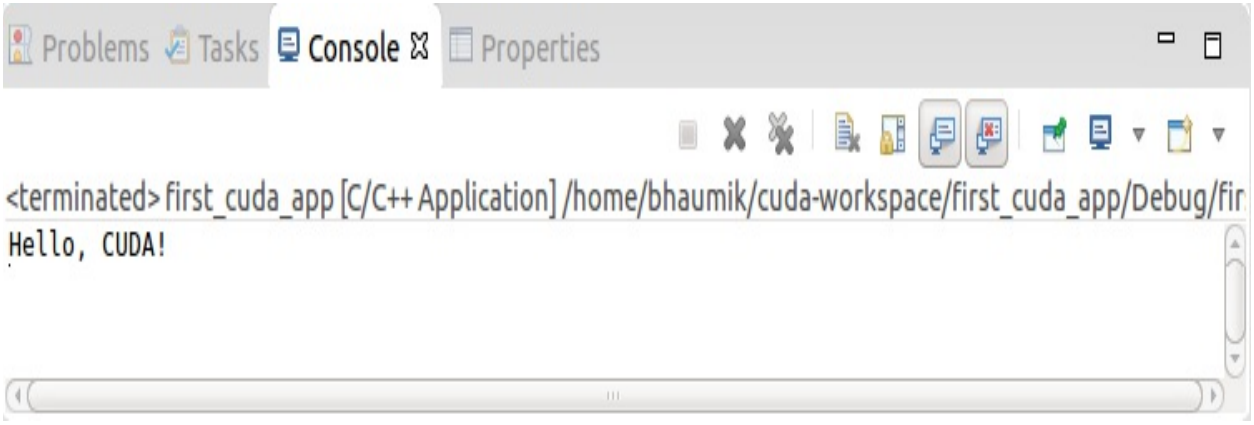
- Open Microsoft Visual Studio.
- Go to File → New → Project.
- Select NVIDIA → CUDA 9.0 → CUDA 9.0 Runtime.
- Give desired name to the project and click on OK.
- It will create a project with sample kernel.cu file. Now open this file by double clicking on it.
- Delete existing code from file and write the code given.
- Build the project from Build Tab and press Ctrl + F5 to debug the code. If everything works correctly, you will see `Hello, CUDA!` displayed on terminal as shown in figure below:



Steps on Ubuntu

This section describes the steps to create and execute basic CUDA C program on Ubuntu using nsight eclipse plugin. The steps are as follows:

- Open nsight by opening terminal and typing nsight on it.
- Go to File → New → Project
- Give desired name to the project and click on OK.
- It will create a project with sample file. Now open this file by double clicking on it.
- Delete existing code from file and write the code given.
- Run the code by pressing play button. if everything works correctly you will see Hello, CUDA! displayed on terminal as shown in figure below:



Summary

To summarize, in this chapter, you were introduced to CUDA and briefed upon the importance of parallel computing. Applications of CUDA and GPUs in various domains were discussed at length. The chapter described the hardware and software setup required to execute CUDA applications on your PCs. It gave a step by step procedure to install CUDA on local PCs. The last section gives a starting guide for application development in CUDA C by developing a simple program and executing it on Windows and Ubuntu. In the next chapter, we will build on this knowledge of programming in CUDA C. You will be introduced to parallel computing using CUDA C by taking several practical examples and how it is faster compared to normal programming. You will also be introduced to concepts of threads and blocks and how synchronization is done between multiple threads and blocks.

Questions

1. Explain three methods to increase the performance of computing hardware. Which method is used to develop GPUs ?
2. State True or False: Improving latency will improve throughput.
3. Fill in the blanks: CPUs are designed to improve ___ and GPUs are designed to improve __ .
4. Take an example of travelling from one place to other, which are 240 km apart. You can take a car which can accommodate 5 persons with 60 kmph speed and bus which can accommodate 40 persons with 40 kmph speed. Which option will provide better latency and which option will provide better throughput?
5. Explain the reasons which make GPU and CUDA particularly useful in Computer Vision applications.
6. State True or False: CUDA compiler cannot compile code with no device code.
7. In the "Hello, CUDA!" example discussed in chapter, printf statement will be executed by host or device?

Parallel programming using CUDA C

Introduction

In the last chapter, we have seen how easy it is to install CUDA and write a program in it. Tough the example was not impressive, it was shown to convince you that it is very easy to get started with CUDA. In this chapter, we will build upon this concept. It will teach you to write advance programs using CUDA for GPUs in detail. It starts with a variable addition program and then incrementally build towards complex vector manipulation examples in CUDA C. It also covers how kernel works and how to use device properties in CUDA programs. The chapter discusses how vectors are operated upon in CUDA programs and how CUDA can accelerate vector operations compared to CPU processing. It also discusses terminologies associated in CUDA Programming.

The following topics will be covered in this chapter:

- The concept of the kernel call
- Creating kernel functions and passing parameters to it in CUDA
- Configuration of kernel Parameters and memory allocation for CUDA programs
- Thread execution in CUDA programs
- Accessing GPU device properties from CUDA programs
- Working with vectors in CUDA programs
- Parallel communication patterns

Technical Requirements

This chapter requires familiarity with the basic C or C++ programming language, particularly dynamic memory allocation functions. All the code used in this chapter can be downloaded from following github link: https://github.com/PacktPublishing/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA. The code can be executed on any operating system though it is only tested on Windows 10 and Ubuntu 16.04.

CUDA program structure

We have seen a very simple "Hello, CUDA!" Program earlier which showcased some important concepts related to CUDA programs. A CUDA program is a combination of functions that are executed either on the host or on the GPU device. The functions that do not exhibit parallelism are executed on CPU, and the functions that exhibit data parallelism are executed on GPU. The GPU Compiler segregates these functions during compilation. As seen in the last chapter, functions meant for execution on the device are defined using the `__global__` keyword and compiled by nvcc compiler, while normal C host code is compiled by C compiler. A CUDA code is basically the same ANSI C code with the addition of some keywords needed for exploiting data parallelism.

So, in this section, a simple two-variable addition program is taken to explain important concepts related to CUDA programming, like kernel call, passing parameters to kernel functions from host to device, configuration of kernel parameters, CUDA APIs needed to exploit data parallelism, and how memory allocation takes place on host and device.

Two-variable addition program in CUDA C

In the simple "Hello, CUDA!" code seen in Chapter 1, the device function was empty. It had nothing to do. So, this section explains a simple addition program that performs addition of two variables on the device. Though it is not exploiting any data parallelism of the device, it is very useful for demonstrating important programming concepts of CUDA C. First, we will see how to write a kernel function for addition of two variables. The code for kernel function is shown below.

```
include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
//Definition of kernel function to add two variables
__global__ void gpuAdd(int d_a, int d_b, int *d_c)
{
 *d_c = d_a + d_b;
}
```

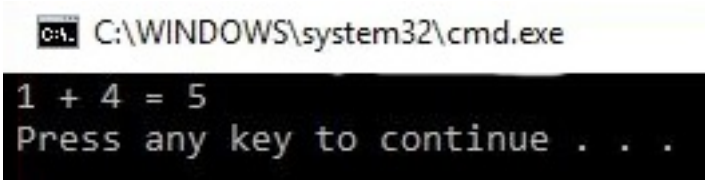
The `gpuAdd` function looks very similar to normal `add` function implemented in ANSI C. It takes two integer variables d_a and d_b as inputs and stores the addition at the memory location indicated by the third integer pointer d_c. The return value for device function is void because it is storing the answer in the memory location pointed by device pointer and not explicitly returning any value. Now we will see how to write main function for this code. The code for main function is shown below.

```
 int main(void)
{
 //Defining host variable to store answer
 int h_c;
 //Defining device pointer
 int *d_c;
 //Allocating memory for device pointer
 cudaMalloc((void**)&d_c, sizeof(int));
 //Kernel call by passing 1 and 4 as inputs and storing answer in d_c
 //<< <1,1> >> means 1 block is executed with 1 thread per block
 gpuAdd << <1, 1 >> > (1, 4, d_c);
 //Copy result from device memory to host memory
 cudaMemcpy(&h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
 printf("1 + 4 = %d\n", h_c);
 //Free up memory
 cudaFree(d_c);
 return 0;
}
```

In the `main` function, first two lines define variables for host and device. The third line allocates memory of the `d_c` variable on the device using the `cudaMalloc` function. The function `cudaMalloc` is similar to the `malloc` function in C. In the fourth line of main function, `gpuAdd` function is called with 1 and 4 as two input variables and `d_c`, which is a device memory pointer as output pointer variable. The weird syntax of `gpuAdd` function, which is also called kernel call, is explained in the next section. If the answer of `gpuAdd` function needs to be used on the host, then it must be copied from device memory to host memory, which is done by the `cudaMemcpy` function. Then, this answer is printed using the `printf` function. The second last line frees the memory used on the device by using the `cudafree` function. It is very important to free up all memory used on device explicitly from program, otherwise you might run out of memory at some point. The lines that start

with `//` are comments for more code readability, and these lines are ignored by compilers.

The two-variable addition program has two functions, `main` and `gpuAdd`. As you can see, `gpuAdd` is defined by using the `__global__` keyword, and hence it is meant for execution on device, while main function will be executed on host. The program adds two variables on the device and prints the output on command line as shown below:



We will use a convention in this book that host variables will be prefixed with `h_` and device variables will be prefixed with `d_`. The functions meant for execution on device will start with `gpu` prefix, and functions, apart from `main` function, meant for execution on host will start with `cpu` prefix. This is not compulsory; it is just done so that readers can understand the concepts easily without any confusion between host and device.

All CUDA APIs like cudaMalloc, cudaMemcpy, and cudaFree along with other important CUDA programming concepts like kernel call, passing parameters to kernels, and memory allocation issues are discussed in the further sections.

A Kernel Call

The device code that is written using ANSI C keywords along with CUDA extension keywords is called as kernel. It is launched from host code by a method called as **Kernel Call**. Basically, the meaning of kernel call is that we are launching device code from the host code. A kernel call typically generates a large number of blocks and threads to exploit data parallelism on GPU. Kernal code is very similar to normal C functions; it is just that this code is executed by several threads in parallel. It has a very weird syntax, which can be shown as follows:

```
kernel << <number of blocks, number of threads per block, size of shared memory > >> (parameters for
kernel)
```

It starts with the name of the kernel that we want to launch. You should make sure that this kernel is defined using `__global__` keyword. Then, it has `<< < > >>` kernel launch operator that contains configuration parameters for kernel. It can include three parameters separated by a comma. The first parameter indicates the number of blocks you want to execute, and the second parameter indicates the number of threads each block will have. So total number of threads started by a kernel launch will be the multiplication of these two numbers. The third parameter, which specifies the size of shared memory used by the kernel, is optional. In the program for variable addition, the kernel launch syntax is as follows:

```
gpuAdd << <1,1> >> (1 , 4, d_c)
```

Here, `gpuAdd` is the name of kernel that we want to launch, `<<<1,1>>>` indicates we want to start one block with one thread per block, which means that we are starting only one thread. Three arguments in round brackets are the parameters that are passed to kernel. Here, we are passing two constants 1 and 4. The third parameter is a pointer to device memory `d_c`. It points at the location on device memory where the kernel will store its answer after addition. One thing that the programmer has to keep in mind is that pointers passed as parameters to kernel should only point to device memory. If it is pointing to host memory, it can crash your program. After kernel execution is completed, the result pointed by device pointer can be copied back into host memory for further use. Starting only one thread for execution on device is not the optimal use of device resources. Suppose you want to start multiple threads in parallel, what is the modification that you have to make in the syntax of the kernel call? This is addressed in the next section and is termed as "configuration of kernel parameters"

Configuration of kernel parameters

For starting multiple threads on the device in parallel, we have to configure parameters in kernel call, which are written inside the kernel launch operator. They specify the number of blocks and number of threads per block. We can launch many blocks in parallel with many threads in each block. Normally, there is a limit of 512 or 1024 threads per block. Each block runs on the streaming multiprocessor, and threads in one block can communicate with each other via shared memory. The programmer can't choose which multiprocessor will execute a particular block and in which order blocks or threads will execute.

Suppose you want to start 500 threads in parallel; what is the modification that you can make to the kernel launch syntax that was shown previously? One option is to start 1 block of 500 threads via the following syntax:

```
gpuAdd<< <1,500> >> (1,4, d_c)
```

We can also start 500 blocks of 1 thread each or 2 blocks of 250 threads each. Accordingly, you have to modify values in the kernel launch operator. The programmer has to be careful that the number of threads per block does not go beyond the maximum supported limit of your GPU device. In this book, we are targetting computer vision applications where we need to work on two and three-dimensional images. In that, it would be great if blocks and threads are not one dimensional but more than that for better processing and visualization.

GPU supports three-dimensional grid of blocks and three-dimensional blocks of threads. It has the following syntax:

```
mykernel<< <dim3(Nbx, Nby,Nbz), dim3(Ntx, Nty,Ntz) > >> ()
```

Where N_{bx}, N_{by}, and N_{bz} indicate the number of blocks in a grid in the direction of X, Y and Z axis, respectively. Similarly, N_{tx}, N_{ty}, and N_{tz} indicate the number of threads in a block in the direction of X, Y and Z axis. If Y and Z dimensions are not specified they are taken as 1 by default. So, for example, to process an image, you can start 16 x 16 grid of blocks with all containing 16 x 16 threads. The syntax will be as follows:

```
mykernel << <dim3(16,16),dim3(16,16)> >> ()
```

To summarize, configuration of the number of blocks and number of threads is very important while launching the kernel. It should be chosen with proper care depending on the application that we are working on and GPU resources. The next section will explain some important CUDA functions added over regular ANSI C functions.

CUDA API functions

In the variable addition program, we have encountered some functions or keywords that are not familiar to regular C/C++ Programmers. These keywords and functions include `__global__` , `cudaMalloc`, `cudaMemcpy` and `cudaFree`. So, in this section, these functions are explained in detail one by one:

- **`__global__`** : It is one of three qualifier keyword aalong with `__device__` and `__host__` . This keyword indicates that function is declared as a device function and will execute on the device when called from the host. It should be kept in mind that this function can only be called from host. If you wan t your function to execute on device and called from device function then you have to use `__device__` keyword. `__host__` keyword is used to define host functions which can only be called from other host function. This is similar to normal C functions. By default all functions in a program aa host functions.Both `__host__` and `__device__` can be simultaneously used to define any function. It generates two copies of the same function. One will execute on the host and the other will execute on the device.

- **cudaMalloc**: It is similar to the `Malloc` function used in C for dynamic memory allocation. This function is used to allocate memory block of a specific size on device. The syntax of `cudaMalloc` with example is as follows:

  ```
  cudaMalloc(void ** d_pointer, size_t size)
  Example: cudaMalloc((void**)&d_c, sizeof(int));
  ```

  As shown in the example, it allocates a memory block of size equal to size of one integer variable and returns the pointer d_c which points to this memory location.

- **cudaMemcpy**: This function is similar to the `Memcpy` function in C. It is used to copy one block of memory to other blocks on host or device. It has the following syntax:

  ```
  cudaMemcpy ( void * dst_ptr, const void * src_ptr, size_t size, enum cudaMemcpyKind kind )
  Example: cudaMemcpy(&h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
  ```

  This function has four arguments. The first and second arguments are destination pointer and source pointer which points to host or device memory location. The third argument indicates size of copy and last argument indicates the direction of copy. It can be from host to device, device to device, host to host or device to host. But be careful that you have to match this direction with approproate pointers as first two arguments. As shown in example, we are coping block of one integer variable from device to host by specifying device pointer d_c as source and host pointer h_c as destination.

- **cudaFree**: It is similar to free function available in C. The syntax of `cudaFree` is as follows:

  ```
  cudaFree ( void * d_ptr )
  Example: cudaFree(d_c)
  ```

It frees the memory space pointed to by `d_ptr`. In the example, it frees the memory location pointed by d_c. Please make sure that d_c is allocated memory using cudaMalloc to free it using cudaFree.

There are many other keywords and functions available in CUDA over and above existing ANSI C functions. We will be frequently using only these three functions, and hence they are discussed in this section. For more details, you can always visit the CUDA programming guide.

Passing parameters to CUDA functions

The `gpuAdd` kernel function of variable addition program is very similar to the normal C function. So, like normal C functions, the kernel functions can also be passed parameters by value or by reference. Hence, in this section, we will see both the methods to pass parameters for CUDA kernels.

Passing parameter by value

To recall, in the `gpuAdd` program, the syntax for calling kernel was as follows:

```
gpuAdd << <1,1> >>(1,4,d_c)
```

On the other hands, `gpuAdd` signature while definition was as follows:

```
__global__  gpuAdd(int d_a, int d_b, int *d_c)
```

So, you can see that we are passing values of `d_a` and `d_b` while calling the kernel. First parameter 1 will be copied to `d_a` and second parameter `4` will be copied to `d_b` while kernel call. The answer after addition will be stored on address pointed by `d_c` on device memory. Instead of directly passing values 1 and 4 as inputs to kernel, we can also write the following:

```
gpuAdd << <1,1> >>(a,b,d_c)
```

Here, `a` and `b` are integer variables that can contain any integer values. Passing parameters by values are not recommended as it creates unnecessary confusion and complication in programs for readers to understand. It is better to pass parameters by reference.

Passing parameter by reference

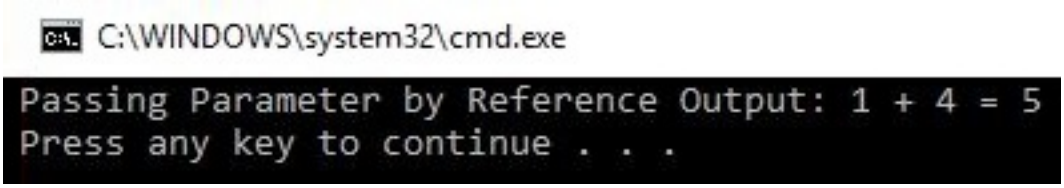
Now, we will see how to write the same program by passing parameters by reference. For that we have to first modify the kernel functions for addition of two variables. The modified kernel for passing parameters by reference is shown below:

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
//Kernel function to add two variables, parameters are passed by reference
 __global__ void gpuAdd(int *d_a, int *d_b, int *d_c)
{
 *d_c = *d_a + *d_b;
}
```

Instead of using integer variables d_a and d_b as inputs to the kernel, the pointers to these variables on device *d_a and *d_b are taken as inputs. The answer after addition is stored at memory location pointed by third integer pointer d_c. The pointers passed as reference to this device function should be allocated memory with cudaMalloc function. The main function for this code is shown below:

```
int main(void)
{
 //Defining host and variables
 int h_a,h_b, h_c;
 int *d_a,*d_b,*d_c;
 //Initializing host variables
 h_a = 1;
 h_b = 4;
 //Allocating memory for Device Pointers
 cudaMalloc((void**)&d_a, sizeof(int));
 cudaMalloc((void**)&d_b, sizeof(int));
 cudaMalloc((void**)&d_c, sizeof(int));
 //Coping value of host variables in device memory
 cudaMemcpy(d_a, &h_a, sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, &h_b, sizeof(int), cudaMemcpyHostToDevice);
 //Calling kernel with one thread and one block with parameters passed by reference
 gpuAdd << <1, 1 >> > (d_a, d_b, d_c);
 //Coping result from device memory to host
 cudaMemcpy(&h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
 printf("Passing Parameter by Reference Output: %d + %d = %d\n", h_a, h_b, h_c);
 //Free up memory
 cudaFree(d_a);
 cudaFree(d_b);
 cudaFree(d_c);
 return 0;
 }
```

`h_a`, `h_b` and `h_c` are variables in host memory. They are defined like normal C code. On the other hand, `d_a`, `d_b` and `d_c` are pointers residing on host memory and they point to device memory. They are allocated memory from host by using the `cudaMalloc` function. The values of `h_a` and `h_b` are copied to device memory pointed by `d_a` and `d_b` by using the `cudaMemcpy` function, and the direction of data transfer is from host to device. Then, in kernel call, these three device pointers are passed to the kernel as parameters. The kernel computes addition and stores result at memory location pointed by `d_c`. The result is copied back to host memory by using cudaMemcpy function again, but this time with the direction of data transfer as device to host. The output of program is as follows:



The memory used by three device pointers is freed by using `cudaFree` function at end of the program. The sample memory map on host and device will look something as follows:

Host Memory (CPU)		**Device Memory (GPU)**	
Address	Value	Address	Value
#01	h_a=1	#01	1
#02	h_b=4	#02	4
#03	h_c=5	#03	5
#04	d_a=#01	#04	
#05	d_b=#02	#05	
#06	d_c=#03	#06	

As you can see from the table, `d_a`, `d_b`, and `d_c` are residing on host and pointing to values on device memory. While passing parameters by reference to kernels, you should take care that all pointers are pointing to device memory only. If it is not the case, then there are chances that the program may crash.

While using device pointers and passing it to kernels, there are some restrictions that have to be followed by the programmer. The device pointers which are allocated memory using cudaMalloc can only be used to read or write from device memory. It can be passed as parameters to device function. It should not be used to read and write memory from host functions. To simplify, device pointers should be used to read and write device memory from device function and host pointers should be used to read and write host memory from host functions. So in this book, you will always find device pointer prefixed by 'd_' in kernel functions.

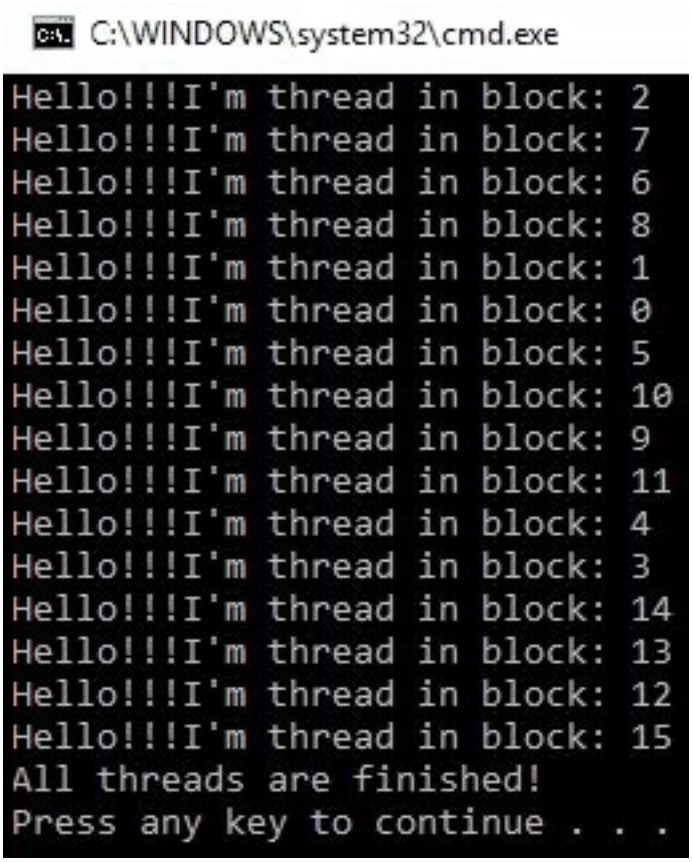
To summarize, in this section, concepts related to CUDA programming are explained in detail by taking two-variable addition program as an example. After this section, you should be familiar with basic CUDA programming concepts and terminology associated with CUDA programs. In the next section, you will learn how threads are executed on the device.

Thread execution on a device

We have seen that while configuring kernel parameters, we can start multiple blocks and multiple threads in parallel. So, in which order do these blocks and threads start and finish their execution? It is important to know this if we want to use the output of one thread in other threads. To understand this, we have modified kernel in `hello,CUDA!` program seen in the first chapter by including a print statement in kernel call which prints block number. The modified code is shown as follows:

```
#include <iostream>
#include <stdio.h>
__global__ void myfirstkernel(void)
{
 //blockIdx.x gives the block number of current kernel
 printf("Hello!!!I'm thread in block: %d\n", blockIdx.x);
}
int main(void)
{
 //A kernel call with 16 blocks and 1 thread per block
 myfirstkernel << <16,1>> >();

//Function used for waiting for all kernels to finish
 cudaDeviceSynchronize();

 printf("All threads are finished!\n");
 return 0;
}
```

As can be seen from the code, we are launching kernel with 16 blocks in parallel with each block having a single thread. In the kernel code, we are printing block id of the kernel execution. We can think that 16 copies of the same `myfirstkernel` start execution in parallel. Each of these copies will have a unique block id, which can be accessed by the `blockIdx.x CUDA` directive, and unique thread id, which can be accessed by `threadIdx.x`. These ids will tell us which block and thread are executing kernel. When you run the program many times, you will find that each time, blocks execute in different order. One sample output can be shown as follows:



So, one question you should ask is how many different output patterns will the previous program produce? The correct answer is 16!. It will produce n factorial number of outputs, where n indicates the number of blocks started in parallel. So, whenever you are writing the program in CUDA, you should be careful that blocks execute in random order.

The as-mentioned program also contains one more CUDA directive: `cudaDeviceSynchronize()`. Why is it used? It is used because a kernel launch is an asynchronous process, which means it returns control to the CPU thread immediately after starting up the GPU process before the kernel has finished executing. In the previous code, the next line in CPU thread is print and application exit that will terminate console before the kernel has finished execution. So, if we do not include this directive, you will not see any print statements of kernel execution. The output that is generated later by the kernel has nowhere to go, and you won't see it. To see the outputs generated by kernel we will include this directive, which ensures that the kernel finishes before the application is allowed to exit, and the output from the kernel will find a waiting standard output queue.

Accessing GPU device properties from CUDA programs

CUDA provides a simple interface to determine the information such as determining which CUDA-enabled GPU devices (if any) are present and what capabilities each device supports. First, It is important to get a count of how many CUDA-enabled devices are present on the system, as a system may contain more than one GPU-enabled devices. This count can be determined by CUDA API `cudaGetDevideCount()`. The program for getting number of CUDA enable devices on system is shown below:

```
#include <memory>
#include <iostream>
#include <cuda_runtime.h>
// Main Program
int main(void)
{
  int device_Count = 0;
  cudaGetDeviceCount(&device_Count);
  // This function returns count of number of CUDA enable devices and 0 if there are no CUDA capable
devices.
  if (device_Count == 0)
  {
    printf("There are no available device(s) that support CUDA\n");
  }
  else
  {
    printf("Detected %d CUDA Capable device(s)\n", device_Count);
  }
}
```

The relevant information about each device can be found by querying the `cudaDeviceProp` structure, which returns all device properties. If you have more than one cuda capable devices than you can start a for loop to iterate over all device properties. The following section contains the list of Device properties divided into different sets and small code snippets used to access them from CUDA programs. These properties are provided by the cudaDeviceProp structure in CUDA 9 runtime. :

For more details about properties in the different version of CUDA, you can check programming guide for that version.

General Device Properties:

cudaDeviceProp provides many properties that are used to identify name and versions of the device we are having. It provides 'name' property which returns the name of the device as a string. We can also get a version of driver and runtime engine the device is using by querying cudaDriverGetVersion and cudaRuntimeGetVersion properties. Sometimes if you have more than one device, you want to use device which has more multiprocessors. multiProcessorCount property returns the count of the number of multiprocessor on the device. The speed of GPU in terms of clock rate can be fetched by using clockRate property. It returns clock rate in Khz. Following code snippet shows how to use these properties from CUDA program.

```
cudaDeviceProp device_Property;
cudaGetDeviceProperties(&device_Property, device);
printf("\nDevice %d: \"%s\"\n", device, device_Property.name);
cudaDriverGetVersion(&driver_Version);
cudaRuntimeGetVersion(&runtime_Version);
printf(" CUDA Driver Version / Runtime Version %d.%d / %d.%d\n", driver_Version / 1000, (driver_Version
% 100) / 10, runtime_Version / 1000, (runtime_Version % 100) / 10);
printf( " Total amount of global memory: %.0f MBytes (%llu bytes)\n",
 (float)device_Property.totalGlobalMem / 1048576.0f, (unsigned long long)
device_Property.totalGlobalMem);
 printf(" (%2d) Multiprocessors", device_Property.multiProcessorCount );
printf("  GPU Max Clock rate: %.0f MHz (%0.2f GHz)\n", device_Property.clockRate * 1e-3f,
device_Property.clockRate * 1e-6f);
```

Memory Related Properties

Memory on GPU device is having a hierarchical structure. It can be divided in terms of L1 cache, L2 cache, global memory, texture memory and shared memory. cudaDeviceProp provided many properties that help in identifying memory available with the device. memoryClockRate and memoryBusWidth provides clock rate and bus width of the memory respectively. Speed of memory is very important. It affects the overall speed of your program. totalGlobalMem property returns the size of global memory avaiable with the device. totalConstMem returns the total constant memory available with the device. sharedMemPerBlock returns the total shared memory can be used in device. Total number of registers available per block can be identified by using regsPerBlock property. Size of l2cahche can be identified using l2CacheSize property. Following code snippet shows how to use memory related properties from CUDA program.

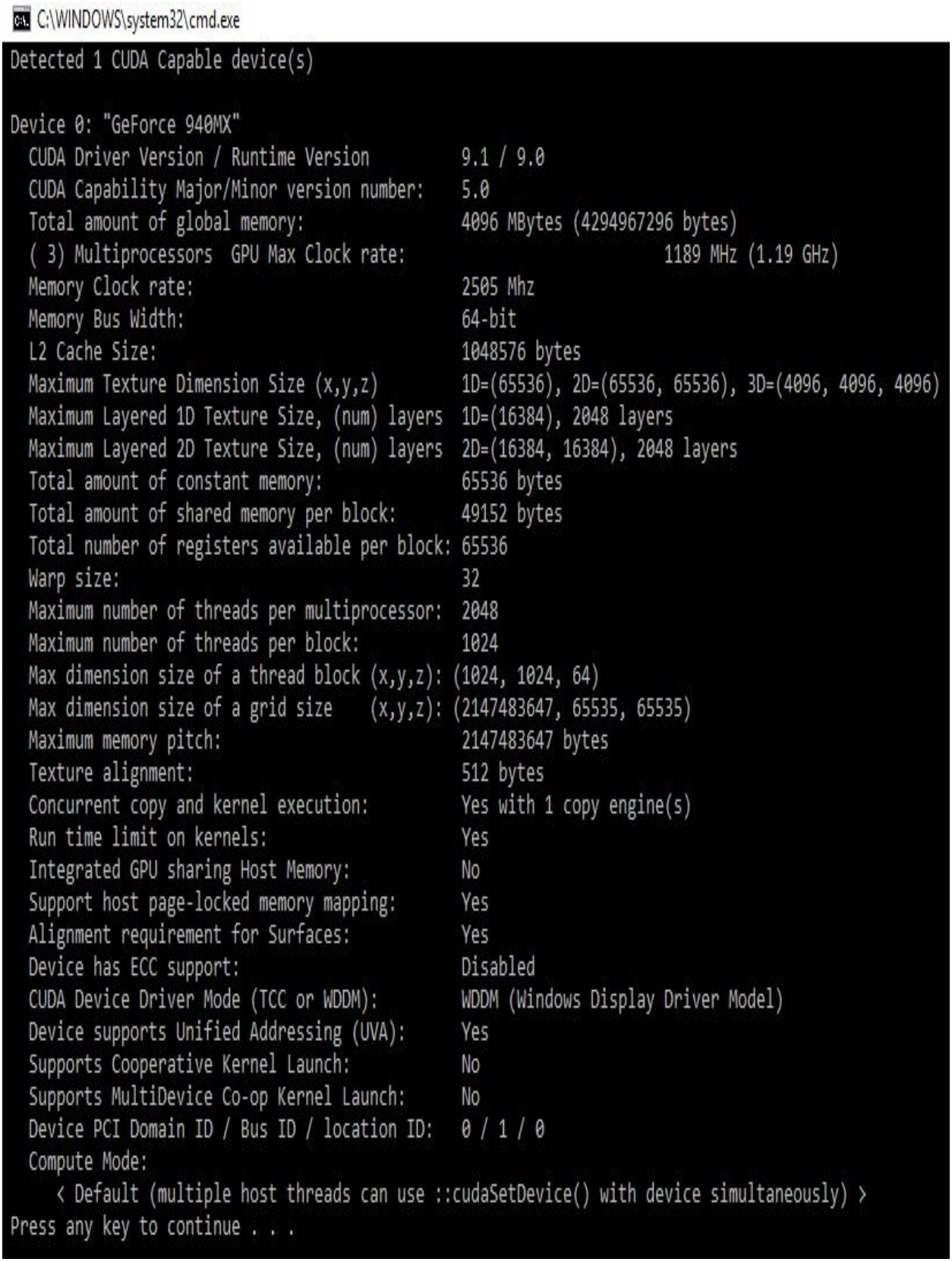
```
printf( " Total amount of global memory: %.0f MBytes (%llu bytes)\n",
(float)device_Property.totalGlobalMem / 1048576.0f, (unsigned long long)
device_Property.totalGlobalMem);
printf(" Memory Clock rate: %.0f Mhz\n", device_Property.memoryClockRate * 1e-3f);
printf(" Memory Bus Width: %d-bit\n", device_Property.memoryBusWidth);
if (device_Property.l2CacheSize)
{
    printf(" L2 Cache Size: %d bytes\n", device_Property.l2CacheSize);
}
printf(" Total amount of constant memory: %lu bytes\n",         device_Property.totalConstMem);
printf(" Total amount of shared memory per block: %lu bytes\n", device_Property.sharedMemPerBlock);
printf(" Total number of registers available per block: %d\n", device_Property.regsPerBlock);
```

Thread Related Properties

As seen in earlier sections, blocks and threads can be multidimensional. So it would be nice to know how many threads and blocks can be launched in parallel in each dimension. There is also limit on number of threads per multiprocessor and number of threads per block. This number can be found by using maxThreadsPerMultiProcessor and maxThreadsPerBlock property. It is very important in configuration of kernel parameters. If you launch more threads per block than maximum threads possible per block, your program can crash. The maximum threads per block in each dimension can be identified by maxThreadsDim property. Same way maximum blocks per grid in each dimension can be identified by using maxGridSize property. Both of them return an Array with three values which shows maximum value in X, Y and Z dimension respectively. Following code snippet shows how to use thread related properties from CUDA code.

```
printf(" Maximum number of threads per multiprocessor: %d\n",
device_Property.maxThreadsPerMultiProcessor);
printf(" Maximum number of threads per block: %d\n",         device_Property.maxThreadsPerBlock);
printf(" Max dimension size of a thread block (x,y,z): (%d, %d, %d)\n",
    device_Property.maxThreadsDim[0],
    device_Property.maxThreadsDim[1],
    device_Property.maxThreadsDim[2]);
printf(" Max dimension size of a grid size (x,y,z): (%d, %d, %d)\n",
    device_Property.maxGridSize[0],
    device_Property.maxGridSize[1],
    device_Property.maxGridSize[2]);
```

There are many other properties available in cudaDeviceProp structure. You can check CUDA programming guide for details of other properties. The output of the all preceding code sections combined and executed on NVIDIA Geforce 940MX GPU and CUDA 9.0 is as follows:



One question you might ask is why I am interested in knowing the device properties. The answer

is that it will help you in choosing a GPU device with more number of multiprocessors in case multiple GPU devices are present. If in your application the kernel needs close interaction with CPU, then you might want your kernel to run on an integrated GPU that shares system memory with CPU. These properties will also help you in finding the number of blocks and number of threads per block available on your device. This will help you in the configuration of kernel parameters. To show you one use of device properties, suppose you have an application that requires double precision for floating point operation. Not all GPU devices support this operation. To know if your device supports double precision floating point operation or not and set that device for your application following code can be used:

```
#include <memory>
#include <iostream>
#include <cuda_runtime.h>
// Main Program
int main(void)
{
int device;
cudaDeviceProp device_property;
cudaGetDevice(&device);
printf("ID of device: %d\n", device);
memset(&device_property, 0, sizeof(cudaDeviceProp));
device_property.major = 1;
device_property.minor = 3;
cudaChooseDevice(&device, &device_property);
printf("ID of device which supports double precision is: %d\n", device);
cudaSetDevice(device);
}
```

This code uses two properties available in cudaDeviceprop structure that helps in identifying whether the device supports double precision operations or not. These two properties are major and minor. CUDA documentation tells that if major is greater than 1 and minor is greater than 3 then that device will support double precision operations. So in the program device_property structure is filled with these two values. CUDA also provides cudaChooseDevice API that helps in choosing a device with particular properties. So that API is used on current device to identify whether it contains these two properties or not. If it contains then that device is selected for your application using cudaSetDevice API. If more than one device is present in the system, this code should be written inside for loop to iterate over all devices.

Though trivial, this section is very important for you in finding out which applications can be supported by your GPU device and which cannot.

Vector Operation in CUDA

Until now, programs that we have seen were not leveraging any advantage of parallel processing capabilities of GPU device. They were just written to get you familiar with the programming concepts in CUDA. From this section, we will start utilizing parallel processing capabilities of GPU by doing vector or Array operations on it.

Two-vector addition Program

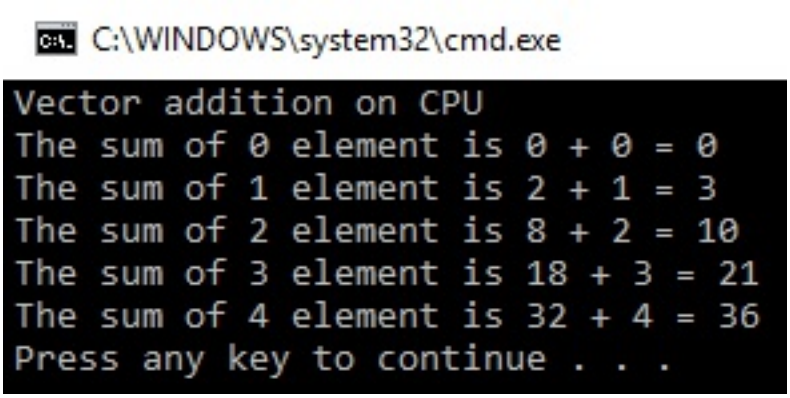
To understand vector operation on GPU, we will start by writing a vector addition program on CPU and then modify it to utilize the parallel structure of GPU. We will take two arrays of some numbers and store the answer of element-wise addition in the third array. The vector addition function on CPU is shown below:

```
#include "stdio.h"
#include<iostream>
 //Defining Number of elements in Array
#define N 5
 //Defining vector addition function for CPU
void cpuAdd(int *h_a, int *h_b, int *h_c)
{
     int tid = 0;
     while (tid < N)
     {
         h_c[tid] = h_a[tid] + h_b[tid];
         tid += 1;
     }
 }
```

`cpuAdd` function should be very simple to understand. One thing you might find difficult to understand is the use of `tid`. It is included to make the program similar to the GPU program, in which `tid` indicated a particular thread id. Here also, if you are having a multicore CPU, then you can initialize `tid` equal to 0 and 1 for each of them and then add two to it in the loop so that one CPU will perform sum on even elements and one CPU will perform addition on odd elements. The main function for the code is shown below:

```
int main(void)
{
     int h_a[N], h_b[N], h_c[N];
     //Initializing two arrays for addition
     for (int i = 0; i < N; i++)
    {
         h_a[i] = 2 * i*i;
         h_b[i] = i;
     }
     //Calling CPU function for vector addition
     cpuAdd (h_a, h_b, h_c);
     //Printing Answer
     printf("Vector addition on CPU\n");
     for (int i = 0; i < N; i++)
    {
     printf("The sum of %d element is %d + %d = %d\n", i, h_a[i], h_b[i],             h_c[i]);
    }
     return 0;
 }
```

There are two functions in the program: `main` and `cpuAdd`. In main function, we start by defining two arrays to hold inputs and initialize it to some random numbers. Then, we pass these two arrays as input to the `cpuAdd` function. The `cpuAdd` function stores the answer in the third array. Then, we print this answer on the console which is shown below:



This explanation of using tid in cpuadd function may give you an idea of how to write the same function for GPU execution which can have many cores in parallel. So, if we initialize this add function with the id of that core, then we can do the addition of all elements in parallel. So the modifed kernel function for addition on GPU is shown below:

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
 //Defining number of elements in Array
#define N 5
 //Defining Kernel function for vector addition
__global__ void gpuAdd(int *d_a, int *d_b, int *d_c)
{
 //Getting block index of current kernel
     int tid = blockIdx.x; // handle the data at this index
     if (tid < N)
     d_c[tid] = d_a[tid] + d_b[tid];
 }
```

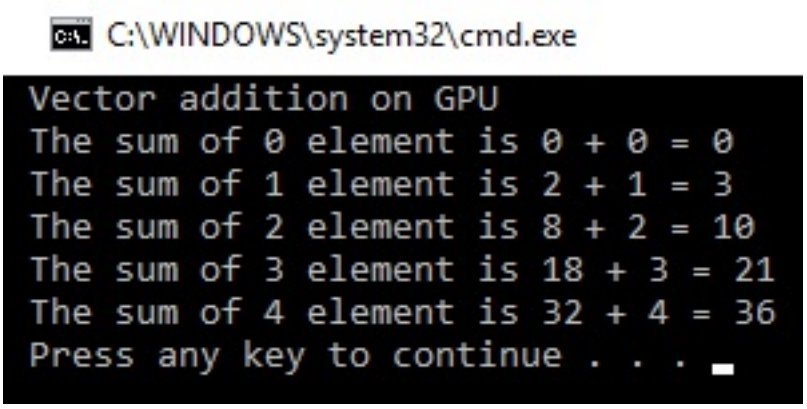
In gpuAdd kernel function, tid is initialize with block id of current block in which kernel is executing. All kernels will add array element indexed by this block id. So if the number of blocks are equal to number of elements in an Array then all addition operations will be done in parallel. How this kernel is called from main function is explained below. The code for main function is as follows:

```
int main(void)
{
 //Defining host arrays
 int h_a[N], h_b[N], h_c[N];
 //Defining device pointers
 int *d_a, *d_b, *d_c;
 // allocate the memory
 cudaMalloc((void**)&d_a, N * sizeof(int));
 cudaMalloc((void**)&d_b, N * sizeof(int));
 cudaMalloc((void**)&d_c, N * sizeof(int));
 //Initializing Arrays
 for (int i = 0; i < N; i++)
    {
     h_a[i] = 2*i*i;
     h_b[i] = i ;
     }

// Copy input arrays from host to device memory
 cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);

//Calling kernels with N blocks and one thread per block, passing device pointers as parameters
gpuAdd << <N, 1 >> >(d_a, d_b, d_c);
 //Copy result back to host memory from device memory
cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);
printf("Vector addition on GPU \n");
```

```
 //Printing result on console
for (int i = 0; i < N; i++)
{
     printf("The sum of %d element is %d + %d = %d\n", i, h_a[i], h_b[i],             h_c[i]);
}
 //Free up memory
 cudaFree(d_a);
 cudaFree(d_b);
 cudaFree(d_c);
 return 0;
}
```

The GPU main function has the fowell-knownll known structure as explained in the first section of this chapter:

- It starts with defining arrays and pointers for host and device. The device pointers are allocated memory using the `cudaMalloc` function.
- The arrays, which are to be passed to the kernel, are copied from host memory to device memory by using the `cudaMemcpy` function.
- The kernel is launched by passing device pointers as parameters to it. If you see the values inside the kernel launch operator, it is N and 1, which indicates we are launching N blocks with one thread per each block.
- The answer stored by the kernel on device memory is copied back to host memory by again using cudaMemcpy function but this time with the direction of data transfer from device to host.
- And finally, memory allocated to three device pointers is freed up by using the `cudaFree` function. The output of the program is shown as follows.



All CUDA programs follow the same pattern as shown before. We are launching N blocks in parallel. The meaning of this is that we are launching N copies of the same kernel simultaneously. You can understand this by taking a real-life example: Suppose you want to transfer five big boxes from one place to another. In the first method, you can do this task by hiring one person that takes one block from one place to the other and repeat this five times. This option will take time and it is similar to how vectors are added on CPU. Now suppose you are hiring five persons and each of them carries one box from. Each of them also knows the id of the box they are carrying. This option will be much faster than the previous one. Each one of them just needs to be told that you have to carry one box with a particular id from one place to the other.

This is exactly how kernels are defined and executed on the device. Each kernel copy knows the

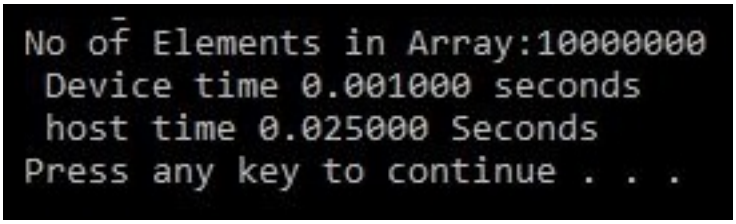
id of it. This can be known by the `blockIdx.x` command. Each copy works on array element indexed by its id. So, all copies add all elements in parallel, which significantly reduces processing time for the entire array. So, in a way we are improving the throughput by doing operations in parallel over CPU sequential execution. The comparison of throughput between CPU code and GPU code is explained in next section.

Comparison of throughput between CPU and GPU code

The programs for CPU and GPU addition are written in a modular way so you can play around with the value of N. If N is small then you will not notice any significant time difference between CPU and GPU code. But if you take N sufficiently large, then you will notice the significant difference in CPU execution time and GPU execution time for same vector addition. The time taken for execution of a particular block can be measured by adding following lines in existing code:

```
clock_t start_d = clock();
printf("Doing GPU Vector add\n");
gpuAdd << <N, 1 >> >(d_a, d_b, d_c);
cudaThreadSynchronize();
clock_t end_d = clock();
double time_d = (double)(end_d - start_d) / CLOCKS_PER_SEC;
printf("No of Elements in Array:%d \n Device time %f seconds \n host time %f Seconds\n", N, time_d,
time_h);
```

Time is measured by calculating total number of clock cycles taken to perform a particular operation. This can be done by taking the difference of starting and ending clock tick count measured using clock() function. This is dived by number of clock cycles per second to get execution time. When N is taken as 10,00,000 in above vector addition programs of CPU and GPU and executed simultaneously. The output is as follows:



As can be seen from the ouput, execution time or throughput is improved from 25ms to almost 1ms when the same function is implemented on GPU. This proves what we have seen in theory earlier that executing code in parallel on GPU helps in improvement of throughput.

Elementwise squaring of Vector in CUDA

Now, one question you can ask here is now that we are launching N blocks in parallel with one thread in each block, can we work in the reverse way? The answer is Yes. We can launch only one block with N threads in parallel. To show that and make you more familiar with working around vectors in CUDA, we take the second example of element-wise squaring of numbers in an Array. We take one Array of numbers and return an Array that contains the square of these numbers. The kernel function to find element-wise square is shown below:

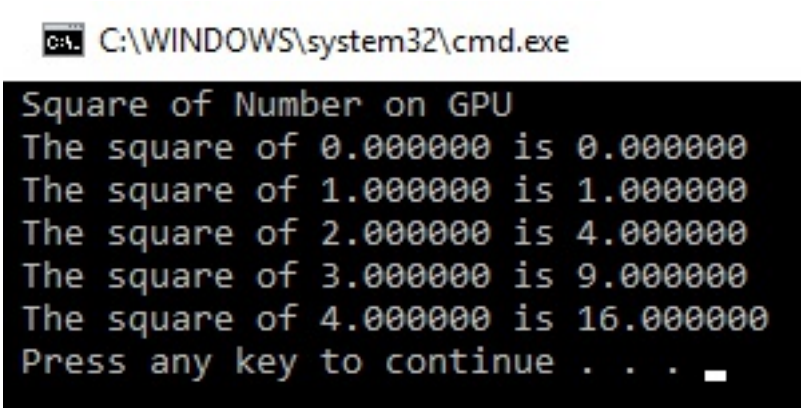
```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
 //Defining number of elements in Array
#define N 5
//Kernel function for squaring number
__global__ void gpuSquare(float *d_in, float *d_out)
{
     //Getting thread index for current kernel
     int tid = threadIdx.x; // handle the data at this index
     float temp = d_in[tid];
     d_out[tid] = temp*temp;
 }
```

gpuSquare kernel function has pointers to two arrays as arguments. The first pointer d_in points to the memory location where input array is stored while second pointer d_out points to the memory location where output will be stored. In this program, instead of launching multiple blocks in parallel we want to launch multiple threads in parallel so tid is initialized with a particular thread id using threadIdx.x. The main function for this program is as follows:

```
int main(void)
{
 //Defining Arrays for host
     float h_in[N], h_out[N];
     float *d_in, *d_out;
// allocate the memory on the cpu
     cudaMalloc((void**)&d_in, N * sizeof(float));
     cudaMalloc((void**)&d_out, N * sizeof(float));
 //Initializing Array
     for (int i = 0; i < N; i++)
    {
         h_in[i] = i;
     }
 //Copy Array from host to device
     cudaMemcpy(d_in, h_in, N * sizeof(float), cudaMemcpyHostToDevice);
 //Calling square kernel with one block and N threads per block
     gpuSquare << <1, N >> >(d_in, d_out);
 //Coping result back to host from device memory
     cudaMemcpy(h_out, d_out, N * sizeof(float), cudaMemcpyDeviceToHost);
 //Printing result on console
     printf("Square of Number on GPU \n");
     for (int i = 0; i < N; i++)
     {
         printf("The square of %f is %f\n", h_in[i], h_out[i]);
     }
 //Free up memory
     cudaFree(d_in);
     cudaFree(d_out);
     return 0;
 }
```

This main function follows a similar structure to the vector addition program. One difference that you will see here from the vector addition program is that we are launching single block with N

threads in parallel. The output of the program is shown as follows:



Whenever you are using this way of launching N threads in parallel, you should take care that maximum threads per block are limited to 512 or 1024. So, the value of N should be less than this value. If N is 2000 and the maximum number of threads per block for your device is 512 then you can't write `<< <1,2000 > >>`. Instead, you should use something like `<< <4,500> >>`. So, the choice of number of blocks and number of threads per block should be made judiciously.

To summarize, we have learnt how to work with vectors and how we can launch multiple blocks and multiple threads in parallel. We have also seen that by doing vector operations on GPU, it improves throughput compared to the same operation on CPU. In the last section of this chapter, we discuss the various parallel communication patterns that are followed by threads executing in parallel.

Parallel Communication Patterns

When more than one threads are executed in parallel, they follow a certain communication pattern that indicates where it is taking inputs and where it is writing its output in memory. We will discuss each communication pattern one by one. It will help you to identify communication patterns related to your application and how to write code for that.

Map

In this communication pattern, each thread or task takes a single input and produces a single output. Basically, it is a one to one operation. The vector addition program and element-wise squaring program, seen in the previous section, are examples of map pattern. The code of map pattern will look like this:

```
d_out[i] = d_in[i] * 2
```

Gather

In this pattern, each thread or task has multiple inputs and it produces a single output to be written at a single location in memory. Suppose, you want to write a program that finds moving average of three numbers then this is an example of gather operation. It takes three inputs from memory and writes single output to memory. So there is data reuse on the input side. It is basically many to one operation. The code for gather pattern will look like:

```
out[i] = (in [i-1] + in[i] + in[i+1])/3
```

Scatter

In scatter pattern, thread or task takes a single input and computes wherein the memory it should write the output. Array sorting is an example of scatter operation. It can also be one to many operation. The code for scatter pattern will look like:

```
out[i-1] += 2 * in[i] and out[i+1] += 3*in[i]
```

Stencil

When threads or tasks read input from a fixed set of neighbourhood in an Array then this is called as stencil communication pattern. It is very useful in image processing examples where we work on 3x3 or 5x5 neighbourhood windows. It is a special form of gather operation so code syntax is similar to it.

Transpose

When the input is in form of a row-major matrix and we want the output to be in column-major form than we have to use this transpose communication pattern. It is particularly useful if you have a structure of arrays and you want to convert it in form of an array of structures. It is also a one to one operation. The code for transpose pattern will look like:

```
out[i+j*128] = in [j +i*128]
```

In this section, various communication patterns that CUDA programming follows is discussed. it is useful to find out communication pattern related to your application and use code syntax of that pattern shown as an example.

Summary

To summarize, in this chapter, you were introduced to programming concepts in CUDA C and how parallel computing can be done using CUDA. It was shown that CUDA programs can run on any GPU hardware efficiently and in parallel. So, CUDA is both efficient and scalable. The CUDA API functions over and above existing ANSI C functions needed for parallel data computations were discussed in detail. How to call device code from host code via a kernel call, configuration of kernel parameters and a passing of parameters to the kernel was also discussed by taking a simple two-variable addition example. It was also shown that CUDA does not guarantee the order in which the blocks or thread will run and which block is assigned to which multi-processor in hardware. Moreover, vector operations, which take advantage of parallel processing capabilities of GPU and CUDA, were discussed. It can be seen that by doing vector operations on GPU, it can improve the throughput drastically compared to CPU. In the last section, various common communication patterns followed in parallel programming were discussed in detail. Still, we have not discussed memory architecture and how threads can communicate with each other in CUDA. If one thread needs data of the other thread then what can be done is also not discussed. So in the next chapter, we will discuss memory architecture and thread synchronization in detail.

Questions

1. Write a CUDA program to subtract two numbers. Pass parameters by value in the kernel function.
2. Write a CUDA program to multiply two numbers. Pass parameters by reference in the kernel function.
3. Suppose you want to launch 5000 threads in parallel. Configure kernel parameters in three different ways to accomplish this. Maximum 512 threads are possible per block.
4. State True or False: The programmer can decide in which order blocks will execute on the device and block will be assigned to which streaming multiprocessor.
5. Write a CUDA program to find out that your system contains GPU device that has a major-minor version of 5.0 or greater.
6. Write a CUDA program to find a cube of a vector that contains numbers starting from 0 to 49.
7. For following applications which communication pattern is useful?
    1. Image Processing
    2. Moving Average
    3. Sorting Array in ascending order
    4. Finding cube of numbers in Array

Threads, Synchronization and Memory

Introduction

In the last chapter, we have seen how to write CUDA programs that take leverage of parallel processing capabilities of GPU by executing multiple threads and blocks in parallel. In programs seen in the last chapter, all threads were independent of each other and there was no communication between multiple threads. Most of the real-life applications need communication between intermediate threads. So, in this chapter, how communication between different threads can be done is explained in detail. The synchronization between multiple threads working on the same data has also been explained in this chapter. It also describes hierarchical memory architecture of CUDA and how different memories can be used to accelerate CUDA programs. The last part of the chapter explains a very useful application of CUDA in dot product of vectors and Matrix multiplication using all the concepts seen earlier.

The following topics will be covered in this chapter:

- Thread Calls
- CUDA Memory Architecture
- Global, Local and Cache Memory
- Shared Memory and Thread Synchronization
- Atomic operations
- Constant and Texture Memory
- Dot Product and Matrix Multiplication Example

Technical Requirements

This chapter requires familiarity with the basic C or C++ programming language and codes explained in previous chapters. All the code used in this chapter can be downloaded from following github link: https://github.com/PacktPublishing/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA. The code can be executed on any operating system though it is only tested on Windows 10.

Threads

CUDA has a hierarchical architecture in terms of parallel execution. The kernel execution can be done in parallel with multiple blocks. Each block is further divided into multiple threads. In the last chapter, we have seen that CUDA runtime can carry out parallel operations by launching same copies of kernel multiple times. We have seen that it can be done in two ways: either by launching multiple blocks in parallel with one thread per block or by launching a single block with many threads in parallel. So, one question you might ask is which method to use in your code and is there any limitation on the number of blocks and threads that can be launched in parallel.

The answer to the question is not that trivial. As we will see later on in this chapter, threads in same blocks can communicate with each other via shared memory. So, there is an advantage of launching one block with many threads in parallel so that they can communicate with each other. In the last chapter, we have also seen maxThreadPerBlock property that limits the number of threads that can be launched per block. Its value is 512 or 1024 for latest GPUs. Same way, in the second method, the maximum number of blocks that can be launched in parallel is also limited to 65,535.

Ideally, instead of launching multiple threads per single block or multiple blocks with a single thread, we launch multiple blocks with each having multiple threads (which can be equal to maxThreadPerBlock) in parallel. So suppose, if you want to launch N = 50000 threads in parallel in the vector add example, seen in the last chapter, then kernel call can be as follows:

```
gpuAdd<< <((N +511)/512),512 > >>(d_a,d_b,d_c)
```

Maximum threads per block are 512, and hence the total number of blocks is calculated by dividing the total number of threads (N) by 512, But if N is not an exact multiple of 512, then N/512 may give a wrong number of blocks, which is one less then the actual count. So, to get the next highest integer value for the number of blocks, 511 is added to N and then it is divided by 512. It is basically 'ceil' operation on division.

Now the question is, will this work for all values of N? The answer sadly is No. From the discussion above, the total number of blocks can't go beyond 65,535. So in the as-mentioned kernel call, if `(N+5111)/512` is above 65,535 then again code will fail. To overcome this, small constant number of blocks and threads are launched with some modification in kernel code, which we will see further by rewriting kernel for our vector addition program seen in the last chapter:

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
//Defining number of elements in Array
#define N 50000
__global__ void gpuAdd(int *d_a, int *d_b, int *d_c)
{
    //Getting index of current kernel
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // handle the data at this
index
```

```
while (tid < N)
    {
        d_c[tid] = d_a[tid] + d_b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

This kernel code is almost similar to what we have written in the last chapter. It has two modifications. One modification is in the calculation of thread id and second is the inclusion of while loop in the kernel function. The change in thread id calculation is due to the launching of multiple threads and blocks in parallel. This calculation can be understood by considering blocks and threads as a two-dimensional matrix with the number of blocks equal to the number of rows and number of columns equal to the number of threads per block. We will take an example of 3 blocks and 3 threads/block as shown in the following figure:

Block 0	Thread 0	Thread 1	Thread 2
Block 1	Thread 0	Thread 1	Thread 2
Block 2	Thread 0	Thread 1	Thread 2

We can get the id of each block by blockIdx.x and the id of each thread in current block by threadIdx.x command. So for thread shown in green, block id will be two and thread id will be 1. But what if we want unique index for this thread among all threads. This can be calculated by multiplying its block id with a total number of threads per block, which is given by blockDim.x, and then summing it with its thread id. This can be represented mathematically as follows:

```
tid = threadIdx.x + blockIdx.x * blockDim.x;
```

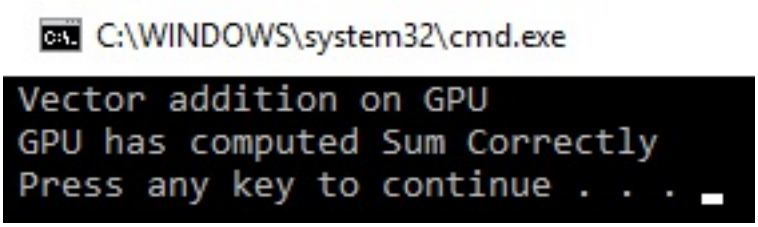
For example, in green, threadIdx.x = 1, blockIdx.x = 2 and blockDim.x = 3 so tid = 7. This calculation is very important to learn as it will be used widely in your codes.

The while loop is included in the code because when N is very large, the total number of threads can't be equal to N because of the limitation seen earlier. So, one thread has to do multiple operations separated by the total number of threads launched. This value can be calculated by multiplying blockDim.x with gridDim.x, which gives block and grid dimensions, respectively. Inside while loop thread id is incremented by this offset value. Now, this code will work for any value of N. To complete the program, we will write main function for this code as follows:

```
int main(void)
{
    //Declare host and device Arrays
    int h_a[N], h_b[N], h_c[N];
    int *d_a, *d_b, *d_c;

    //Allocate Memory on Device
    cudaMalloc((void**)&d_a, N * sizeof(int));
    cudaMalloc((void**)&d_b, N * sizeof(int));
    cudaMalloc((void**)&d_c, N * sizeof(int));
    //Initialize host Array
    for (int i = 0; i < N; i++)
    {
        h_a[i] = 2 * i*i;
        h_b[i] = i;
    }

    cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);
    //Kernel Call
    gpuAdd << <512, 512 >> >(d_a, d_b, d_c);

    cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);
    //This ensures that kernel execution is finishes before going forward
    cudaDeviceSynchronize();
    int Correct = 1;
    printf("Vector addition on GPU \n");
    for (int i = 0; i < N; i++)
    {
        if ((h_a[i] + h_b[i] != h_c[i]))
        { Correct = 0; }
    }
    if (Correct == 1)
    {
        printf("GPU has computed Sum Correctly\n");
    }
    else
    {
        printf("There is an Error in GPU Computation\n");
    }
    //Free up memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}
```

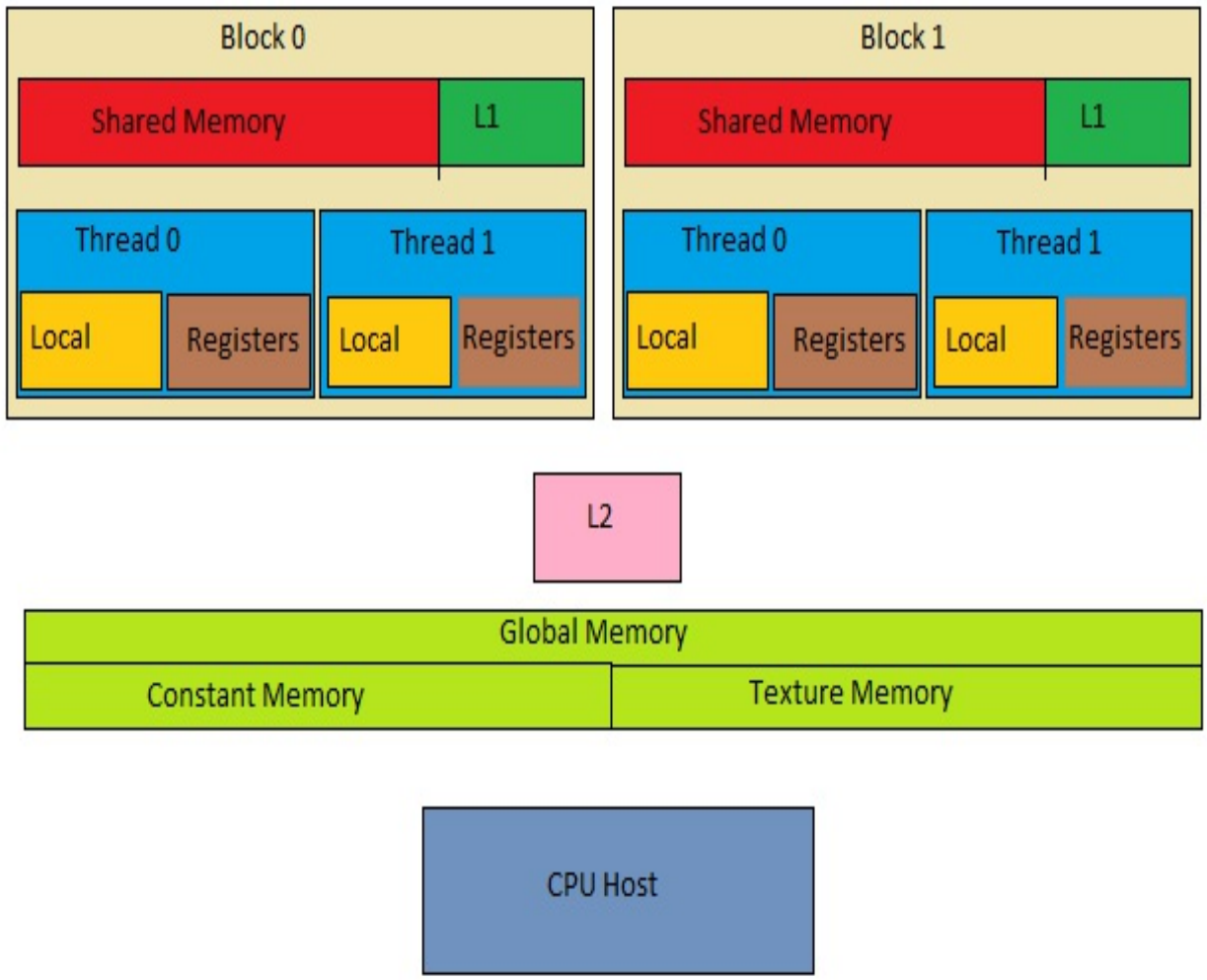
Again, the main function is very similar to what we have written last time. The only changes are in terms of how we are launching kernel function. Kernel is launched by launching 512 blocks, each having 512 threads in parallel. This will solve the problem for large values of N. In this main, instead of printing addition of very long vector, only one print statement, which indicates whether the answer is right or wrong, is printed. The output of the code will be seen as follows:



This section explained the hierarchical execution concept in CUDA. The next section will take this concept further by explaining hierarchical memory architecture.

Memory Architecture

The execution of code on GPU is divided among streaming multiprocessors, blocks and threads. The GPU has several different memory spaces with each having particular features and uses and different speeds and scopes. This memory space is hierarchically divided into different chunks like global memory, shared memory, local memory, constant memory and texture memory, and each of them can be accessed from different points in the program. This memory architecture is shown in the following image:



As shown in the figure, each thread has its own local memory and a register file. Unlike processors, GPU cores have lots of registers to store local data. When data of thread do not fit in the register file, the local memory is used. Both of them are unique to each thread. Register file is the fastest memory. Threads in same blocks have shared memory that can be accessed by all threads in that block. It is used for communication between threads. There is a global memory that can be accessed by all blocks and all threads. Global memory has a large memory access latency. There is a concept of caching to speed up this operation. L1 and L2 caches are available

as shown in the figure. There is a read-only constant memory that is used to store constants and kernel parameters. Finally, There is a texture memory that can take advantage of different 2D or 3D access patterns.

The features of all memories are summarized in the following table:

Memory	**Access Pattern**	**Speed**	**Cached?**	**Scope**	**Lifetime**
Global	Read and Write	Slow	Yes	Host and All Threads	Entire Program
Local	Read and Write	Slow	Yes	Each Thread	Thread
Registers	Read and Write	Fast	-	Each Thread	Thread
Shared	Read and Write	Fast	No	Each Block	Block
Constant	Read only	Slow	Yes	Host and All Threads	Entire Program
Texture	Read only	Slow	Yes	Host and All Threads	Entire Program

The table describes important features of all memories. The scope defines the part of the program that can use this memory and lifetime defines the time for which data in that memory will be visible to the program. Apart from this, L1 and l2 cache is also available for GPU programs for faster memory access.

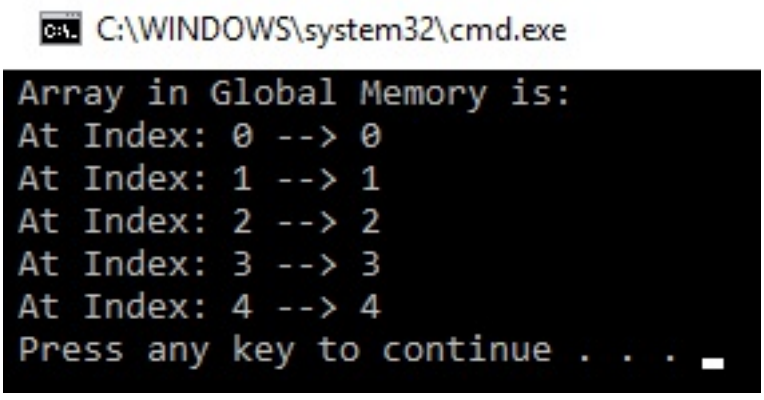
To summarize, all threads have a register file that is fastest. Multiple threads in same blocks have shared memory which is faster than global memory. All blocks can access global memory which will be slowest. Constant and texture memory are used for a special purpose, which will be discussed in the next section. Memory access is the biggest bottleneck in the fast execution of the program. So how these memories can be used to speed up your program is explained in the next section.

Global Memory

All blocks have read and write access to global memory. This memory is slow but can be accessed from anywhere in your device code. The concept of caching is used to speed up access to global memory. All memories allocated using cudaMalloc will be a global memory. The following simple example demonstrates how you can use global memory from your program:

```
#include <stdio.h>
#define N 5

__global__ void gpu_global_memory(int *d_a)
{
  d_a[threadIdx.x] = threadIdx.x;
}

int main(int argc, char **argv)
{
  int h_a[N];
  int *d_a;

  cudaMalloc((void **)&d_a, sizeof(int) *N);
  cudaMemcpy((void *)d_a, (void *)h_a, sizeof(int) *N, cudaMemcpyHostToDevice);

  gpu_global_memory << <1, N >> >(d_a);
  cudaMemcpy((void *)h_a, (void *)d_a, sizeof(int) *N, cudaMemcpyDeviceToHost);

  printf("Array in Global Memory is: \n");
  for (int i = 0; i < N; i++)
  {
    printf("At Index: %d --> %d \n", i, h_a[i]);
  }
  return 0;
}
```

This code demonstrates how you can write in global memory from your device code. The memory is allocated using cudaMalloc from host code and pointer to this array is passed as a parameter to the kernel function. The kernel function populates this memory chunk with values of thread id. This is copied back to host memory for printing. The result is shown as follows:



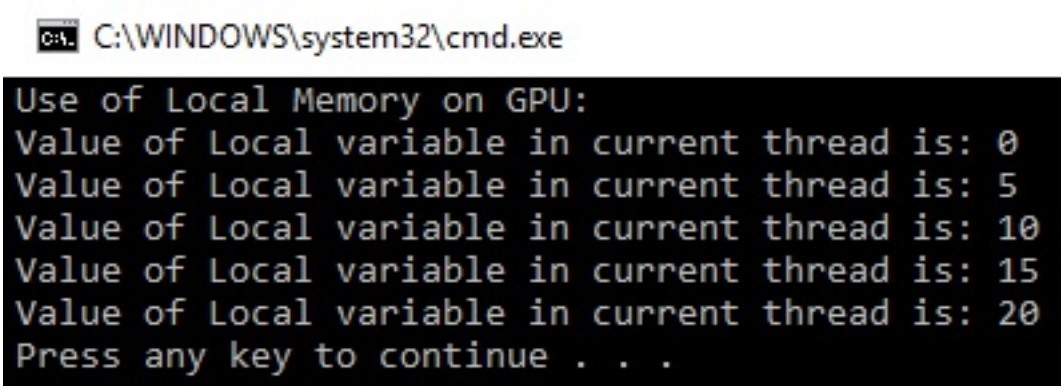
As we are using global memory, this operation will be slow. There are advance concepts to speed up this operation which will be explained later on. In the next section, we explain local memory and registers which is unique to all threads.

Local Memory and Registers

Local memory and register files are unique to each thread. Register files are the fastest memory available for each thread. When variables of the kernel do not fit in register files, it uses local memory. This is called as register spilling. Basically, local memory is a part of global memory that is unique for each thread. Access to local memory will be slow compared to register files. Though Local memory is cached in L1 and L2 caches, register spilling might not affect your program adversely. A simple program to understand how to use local memory is shown as follows:

```
#include <stdio.h>
#define N 5

__global__ void gpu_local_memory(int d_in)
{
  int t_local;
  t_local = d_in * threadIdx.x;
  printf("Value of Local variable in current thread is: %d \n", t_local);
}
int main(int argc, char **argv)
{

  printf("Use of Local Memory on GPU:\n");
  gpu_local_memory << <1, N >> >(5);
  cudaDeviceSynchronize();
  return 0;
}
```

The variable t_local will be local to each thread and stored in a register file. When this variable is used for computation in kernel function, the computation will be fastest. The output of the code is shown as follows:



Cache Memory

On latest GPUs, there is a L1 cache per multiprocessor and L2 cache, which is shared between all multiprocessors. Both global and local memories use these caches. As L1 is near to thread execution, it is very fast. As shown in the figure for memory architecture earlier, L1 cache and shared memory use same 64-kilo bytes. Both can be configured for how many bytes they will use out of the 64 KB. All global memory access go through L2 cache. Texture memory and constant memory have their separate caches.

Thread Synchronization

Up till Now, whatever examples we have seen in this book had all threads independent of each other. But rarely in real life, you will find examples where thread operates on data and terminates without passing results to any other threads. So, there has to be some mechanism for threads to communicate with each other. Therefore, the concept of shared memory is explained in this section. When many threads work in parallel and operate on same data or read and write from the same memory location, there should be synchronization between all threads. Thus, thread synchronization is also explained in this section. Last part of this section explains atomic operations, which are very useful in read-modified write conditions.

Shared Memory

Shared memory is available on-chip, and hence it is much faster than global memory. Shared memory latency is roughly 100 times lower than uncached global memory latency. All the threads from the same block can access shared memory. This is very useful in many applications where threads need to share their results with other threads. However, it can also create chaos or false result if this is not synchronized. If one thread reads data from memory before the other thread has written to it, it can lead to false results. So, this memory access should be controlled or managed properly. This is done by `__syncthreads()` directive, which ensures all write operation to memory is completed before moving ahead in the programs. This is also called as a barrier. The meaning of barrier is all threads will reach at this line and wait for other threads to finish. After all threads have reached this barrier, it can move further. To demonstrate the use of shared memory and thread synchronization, an example of moving average is taken. The kernel function for that is shown as follows:

```
#include <stdio.h>
__global__ void gpu_shared_memory(float *d_a)
{
      int i, index = threadIdx.x;
      float average, sum = 0.0f;
      //Defining shared memory
      __shared__ float sh_arr[10];

      sh_arr[index] = d_a[index];
 // This directive ensure all the writes to shared memory have completed

      __syncthreads();
      for (i = 0; i<= index; i++)
      {
        sum += sh_arr[i];
      }
      average = sum / (index + 1.0f);
      d_a[index] = average;

       //This statement is redundant and will have no effect on overall code execution
      sh_arr[index] = average;
}
```

Moving average operation is nothing but finding an average of all elements in an Array up to current element. So, many threads will need same data of an Array. This is an ideal case of using shared memory, and it will provide faster data than global memory. This will reduce the number of global memory access per thread which in turn will reduce the latency of the program. The shared memory location is defined using `__shared__` directive. In this example the shared memory of ten float elements is defined. Normally, the size of shared memory should be equal to the number of threads per block. Here, we are working on an array of 10, and hence we have taken the shared memory of this size.

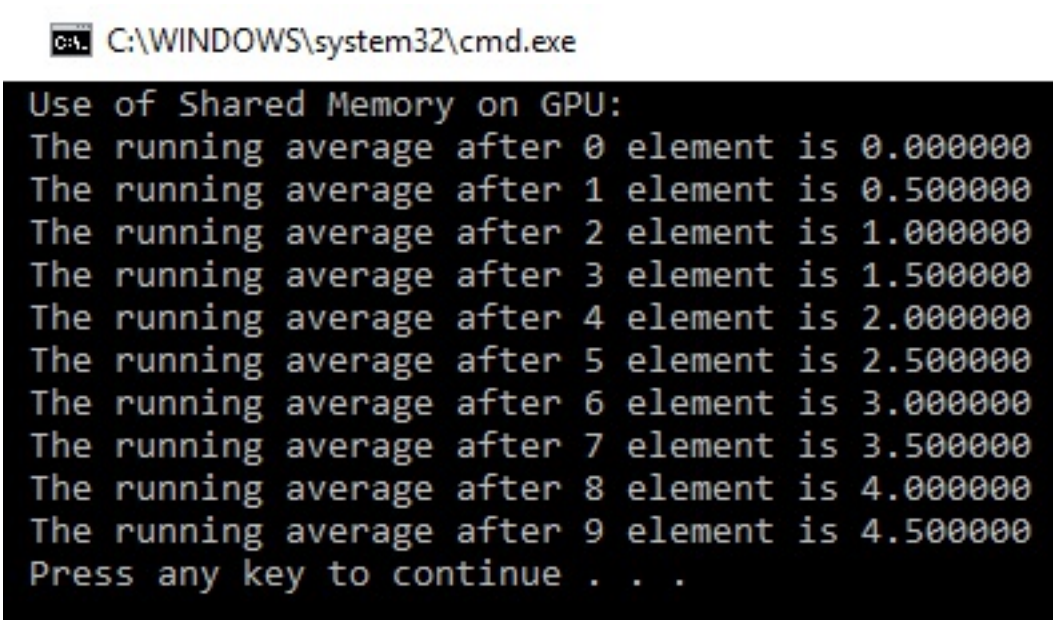
The next step is to copy data from global memory to this shared memory. All the threads copy the element indexed by its thread id to shared array. Now, this is a shared memory write operation and in the next line, we will read from this shared array. So before proceeding, we should ensure that all share memory write operation is completed. so, let us get introduced to the `__synchronizethreads()` barrier.

Next, the for loop calculates the average of all elements up to current elements using the values

in shared memory and stores the answer in global memory, indexed by current thread id. The last line copies the calculated value in shared memory also. This line will have no effect on the overall execution of the code, because shared memory has a lifetime up till the end of the current block execution, and this is the last line after which block execution is complete. It is just used to demonstrate this concept about shared memory. Now, we will try to write the main function for this code as follows:

```
int main(int argc, char **argv)
{
      float h_a[10];
      float *d_a;

      //Initialize host Array
      for (int i = 0; i < 10; i++)
      {
        h_a[i] = i;
      }

    // allocate global memory on the device
      cudaMalloc((void **)&d_a, sizeof(float) * 10);

       // copy data from host memory  to device memory
      cudaMemcpy((void *)d_a, (void *)h_a, sizeof(float) * 10,         cudaMemcpyHostToDevice);
      gpu_shared_memory << <1, 10 >> >(d_a);

    // copy the modified array back to the host
      cudaMemcpy((void *)h_a, (void *)d_a, sizeof(float) * 10, cudaMemcpyDeviceToHost);
      printf("Use of Shared Memory on GPU: \n");

      for (int i = 0; i < 10; i++)
      {
        printf("The running average after %d element is %f \n", i, h_a[i]);
      }
      return 0;
}
```

In the main function, after allocating memory for host and device arrays, host array is populated with values from 0 to 9. This is copied to device memory where moving average is calculated and the result is stored. The result from device memory is copied back to host memory and then printed on the console. The output on the console is shown as follows:



This section demonstrated the use of shared memory when multiple threads use data from the same memory location. The next section demonstrates the use of atomic operations, which are very important in read-modified write operations.

Atomic Operations

Consider a situation in which a large number of threads tries to modify a small portion of memory. This is a frequently occurring phenomenon. It creates more problems when we try to perform read-modify write operation. The example of this operation is `d_out[i] ++` where first d_out[i] is read from memory, then incremented and then written back to the memory. However, when multiple threads are doing this operation on the same memory location, it can give wrong output.

Suppose one memory location has initial value 6 and threads p and q try to increment this memory location, then the final answer should be 8. But at time of execution, it may happen that both p and q thread read this value simultaneously, then both will get value 6. They increment it to 7 and both will store this 7 on memory. So instead of 8, our final answer is 7, which is wrong. How this can be dangerous is understood by taking an example of ATM cash withdrawal. Suppose you have a balance of 5000 in your account. You have two ATM cards of the same account. You and your friend go to two different ATMs simultaneously to withdraw Rs. 4000. Both of you swipe card simultaneously; so, when ATM checks for balance both will get Rs. 5000 balance. When both of you withdraw Rs. 4000 then both machines will look at the initial balance, which was Rs 5000. The amount to withdraw is less than the balance, and hence both machines will give Rs. 4000. Even though your balance was Rs 5000, you got Rs 8000 which is dangerous. To demonstrate this phenomenon, one example of large threads trying to access small array is taken. The kernel function for example is shown as follows:

```
include <stdio.h>

#define NUM_THREADS 10000
#define SIZE 10

#define BLOCK_WIDTH 100

__global__ void gpu_increment_without_atomic(int *d_a)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Each thread increment elements which wraps at SIZE
  tid = tid % SIZE;
  d_a[tid] += 1;
}
```

The kernel function is just incrementing memory location in the line `d_a[tid] +=1`. The issue is how many times this memory location is incremented. Total number of threads are 10000 and array is only of size 10. We are indexing array by taking modulo operation between thread id and size of an Array. So, 1000 threads will try to increment the same memory location. Ideally, every location in the Array should be incremented 1000 times. But as we will see in output this is not the case. Before seeing output we will try to write the main function:

```
int main(int argc, char **argv)
{
  printf("%d total threads in %d blocks writing into %d array elements\n",
  NUM_THREADS, NUM_THREADS / BLOCK_WIDTH, SIZE);

  // declare and allocate host memory
  int h_a[SIZE];
```

```
  const int ARRAY_BYTES = SIZE * sizeof(int);
  // declare and allocate GPU memory
  int * d_a;
  cudaMalloc((void **)&d_a, ARRAY_BYTES);

  // Initialize GPU memory with zero value.
  cudaMemset((void *)d_a, 0, ARRAY_BYTES);
  gpu_increment_without_atomic << <NUM_THREADS / BLOCK_WIDTH, BLOCK_WIDTH >> >(d_a);

  // copy back the array of sums from GPU and print
  cudaMemcpy(h_a, d_a, ARRAY_BYTES, cudaMemcpyDeviceToHost);

  printf("Number of times a particular Array index has been incremented without atomic add is: \n");
  for (int i = 0; i < SIZE; i++)
  {
    printf("index: %d --> %d times\n ", i, h_a[i]);
  }
  cudaFree(d_a);
  return 0;
}
```

In the main function, the device array is declared and initialized to zero. Here, a special function cudaMemSet is used to initialize memory on the device. This is passed as a parameter to the kernel, which increments these 10 memory locations. Here, total 10000 threads are launched as 1000 blocks and 100 threads per block. The answer stored on the device after kernel execution is copied back to host, and the value of each memory location is displayed on the console. The output is as follows:

As discussed, ideally, each memory location should have been incremented 1000 times, but most of the memory locations are having values of 16 and 17. This is because many threads are reading same locations simultaneously and hence incrementing the same value and storing it on memory. As the timing of thread execution is beyond the control of the programmer, how many times simultaneous memory access will happen is not known. If you run your program second time, then will your output be same as first time? Your output might look like the following one:

As you might have guessed, every time you run your program, the memory locations may have different values. This happens because of random execution of all threads on the device.

To solve this problem, CUDA provides an API called `atomicAdd` operations. It is a blocking operation, which means that when multiple threads try to access same memory location, only one thread can access memory location at a time. Other threads have to wait for this thread to finish and write its answer on memory. The kernel function to use `atomicAdd` operation is shown as follows:

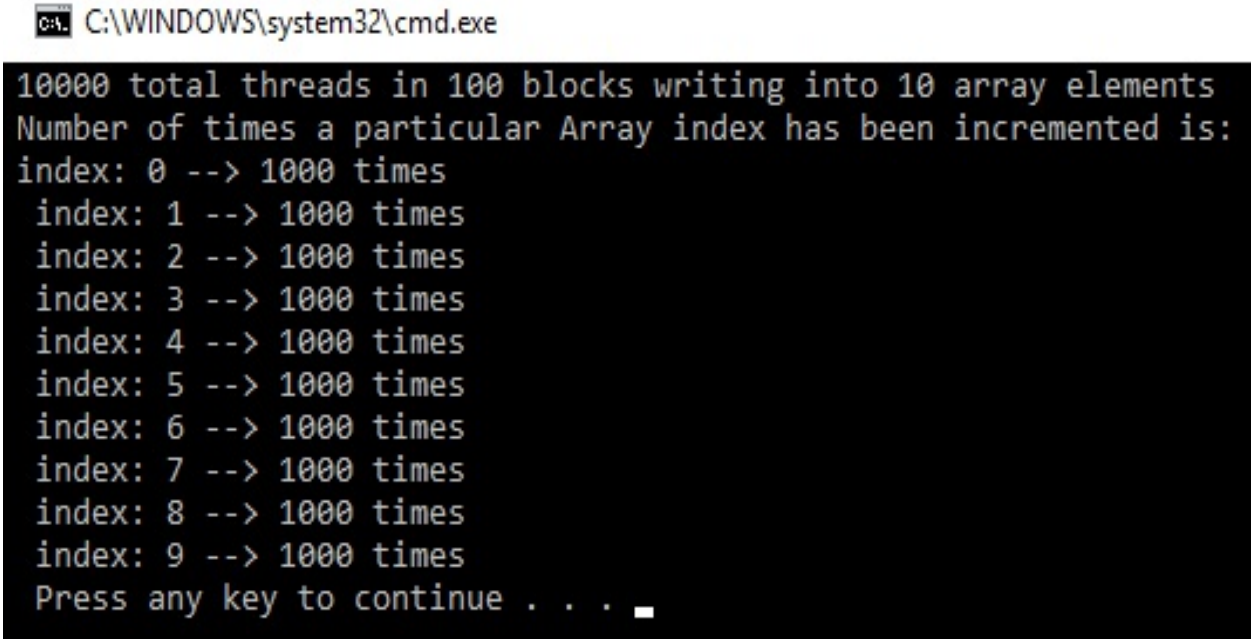
```
#include <stdio.h>
#define NUM_THREADS 10000
#define SIZE 10
#define BLOCK_WIDTH 100

__global__ void gpu_increment_atomic(int *d_a)
{
  // Calculate thread index
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Each thread increments elements which wraps at SIZE
  tid = tid % SIZE;
  atomicAdd(&d_a[tid], 1);
}
```

The kernel function is almost similar to what we have seen earlier. Instead of incrementing memory location using += operator, the `atomicAdd` function is used. It takes two arguments. First is the memory location we want to increment and second is the value by which this location has to be incremented. In this code, again 1000 threads will try to access the same location; so when one thread is using this location other 999 threads have to wait. This will increase the cost in terms of execution time. The main function of increment using atomic operations is shown as follows:

```
int main(int argc, char **argv)
{
```

```
  printf("%d total threads in %d blocks writing into %d array elements\n",
    NUM_THREADS, NUM_THREADS / BLOCK_WIDTH, SIZE);

  // declare and allocate host memory
  int h_a[SIZE];
  const int ARRAY_BYTES = SIZE * sizeof(int);

  // declare and allocate GPU memory
  int * d_a;
  cudaMalloc((void **)&d_a, ARRAY_BYTES);

   // Initialize GPU memory withzero value
  cudaMemset((void *)d_a, 0, ARRAY_BYTES);

  gpu_increment_atomic << <NUM_THREADS / BLOCK_WIDTH, BLOCK_WIDTH >> >(d_a);
    // copy back the array from GPU and print
  cudaMemcpy(h_a, d_a, ARRAY_BYTES, cudaMemcpyDeviceToHost);

  printf("Number of times a particular Array index has been incremented is: \n");
  for (int i = 0; i < SIZE; i++)
  {
    printf("index: %d --> %d times\n ", i, h_a[i]);
  }

  cudaFree(d_a);
  return 0;
}
```

In the main function, the array with 10 elements is initialized with zero value and passed to the kernel. But now, kernel will do atomic add operation. So, the output of this program should be accurate. Each element in the array should be incremented 1000 times. The following will be the output:



If you measure the execution time of program with atomic operations, it may take longer time than that taken by simple programs using global memory. This is because many threads are waiting for memory access in the atomic operation. Use of shared memory can help to speed up operation. Also, if the same number of threads are accessing more memory locations, then the atomic operation will incur less time overhead as less number of threads have to wait for memory access.

So in this section, we have seen that atomic operations help in avoiding race conditions in memory operations and make the code simpler to write and understand. In the next section, we will explain two special types of memories, constant and texture, which help in accelerating certain type of code.

Constant Memory

The CUDA language makes another type of memory available to the programmer, which is known as constant memory. NVIDIA hardware provides 64 KB of this constant memory which is used to store data that remains constant throughout the execution of the kernel. This constant memory is cached on-chip so that the use of constant memory instead of global memory can speed up execution. The use of constant memory will also reduce memory bandwidth to device global memory. In this section, we will see how to use constant memory in CUDA programs. A simple program that performs simple math operation a*x + b, where a and b are constants, is taken as example. The kernel function code for this program is shown below:

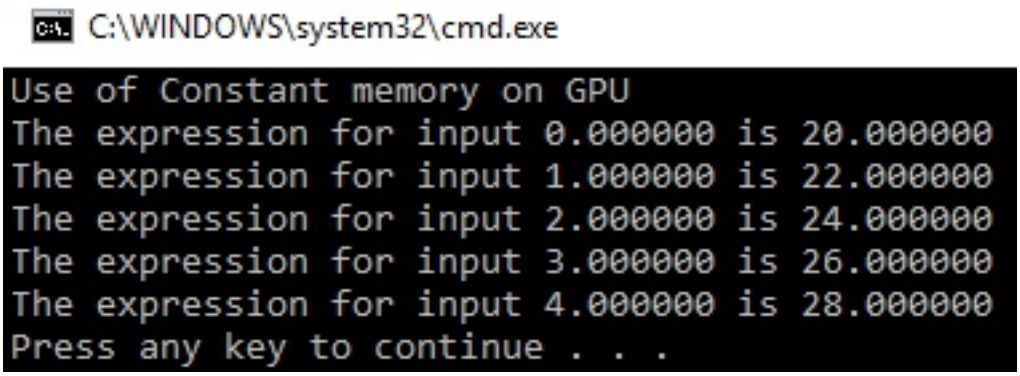
```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>

//Defining two constants
__constant__ int constant_f;
__constant__ int constant_g;
#define N 5

//Kernel function for using constant memory
__global__ void gpu_constant_memory(float *d_in, float *d_out)
{
  //Getting thread index for current kernel
  int tid = threadIdx.x;
  d_out[tid] = constant_f*d_in[tid] + constant_g;
}
```

Constant memory variables are defined using `__constant__` keyword. In the preceding code, two float variables, constant_f and constant_g, are defined as constants which will not change throughout kernel execution. The second thing to note is that once variables are defined as constants, they should not be defined again in the kernel function. The kernel function computes a simple mathematical operation using these two constants. There is a special way in which constant variables are copied to memory from the main function. This is shown in the following code:

```
int main(void)
{
  //Defining Arrays for host
  float h_in[N], h_out[N];
  //Defining Pointers for device
  float *d_in, *d_out;
  int h_f = 2;
  int h_g = 20;

  // allocate the memory on the cpu
  cudaMalloc((void**)&d_in, N * sizeof(float));
  cudaMalloc((void**)&d_out, N * sizeof(float));

  //Initializing Array
  for (int i = 0; i < N; i++)
  {
    h_in[i] = i;
  }

  //Copy Array from host to device
  cudaMemcpy(d_in, h_in, N * sizeof(float), cudaMemcpyHostToDevice);
  //Copy constants to constant memory
  cudaMemcpyToSymbol(constant_f, &h_f, sizeof(int),0,cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(constant_g, &h_g, sizeof(int));
```

```
  //Calling kernel with one block and N threads per block
  gpu_constant_memory << <1, N >> >(d_in, d_out);

  //Coping result back to host from device memory
  cudaMemcpy(h_out, d_out, N * sizeof(float), cudaMemcpyDeviceToHost);

  //Printing result on console
  printf("Use of Constant memory on GPU \n");
  for (int i = 0; i < N; i++)
  {
    printf("The expression for index %f is %f\n", h_in[i], h_out[i]);
  }

  cudaFree(d_in);
  cudaFree(d_out);
  return 0;
}
```

In the main function, two constants h_f and h_g are defined and initialized on the host, which will be copied to constant memory. The instruction cudaMemcpyToSymbol is used to copy these constants on to constant memory for kernel execution. It has five arguments. First is the destination which is defined using `__constant__` keyword. Second is the host address, third is the size of the transfer, fourth is memory offset, which is taken as zero, and fifth is the direction of data transfer, which is taken as host to device. The last two arguments are optional and hence they are omitted in the second call to cudaMemcpyToSymbol instruction. The output of the code is shown as follows:



One thing to note is that constant memory is a Read-only memory. This example is used just to explain the use of constant memory from CUDA program. It is not the optimal use of constant memory. As discussed earlier, constant memory helps in conserving memory bandwidth to global memory. To understand this, you have to understand the concept of warp. One warp is a collection of 32 thread woven together and gets executed in lockstep. A single read from constant memory can be broadcasted to half warp which can reduce up to 15 memory transactions. Also, constant memory is cached so that memory access to nearby location will not incur additional memory transaction. When each half warp, which contains 16 threads, operates on same memory locations, the use of constant memory saves a lot of execution time. It should also be noted that if half warp threads use completely different memory locations then use of constant memory may increase the execution time. So the constant memory should be used with proper care.

Texture Memory

Texture memory is another read-only memory available that can accelerate the program and reduce memory bandwidth when data are read in a certain pattern. Like constant memory, it is also cached on a chip. This memory was originally designed for rendering graphics but it can also be used for general purpose computing applications. It is very effective when applications have memory access that exhibits a great deal of spatial locality. The meaning of spatial locality is that each thread is likely to read from the nearby location of what other nearby threads read. This is great in image processing applications where we work on 4-point connectivity and 8-point connectivity. A 2-d spatial locality for accessing memory location by threads may look something like this:

Thread 0	Thread 2
Thread 1	Thread 3

General global memory cache will not be able to capture this spatial locality and will result in lots of memory traffic to global memory. Texture cache is designed for this kind of access pattern so that it will only read from memory once, and then it will be cached so that execution will be much faster. Texture memory supports 1-D and 2-D fetch operations. Using texture memory in your CUDA program is not trivial especially for those who are not programming experts. In this section, a simple example of how to copy array values using texture memory is explained. The kernel function for using texture memory is explained as follows:

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>

#define NUM_THREADS 10
#define N 10

//Define texture reference for 1-d access
texture <float, 1, cudaReadModeElementType> textureRef;

__global__ void gpu_texture_memory(int n, float *d_out)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx < n) {
    float temp = tex1D(textureRef, float(idx));
    d_out[idx] = temp;
  }
}
```

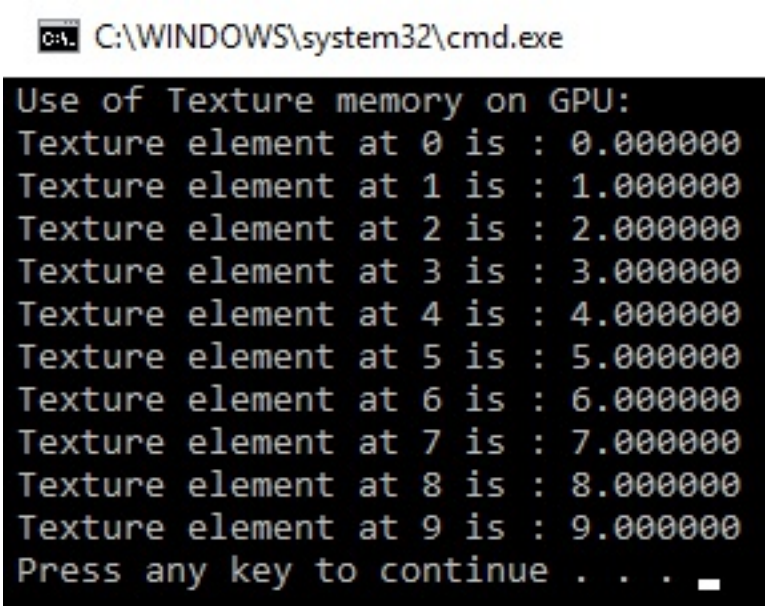
The part of texture memory that should be fetched is defined by texture reference. In code, it is defined using texture API. It has three arguments. The first argument indicates the data type of texture elements. In this example, it is a float. The second argument indicates the type of texture reference which can be 1-D, 2-D etc. Here, it is 1-D reference. The third argument specifies the read mode and it is optional arguments. Please make sure that this texture reference is declared as static global variable and it should not be passed as parameters to any function. In kernel function, data stored at thread id is read from this texture reference and copied to global memory

pointer d_out. Here, we are not using any spatial locality, this example is only taken to show you how to use texture memory from CUDA programs. The spatial locality will be explained in next chapter when we see some image processing applications with CUDA. The main function for this example is shown as follows:

```
int main()
{
  //Calculate number of blocks to launch
  int num_blocks = N / NUM_THREADS + ((N % NUM_THREADS) ? 1 : 0);
  float *d_out;
  // allocate space on the device for the results
  cudaMalloc((void**)&d_out, sizeof(float) * N);
  // allocate space on the host for the results
  float *h_out = (float*)malloc(sizeof(float)*N);
  float h_in[N];
  for (int i = 0; i < N; i++)
  {
    h_in[i] = float(i);
  }
  //Define CUDA Array
  cudaArray *cu_Array;
  cudaMallocArray(&cu_Array, &textureRef.channelDesc, N, 1);

  cudaMemcpyToArray(cu_Array, 0, 0, h_in, sizeof(float)*N, cudaMemcpyHostToDevice);

  // bind a texture to the CUDA array
  cudaBindTextureToArray(textureRef, cu_Array);


  gpu_texture_memory << <num_blocks, NUM_THREADS >> >(N, d_out);

  // copy result to host
  cudaMemcpy(h_out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);
  printf("Use of Texture memory on GPU: \n");
  // Print the result
  for (int i = 0; i < N; i++)
  {
    printf("Average between two nearest element is : %f\n", h_out[i]);
  }
  free(h_out);
  cudaFree(d_out);
  cudaFreeArray(cu_Array);
  cudaUnbindTexture(textureRef);
}
```

In the main function, after declaring and allocating memory for host and device arrays, host array is initialized with values from 0 to 9. In this example, you will see the first use of CUDA arrays. They are similar to normal arrays but they are dedicated to textures. They are read only to kernel functions and can be written to device memory from host by using cudaMemcpyToArray function, as shown in the code. The second and third argument in that function are width and height offset which is taken as 0, 0, meaning we are starting from the top left corner. They are opaque memory layouts optimized for texture memory fetches.

cudaBindTextureToArray functions bind texture reference to this CUDA Array. It means it copies this array to texture reference starting from top left corner. After binding texture reference, the kernel is called which uses this texture reference and computes the Array to be stored on device memory. After kernel finishes, output Array is copied back to host for displaying on the console. When using texture memory, we have to unbind texture from our code. This is done by using cudaUnbindTexture function. cudaFreeArray function is used to free up memory used by CUDA Array. The output of the program displayed on console is shown as follows:



This section finishes our discussion on memory architecture in CUDA. When memories available in CUDA are used judiciously according to your application, it improves the performance of the program drastically. You need to carefully look at the memory access pattern of all threads in your application and then select on which memory you should use for your application. The last section of this chapter describes a little bit complex CUDA program which uses all concepts we have used up till this point.

Dot Product and Matrix Multiplication Example

Up to this point, we have learned almost all important concepts related to Basic parallel programming using CUDA. In this section, we will show you how to write CUDA programs for important mathematical operations like dot product and matrix multiplication, which is used in almost all application. This will make use of all concepts we have seen earlier and help you in writing code for your applications.

Dot Product

The dot product between two vectors is an important mathematical operation. It will also explain one important concept in CUDA programming which is called reduction operation. Dot product between two vectors can be defined as follows:

```
(x1,x1,x3) . (y1,y2,y3) = x1y1 + x2y2 +x3y3
```

Now, if you see this operation, it is very similar to element-wise addition operation on vectors. Instead of addition, you have to perform element-wise multiplication. All threads also have to keep running sum of multiplication they have performed because all individual multiplication needs to be summed up to get a final answer of dot product. The answer of dot product will be a single number. This operation where the final answer is the reduced version of original two arrays is called as reduce operation in CUDA. It is useful in many applications. To perform this operation in CUDA, we will start by writing a kernel function for it as follows

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
#define N 1024
#define threadsPerBlock 512


__global__ void gpu_dot(float *d_a, float *d_b, float *d_c)
{
  //Define Shared Memory
  __shared__ float partial_sum[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int index = threadIdx.x;

  float sum = 0;
  while (tid < N)
  {
    sum += d_a[tid] * d_b[tid];
    tid += blockDim.x * gridDim.x;
  }

  // set the partial sum in shared memory
  partial_sum[index] = sum;

  // synchronize threads in this block
  __syncthreads();

  //Calculate Patial sum for a current block using data in shared memory
  int i = blockDim.x / 2;
  while (i != 0) {
    if (index < i)
      {partial_sum[index] += partial_sum[index + i];}
    __syncthreads();
    i /= 2;
  }
  //Store result of partial sum for a block in global memory
  if (index == 0)
    d_c[blockIdx.x] = partial_sum[0];
}
```

The kernel function takes two input Arrays as input and stores the final partial sum in the third Array. Shared memory is defined to store intermediate answers of the partial answer. The size of the shared memory is taken equal to the number of threads per block as all separate blocks will

have the separate copy of this shared memory. After that, two indexes are calculated; the first one which calculates unique thread id is similar to what we have done in vector addition example. The second index is used to store partial product answer on shared memory. Again, every block has a separate copy of shared memory so only thread id is used to index into the shared memory of a given block.

The while loop will perform element-wise multiplication of elements indexed by thread id. It will also do multiplication of elements which is offset by total threads to the current thread id. The partial sum of this element is stored in the shared memory. We are going to use these results from shared memory to calculate the partial sum for a single block. So, before reading this shared memory block, we must ensure that all threads have finished writing to this shared memory. This is ensured by using `__syncthreads()` directive.

Now, one method to get an answer of the dot product is that one thread iterates over all these partial sums to get a final answer. One thread can perform reduce operation. This will take N operations to complete where N is the number of partial sums to be added (equal to the number of threads per block) to get a final answer.

The question is can we do this reduce operation in parallel? The answer is yes. The idea is that every thread will add two elements of the partial sum and store the answer in the location of the first element. Since each thread combines two entries in one, the operation can be completed in half entries. Now, we will repeat this operation for the remaining half until we get final answer that calculates the partial sum for this entire block. The complexity of this operation is $\log_2(N)$ which is far better than the complexity of N when one thread performs reduce operation.

The operation explained is calculated by the block starting with `while (i != 0)` . The block sums partial answer of current thread and thread offset by blockdim/2. It continues this addition till we get a final single answer which is a sum of all partial products in a given block. The final answer is stored in the global memory. Each block will have a separate answer to be stored on global memory so that it is indexed by block id which is unique for each block. Still, we have not got the final answer. This can be performed in device function or the main function.

Normally, last few additions in reduce operation need very little resources. Much of the GPU resource remains idle and that is not the optimal use of GPU. So, final addition operation of all partial sums for an individual block is done in the main function. The main function is as follows:

```
int main(void)
{
  float *h_a, *h_b, h_c, *partial_sum;
  float *d_a, *d_b, *d_partial_sum;

  //Calculate number of blocks and number of threads
  int block_calc = (N + threadsPerBlock - 1) / threadsPerBlock;
  int blocksPerGrid = (32 < block_calc ? 32 : block_calc);
  // allocate memory on the cpu side
  h_a = (float*)malloc(N * sizeof(float));
  h_b = (float*)malloc(N * sizeof(float));
  partial_sum = (float*)malloc(blocksPerGrid * sizeof(float));

  // allocate the memory on the gpu
  cudaMalloc((void**)&d_a, N * sizeof(float));
  cudaMalloc((void**)&d_b, N * sizeof(float));
  cudaMalloc((void**)&d_partial_sum, blocksPerGrid * sizeof(float));

  // fill in the host mempory with data
```

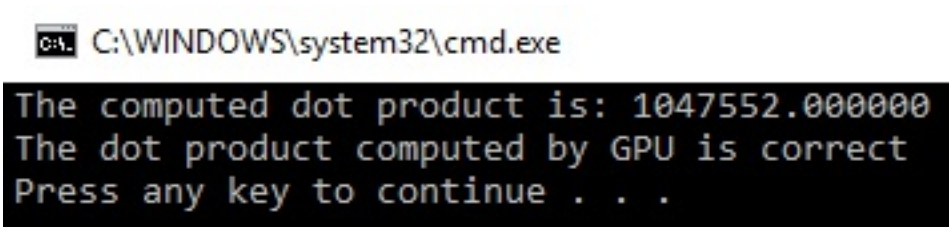
```
  for (int i = 0; i<N; i++) {
    h_a[i] = i;
    h_b[i] = 2;
  }


  // copy the arrays to the device
  cudaMemcpy(d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice);

  gpu_dot << <blocksPerGrid, threadsPerBlock >> >(d_a, d_b, d_partial_sum);

  // copy the array back to the host
  cudaMemcpy(partial_sum, d_partial_sum, blocksPerGrid * sizeof(float), cudaMemcpyDeviceToHost);

  // Calculate final dot prodcut
  h_c = 0;
  for (int i = 0; i<blocksPerGrid; i++)
 {
    h_c += partial_sum[i];
  }

}
```

Three arrays are defined and memory is allocated for both host and device to store inputs and output. The two host arrays are initialized inside a for loop. One array is initialized with 0 to N and second is initialized with a constant value 2. The calculation of the number of blocks in a grid and number of threads in a block is also done. It is similar to what we have done in the starting of this chapter. keep in mind, you can also keep this value as constants like we have done in the first program of this chapter to avoid complexity.

These arrays are copied to device memory and passed as parameters to the kernel function. The kernel function will return an array which has answers of partial products of individual blocks indexed by their block id. This array is copied back to host in partial_sum array. The final answer of the dot product is calculated by iterating over this partial_sum array using for loop starting from 0 to the number of blocks per grid. The final dot product is stored in h_c. To check whether the calculated dot product is correct or not, follwing code can be added to main function.

```
printf("The computed dot product is: %f\n", h_c);
#define cpu_sum(x) (x*(x+1))
  if (h_c == cpu_sum((float)(N - 1)))
  {
    printf("The dot product computed by GPU is correct\n");
  }
  else
  {
    printf("Error in dot product computation");
  }
  // free memory on the gpu side
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_partial_sum);
  // free memory on the cpu side
  free(h_a);
  free(h_b);
  free(partial_sum);
```

The answer is verified with answer calculated mathematically of two input arrays . In two input Arrays, if one Array has values from 0 to N-1 and second Array has constant value 2, then dot product will be N*(N+1). We are printing answer of the dot product along with whether it is calculated correctly or not. The host and device memory is freed up in the end.The output of the program is as follows:


C:\WINDOWS\system32\cmd.exe
The computed dot product is: 1047552.000000
The dot product computed by GPU is correct
Press any key to continue . . .

Matrix Multiplication

The second most important mathematical operation performed on GPU using CUDA is matrix multiplication. It is a very complicated mathematical operation when sizes of the matrix are very large. It should be kept in mind that for matrix multiplication, number of columns in the first matrix should be equal to the number of rows in the second matrix. Matrix multiplication is not a cumulative operation. To avoid complexity, in this example, we are taking a square matrix of same size. If you are familiar with the mathematics of matrix multiplication, then you may recall that a row in the first matrix will be multiplied with all the columns in the second matrix. This is repeated for all rows in the first matrix. It is shown as follows:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0*0+0*0+0*0+0*0 & 0 & 0 & 0 \\ 1*0+1*0+1*0+1*0 & 4 & 8 & 12 \\ 2*0+2*0+2*0+2*0 & 8 & 16 & 24 \\ 3*0+3*0+3*0+3*0 & 12 & 24 & 36 \end{bmatrix}$$

Same data are reused many times. So this is an ideal case of using shared memory. In this section, we will make two separate kernel functions with and without using shared memory. You can compare the execution of two kernels to get an idea of how shared memory improve the performance of the program. We will first start by writing a kernel function without using shared memory.

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>

//This defines size of a small sqaure box or thread dimensions in one block
#define TILE_SIZE 2

//Matrix multiplication using non shared kernel
__global__ void gpu_Matrix_Mul_nonshared(float *d_a, float *d_b, float *d_c, const int size)
{
  int row, col;
  col = TILE_SIZE * blockIdx.x + threadIdx.x;
  row = TILE_SIZE * blockIdx.y + threadIdx.y;

  for (int k = 0; k< size; k++)
  {
    d_c[row*size + col] += d_a[row * size + k] * d_b[k * size + col];
  }
}
```

Matrix multiplication is performed using two-dimensional threads. If we launch two-dimensional threads with each thread performing a single element of the output matrix then up to 16x16 matrix can be multiplied. If the size is greater than this, then it will need more than 512 threads for computation, which is not possible on most GPUs. So, we need to launch multiple blocks with each containing less than 512 threads. To accomplish this, output matrix is divided into small square blocks having dimensions of TILE_SIZE in both directions. Each thread in a block will calculate elements of this square block. The total number of blocks for matrix multiplication

will be calculated by dividing the size of the matrix by size of this small square defined by TILE_SIZE.

If you could understand this then calculating row and column index for output will be very easy. It is similar to what we have done up till now with blockdim.x will be equal to TILE_SIZE. Now every element in the output will be the dot product of one raw in the first matrix and one column in the second matrix. Both the matrix have the same size so dot product has to be performed for a number of elements equal to size variable. So for loop in the kernel function is running from 0 to size.

To calculate the individual index of both matrices consider that this matrix is stored as a linear Array in system memory in row-major fashion. Its meaning is that all elements in first row are are placed in a consecutive memory location and then rows are placed one after the other as follows:

$$\begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \longrightarrow [M_{00} \quad M_{01} \quad M_{10} \quad M_{11}]$$

Index of linear Array can be calculated by its row id multiply by the size of matrix plus its column id . So index for $M_{1,0}$ will be 2 as its row id is 1, matrix size is 2 and column id is zero. This method is used to calculate element index in both the matrix.

To calculate element at [row, col] in the resultant matrix, the index in the first matrix will be equal to row*size + k and for second matrix it will be k*size + col. This is a very simple kernel function. There is a large amount of data reuse in matrix multiplication. This function is not utilizing the advantage of shared memory. So we will try to modify the kernel function that makes use of shared memory. The modified kernel function is shown as follows:

```
// shared
__global__ void gpu_Matrix_Mul_shared(float *d_a, float *d_b, float *d_c, const int size)
{
  int row, col;

  __shared__ float shared_a[TILE_SIZE][TILE_SIZE];

  __shared__ float shared_b[TILE_SIZE][TILE_SIZE];

  // calculate thread id
  col = TILE_SIZE * blockIdx.x + threadIdx.x;
  row = TILE_SIZE * blockIdx.y + threadIdx.y;

  for (int i = 0; i< size / TILE_SIZE; i++)
  {
    shared_a[threadIdx.y][threadIdx.x] = d_a[row* size + (i*TILE_SIZE + threadIdx.x)];
    shared_b[threadIdx.y][threadIdx.x] = d_b[(i*TILE_SIZE + threadIdx.y) * size + col];
    }
    __syncthreads();


    for (int j = 0; j<TILE_SIZE; j++)
      d_c[row*size + col] += shared_a[threadIdx.x][j] * shared_b[j][threadIdx.y];
    __syncthreads(); // for syncronizeing the threads

  }
}
```

Two shared memory with size equal to the size of a small square block which, is TILE_SIZE, is defined for storing data for reuse. Row and column indexes are calculated in the same way as seen earlier. First, this shared memory is filled up in the first for loop. After that __syncthreads() is included so that memory read from shared memory only happens when all threads have finished writing to it. The last for loop again calculates dot product. As this is done by only using shared memory, this considerably reduces memory traffic to global memory which in turns improves performance of the program for larger matrix dimensions. The main function for this program is shown as follows:

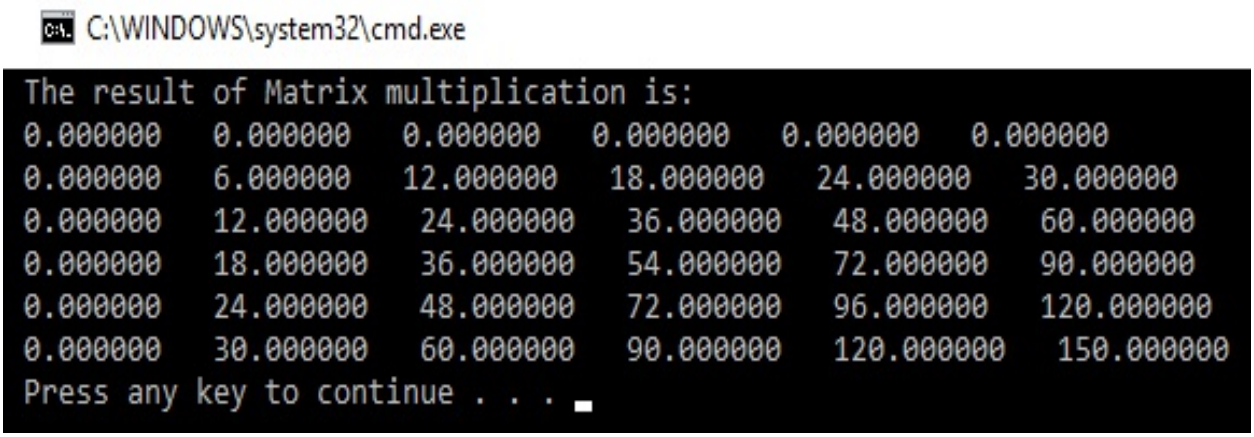
```
int main()
{
   //Define size of the matrix
  const int size = 4;
   //Define host and device arrays
  float h_a[size][size], h_b[size][size],h_result[size][size];
  float *d_a, *d_b, *d_result; // device array
  //input in host array
  for (int i = 0; i<size; i++)
  {
    for (int j = 0; j<size; j++)
    {
      h_a[i][j] = i;
      h_b[i][j] = j;
    }
  }


  cudaMalloc((void **)&d_a, size*size*sizeof(int));
  cudaMalloc((void **)&d_b, size*size * sizeof(int));
  cudaMalloc((void **)&d_result, size*size* sizeof(int));
  //copy host array to device array
  cudaMemcpy(d_a, h_a, size*size* sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, size*size* sizeof(int), cudaMemcpyHostToDevice);
  //calling kernal
  dim3 dimGrid(size / TILE_SIZE, size / TILE_SIZE, 1);
  dim3 dimBlock(TILE_SIZE, TILE_SIZE, 1);

  gpu_Matrix_Mul_nonshared << <dimGrid, dimBlock >> > (d_a, d_b, d_result, size);
  //gpu_Matrix_Mul_shared << <dimGrid, dimBlock >> > (d_a, d_b, d_result, size);

  cudaMemcpy(h_result, d_result, size*size * sizeof(int), cudaMemcpyDeviceToHost);

  return 0;
}
```

After defining and allocating memory for host and device arrays, host Array is filled with some random values. These Arrays are copied to device memory so that it can be passed to kernel functions. The number of grid blocks and number of block threads are defined using the dim3 structure with dimensions equal to that calculated earlier. You can call any of the kernel. The calculated answer is copied back to host memory.To display output on console following code ia added to main function.

```
printf("The result of Matrix multiplication is: \n");

  for (int i = 0; i< size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      printf("%f ", h_result[i][j]);
    }
    printf("\n");
  }
cudaFree(d_a)
cudaFree(d_b)
cudaFree(d_result)
```

The memory used to store matrices on device memory is also freed up. The output on console is

as follows:



This section demonstrated CUDA programs for two important mathematical operations used in wide range of applications. It also explained the use of shared memory and multidimensional threads.

Summary

This chapter explained the launch of multiple blocks with each having multiple threads from kernel function. It also showed the method of choosing these two parameters for a large value of threads. It also explained the hierarchical memory architecture that can be used by CUDA programs. The memory nearer to thread execution is fast and as we move away from it memories get slower. When multiple threads want to communicate with each other then CUDA provides the flexibility of using shared memory by which threads from same blocks can communicate with each other. When multiple threads use same memory location then there should be synchronization between this memory access otherwise the final result will not be as expected. We have also seen use of atomic operation to accomplish this synchronization. If some parameters are remaining constant throughout the kernel execution then it can be stored in constant memory for speed up. When CUDA programs exhibit a certain communication pattern like spatial locality then texture memory should be used to improve the performance of the program. To summarize, to improve the performance of CUDA programs we should reduce memory traffic to slow memories. If this is done efficiently drastic improvement in the performance of the program can be achieved.

In next chapter, the concept of CUDA streams is discussed which is similar to multitasking in CPU programs. How we can measure the performance of CUDA programs will also be discussed in next chapter. It will also show use of CUDA in simple Image processing applications.

Questions

1. Suppose you want to launch 1,00,000 threads in parallel, then what is the best choice of the number of blocks in a grid and number of threads in blocks and why?
2. Write a CUDA program to find out cube of every element in an Array when the number of elements in Array is 1,00,000.
3. State whether the following statement is True or False with reason: Assignment operator between local variables will be faster than assignment operation between global variables.
4. What is register spilling? How it can harm the performance of your CUDA program.
5. State whether following line of code will give required output or not: d_out[i] = d_out[i-1]
6. State whether the following statement is True or False with reason: Atomic Operations increases the execution time for CUDA Program.
7. Which kind of communication patterns are ideal for using texture memory in your CUDA programs.
8. What will be the effect of using __syncthreads directive inside if statement ?

Advanced concepts in CUDA

Introduction

In the last chapter, we have seen memory architecture in CUDA and how it can be used efficiently to accelerate applications. Up till now, we have not seen a method to measure the performance of CUDA programs. In this chapter, we will discuss how we can measure the performance of CUDA programs using CUDA events. The Nvidia visual profiler will also be discussed. How to resolve errors in CUDA programs from within the CUDA code and using debugging tools will be discussed in this chapter. How we can improve the performance of CUDA programs will also be discussed. This chapter will describe how CUDA streams can be used for multitasking and how we can use it to accelerate applications. You will also know how array sorting algorithms can be accelerated using CUDA. Image processing is an application where we need to process a large amount of data in a very small time so CUDA can be an ideal choice for this kind of application to manipulate pixel values in an image. This chapter describes the acceleration of simple and widely used image processing function histogram calculation using CUDA.

The following topics will be covered in this chapter:

- Performance measurement in CUDA
- Error handling in CUDA
- Performance improvement of CUDA Programs
- CUDA streams and how it can be used to accelerate applications
- Acceleration of sorting algorithms using CUDA
- Introduction to Image Processing Application with CUDA

Technical Requirements

This chapter requires familiarity with the basic C or C++ programming language and all the codes explained in previous chapters. All the code used in this chapter can be downloaded from following github link: https://github.com/PacktPublishing/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA. The code can be executed on any operating system though it has only been tested on Windows 10.

Performance measurement of CUDA Programs

Up till now, we have not determined the performance of the CUDA programs explicitly. In this section, we will see how to measure the performance of CUDA programs using CUDA events and also visualize the performance using Nvidia visual profiler. This is a very important concept in CUDA because it will allow you to chose best-performing algorithms for a particular application from many options. First, we will try to measure performance using CUDA events.

CUDA Events

We can use CPU timer for measuring the performance of CUDA programs but it will not give accurate results. It will include time overhead of thread latency in OS and scheduling in OS among many other factors. The time measured using CPU will also depend on the availability of high precision CPU timer. Many times host is performing asynchronous computation while GPU kernel is running, and hence CPU timers may not give correct time for kernel executions. So, to measure time for GPU kernel computation, CUDA provides an Event API.

A CUDA event is a GPU timestamp recorded at a specified point from your CUDA program. In this API, GPU records the time stamp which eliminates the issues that were present when using CPU timers for measuring performance. There are two steps to measure time using CUDA events: creating an event and recording an event. We can record two events, one at the starting of your code and one at the end. Then we will try to calculate the difference in time between these two events that will give an overall performance of our code.

In your CUDA code, you can include the following lines to measure performance using CUDA event API.

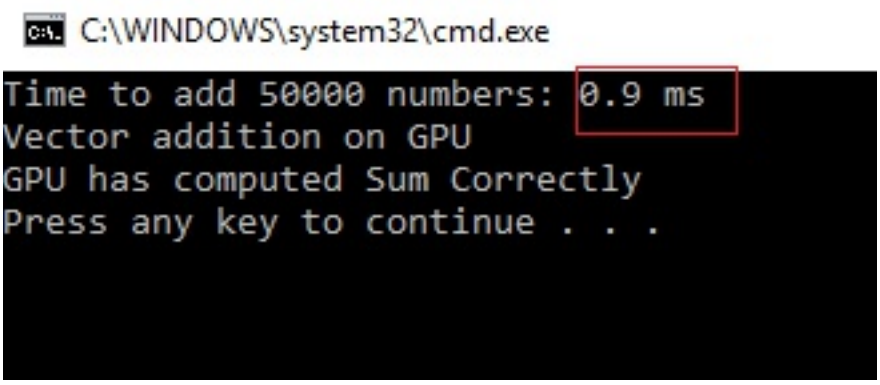
```
cudaEvent_t e_start, e_stop;
cudaEventCreate(&e_start);
cudaEventCreate(&e_stop);
cudaEventRecord(e_start, 0);
//All GPU code for which performance needs to be measured allocate the memory
cudaMalloc((void**)&d_a, N * sizeof(int));
cudaMalloc((void**)&d_b, N * sizeof(int));
cudaMalloc((void**)&d_c, N * sizeof(int));

  //Copy input arrays from host to device memory
cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);

gpuAdd << <512, 512 >> >(d_a, d_b, d_c);
//Copy result back to host memory from device memory
cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
cudaEventRecord(e_stop, 0);
cudaEventSynchronize(e_stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, e_start, e_stop);
printf("Time to add %d numbers: %3.1f ms\n",N, elapsedTime);
```

We are creating two events, e_start and e_stop, for starting and ending of code. cudaEvent_t is used to define event objects. To create an event we will use cudaEventCreate API. We can pass in event objects as an argument to this API. At the beginning of the code, we will record GPU time stamp in e_start event; this will be done using cudaEventRecord API. Second argument to this function is zero which indicates CUDA stream number that we will see later in this chapter.

After recording time stamp in the beginning, you can start writing your GPU code. After the code finishes, we will again record time in e_stop event. This will be done by cudaEventRecord(e_stop, 0) line. Now that we have recorded both start time and end time, the difference between them should give us the actual performance of the code. But there will be one issue in directly calculating the time difference between these two events.

As we have discussed in previous chapters, execution in CUDA C can be asynchronous. When GPU is executing the kernel, CPU might be executing next lines of our code until GPU finishes its execution. So, measuring time directly without syncronizing GPU and CPU may give wrong results. CudaEventRecord() will record time stamp when all GPU instruction prior to its call finishes. We should not read this e_stop event until this point when prior work on GPU is finished. So, to synchronize CPU operation with GPU, we have used cudaEventSynchronize(e_stop). It ensures that correct time stamp is recorded in e_stop event.

Now to calculate the difference between these two time stamps, CUDA provides an API called cudaEventElapsedTime. It has three arguments. First is the variable in which we want to store the difference, second is start event and third is end event. After calculating this time, we are printing it on the console in next line. We have added this performance measurement code to the code of vector addition using multiple threads and blocks seen in the last chapter. The output after adding these lines is as follows:

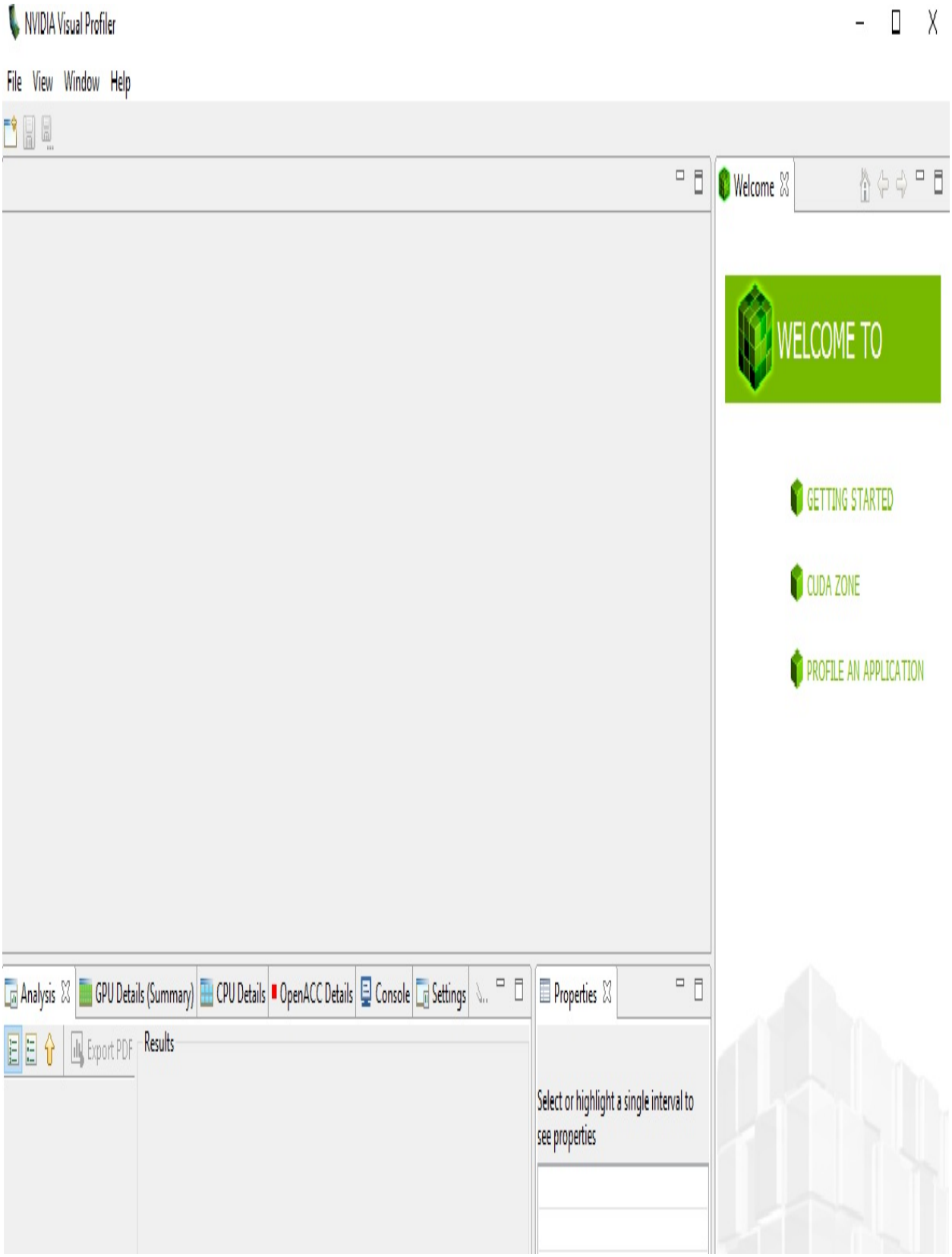


The time taken to add 50,000 elements on GPU is around 0.9ms. This output will depend on your system configurations, and hence you might get different output in the red box. So you can include this performance measurement code in all the codes we have seen in this book to measure the performance of them. You can also quantify performance gains by using constant and texture memories by using this event API.

It should be kept in mind that CUDA events can only be used to measure the timing of device codes. This only includes memory allocation, memory copies, and kernel execution. It should not be used to measure timings of host code. As GPU is recording time in Event API, using it to measure the performance of host code may give wrong results.

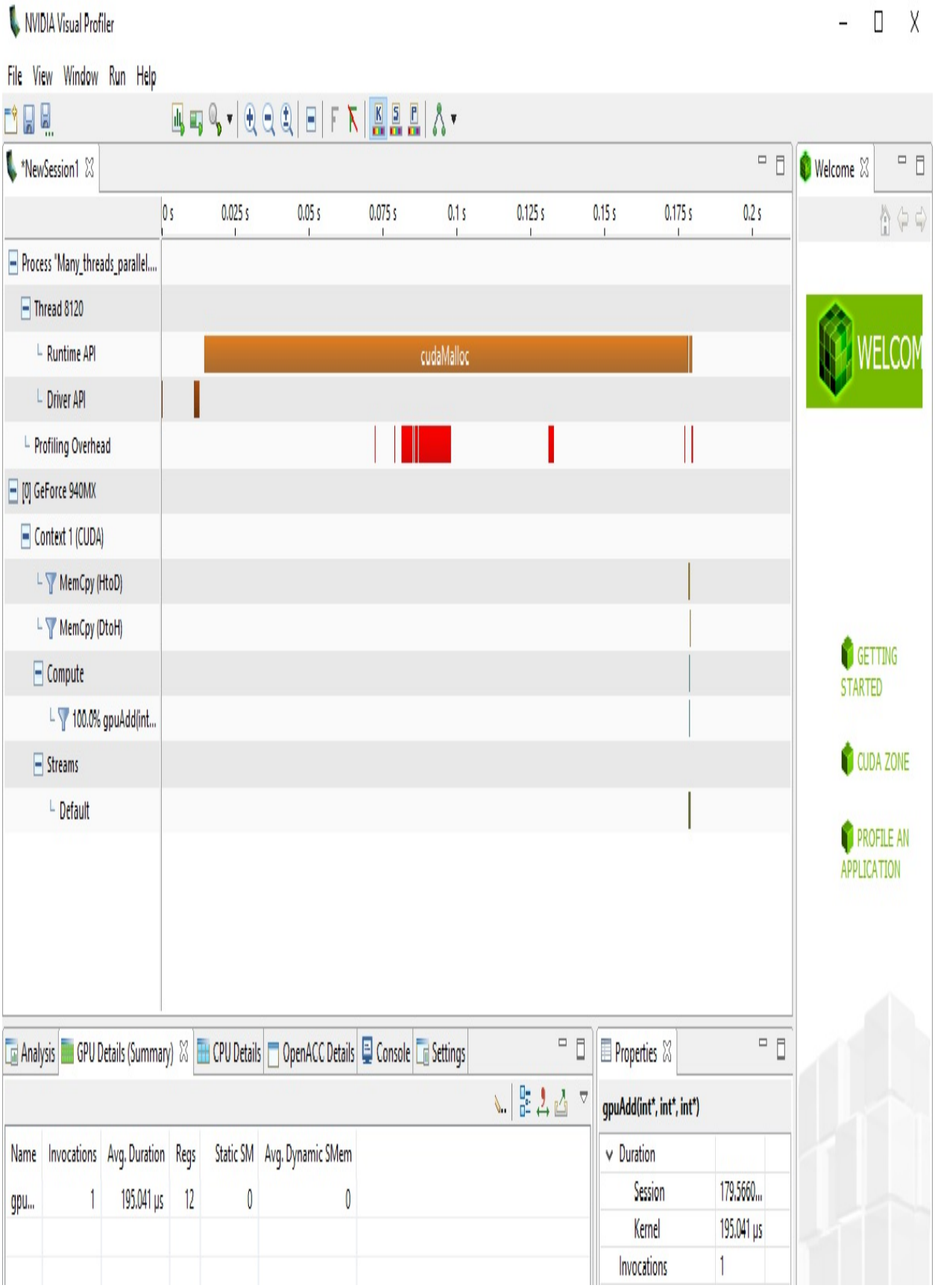
Nvidia Visual Profiler

We now know that CUDA provides an efficient way to improve the performance of parallel computing application. However, sometimes it may happen that even after incorporating CUDA in your application, the performance of the code does not improve. In these kind of scenarios, it is very useful to visualize which part of the code is taking maximum time. This is called profiling of kernel execution code. Nvidia provides a tool for this and it comes with standard CUDA installation. This tool is called Nvidia Visual Profiler. In standard CUDA 9.0 installation on Windows 10, it can be found on following path: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\libnvvp`. You can run nvvp application available on this path which will open Nvidia Visual Profile tool as follows:

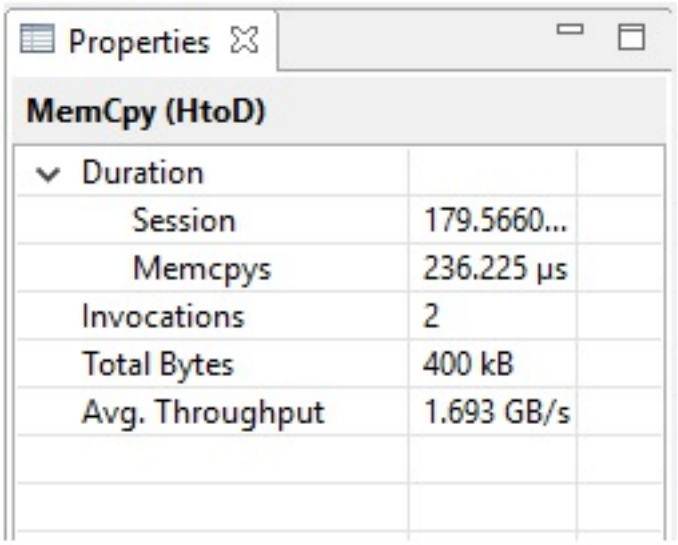


This tool will execute your code, and based on the performance of your GPU, it will give you a

detail report on the execution time of each kernel, detail time stamp for each operation in your code, memory used in your code, memory bandwidth etc. To visualize and get detailed report for any applications you have developed, you can go to File -> New Session. Select .exe file of the application. We have selected the vector add example seen in previous chapter. The result will be as follows:


NVIDIA Visual Profiler
File View Window Run Help
*NewSession1
Welcome
0 s
0.025 s
0.05 s
0.075 s
0.1 s
0.125 s
0.15 s
0.175 s
0.2 s
Process "Many_threads_parallel....
Thread 8120
Runtime API
cudaMalloc
Driver API
Profiling Overhead
[0] GeForce 940MX
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% gpuAdd(int...
Streams
Default
WELCOM
GETTING STARTED
CUDA ZONE
PROFILE AN APPLICATION
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Settings
Properties
gpuAdd(int*, int*, int*)
Name
Invocations
Avg. Duration
Regs
Static SM
Avg. Dynamic SMem
gpu...
1
195.041 µs
12
0
0
Duration
Session
179.5660...
Kernel
195.041 µs
Invocations
1

The result displays timing of all operations in the program. It can be seen that cudaMalloc operation takes maximum time to complete. It also displays the order in which each operation is performed in your code. It displays that kernel is called only once and it needs an average of 192.041 microseconds to execute. The details of memory copy operations can also be visualized. The property of memory copy operation from host to device is shown as follows:



It can be seen that as we are coping two arrays from host to device, so memory copy operation is invoked two times. Total number of Bytes copied are 400 kB with a throughput of 1.693 GB/s. This tool is very important in the analysis of kernel execution. It can also be used to compare the performance of two kernels. It will show you the exact operation that is slowing down the performance of your code.

To summarize, in this section we have seen two methods to measure and analyze CUDA code. CUDA event is an efficient API to measure the timing of device code. CUDA visual profiler gives a detailed analysis and profiling of CUDA code, which can be used to analyze performance. In the next section, we will see how to handle errors in CUDA code.

Error handling in CUDA

We have not checked the availability of GPU device or memory for our CUDA programs. It may happen that when you run your CUDA program, the GPU device is not available or it is out of memory. In that case, you may find it difficult to understand the reason for termination of your program. Therefore, it is a good practice to add error handling code in CUDA programs. In this section, first we will try to understand how we can add this error handling code to CUDA functions. When the code is not giving the intended output, it is useful to check the functionality of the code line-by-line or by adding a break point in the program. This is called as debugging. CUDA provides debugging tools that can help. So in the second section, we will see some debugging tools provided by Nvidia with CUDA.

Error handling from within the code

When we discussed CUDA API functions in the second chapter, we have seen that it also returns the flag that indicates whether the operation has finished successfully or not. This can be used to handle errors from within CUDA programs. Of course, it will not help in resolving errors but it will indicate which CUDA operation is causing errors. It is a very good practice to wrap CUDA function with this error handling code. The sample error handling code for a cudaMalloc function is shown:

```
cudaError_t cudaStatus;
cudaStatus = cudaMalloc((void**)&d_a, sizeof(int));
if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
}
```

cudaError_t API is used to create an error object that will store the return value of all CUDA operation. So the output of a cudaMalloc function is assigned to this error object. If error object is not equal to cudaSuccess then there was some error in assigning memory on the device. So this is handled by if statement. It will print the error on console and jump to end of the program. The wrapper code for error handling during memory copy operation is shown as follows:

```
cudaStatus = cudaMemcpy(d_a,&h_a, sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
```

Again, it has a similar structure to error handling code for cudaMalloc. The wrapper code for kernel call is shown as follows:

```
gpuAdd<<<1, 1>>>(d_a, d_b, d_c);
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }
```

Kernel call does not return flag that indicates success or failure. So it is not directly assigned to error object. Instead, if there is any error during kernel launch then we can fetch it by using cudaGetLastError() API. This API is used to handle an error during kernel call. It is assigned to error object cudaStatus and if it is not equal to cudaSuccess then it prints the error on the console and jumps to end of the program. All the error handling code jumps to the code section defined by Error label. So it can be defined as follows:

```
Error:
    cudaFree(d_a);
```

Whenever any error is encountered in a program, we are jumping to this section. We are freeing up memory allocated on the device and then exiting main function. This is a very efficient way of writing CUDA programs. We suggest you to use this method for writing your CUDA code. It was not explained earlier to avoid unnecessary complexity in codes explained. Addition of error handling codes in CUDA program will make it longer but it will be able to pinpoint which

CUDA operation is causing problems in the code.

Debugging tools

There are always two types of errors that we may encounter in programming. Syntax errors and Semantic errors. Syntax errors can be handled by compilers but semantic errors are difficult to find and debug. The semantic errors cause the program to work unexpectedly. When your CUDA program is not working as intended then there is a need for executing your code line-by line to visualize output after every line. This is called as debugging. It is a very important operation for any kind of programming. CUDA provides debugging tools that help resolve these kind of errors.

For Linux based systems, Nvidia provides a very helpful debugger known as CUDA-GDB. It has a similar interface to normal gdb debugger used for C code. It helps you in debugging your kernel directly on GPU with features like setting breakpoints, inspecting GPU memory, inspecting blocks and thread etc. It also provides memory checker to check illegal memory accesses.

For the Windows-based system, Nvidia provides Nsight debugger integrated with Microsoft visual studio. Again, it has features for adding breakpoints in the program and inspecting blocks/thread execution. The device global memory can be viewed from visual studio memory interface.

To summarize, in this sections we have seen two methods for handling errors in CUDA. One method helps in solving GPU hardware related errors like device not available or memory not available from CUDA programs. The second method of using debugging helps when the program is not working as per expectation. In the next section, we will try to see some advanced concepts that can help in performance improvement of CUDA Programs.

Performance improvement of CUDA Programs

In this section, we will see some basic guidelines that we can follow to improve the performance of CUDA programs. It is explained one by one:

Using an optimum number of blocks and threads

We have seen two parameters that need to be specified during kernel call: Number of blocks and Number of Threads per block. GPU resources should not be idle during kernel call then only it will give the optimum performance. If resources remain idle than it may degrade the performance of the program. The number of blocks and Threads per block helps in keeping GPU resources busy. It has been researched that if a number of blocks are taken as two times the number of multiprocessors on GPU then it will give the best performance. Total multiprocessor on GPU can be found out by using device properties as seen in the second chapter. Same way maximum number of threads per block should be taken equal to maxThreadperblock device property. These values are just for guidance. You can play around with these two parameters to get optimum performance in your application.

Maximizing Arithmetic Efficiency

Arithmetic efficiency is defined as the ratio of the number of mathematical computation to the number of memory access operations. The value of arithmetic efficiency should be as high as possible for good performance. It can be increased by maximizing computation per thread and minimizing time spent on memory per thread. Sometimes the opportunity to maximize computation per thread is limited but certainly, you can reduce time spent on memory. You can minimize it by storing frequently accessed data in to fast memory. We have seen in the last chapter that local memory and register files are the fastest memory available on GPU. So, it can be used to store data that need frequent access. We have also seen the use of shared memory, constant memory, and texture memory for performance improvement. Caching also helps in reducing memory access time. Ultimately, if we reduce bandwidth to global memory then we can reduce time spent on memory. Efficient memory usage is very important in improving the performance of CUDA programs as memory bandwidth is the biggest bottleneck in fast execution.

Using Coalesce or strided memory access

The Coalesce memory access means that every thread reads or writes to contiguous memory locations. GPU is most efficient when this memory access method is used. If threads use memory locations that are offset by a constant value then this is called as strided memory access. It still gives better performance than random memory access. So, if one tries to use coalesce memory access in the program then it can drastically improve performance. Following are examples of these memory access patterns:

```
Coalesce Memory Access: d_a[i] = a
Strided Memory Access: d_a[i*2] = a
```

Avoiding Thread Divergence

Thread divergence happens when all threads in kernel call follow different paths of execution. It may happen in following kernel code scenarios:

```
Thread divergence by way of branching
tid = ThreadId
if (tid%2 == 0)
{
 Some Branch code;
}
else
{
 Some other code;
}
```

```
Thread divergence by way of looping
Pre-loop code
for (i=0; i<tid;i++)
{
Some loop code;
}
Post loop code;
```

In the first code, there is a separate code for odd and even number threads because of the condition in if statement. This makes odd and even number threads follow a different path for execution. After if statement, these threads will again merge. This will incur time overhead because fast threads have to wait for slow threads.

In the second example using for loop, each thread runs for loop for a different number of iterations, and hence all threads will take a different amount of time to finish. Post loop code has to wait for all these threads to finish. It will incur time overhead. So as far as possible, avoid this kind of thread divergence in your code.

Using Page-locked Host Memory

In every example till this point, we have used malloc function to allocate memory on the host, which allocates standard pageable memory on the host. CUDA provides another API called cudaHostAlloc(), which allocates page-locked host memory or sometimes referred as pinned memory. It guarantees that operating system will never page this memory out of this disk and it will remain in physical memory. So any application can access the physical address of the buffer. This property helps GPU in copying data to and from host via Direct Memory Access (DMA) without CPU intervention. This helps in improvement of performance in memory transfer operation. But page-locked memory should be used with proper care because this memory is not swapped out of disk; your system may run out of memory. It may effect the performance of other applications running on the system. You can use this API to allocate memory which is used to transfer data to device using Memcpy operation. The syntax of using this API is shown below:

```
Allocate Memory: cudaHostAlloc ( (void **) &h_a, sizeof(*h_a), cudaHostAllocDefault);
Free Memory: cudaFreeHost(h_a);
```

The syntax of cudaHostAlloc is similar to simple malloc function. The last argument cudaHostAllocDefault which is a flag used to modify the behavior of pinned memory is added. cudaFreeHost is used to free memory allocated using a cudaHostAlloc function.

CUDA Streams

We have seen that GPU provides a great performance improvement in data parallelism when single instruction operates on multiple data. We have not seen task parallelism where more than one kernel function, which are independent of each other, operate in parallel. For example, one function may be computing pixel values while another function is downloading something from the internet. We know that CPU provides a very flexible way for this kind of task parallelism. GPU also provides this capability of task parallelism but it is not as flexible as CPU. This task parallelism is achieved by using CUDA streams which we will see in detail in this section.

A CUDA stream is nothing but a queue of GPU operations which executes in a specific order. These functions include kernel functions, memory copy operations, and CUDA event operations. The order in which they are added to the queue will determine the order of their execution. Each CUDA stream can be considered as a single task so we can start multiple streams to do multiple tasks in parallel. We will look at how multiple streams work in CUDA in next section.

Using Multiple CUDA streams

We will understand the working of CUDA streams by using multiple CUDA streams in vector addition program that we have developed in the last chapter. The kernel function for this is shown below:

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>
//Defining number of elements in Array
#define N 50000

//Defining Kernel function for vector addition
__global__ void gpuAdd(int *d_a, int *d_b, int *d_c) {
  //Getting block index of current kernel

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  while (tid < N)
  {
    d_c[tid] = d_a[tid] + d_b[tid];
    tid += blockDim.x * gridDim.x;
  }
}
```

The kernel function is similar to what we have developed earlier. It is just that multiple streams will execute this kernel in parallel. It should be noted that not all GPU devices support CUDA stream. GPU devices, which support deviceOverlap property, can perform memory transfer operation and kernel execution simultaneously. This property will be used in CUDA streams for task parallelism. Before proceeding further in this code, please ensure that your GPU device supports this property. You can use code from chapter 2 for verifying this property. We will use two parallel streams, which execute this kernel in parallel and operate on half input data each. We will start by creating these two streams in main function as follows:

```
int main(void) {
  //Defining host arrays
  int *h_a, *h_b, *h_c;
  //Defining device pointers for stream 0
  int *d_a0, *d_b0, *d_c0;
  //Defining device pointers for stream 1
  int *d_a1, *d_b1, *d_c1;
  cudaEvent_t e_start, e_stop;
  cudaEventCreate(&e_start);
  cudaEventCreate(&e_stop);
  cudaEventRecord(e_start, 0);
  cudaStream_t stream0, stream1;
  cudaStreamCreate(&stream0);
  cudaStreamCreate(&stream1);
```

Two stream objects, stream 0 and stream 1, are defined using cudaStream_t and cudaStreamCreate APIs. We have also defined host pointers and two sets of device pointers which will be used for each streams separately. We are defining and creating two events for performance measurement of this program. Now we need to allocate memory for these pointers. The code is shown below:

```
  //Allocate memory for host pointers
  cudaHostAlloc((void**)&h_a, 2*N* sizeof(int),cudaHostAllocDefault);
  cudaHostAlloc((void**)&h_b, 2*N* sizeof(int), cudaHostAllocDefault);
  cudaHostAlloc((void**)&h_c, 2*N* sizeof(int), cudaHostAllocDefault);
  //Allocate memory for device pointers
```

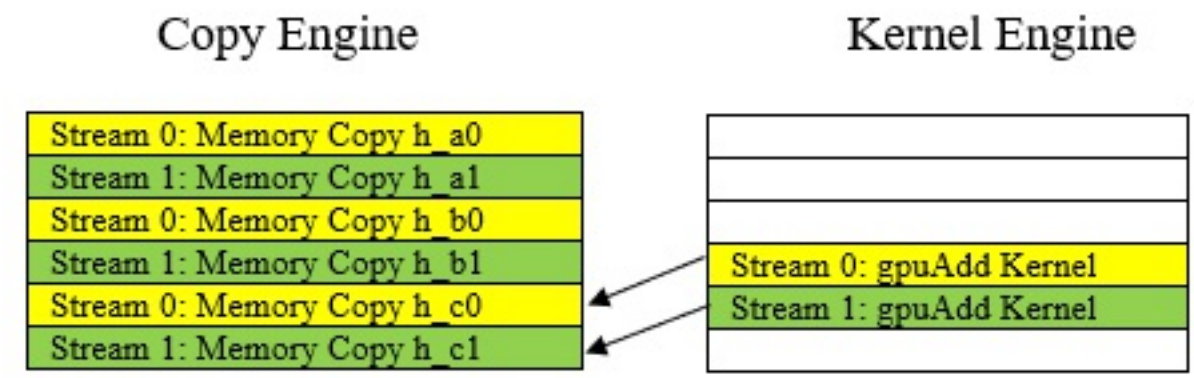
```
  cudaMalloc((void**)&d_a0, N * sizeof(int));
  cudaMalloc((void**)&d_b0, N * sizeof(int));
  cudaMalloc((void**)&d_c0, N * sizeof(int));
  cudaMalloc((void**)&d_a1, N * sizeof(int));
  cudaMalloc((void**)&d_b1, N * sizeof(int));
  cudaMalloc((void**)&d_c1, N * sizeof(int));
for (int i = 0; i < N*2; i++) {
    h_a[i] = 2 * i*i;
    h_b[i] = i;
  }
```

CUDA streams need an access to page locked memory for its memory copy operation, and hence we are defining host memory using a cudaHostAlloc function instead of simple malloc. We have seen the advantage of page-locked memory in the last section. Two sets of device pointers are allocated memory using cudaMalloc. It should be noted that host pointers hold the entire data so its size is 2*N*sizeof(int) whereas each device pointers operate on half data elements so their size is only N*sizeof(int). We have also initialize host arrays with some random values for addition. Now we will try to enqueue memory copy operation and kernel execution operations in both the streams. The code for this is shown below:

```
//Asynchrnous Memory Copy Operation for both streams
cudaMemcpyAsync(d_a0, h_a , N * sizeof(int), cudaMemcpyHostToDevice, stream0);
cudaMemcpyAsync(d_a1, h_a+ N, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(d_b0, h_b , N * sizeof(int), cudaMemcpyHostToDevice, stream0);
cudaMemcpyAsync(d_b1, h_b + N, N * sizeof(int), cudaMemcpyHostToDevice, stream1);

//Kernel Call
gpuAdd << <512, 512, 0, stream0 >> > (d_a0, d_b0, d_c0);
gpuAdd << <512, 512, 0, stream1 >> > (d_a1, d_b1, d_c1);

//Copy result back to host memory from device memory
cudaMemcpyAsync(h_c , d_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0);
cudaMemcpyAsync(h_c + N, d_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream0);
```

Instead of using a simple cudaMemcpy, we are using cudaMemcpyAsync API which is used for asynchronous memory transfer. It enqueues a request of memory copy operation in a given stream specified as the last argument to the function. When this function returns, the memory copy operation may not have even started, and hence it is called asynchronous operation. It just puts a request of memory copy in a queue. As we can see in memory copy operations, stream 0 operates on data between 0 to N and stream 1 operates on data N+1 to 2N.

The order of operation is important in stream operations as we want to overlap memory copy operations with kernel execution operations. So, instead of enqueuing all stream 0 operations and then enqueuing stream 1 operations, we are first enqueuing memory copy operation in both streams and then enqueuing both kernel computation operations. This will ensure that memory copy and kernel computation overlaps with each other. If both the operations take the same amount of time then we can achieve two times speed up. We can get more idea about the order of operation by looking at figure shown below:



The time increases from top to bottom. We can see that two memory copy operations and kernel execute operations are performed in the same time period which will accelerate your program. We have also seen that memory copy operation, defined by cudaMemcpyAsync, is asynchronous; so, when it returns, memory copy operation may not have started. If we want to use the result of last memory copy operation then we have to wait for both streams to finish their queue operations. This can be ensured by the code below:

```
cudaDeviceSynchronize();
cudaStreamSynchronize(stream0);
cudaStreamSynchronize(stream1);
```

cudaStreamSynchronize ensures that all operations in the stream are finished before proceeding to the next line. To measure the performance of the code, the following code is inserted:

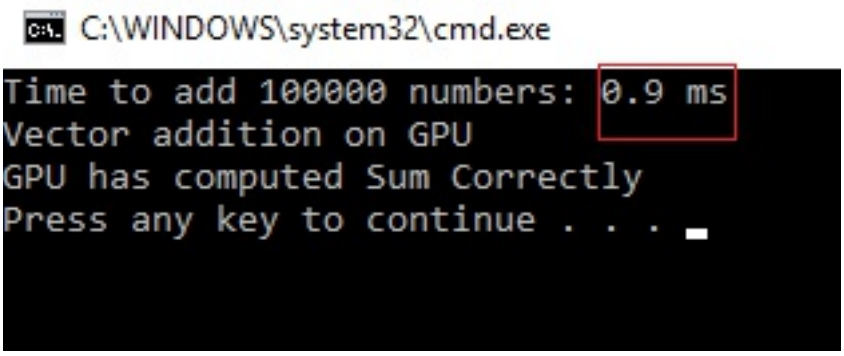
```
cudaEventRecord(e_stop, 0);
cudaEventSynchronize(e_stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, e_start, e_stop);
printf("Time to add %d numbers: %3.1f ms\n",2* N, elapsedTime);
```

It will record the stop time and based on the difference between start and the stop time, it will calculate the overall execution time for this program and print the output on the console. To check whether the program has calculated correct output, we will insert the following code for verification:

```
int Correct = 1;
printf("Vector addition on GPU \n");
//Printing result on console
for (int i = 0; i < 2*N; i++)
{
    if ((h_a[i] + h_b[i] != h_c[i]))
    {
      Correct = 0;
    }
}

if (Correct == 1)
  {
    printf("GPU has computed Sum Correctly\n");
  }
else
  {
    printf("There is an Error in GPU Computation\n");
  }
  //Free up memory
  cudaFree(d_a0);
  cudaFree(d_b0);
```

```
  cudaFree(d_c0);
  cudaFree(d_a0);
  cudaFree(d_b0);
  cudaFree(d_c0);
  cudaFreeHost(h_a);
  cudaFreeHost(h_b);
  cudaFreeHost(h_c);
  return 0;
}
```

The verification code is similar to what we have seen earlier. Memory allocated on the device is freed up using cudaFree and memory allocated on the host using cudaHostAlloc is freed up using the cudaFreeHost function. This is mandatory otherwise your system may run out of memory very quickly. The output of the program is shown below:



As can be seen from the figure, 0.9 ms is needed to add 1,00,000 elements, which is 2x increamant over code without streams that needed 0.9 ms to add 50,000 numbers, as seen in the first section of this chapter.

To summarize, we have seen CUDA streams in this section which helps in achieving task parallelism on GPU. The order in which operations are queued in streams is very important to achieve speed up using CUDA stream.

Acceleration of sorting algorithms using CUDA

Sorting algorithms are widely used in many computing applications. There are many sorting algorithms like enumeration or rank short, bubble short, merge short etc. All algorithms have different levels of complexity so it takes a different amount of time to sort a given Array. For large sizes of any Array, all algorithms take a long time to complete. If this can be accelerated using CUDA then it can be of a great help in any computing application.

To show an example of how CUDA can accelerate different sorting algorithms, Rank sort algorithm is implemented in CUDA.

Enumeration or Rank Sort Algorithm

In this algorithm, we count every element in an Array to find out how many elements in an Array are less than current element. From that, we can get the position of the current element in a sorted Array. Then we put this element in that position. We repeat this process for all elements in an Array to get a sorted Array. This is implemented as kernel function shown below:

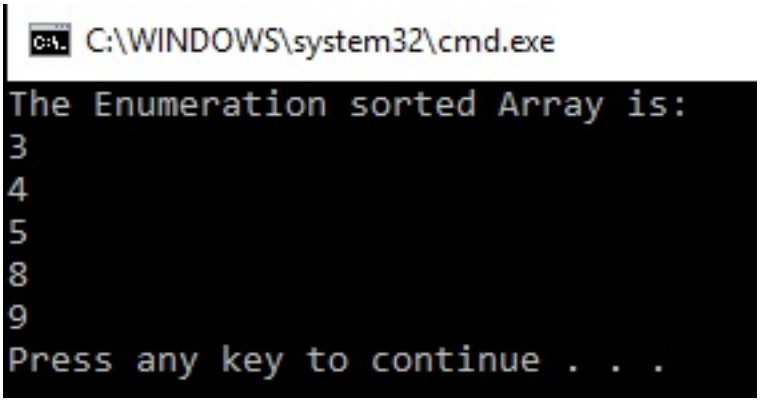
```
#include "device_launch_parameters.h"
#include <stdio.h>

#define arraySize 5
#define threadPerBlock 5
//Kernel Function for Rank sort
__global__ void addKernel(int *d_a, int *d_b)
{
  int count = 0;
  int tid = threadIdx.x;
  int ttid = blockIdx.x * threadPerBlock + tid;
  int val = d_a[ttid];
  __shared__ int cache[threadPerBlock];
  for (int i = tid; i < arraySize; i += threadPerBlock) {
    cache[tid] = d_a[i];
    __syncthreads();
      for (int j = 0; j < threadPerBlock; ++j)
          if (val > cache[j])
            count++;
    __syncthreads();
  }
  d_b[count] = val;
}
```

Kernel function takes two arrays as parameters. d_a is input array and d_b is the output array. 'count' variable is taken which stores the position of the current element in the sorted Array. Current thread index in block is stored in tid and unique thread index among all blocks is stored in ttid. Shared memory is used to reduce the time in accessing data from global memory. The size of shared memory is equal to the number of threads in a block as discussed earlier. 'value' variable holds the current element. Shared memory is filled up with values of global memory. These values are compared with 'value' variable and count of the number of values which are less is stored in 'count' variable. This continues till all elements in an Array are compared with 'value' variable. After loop finishes, 'count' variable has the position of the element in a sorted array, and hence the current element is stored at that position in d_b which is an output array. The main function for this code is shown below:

```
int main()
{
    //Define Host and Device Array
    int h_a[arraySize] = { 5, 9, 3, 4, 8 };
    int h_b[arraySize];
    int *d_a, *d_b;

    //Allocate Memory on the device
    cudaMalloc((void**)&d_b, arraySize * sizeof(int));
    cudaMalloc((void**)&d_a, arraySize * sizeof(int));

    // Copy input vector from host memory to device memory.
    cudaMemcpy(d_a, h_a, arraySize * sizeof(int), cudaMemcpyHostToDevice);

    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<arraySize/threadPerBlock, threadPerBlock>>>(d_a, d_b);

    //Wait for device to finish operations
    cudaDeviceSynchronize();
```

```
    // Copy output vector from GPU buffer to host memory.
    cudaMemcpy(h_b, d_b, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
    printf("The Enumeration sorted Array is: \n");
    for (int i = 0; i < arraySize; i++)
    {
    printf("%d\n", h_b[i]);
    }
    //Free up device memory
    cudaFree(d_a);
    cudaFree(d_b);
    return 0;
}
```

The main function should be very familiar to you by now. We are defining host and device Arrays and allocating memory on the device for device Arrays. Host array is initialized with some random values and copied to device memory. Kernel is launched by passing device pointers as parameters. Kernel computes sorted Array by rank sort algorithm and returns it to the host. This sorted Array is printed on the console as shown below:



This is a very trivial case and you might not see any performance improvement between CPU and GPU. But if you go on increasing value of arraySize then GPU will drastically improve the performance of this algorithm. It can be 100x improvement for Array size equal to 15,000. Rank sort is the simplest sorting algorithm available. This discussion will help you in developing code for other sorting algorithms like bubble sort and merge sort.

Image Processing using CUDA

Today we live in the age of high definition camera sensors that capture high-resolution images. An image can have a size of up to 1920 x 1920 pixels. So processing of these pixels on computers in real time involves billions of floating point operations to be performed per second. This is difficult for even the fastest of CPUs. GPU can help in this kind of situations. It provides high computation power which can be leveraged by using CUDA in your code.

Images are stored as multi dimensional Array in a computer with 2-dimension for gray scale image and 3-dimension for a color image. CUDA also support multi dimensional grid blocks and threads. So we can process an image by launching multi dimensional blocks and threads as seen previously. The number of blocks and threads can very depending on the size of an image. It will also depend on your GPU specifications. If it supports 1024 threads per block than 32 x 32 threads per block can be launched. The number of blocks will be determined by dividing image size by these number of threads. As discussed many times previously that choice of these parameters affect the performance of your code so it should be chosen properly.

It is very easy to convert a simple image processing code developed in C/C++ to a CUDA code. It can be done by a naive programmer also by following a set pattern. Image processing code has a set pattern like the code shown below:

```
for (int i=0; i < image_height; i++)
{
        for (int j=0; j < image_width; j++)
        {
                //Pixel Processing code for pixel located at (i,j)
        }
}
```

Images are nothing but multi-dimensional matrix stored on the computer so getting a single pixel value from an image involves using nested for loops to iterate over all pixels. To convert this code to CUDA, we want to launch a number of threads equal to the number of pixels in an image. The pixel value in a thread can be obtained by following code in kernel function:

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

These values i and j can be used as index to an image array to find the pixel values. So as can be seen that a simple conversion of converting for loops to thread index, we can write a device code for CUDA program. We are going to develop many image processing applications using OpenCV library from the next section onwards. So we will not cover actual image manipulation in this chapter but we will end this chapter by developing a CUDA program for very important statistical operation of calculating a histogram. Histogram calculation is very important for image processing applications as well.

Histogram Calculation on GPU using CUDA

A histogram is a very important statistical concept used in a variety of applications like machine learning, computer vision, data science, image processing etc. It represents a count of the frequency of each element in a given data. It shows which data are occurring most frequently and which data are occurring least frequently. You can also get an idea about the distribution of data by just looking at the values of the histogram. In this section, we will try to develop an algorithm that calculates the histogram of a given data distribution.

We will start by calculating histogram on CPU so that you can get an idea of how to calculate histogram. Assume that we have data with 1000 elements with each element has a value between 0 to 15. We want to calculate the histogram of this distribution. The sample code for this calculation on CPU is shown below:

```
int h_a[1000] = Random values between 0 and 15

int histogram[16];
for (int i = 0; i<16; i++)
{
  histogram[i] = 0;
}
for (i=0; i < 1000; i++)
{
  histogram[h_a[i]] +=1;
}
```

We have 1000 data elements and they are stored in h_a. h_a contains values between 0 and 15. It has 16 distinct values. So the number of bins which indicates the number of distinct values for which histogram needs to be calculated is 16. So we have defined a histogram array that will store the final histogram with size equal to the number of bins. This array needs to be initialized to zero as it will be incremented with each occurrence. This is done in the first for loop that runs from 0 to the number of bins.

For calculating histogram, we need to iterate through all elements in h_a. Whichever value is found in h_a, that particular index in histogram array needs to be incremented. That is done by second for loop which calculates final histogram by running from 0 to size of the array and incrementing histogram array indexed by the value found in h_a. Histogram array will contain frequency of each element between 0 to 15 after for loop finishes.

Now we will try to develop this same code for GPU. We will try to develop this code using three different methods. The kernel code for first two methods is shown below:

```
#include <stdio.h>
#include <cuda_runtime.h>

#define SIZE 1000
#define NUM_BIN 16

__global__ void histogram_without_atomic(int *d_b, int *d_a)
{
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int item = d_a[tid];
  if (tid < SIZE)
  {
    d_b[item]++;
```

```
  }
 }

__global__ void histogram_atomic(int *d_b, int *d_a)
{
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int item = d_a[tid];
  if (tid < SIZE)
  {
    atomicAdd(&(d_b[item]), 1);
  }
}
```

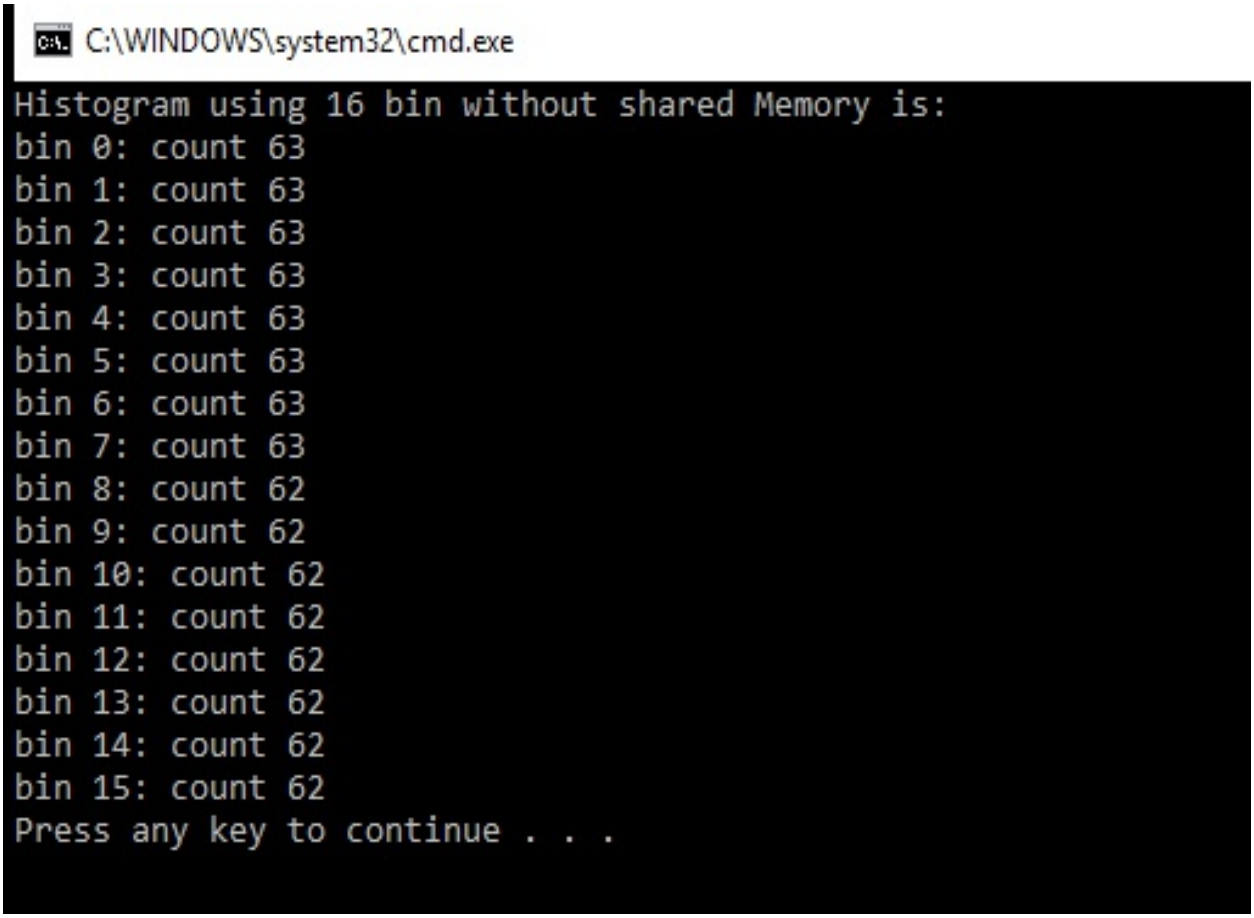
The first function is the simplest kernel function for histogram computation. Each thread is operating on one data element. The value of data element is fetched using thread id as an index to input Array. This value is used as an index into output array d_b which is incremented. d_b should contain the frequency of each value between 0 to 15 in input data. But if you recall from the last chapter, this may not give you correct answer as many threads are trying to modify same memory location simultaneously. In this example, 1000 threads are trying to modify 16 memory locations simultaneously. We need to use atomic add operation for this kind of scenarios.

Second device function is developed using atomic add operation. This kernel function will give you the correct answer but it will take more time to complete as the atomic operation is a blocking operation. All other threads have to wait when one thread is using a particular memory location. So second kernel function will add time overhead which makes it even slower than CPU version. To complete the code, we will try to write the main function for it as shown below:

```
int main()
{

  int h_a[SIZE];
  for (int i = 0; i < SIZE; i++) {

    h_a[i] = i % NUM_BIN;
  }
  int h_b[NUM_BIN];
  for (int i = 0; i < NUM_BIN; i++) {
    h_b[i] = 0;
  }

  // declare GPU memory pointers
  int * d_a;
  int * d_b;

  // allocate GPU memory
  cudaMalloc((void **)&d_a, SIZE * sizeof(int));
  cudaMalloc((void **)&d_b, NUM_BIN * sizeof(int));

  // transfer the arrays to the GPU
  cudaMemcpy(d_a, h_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, NUM_BIN * sizeof(int), cudaMemcpyHostToDevice);


  // launch the kernel

  //histogram_without_atomic << <((SIZE+NUM_BIN-1) / NUM_BIN), NUM_BIN >> >(d_b, d_a);
  histogram_atomic << <((SIZE+NUM_BIN-1) / NUM_BIN), NUM_BIN >> >(d_b, d_a);


  // copy back the sum from GPU
  cudaMemcpy(h_b, d_b, NUM_BIN * sizeof(int), cudaMemcpyDeviceToHost);
  printf("Histogram using 16 bin without shared Memory is: \n");
  for (int i = 0; i < NUM_BIN; i++) {
    printf("bin %d: count %d\n", i, h_b[i]);
  }

  // free GPU memory allocation
  cudaFree(d_a);
  cudaFree(d_b);
```

```
  return 0;
}
```

We have started main function by defining host and device arrays and allocating memory for it. The input data array h_a is initialized with values from 0 to 15 in the first for loop. We are using modulo operation, and hence 1000 elements will be evenly divided between values of 0 10 15. The second array which will store histogram is initialized to zero. These two arrays are copied to device memory. The kernel will compute the histogram and return it to the host. We are printing this histogram on the console. The output is shown below:



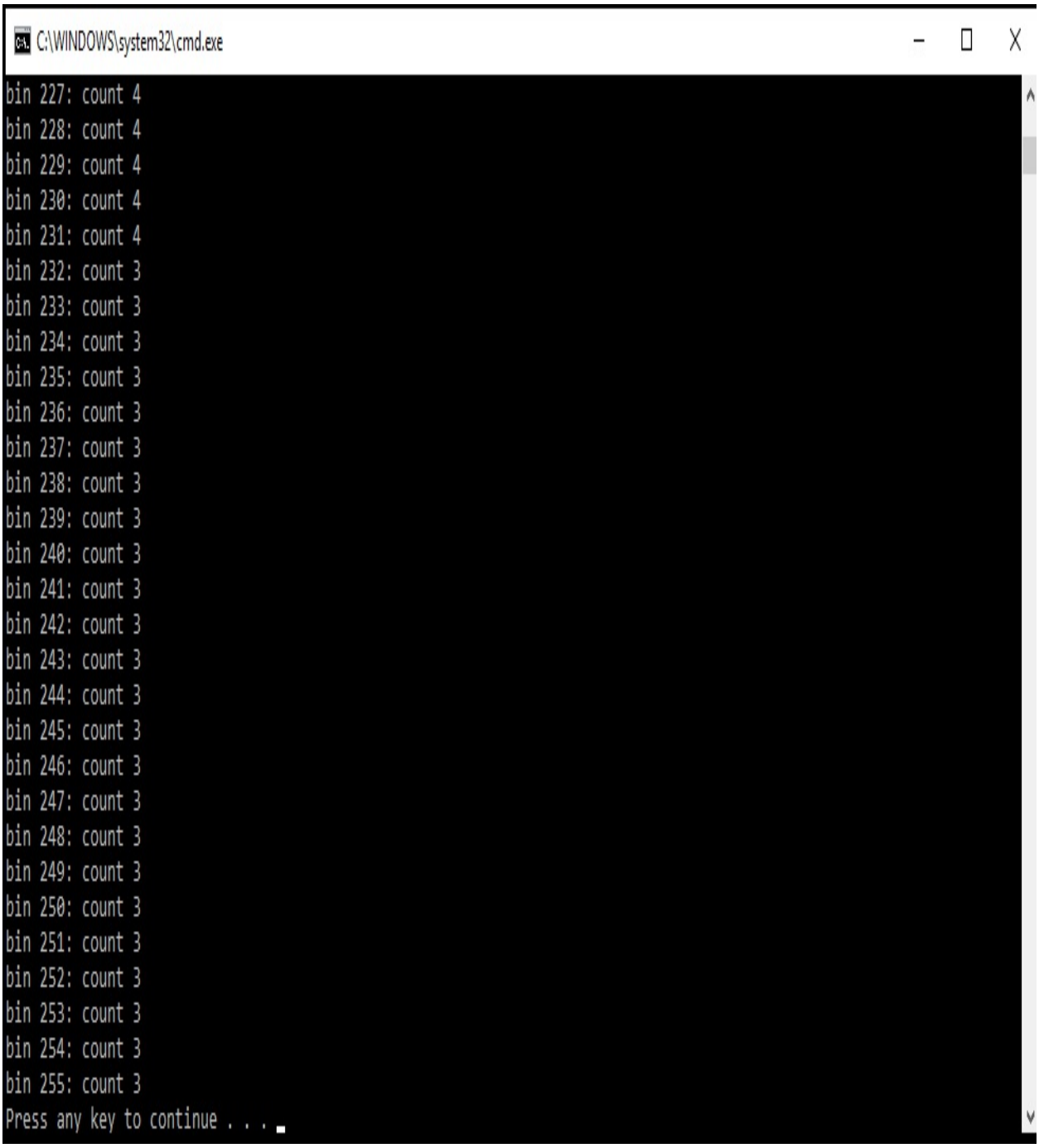
When we try to measure the performance of this code using atomic operation and compare it with CPU performance, it is slower than CPU for large size arrays. That begs a question of why use CUDA for histogram computation or can we make this computation faster.

The answer to this question is Yes. If we use shared memory for computing histogram for a given block and then add this block histogram to the overall histogram on global memory then it can speed up operation. This is possible because addition is a cumulative operation. The kernel code of using shared memory for histogram computation is shown below:

```
#include <stdio.h>
#include <cuda_runtime.h>
#define SIZE 1000
#define NUM_BIN 256
__global__ void histogram_shared_memory(int *d_b, int *d_a)
{
```

```
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int offset = blockDim.x * gridDim.x;
  __shared__ int cache[256];
  cache[threadIdx.x] = 0;
  __syncthreads();

  while (tid < SIZE)
  {
    atomicAdd(&(cache[d_a[tid]]), 1);
    tid += offset;
  }
  __syncthreads();
  atomicAdd(&(d_b[threadIdx.x]), cache[threadIdx.x]);
}
```

In this code, the number of bins are taken as 256 instead of 16 for more understanding. We are defining shared memory of size equal to the number of threads in a block which is equal to a number of bins 256. We will calculate a histogram for the current block so shared memory is initialized to zero and histogram is computed for this block in the same way as discussed earlier. But this time, the result is stored in shared memory and not in global memory. In this case, only 256 threads are trying to access 256 memory elements in shared memory instead of 1000 elements in the previous code. This will help in reducing time overhead in the atomic operation. The final atomic add in the last line will add histogram of one block to overall histogram values. As addition is cumulative operation we do not have to worry about the order in which each block is executed. The main function for this is similar to the previous function shown. The output for this kernel function is shown below:



If you measure the performance of the preceding program, it will beat both GPU version without shared memory and CPU implementation for large Array sizes. You can check whether the computed histogram by GPU is correct or not by comparing results with CPU computation.

This section demonstrated the implementation of the histogram on GPU. It also reemphasizes the use of shared memory and atomic operation in CUDA programs. It also demonstrates how CUDA is helpful in image processing application and how easy it is to convert existing CPU code to CUDA code.

Summary

In this chapter, we have seen some advanced concepts in CUDA that help us in developing a complex application using CUDA. We have seen the method for measuring the performance of the device code and how to see a detailed profile of kernel function using Nvidia Visual profiler tool. It helps us in identifying the operation that slows down the performance of our program. We have seen the methods to handle errors in hardware operation from CUDA code itself and also seen methods to debug the code using tools. CPU provides efficient task parallelism where two completely different functions execute in parallel. We have seen that GPU also provides this functionality using CUDA streams and achieve 2x speed up on the same vector addition program using CUDA streams. Then we have seen an acceleration of sorting algorithm using CUDA which is an important concept to understand to build complex computing applications. Image processing is a computationally intensive task which needs to be performed in Real-time. Almost all image processing algorithms can utilize parallelism of GPU and CUDA. So in the last section, we have seen the use of CUDA in the acceleration of image processing application and how we can convert existing C++ code to CUDA code. We have also developed CUDA code for histogram calculation which is an important image processing application. This section also marks an end to concepts related to CUDA programming. From the next chapter, we will start with the exciting part of developing computer vision applications using OpenCV library which utilizes CUDA acceleration concepts that we have seen till this point. We will start by dealing with real images and not matrices from the next chapter.

Questions

1. Why CPU timers are not used to measure the performance of kernel functions?
2. Try to visualize the performance of matrix multiplication code implemented in the last chapter using Nvidia Visual profiler tool.
3. Give different examples of semantic errors encountered in programs.
4. What is the drawback of thread divergence in kernel function? Explain with example.
5. What is the drawback of using cudahostAlloc function to allocate memory on the host ?
6. Justify the statement: Order of operation in Cuda Stream is very important to improve the performance of a program.
7. How many blocks and threads should be launched for 1024x1024 image for a good performance using CUDA?

Getting started with OpenCV with CUDA support

Introduction

So far, we have seen all the concepts related to parallel programming using CUDA and how it can leverage GPU for acceleration. From this chapter, we will try to use the concept of parallel programming in CUDA for computer vision applications. Though we have worked on matrices, we have not worked on actual images. Basically, working on images is similar to manipulation of two-dimensional matrices. We will not develop entire code from scratch for computer vision applications in CUDA but we will use popular computer vision library that is called OpenCV. Though this book assumes that the reader has some familiarity in working with OpenCV still this chapter revises the concepts of using OpenCV in C++. This chapter describes the installation of OpenCV library with CUDA support on Windows and Ubuntu. Then it describes how to test this installation and run a simple program. This chapter describes the use of OpenCV in working with images and videos by developing simple codes for it. This chapter also describes the comparison between the performance of the program with CUDA support t0 one without it.

The following topics will be covered in this chapter:

- Introduction to Image Processing and Computer Vision
- Introduction to OpenCV with CUDA support
- Installation of OpenCV with CUDA support on Windows and Ubuntu
- Working with Images using OpenCV
- Working with Videos using OpenCV
- Arithmatic and Logical operation on Images
- Color-space conversions and Image Thresholding
- Performance Comparison between CPU and GPU OpenCV programs

Technical Requirements

This chapter requires basic understanding of image processing and computer vision. It needs familiarity with the basic C or C++ programming language and all the codes explained in previous chapters. All the code used in this chapter can be downloaded from following github link: https://github.com/PacktPublishing/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA. The code can be executed on any operating system though it has only been tested on Ubuntu 16.04.

Introduction to Image Processing and Computer Vision

The volume of image and video data available in the word is increasing day by day. The increasing use of mobile devices to capture images and use of the internet to post them has allowed production of the enormous amount of video and image data every day. Image processing and computer vision are used in many applications across various domains. Doctors use MRI and X-ray images for medical diagnosis. Space scientists and chemical engineers used images for space exploration and analysis of various genes at molecular levels. Images can be used to develop autonomous vehicles and video surveillance applications. Images can also be used in agricultural applications. Images can be used to identify faulty products during manufacturing. Al these applications need to process images on a computer at a high speed. We are not going into detail about how images are captured by a camera sensor and converted into digital images to be stored on the computer. In this book, we will only cover the processing of the image on the computer by assuming that we already have it stored on the computer.

Many people use the terms **image processing** and **computer vision** interchangeably. There is a difference between these two fields. Image processing is concerned with improving the visual quality of images by modifying pixel values whereas computer vision is concerned with extracting important information from the images. So in image processing, both input and output are images while in computer vision input is an image but the output is the information extracted from that image. Both have a wide variety of applications, mainly image processing is used as a pre-processing stage of a computer vision applications.

Images are stored as a multidimensional matrix on computers. So processing of images on a computer is nothing but manipulating this matrix. We have seen how to work with matrices in CUDA in previous chapters. The code for reading, manipulating and displaying images in CUDA might get very long, tedious and hard to debug. So We will use a library that contains API for all these functions and which can leverage advantage of CUDA-GPU acceleration for processing images. This library is called as OpenCV which is an acronym for Open Computer Vision. In the next section, this library is explained in detail.

Introduction to OpenCV

OpenCV is a computer vision library developed with computational efficiency in mind and keeping the focus on real-time performance. It is written in C/C++ and it contains more than hundred functions that help in computer vision applications. The main advantage of OpenCV is that it is open source and released under Berkley software distribution (BSD) license which allows free use of OpenCV in research and commercial applications. This library has an interface in C, C++, Java and Python languages and it can be used in all operating systems like Windows, Linux, Mac, and Android without modifying the single line of code.

This library can also take advantage of multi-core processing. It can take advantage of OpenGL and CUDA for parallel processing. As OpenCV is lightweight, it can be used on embedded platforms like raspberry pi as well. This makes it ideal for deploying computer vision applications on embedded systems in real life scenarios. We are going to explore this in the next few chapters. These features have made OpenCV a default choice for computer vision developers. It has a wide developer base and user community that helps in improving the library constantly. The downloads for OpenCV is in millions and its increasing day by day. The other popular computer vision and image processing tool is MATLAB so you might wonder what are the advantages of using OpenCV over MATLAB. The table below shows the comparison between these two tools.

Parameter	OpenCV	MATLAB
The speed of Program	Higher because it is developed using C/C++	Lower than OpenCV
Resource needed	OpenCV is a lightweight library so it consumes very little memory both in terms of hard disk and RAM. A normal OpenCV program will require less than 100MB RAM.	MATLAB is very bulky. The latest MATLAB version installation can consume more than 15GB space on the hard disk and large chunk of RAM (More than 1 GB) when it is in use.
Portability	OpenCV can run on all operating systems that can run C language.	MATLAB can only run on Windows, Linux and MAC.
Cost	The use of OpenCV in commercial or academic applications is completely free.	MATLAB is a licensed software so you have to pay a large amount to use it in your academic or commercial applications.
Ease of use	OpenCV is comparatively difficult to use as it has less documentation and syntax which is difficult to remember. It also does not have its own development environment.	MATLAB has its own integrated development environment with inbuilt help resource which makes it easy to use for a new programmer.

MATLAB and OpenCV both have their pros and cons. But when we want to use computer vision in embedded applications and take advantage of parallel processing then OpenCV is the

ideal choice. So in this book, OpenCV is described for accelerating computer vision applications using GPU and CUDA. OpenCV has APIs in C, C++, Python, and Java. It is written in C/C++ so API in that languages will be the fastest. Moreover, CUDA acceleration is more supported in C/C++ API so we will use OpenCV with C/C++ API in this book. In the next section, we will see how to install OpenCV on various operating systems.

Installation of OpenCV with CUDA support

Installation of OpenCV with CUDA is not trivial as you might think. It involves many steps. In this section, all the steps to install OpenCV in windows and Ubuntu are explained with snapshots so that you can setup your environment easily by following these steps.

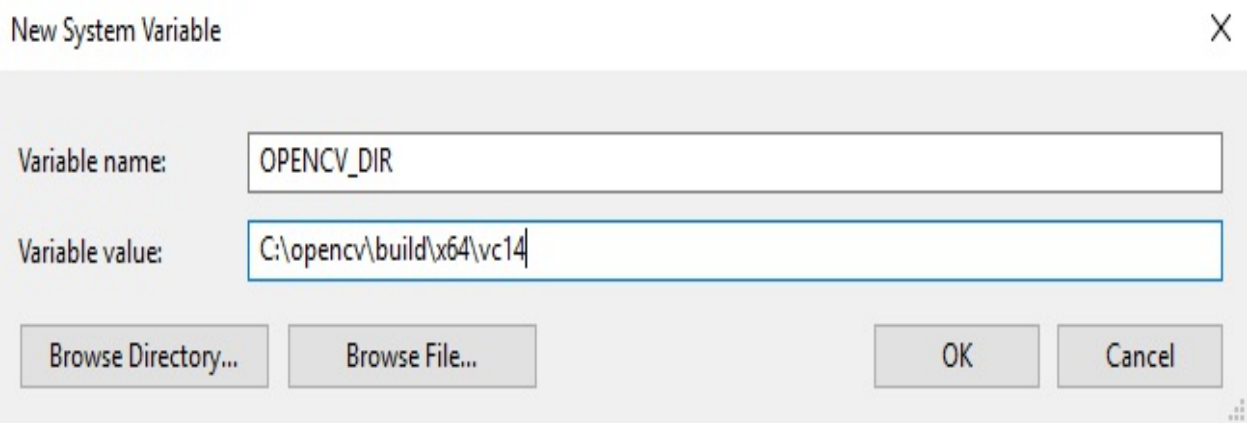
Installation of OpenCV on Windows

This section explains the steps needed to install OpenCV with CUDA on Windows Operating system. The steps are performed on Windows 10 operating system but it will work on any windows operating system.

Using pre-built binaries

There are pre-built binaries available of OpenCV which can be downloaded and used directly in your program. It can't leverage advantage of CUDA so it is not recommended for this book. Following steps describe the procedure for installing OpenCV without CUDA support on Windows.

1. Make sure that Microsoft visual studio is installed for a compilation of C Programs.
2. Download the latest version of OpenCV from https://sourceforge.net/projects/opencvlibrary/files/opencv-win/
3. Double click on downloaded .exe file and extract it to the folder of choice. Here we are extracting it in `c://opencv` folder.
4. Setup environment variable OPENCV_DIR by right clicking on my computer -> advance setting -> environment variables -> new. Set its value as `c:\opencv\build\x64\vc14` as shown in the following figure. Here vc14 will depend on the version of Microsoft visual studio.



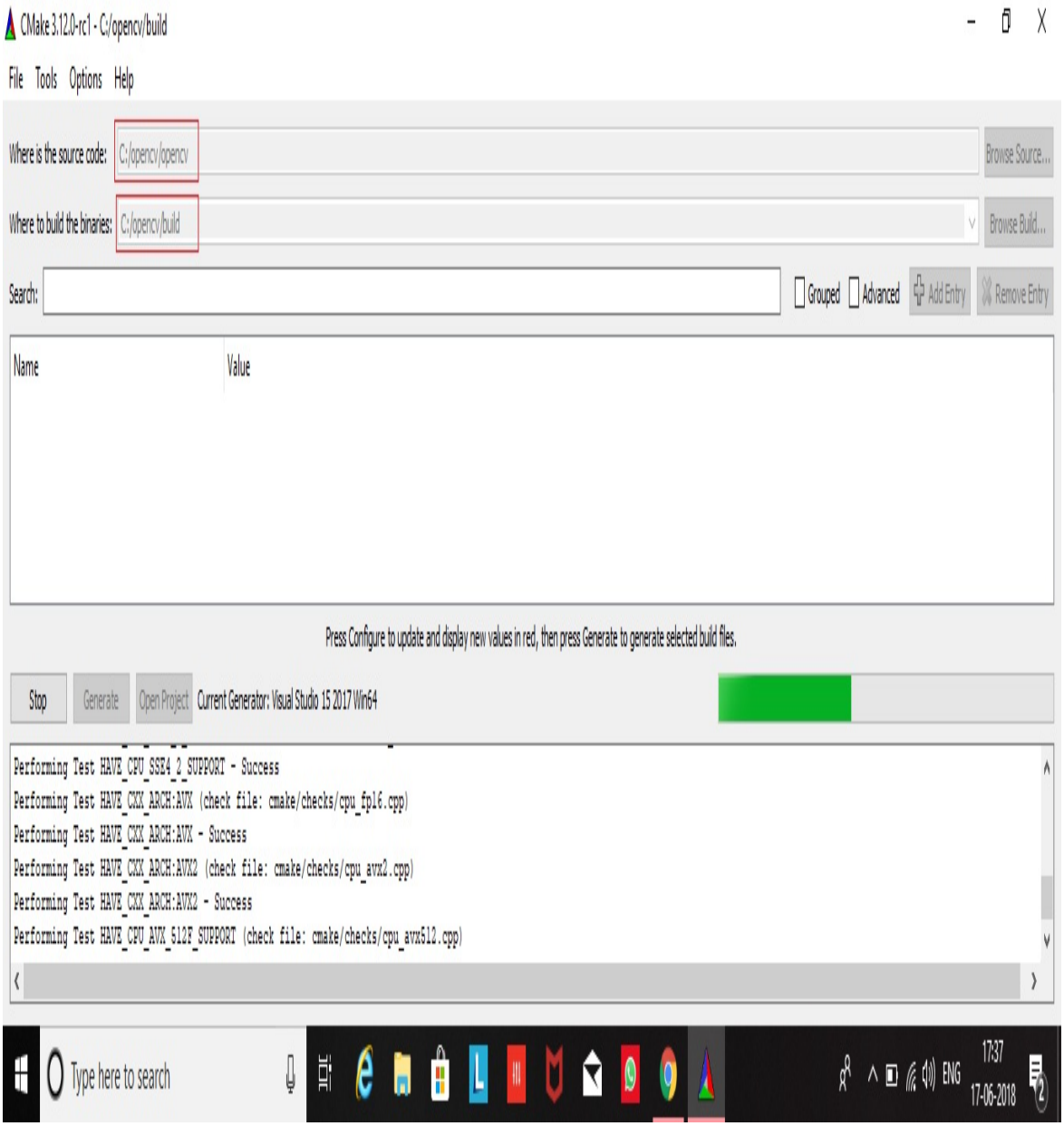
Now you can use this installation for OpenCV applications using C/C++.

Building libraries from Source

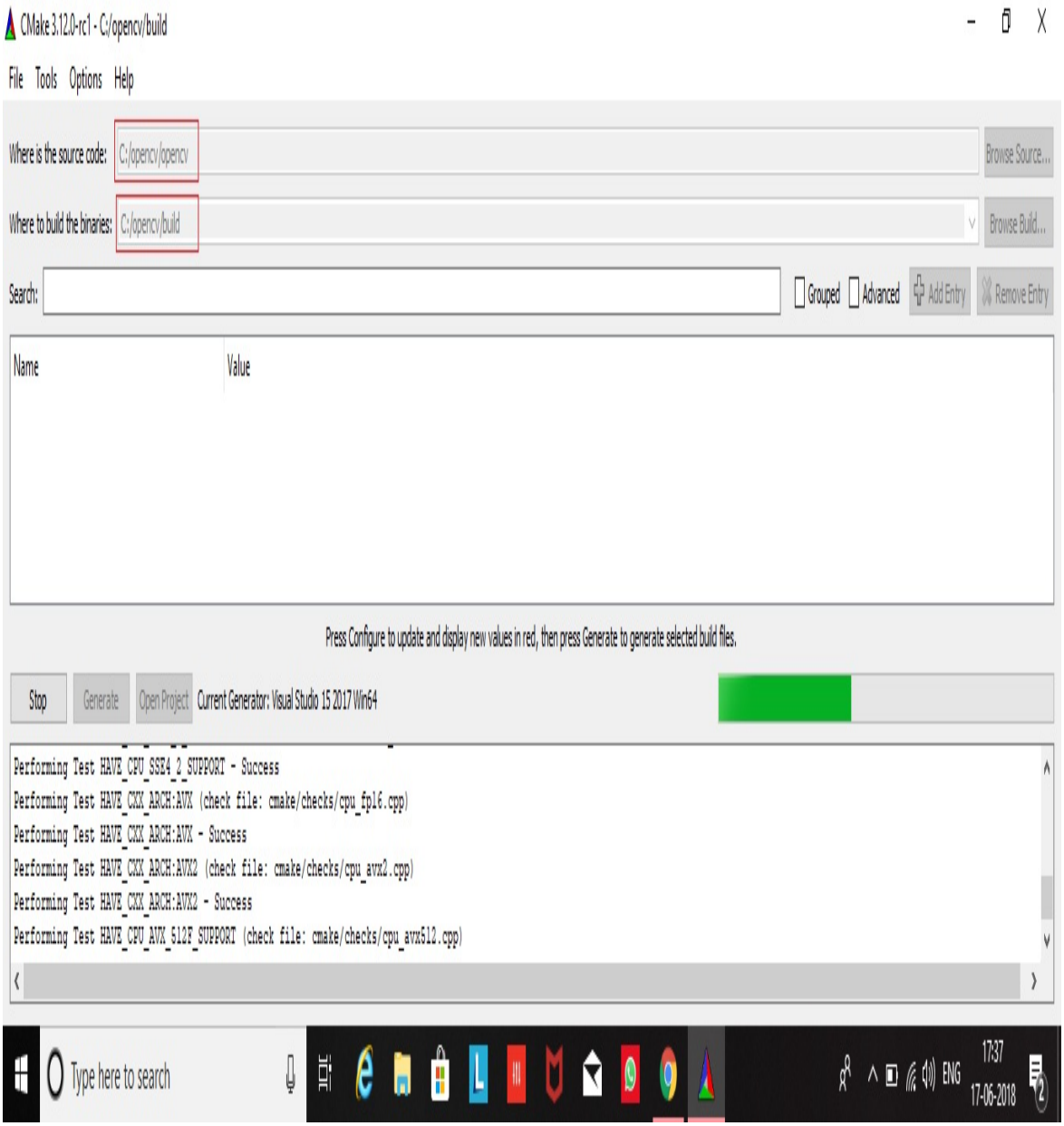
If you want to compile OpenCV with CUDA support, you have to follow these steps for installation:

1. OpenCV with CUDA will require C compiler and GPU compiler. It will require Microsoft visual studio and latest CUDA installation. The procedure to install the same is covered in the first chapter. So before moving ahead, please check that they are installed properly or not.
2. Download the source for the latest version of OpenCV by visiting the link: https://github.com/opencv/opencv/
3. There are some extra modules which are not included in OpenCV but they are available in the extra module call opencv_contrib which can be installed along with OpenCV. The functions available in this module are not stable, once they get stable they are moved to actual OpenCV source. If you want to install this module download it from https://github.com/opencv/opencv_contrib link.
4. Install cmake from following link: https://cmake.org/download/ .It is needed for compilation of OpenCV library.
5. Extract zip files of opencv and opencv_contrib in any folder. Here they are extracted in `C://opencv` and `C://opencv_contrib` folder.
6. Open cmake to compile OpenCV. In that, you need to select the path for OpenCV source and select the folder in which this source has to be built. It is shown in the figure below:

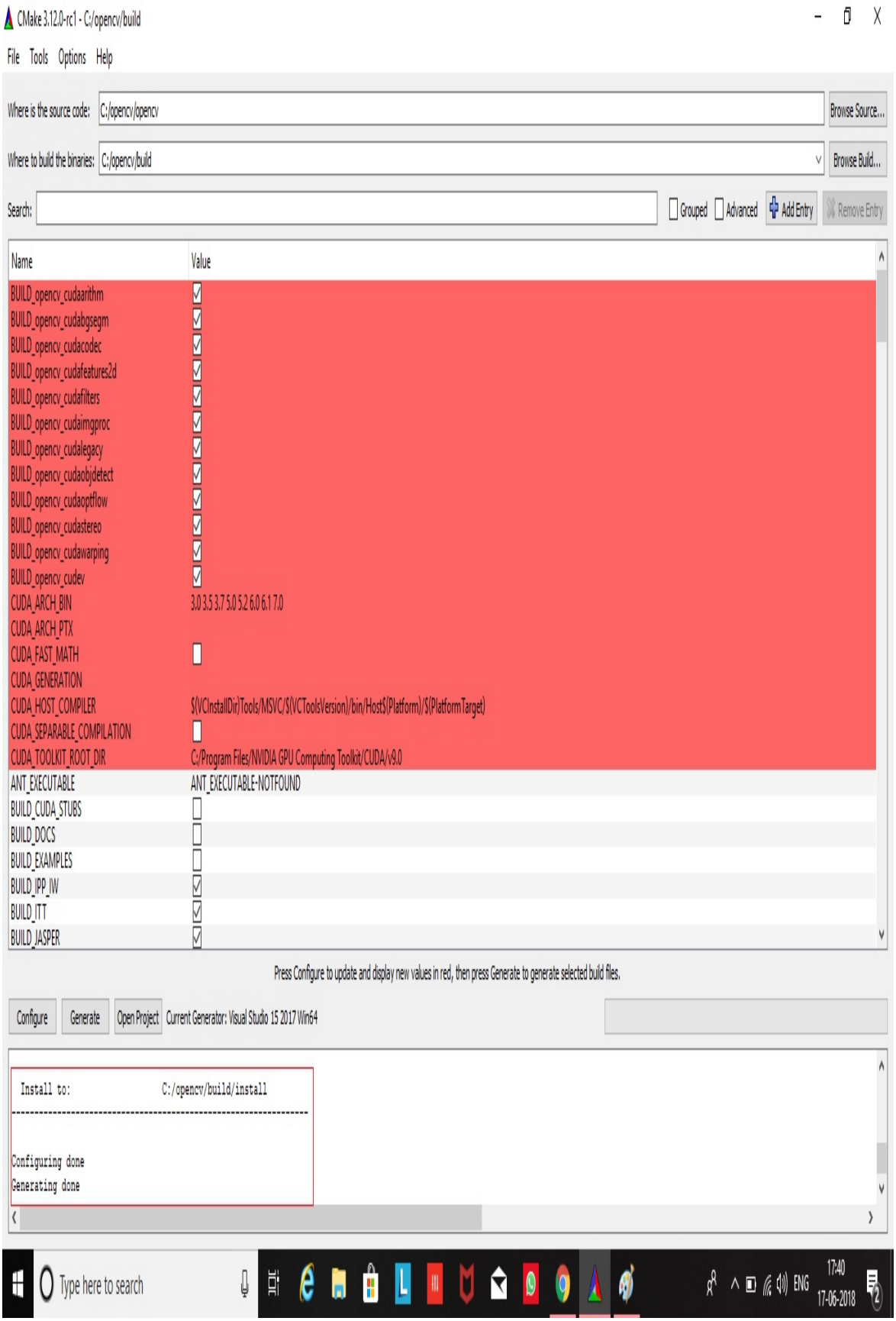
7. After that click on configure. It will start configuring the source. CMake will try to locate as many packages as possible based on the path settings in system variables. The configuration process is shown in the figure below:



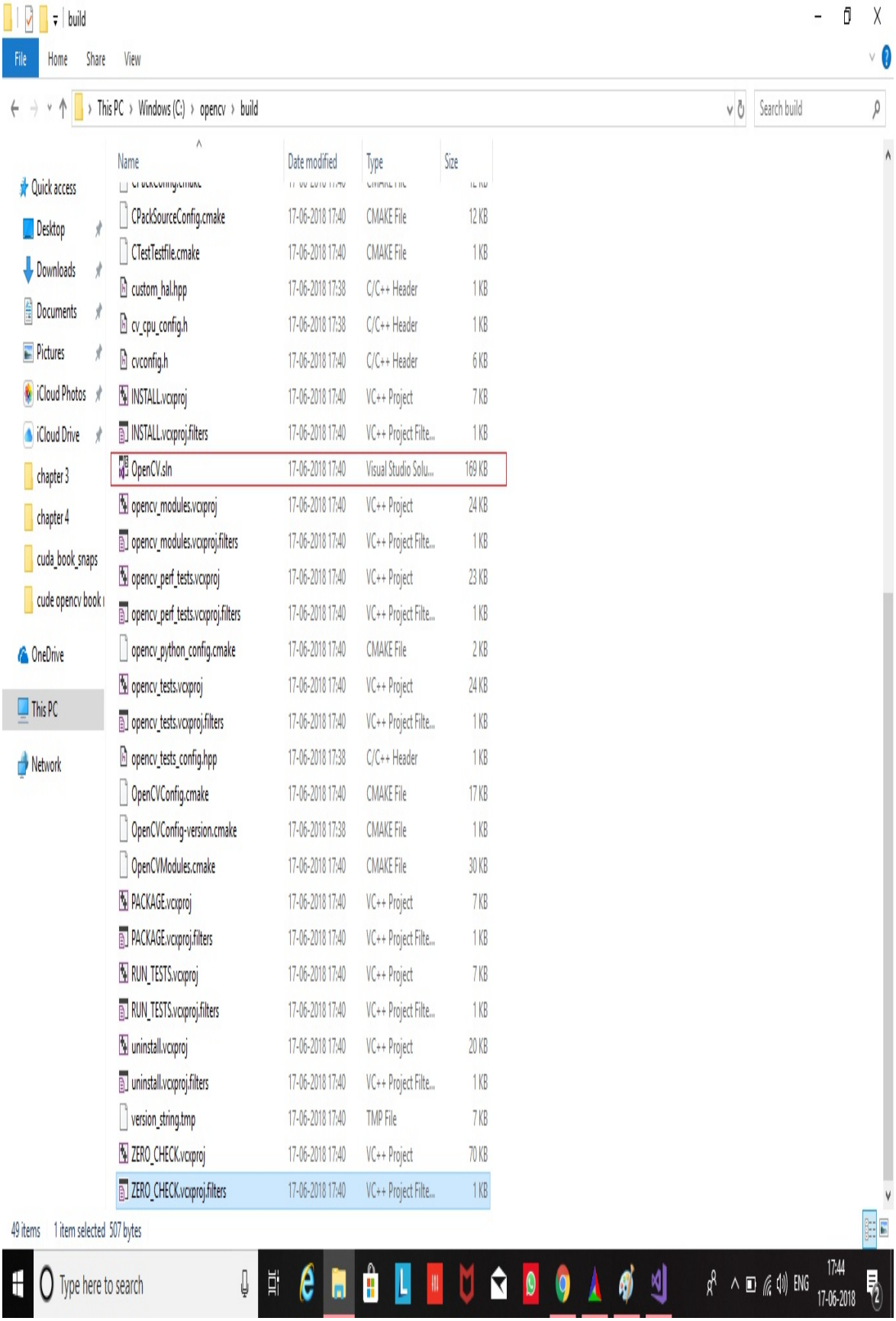
8. If some of the packages are not located then you can locate it manually. To configure OpenCV for installation with CUDA support, you have to check WITH_CUDA variable as shown in the figure below and then again click on configure again:



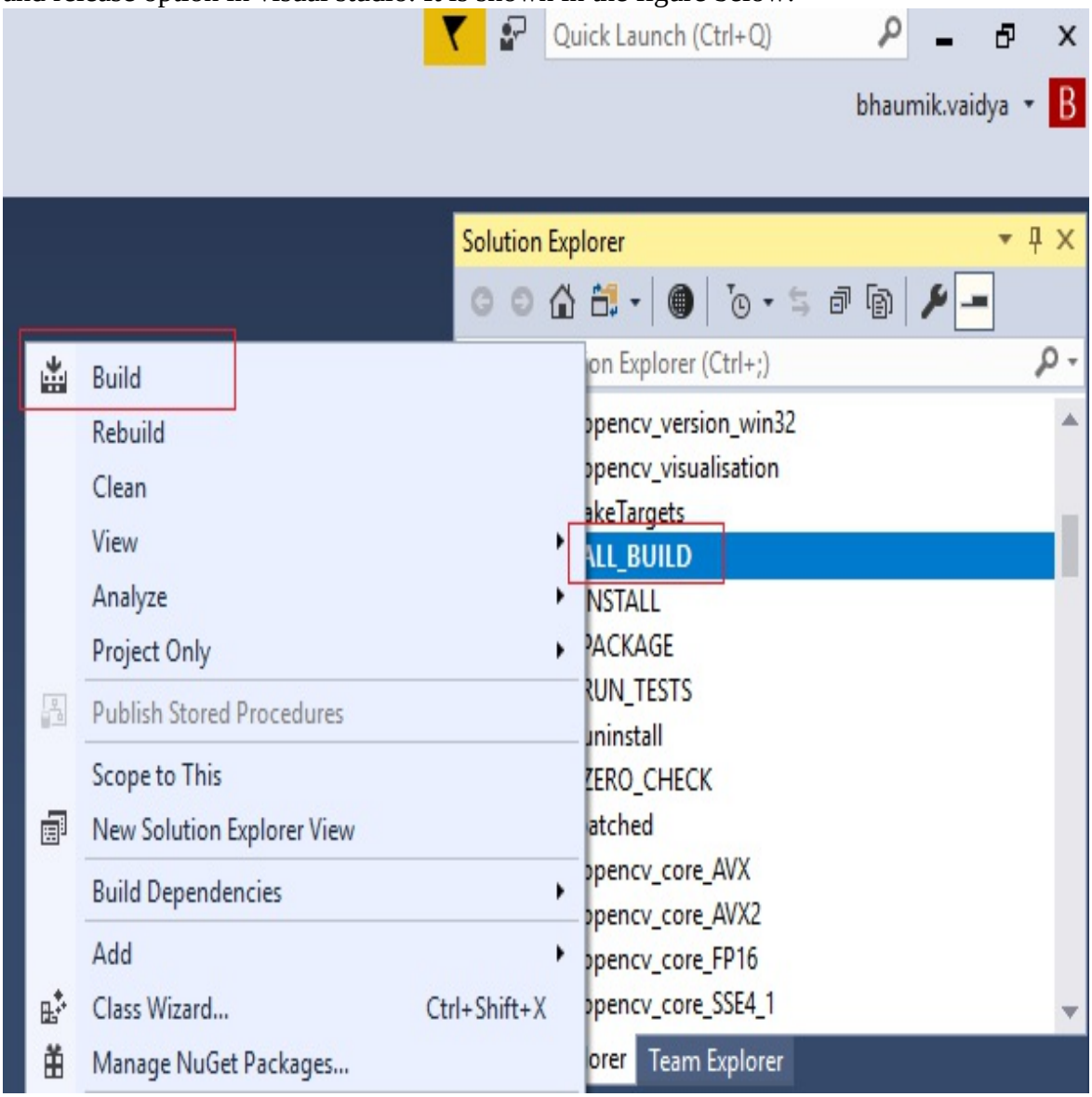
9. After configuration is finished, click on generate. This will create the visual studio project file based on your selection of version for visual studio. When generating is finished, the window should be something like what is shown below:


CMake 3.12.0-rc1 - C:/opencv/build
File Tools Options Help
Where is the source code: C:/opencv/opencv
Browse Source...
Where to build the binaries: C:/opencv/build
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
BUILD_opencv_cudaarithm
BUILD_opencv_cudabgsegm
BUILD_opencv_cudacodec
BUILD_opencv_cudafeatures2d
BUILD_opencv_cudafilters
BUILD_opencv_cudaimgproc
BUILD_opencv_cudalegacy
BUILD_opencv_cudaobjdetect
BUILD_opencv_cudaoptflow
BUILD_opencv_cudastereo
BUILD_opencv_cudawarping
BUILD_opencv_cudev
CUDA_ARCH_BIN
3.0 3.5 3.7 5.0 5.2 6.0 6.1 7.0
CUDA_ARCH_PTX
CUDA_FAST_MATH
CUDA_GENERATION
CUDA_HOST_COMPILER
$(VCInstallDir)Tools/MSVC/$(VCToolsVersion)/bin/Host$(Platform)/$(PlatformTarget)
CUDA_SEPARABLE_COMPILATION
CUDA_TOOLKIT_ROOT_DIR
C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v9.0
ANT_EXECUTABLE
ANT_EXECUTABLE-NOTFOUND
BUILD_CUDA_STUBS
BUILD_DOCS
BUILD_EXAMPLES
BUILD_IPP_IW
BUILD_ITT
BUILD_JASPER
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Open Project
Current Generator: Visual Studio 15 2017 Win64
Install to: C:/opencv/build/install
Configuring done
Generating done
Type here to search
ENG
17:40
17-06-2018

10. Go to build directory of opencv folder and find visual studio project with name OpenCV.sln as shown in the figure below:

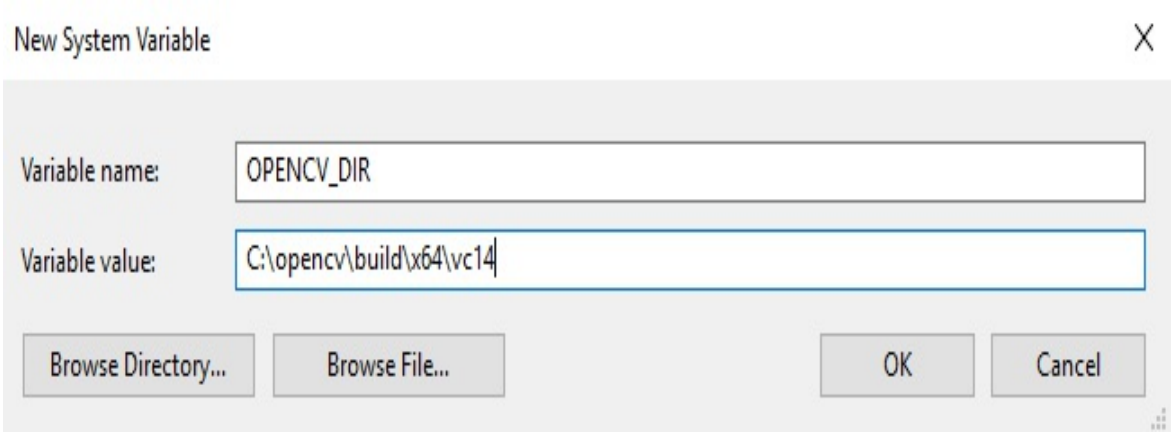

build
File
Home
Share
View
This PC › Windows (C:) › opencv › build
Search build
Quick access
Desktop
Downloads
Documents
Pictures
iCloud Photos
iCloud Drive
chapter 3
chapter 4
cuda_book_snaps
cude opencv book
OneDrive
This PC
Network
Name
Date modified
Type
Size
CPackSourceConfig.cmake 17-06-2018 17:40 CMAKE File 12 KB
CTestTestfile.cmake 17-06-2018 17:40 CMAKE File 1 KB
custom_hal.hpp 17-06-2018 17:38 C/C++ Header 1 KB
cv_cpu_config.h 17-06-2018 17:38 C/C++ Header 1 KB
cvconfig.h 17-06-2018 17:40 C/C++ Header 6 KB
INSTALL.vcxproj 17-06-2018 17:40 VC++ Project 7 KB
INSTALL.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
OpenCV.sln 17-06-2018 17:40 Visual Studio Solu... 169 KB
opencv_modules.vcxproj 17-06-2018 17:40 VC++ Project 24 KB
opencv_modules.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
opencv_perf_tests.vcxproj 17-06-2018 17:40 VC++ Project 23 KB
opencv_perf_tests.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
opencv_python_config.cmake 17-06-2018 17:40 CMAKE File 2 KB
opencv_tests.vcxproj 17-06-2018 17:40 VC++ Project 24 KB
opencv_tests.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
opencv_tests_config.hpp 17-06-2018 17:38 C/C++ Header 1 KB
OpenCVConfig.cmake 17-06-2018 17:40 CMAKE File 17 KB
OpenCVConfig-version.cmake 17-06-2018 17:38 CMAKE File 1 KB
OpenCVModules.cmake 17-06-2018 17:40 CMAKE File 30 KB
PACKAGE.vcxproj 17-06-2018 17:40 VC++ Project 7 KB
PACKAGE.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
RUN_TESTS.vcxproj 17-06-2018 17:40 VC++ Project 7 KB
RUN_TESTS.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
uninstall.vcxproj 17-06-2018 17:40 VC++ Project 20 KB
uninstall.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
version_string.tmp 17-06-2018 17:40 TMP File 7 KB
ZERO_CHECK.vcxproj 17-06-2018 17:40 VC++ Project 70 KB
ZERO_CHECK.vcxproj.filters 17-06-2018 17:40 VC++ Project Filte... 1 KB
49 items
1 item selected 507 bytes
Type here to search
ENG
17:44
17-06-2018

11. This will open the project in Microsoft visual studio. In solution explorer, find the project with name ALL_BUILD. Right click on it and build it. Build this project for both debug and release option in visual studio. It is shown in the figure below:



12. It will take a long time to build this entire project. Depending on your processor and visual studio version, it may vary. After successful completion of build operation, you are ready to use OpenCV library in your C/C++ Projects.

13. Setup environment variable OPENCV_DIR by right click on My computer -> advance system settings -> environment variables -> new. Set its value as `C:\opencv\build\x64\vc14`. Here vc14 will depend on the version of Microsoft visual studio.

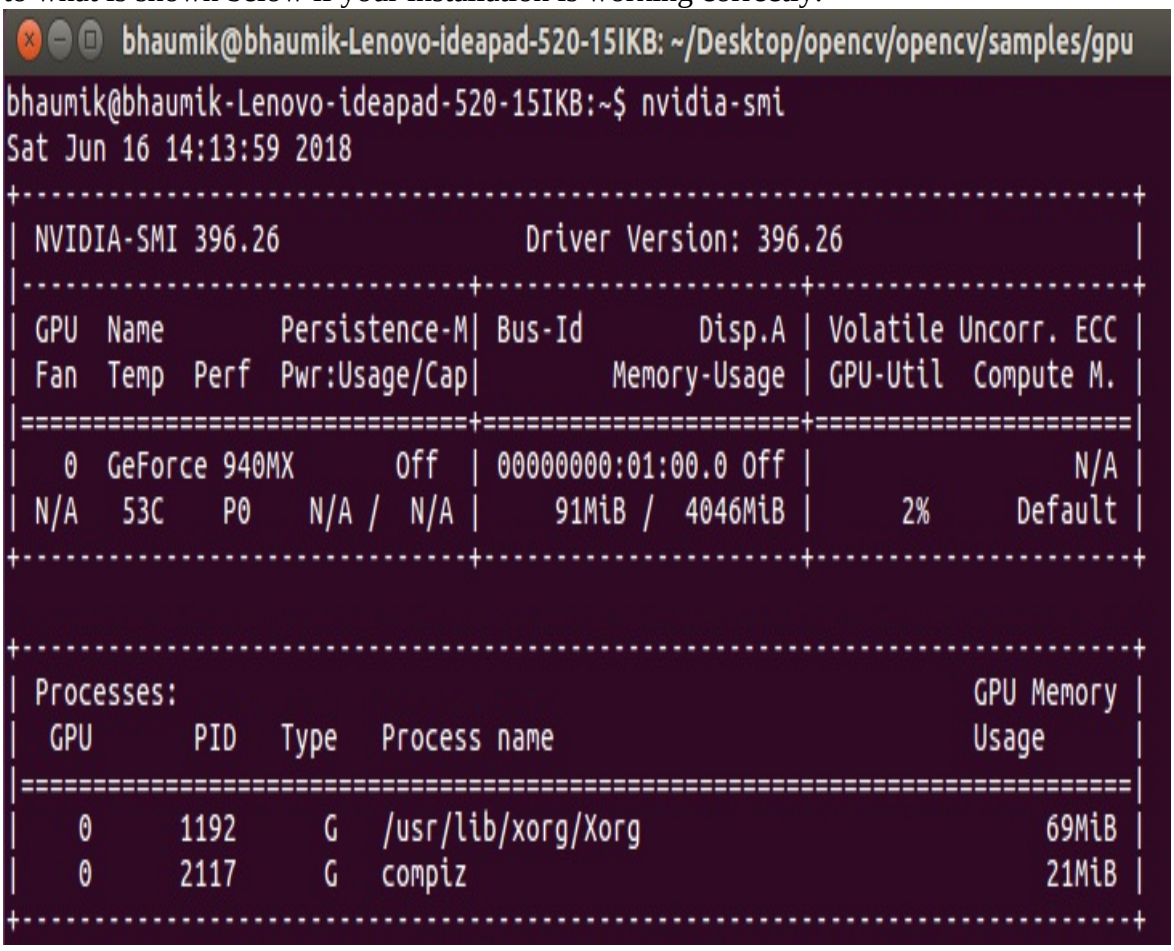


You can check the installation by going to `c://opencv/build/bin/Debug` directory and run any .exe applications.

Installation of OpenCV with CUDA support on Linux

This section covers the installation steps for OpenCV with CUDA support on Linux operating system. The steps are tested on Ubuntu 16.04 but they should work on any Unix distributions.

1. OpenCV with CUDA will require latest CUDA installation. The procedure to install the same is covered in the first chapter. So before moving ahead, please check that it is installed properly or not. You can check the installation of CUDA toolkit and supporting NVIDIA device driver by executing nvidia-sim command. You should see output similar to what is shown below if your installation is working correctly.



2. Download the source for the latest version of OpenCV by visiting the link: https://github.com/opencv/opencv/ . Extract it in opencv folder.
3. There are some extra modules which are not included in OpenCV but they are available in the extra module call opencv_contrib which can be installed along with OpenCV. The functions available in this module are not stable, once they get stable that are moved to actual OpenCV source. If you want to install this module download it from https://github.com/opencv/opencv_contrib link. Extract it in opencv_contrib folder in the

same directory as opencv folder.

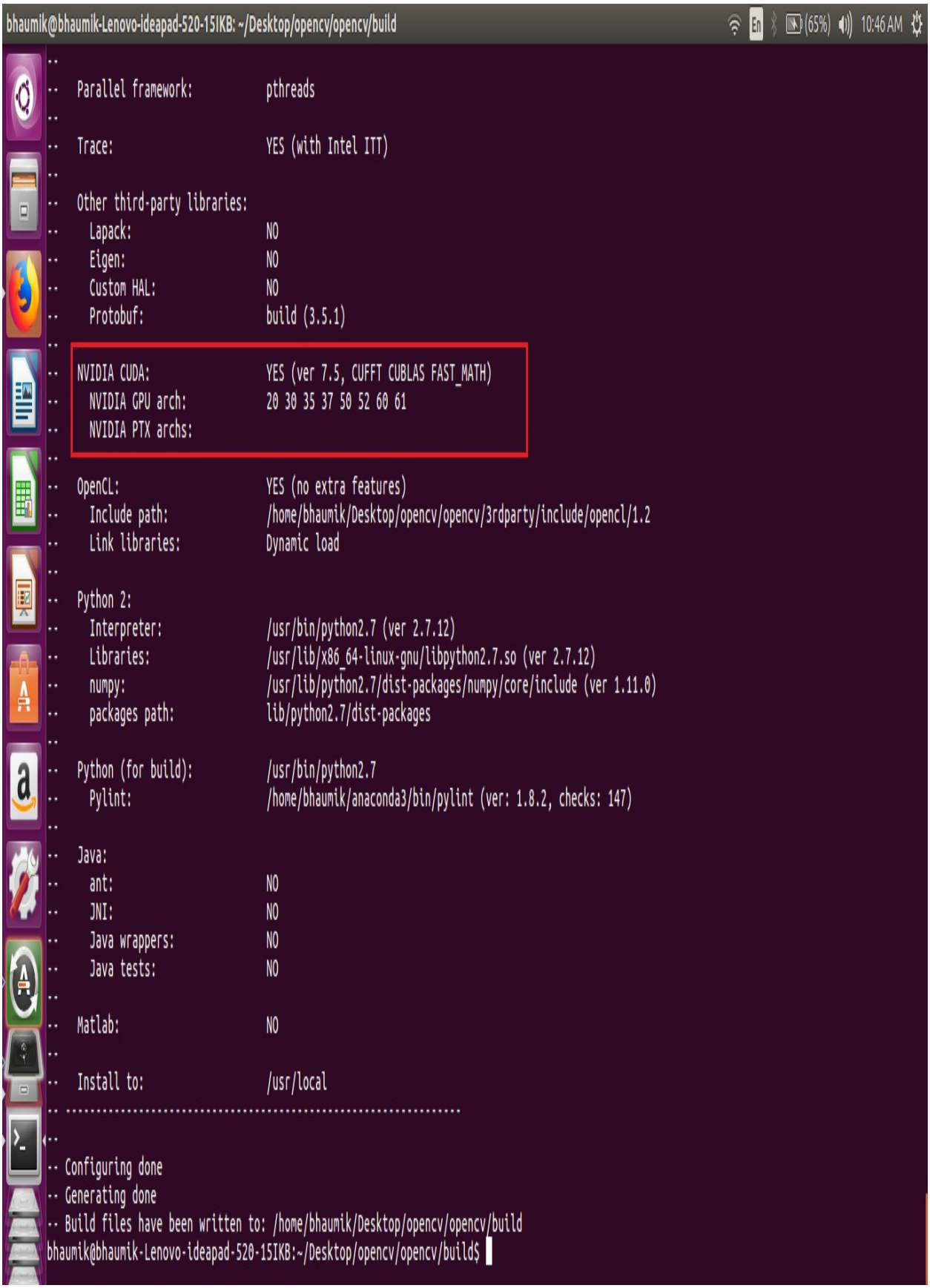
4. Now go inside opencv folder and create build directory. Then go inside this newly created build directory. These can be done by executing following commands from the command prompt.

```
$ cd opencv
$ mkdir build
$ cd build
```

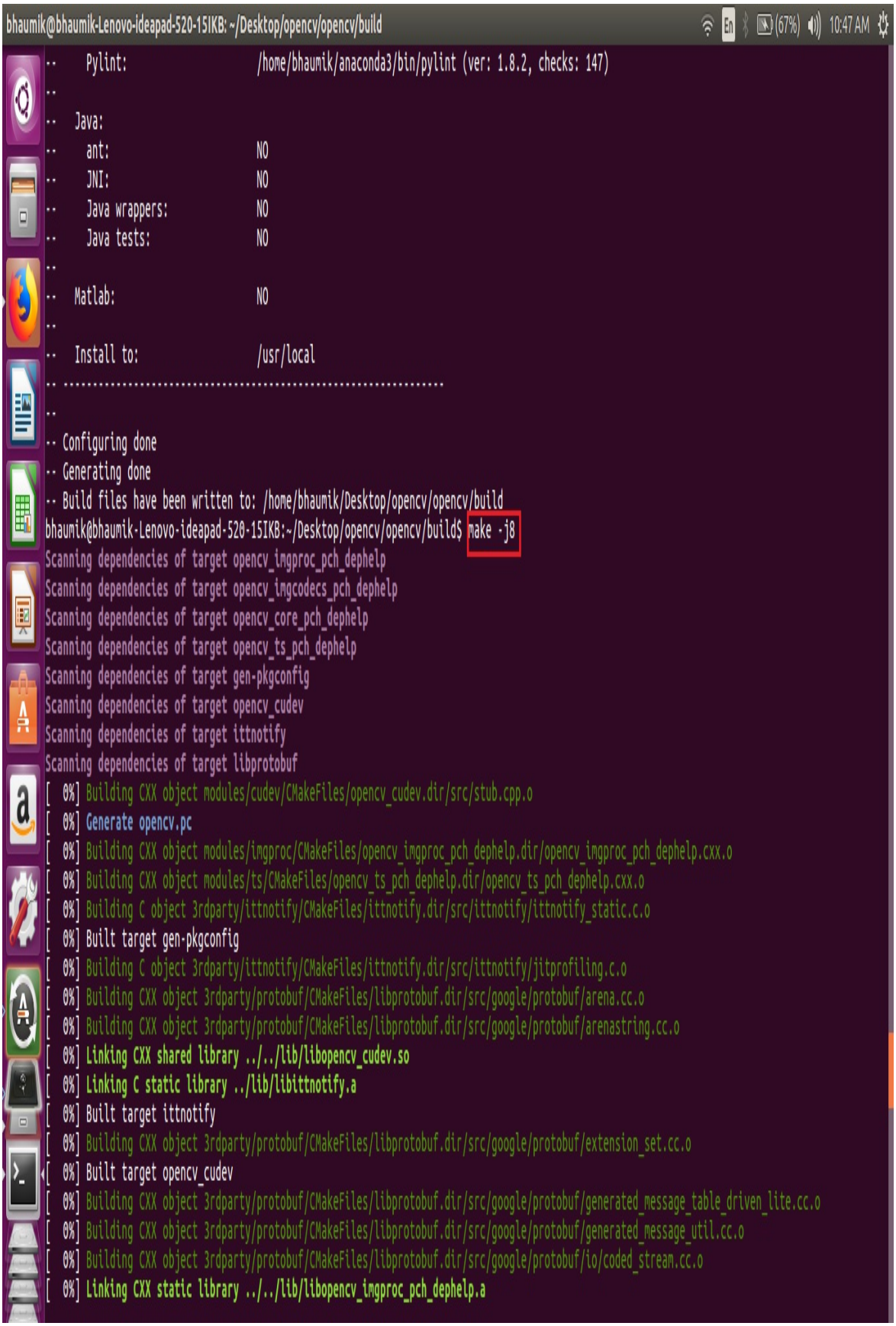
5. cmake command is used to compile opencv with CUDA support. Make sure WITH_CUDA flag is set to ON in this command along with a proper path for extra modules downloaded and saved in opencv_contrib directory. The entire cmake command is shown below:

```
cmake -D CMAKE_BUILD_TYPE=RELEASE CMAKE_INSTALL_PREFIX=/usr/local WITH_CUDA=ON  ENABLE_FAST_MATH=1
CUDA_FAST_MATH=1 -D WITH_CUBLAS=1 OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules
BUILD_EXAMPLES=ON ..
```

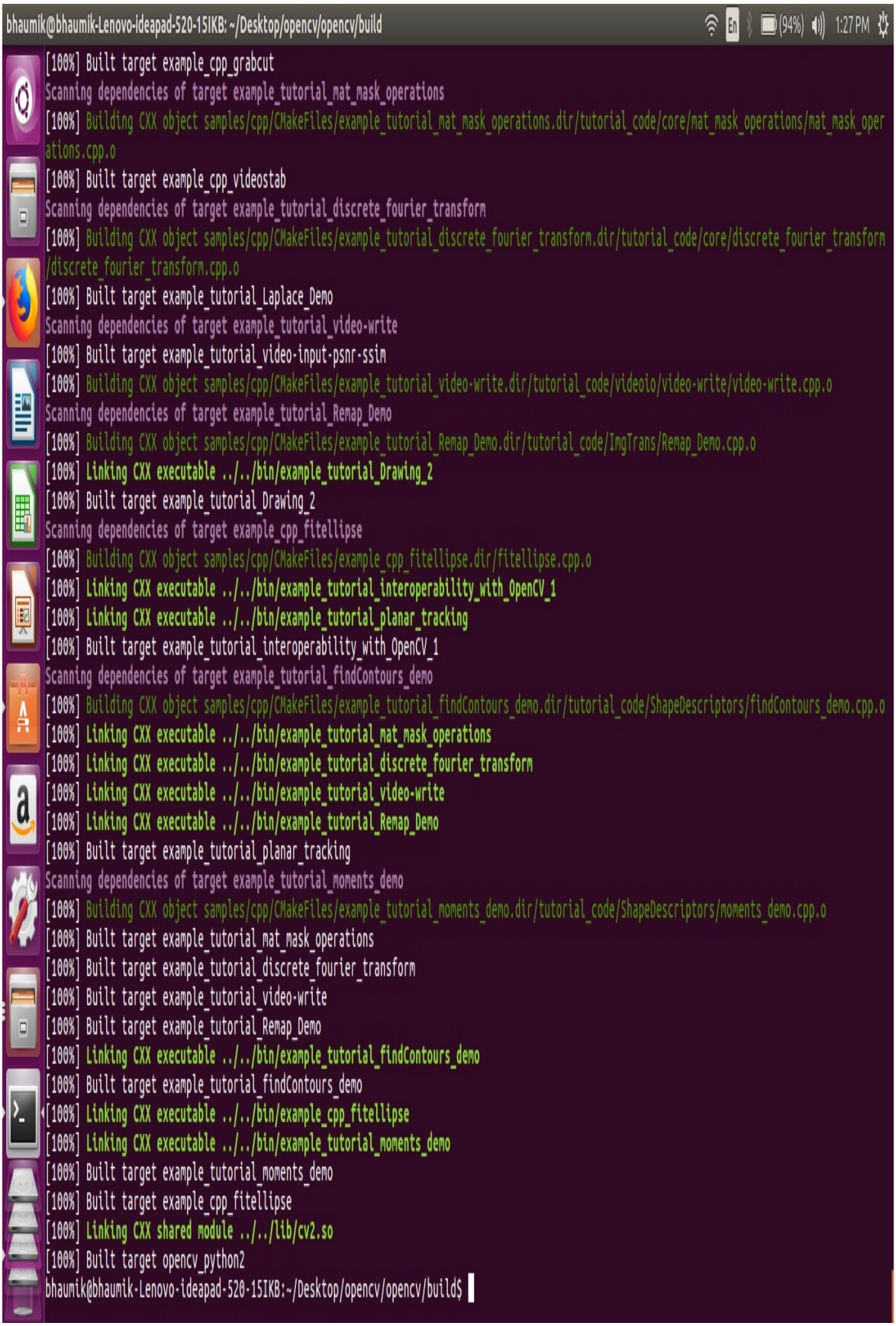
It will start the configuration and creation of makefile. It will locate all the extra modules based on the values in the system path. The output of cmake command with selected cuda installation is shown below:


bhaumik@bhaumik-Lenovo-ideapad-520-15IKB: ~/Desktop/opencv/opencv/build
(65%) 10:46 AM
--
-- Parallel framework: pthreads
--
-- Trace: YES (with Intel ITT)
--
-- Other third-party libraries:
--   Lapack: NO
--   Eigen: NO
--   Custom HAL: NO
--   Protobuf: build (3.5.1)
--
-- NVIDIA CUDA: YES (ver 7.5, CUFFT CUBLAS FAST_MATH)
--   NVIDIA GPU arch: 20 30 35 37 50 52 60 61
--   NVIDIA PTX archs:
--
-- OpenCL: YES (no extra features)
--   Include path: /home/bhaumik/Desktop/opencv/opencv/3rdparty/include/opencl/1.2
--   Link libraries: Dynamic load
--
-- Python 2:
--   Interpreter: /usr/bin/python2.7 (ver 2.7.12)
--   Libraries: /usr/lib/x86_64-linux-gnu/libpython2.7.so (ver 2.7.12)
--   numpy: /usr/lib/python2.7/dist-packages/numpy/core/include (ver 1.11.0)
--   packages path: lib/python2.7/dist-packages
--
-- Python (for build):   /usr/bin/python2.7
--   Pylint: /home/bhaumik/anaconda3/bin/pylint (ver: 1.8.2, checks: 147)
--
-- Java:
--   ant: NO
--   JNI: NO
--   Java wrappers: NO
--   Java tests: NO
--
-- Matlab: NO
--
-- Install to: /usr/local
-- ---
--
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bhaumik/Desktop/opencv/opencv/build
bhaumik@bhaumik-Lenovo-ideapad-520-15IKB:~/Desktop/opencv/opencv/build$

6. cmake will create a makefile in the build directory after successful configuration. To compile OpenCV using this makefile execute the command `make -j8` from the command window as shown below:


bhaumik@bhaumik-Lenovo-ideapad-520-15IKB: ~/Desktop/opencv/opencv/build
10:47 AM
-- Pylint: /home/bhaumik/anaconda3/bin/pylint (ver: 1.8.2, checks: 147)
-- Java:
-- ant: NO
-- JNI: NO
-- Java wrappers: NO
-- Java tests: NO
-- Matlab: NO
-- Install to: /usr/local
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bhaumik/Desktop/opencv/opencv/build
bhaumik@bhaumik-Lenovo-ideapad-520-15IKB:~/Desktop/opencv/opencv/build$ make -j8
Scanning dependencies of target opencv_imgproc_pch_dephelp
Scanning dependencies of target opencv_imgcodecs_pch_dephelp
Scanning dependencies of target opencv_core_pch_dephelp
Scanning dependencies of target opencv_ts_pch_dephelp
Scanning dependencies of target gen-pkgconfig
Scanning dependencies of target opencv_cudev
Scanning dependencies of target ittnotify
Scanning dependencies of target libprotobuf
[0%] Building CXX object modules/cudev/CMakeFiles/opencv_cudev.dir/src/stub.cpp.o
[0%] Generate opencv.pc
[0%] Building CXX object modules/imgproc/CMakeFiles/opencv_imgproc_pch_dephelp.dir/opencv_imgproc_pch_dephelp.cxx.o
[0%] Building CXX object modules/ts/CMakeFiles/opencv_ts_pch_dephelp.dir/opencv_ts_pch_dephelp.cxx.o
[0%] Building C object 3rdparty/ittnotify/CMakeFiles/ittnotify.dir/src/ittnotify/ittnotify_static.c.o
[0%] Built target gen-pkgconfig
[0%] Building C object 3rdparty/ittnotify/CMakeFiles/ittnotify.dir/src/ittnotify/jitprofiling.c.o
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/arena.cc.o
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/arenastring.cc.o
[0%] Linking CXX shared library ../../lib/libopencv_cudev.so
[0%] Linking C static library ../lib/libittnotify.a
[0%] Built target ittnotify
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/extension_set.cc.o
[0%] Built target opencv_cudev
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/generated_message_table_driven_lite.cc.o
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/generated_message_util.cc.o
[0%] Building CXX object 3rdparty/protobuf/CMakeFiles/libprotobuf.dir/src/google/protobuf/io/coded_stream.cc.o
[0%] Linking CXX static library ../../lib/libopencv_imgproc_pch_dephelp.a

7. After successful compilation, to install OpenCV you have to execute the command sudo `make install` from the command line. The output of that command is shown below:


bhaumik@bhaumik-Lenovo-ideapad-520-15IKB: ~/Desktop/opencv/opencv/build
(94%)
1:27 PM
[100%] Built target example_cpp_grabcut
Scanning dependencies of target example_tutorial_mat_mask_operations
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_mat_mask_operations.dir/tutorial_code/core/mat_mask_operations/mat_mask_operations.cpp.o
[100%] Built target example_cpp_videostab
Scanning dependencies of target example_tutorial_discrete_fourier_transform
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_discrete_fourier_transform.dir/tutorial_code/core/discrete_fourier_transform/discrete_fourier_transform.cpp.o
[100%] Built target example_tutorial_Laplace_Demo
Scanning dependencies of target example_tutorial_video-write
[100%] Built target example_tutorial_video-input-psnr-ssim
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_video-write.dir/tutorial_code/videoio/video-write/video-write.cpp.o
Scanning dependencies of target example_tutorial_Remap_Demo
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_Remap_Demo.dir/tutorial_code/ImgTrans/Remap_Demo.cpp.o
[100%] Linking CXX executable ../../bin/example_tutorial_Drawing_2
[100%] Built target example_tutorial_Drawing_2
Scanning dependencies of target example_cpp_fitellipse
[100%] Building CXX object samples/cpp/CMakeFiles/example_cpp_fitellipse.dir/fitellipse.cpp.o
[100%] Linking CXX executable ../../bin/example_tutorial_interoperability_with_OpenCV_1
[100%] Linking CXX executable ../../bin/example_tutorial_planar_tracking
[100%] Built target example_tutorial_interoperability_with_OpenCV_1
Scanning dependencies of target example_tutorial_findContours_demo
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_findContours_demo.dir/tutorial_code/ShapeDescriptors/findContours_demo.cpp.o
[100%] Linking CXX executable ../../bin/example_tutorial_mat_mask_operations
[100%] Linking CXX executable ../../bin/example_tutorial_discrete_fourier_transform
[100%] Linking CXX executable ../../bin/example_tutorial_video-write
[100%] Linking CXX executable ../../bin/example_tutorial_Remap_Demo
[100%] Built target example_tutorial_planar_tracking
Scanning dependencies of target example_tutorial_moments_demo
[100%] Building CXX object samples/cpp/CMakeFiles/example_tutorial_moments_demo.dir/tutorial_code/ShapeDescriptors/moments_demo.cpp.o
[100%] Built target example_tutorial_mat_mask_operations
[100%] Built target example_tutorial_discrete_fourier_transform
[100%] Built target example_tutorial_video-write
[100%] Built target example_tutorial_Remap_Demo
[100%] Linking CXX executable ../../bin/example_tutorial_findContours_demo
[100%] Built target example_tutorial_findContours_demo
[100%] Linking CXX executable ../../bin/example_cpp_fitellipse
[100%] Linking CXX executable ../../bin/example_tutorial_moments_demo
[100%] Built target example_tutorial_moments_demo
[100%] Built target example_cpp_fitellipse
[100%] Linking CXX shared module ../../lib/cv2.so
[100%] Built target opencv_python2
bhaumik@bhaumik-Lenovo-ideapad-520-15IKB:~/Desktop/opencv/opencv/build$

8. Run `sudo ldconfig` command to finish the installation. It creates necessary links and cache to the opencv libraries.
9. You can check the installation by running any example from opencv/samples/gpu folder.

For using OpenCV in the program, you have to include opencv2/opencv.hpp header file. This header file will include all other header files necessary for the program. So all the OpenCV programs have to include this header file on the top.

Working with Images in OpenCV

Now that OpenCV is installed on the system, we can start working with images using OpenCV. In this section, we will try to learn how images are represented inside OpenCV, develop programs to read an image, display an image and save an image on disk. We will also see the method to create synthetic images in OpenCV. We will try to draw different shapes on an image using OpenCV. Along with this, important syntax and features of OpenCV will also be explained.

Image representation inside OpenCV

As described earlier, Images are nothing but two-dimensional arrays so they should be stored as an array inside a computer for processing. OpenCV provides a Mat class which is nothing but an image container used to store an image. The mat object can be created and assigned to an image in two separate lines as shown below:

```
Mat img;
img= imread("cameraman.tif");
```

The datatype of an Image and size of the two-dimensional array can also be defined while creating an object. The datatype of an image is very important as it signifies the number of channels and number of bits used to specify a single pixel value. Grayscale images have single channel while color images are a combination of three separate channels Red, Green, and Blue.

The number of bits used for single pixel specifies the number of discrete gray level values. An 8-bit image can have gray levels between 0 and 255 while 16-bit images can have gray levels between 0 to 65,535. OpenCV supports many data types with CV_8U as default which indicates 8-bit unsigned image with a single channel. It is equivalent to CV_8UC1. The color images can be specified as CV_8UC3 which indicate 8-bit unsigned image with 3 channels. OpenCV supports up to 512 channels. The channels more than 5 has to be defined in round brackets, for example, CV_8UC(5) which indicates 8-bit image with 5 channels. OpenCV also supports signed numbers so datatype can also be like CV_16SC3 which specifies 16-bit signed image with 3 channels.

Mat object can be used to define the size of an image. It is also called resolution of an image. It indicates the number of pixels in horizontal and vertical direction. Normally resolution of an image is defined in terms of width x height. While the size of an Array in Mat object should be defined in terms of the number of rows x number of columns. Some examples of using Mat to define image containers are shown below:

```
Mat img1(6,6,CV_8UC1);
//This defines img1 object with size of 6x6, unsigned 8-bit integers and single channel.

 Mat img2(256,256, CV_32FC1)
//This defines img2 object with size of 256x256, 32 bit floating point numbers and single channel.

Mat img3(1960,1024, CV_64FC3)
//This defines img3 object with size of 1960x1024, 64 bit floating point numbers and three channels.
```

The resolution and size of the image will determine the space needed to save an Image on disk. Suppose the size of a color Image with three channels is 1024 x 1024 then it will take 3 x1024 x1024 bytes= 3 Mbytes to store an Image on disk. In the next section, we will see how to use this Mat object and OpenCV to read and display Image.

Read and Display an Image

In this section, we will try to develop the code for reading and displaying an image using C++ and OpenCV. The entire code for this is shown below and then it is explained line by line.

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
 // Read the image
 Mat img = imread("images/cameraman.tif",0);

 // Check for failure in reading an Image
 if (img.empty())
 {
  cout << "Could not open an image" << endl;
  return -1;
 }
//Name of the window
 String win_name = "My First Opencv Program";

 // Create a window
 namedWindow(win_name);

 // Show our image inside the created window.
imshow(win_name, img);

// Wait for any keystroke in the window
waitKey(0);

//destroy the created window
 destroyWindow(win_name);

 return 0;
}
```

The program starts with including header files for standard input-output and Image Processing.

The functions from std namespace like cout and endl are used in the program so std namespace is added. All the OpenCV classes and functions are defined using cv namespace. So to use functions defined in cv namespace, we are specifying `using namespace cv` line. If that line is omitted then every function in cv namespace have to be used in the following way.

```
Mat img = cv::imread("cameraman.tif")
```

The main function contains the code for reading and displaying an image. The imread command is used to read an image in OpenCV. It returns a Mat object . The imread command has two Arguments. The first argument is the name of an image along with its path. The path can be specified in two ways. You can specify a fully qualified path of an image in your PC or you can specify relative path of an image from your code file. In the example above relative path is used where an image is located in images folder which is in the same directory as the code file.

The second argument is optional which specifies whether the image is to be read is a grayscale image or color image. If the image is to be read as color image then specify IMREAD_COLOR or 1. If image is to be read as a grayscale image then specify IMREAD_GRAYSCALE or 0. If

image is to be read in its saved form then specify IMREAD_UNCHANGED or -1 as second argument. If image is read as a color image then the imread command will return three channels starting with blue, green and red (BGR format). If second argument is not provided then the default value is IMREAD_COLOR which reads an image as a color image.

If somehow image can't be read or it is not available on the disk then imread command will return a Null Mat object. If this happens then there is no need to continue with further image processing code and we can exit at this point with notifying the user about the error. This is handled by the code inside the if loop.

The window should be created in which image will be displayed. OpenCV provides a function called namedWindow for that. It requires two arguments. The first argument is the name of the window. It has to be a string. The second argument specifies the size of the window that is to be created. It can take two values: WINDOW_AUTOSIZE or WINDOW_NORMAL.
If WINDOW_AUTOSIZE is specified then the user will not be able to resize the window and image will be displayed in its original size and if WINDOW_NORMAL is specified then the user will be able to resize the window. This argument is optional and its default value is WINDOW_AUTOSIZE if it is not specified.

To display an image in the created window imshow command is used. This command requires two arguments. The first argument is the name of the window created using namedWindow command and the second argument is the image variable that has to be displayed. This variable has to be a mat object. For displaying multiple images, separate windows with unique names have to be created. The name of the window will appear as a title on the image window.

The imshow function should be provided enough time to display an image in the created window. This is done by using the waitKey function. So imshow function should be followed by waitkey function in all OpenCV programs otherwise image will not be displayed. waitKey is a keyboard binding function and it accepts one argument which is a time in milliseconds. It will wait for the specified time for a keystroke then it will move to the next line of code. If no argument is specified or 0 is specified then it will wait for an indefinite time period for a keystroke. It will move to the next line only when any key is pressed on a keyboard. We can also detect whether a specific key is pressed and depending on the key pressed we can take certain decisions. We will use this feature later on in this chapter.

All the windows created for displaying the windows need to be closed before the termination of the program. This can be done using distroyAllWindows function. It will close all the windows created using namedWindow function during the program for displaying an image. There is a function called distroyWindow which closes specific windows. The name of the window should be provided as an argument to distroyWindow function.

For the execution of a program, just copy the code and paste it in visual studio if using on Windows or make a cpp file of that in Ubuntu. The build method is similar to normal cpp application in visual studio so it is not repeated here. For execution on Ubuntu, execute following commands on command prompt from the folder of a saved cpp file.

```
For compilation:
$ g++ -std = c++11 image_read.cpp 'pkg_config --libs --cflags opencv' -o image_read
For execution:
$./image_read
```

The output of the preceding program after execution is as follows:

Read and Display Color Image

In the above program, the second argument for imread is specified as 0 which means that it will read an image as a grayscale image. Suppose you want to read any color image then you can change imread command in the following way.

```
Mat img = imread("images/autumn.tif",1);
```

The second argument is specified as 1 which means that it will read an image in BGR form. It is important to note that OpenCV's imread and imshow use BGR format for color images which is different than RGB format used by MATLAB and other image processing tools. The output after changing imread is shown below:

The same image can be read as a grayscale image even though it is a color image by providing 0 as a second argument. This will convert an image into grayscale implicitly and then read. The image will look like what is shown below:

It is very important to remember how you are reading the image using imread function because it will affect other image processing code of your program.

To summarize, we have seen how to read an image and display it using OpenCV in this section. In the process, we have also learned some important functions available in OpenCV. In the next section, we will see how to create a synthetic image using OpenCV.

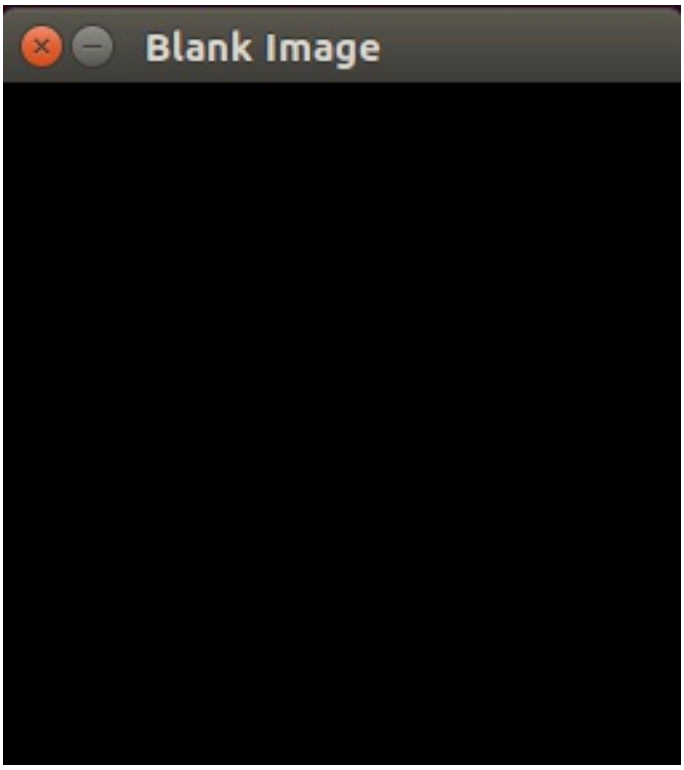
Creating Images using OpenCV

Sometimes, we may encounter the need to create our own image or draw some shapes on top of existing images. It is also needed when we want to draw bounding boxes around the detected object or we want to display labels on the image. So in this section, we will see how to create a blank grayscale and color images. We will also see the functions to draw line, rectangles, ellipse, circle and text on images.

To create an empty black image of the size 256x256, following code can be used.

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
 //Create blank black grayscale Image with size 256x256
 Mat img(256, 256, CV_8UC1, 0);
 String win_name = "Blank Image";
 namedWindow(win_name);
 imshow(win_name, img);
 waitKey(0);
 destroyWindow(win_name);
 return 0;
}
```

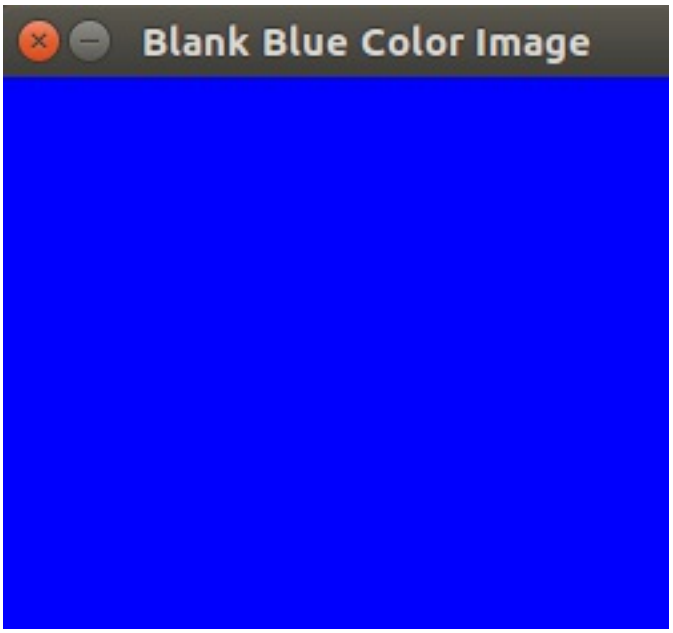
The code is more or less similar to the code developed for reading an image but instead of using imread command here the image is created using the constructor of Mat class only. As discussed earlier, we can provide size and datatype while creating a Mat object. So while creating an img object, we have provided four arguments. The first two arguments specify the size of an image with first defines the number of rows (height) and second defines the number of columns (width). The third argument defines the datatype of an image. We have used CV_8UC1 which means 8-bit unsigned integer image with a single channel. The last argument specifies the initialization value for all pixels in an Array. Here we have used 0 which is value for black color. When this program is executed it will create a black image of size 256x256 as shown below:



Similar code can be used for creating a blank color image as shown below:

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
//Create blank blue color Image with size 256x256
 Mat img(256, 256, CV_8UC3, Scalar(255,0,0));
 String win_name = "Blank Blue Color Image";
 namedWindow(win_name);
 imshow(win_name, img);
 waitKey(0);
 destroyWindow(win_name);
 return 0;
}
```

While creating of Mat object, instead of using CV_8UC1 datatype, CV_8UC3 is used which specifies 8-bit image with three channels. So there are 24 bits for a single pixel. The fourth argument specifies the starting pixel values. It is specified using Scalar keyword and tuple of three values specifying starting values in all three channels. Here, the blue channel is initialized with 255, the green channel is initialized with 0 and red channel is initialized with 0. This will create an image of size 256 x 256 in blue color. Different combinations of values in tuple will create different colors. The output of the above program is shown below:


Blank Blue Color Image

Drawing shapes on the blank image

To start drawing different shapes on an image, we will start by creating a blank black image of arbitrary size by the following command.

```
Mat img(512, 512, CV_8UC3, Scalar(0,0,0));
```

This command will create a black color image with the size of 512 x 512. Now, we will start by drawing different shapes on this image.

Draw a Line

The line can be specified by two points: the starting point and ending point. To draw a line on the image, these two points have to be specified. The function to draw a line on an image is shown below:

```
line(img,Point(0,0),Point(511,511),Scalar(0,255,0),7);
```

The line function has five arguments. The first argument specifies the image on which line needs to be drawn, Second and third arguments define the starting point and ending points respectively. The points are defined using Point class constructor which takes x and y coordinates of an image as argument. The fourth argument specifies the color of the line. It is specified as a tuple of B, G and R values. Here the value taken is (0,255,0) which specifies a green color. The fifth argument is the thickness of the line. Its value is taken as 7 pixels wide. This function also has an optional linetype argument. The function above will draw a diagonal green line of 7 pixels wide from (0,0) to (511,511).

Draw a rectangle

The rectangle can be specified using two extreme diagonal points. OpenCV provides a function to draw the rectangle on an image which has a syntax as shown below:

```
rectangle(img,Point(384,0),Point(510,128),Scalar(255,255,0),5);
```

Rectangle function has five arguments. The first argument is the image on which rectangle is to be drawn. The second argument is a top-left point of the rectangle. The third argument is a bottom-right point of the rectangle. The fourth argument specifies the color of the border. It is specified as (255,255,0) which is a mix of blue and green color. It gives cyan color. The fifth argument is the thickness of the border. If fifth argument is specified as -1 then the shape will be filled with color. So the function above will draw a rectangle with two extreme points (384,0) and (510,128) in cyan color with border thickness of 5 pixels.

Draw a circle

A circle can be specified by a center and its radius. OpenCV provides a function to draw a circle on an image which has a syntax as shown below:

```
circle(img,Point(447,63), 63, Scalar(0,0,255), -1);
```

The circle function has five arguments. the first argument is the image on which circle needs to be drawn. The second argument specifies the center point for a circle and the third argument specifies the radius of the circle. The fourth argument specifies the color of the circle. the value taken is (0,0,255) which is a red color. The fifth argument is a thickness of the border. It is taken as -1 which means the circle will be filled with red color.

Draw an ellipse

OpenCV provides a function to draw an ellipse on an image which has a syntax as shown below:

```
ellipse(img,Point(256,256),Point(100,100),0,0,180,255,-1);
```

The ellipse function has many arguments. The first argument specifies the image on which ellipse needs to be drawn. The second argument specifies the center of the ellipse. The third argument specifies the box size under which ellipse will be drawn. The fourth argument specifies the angle by which ellipse needs to be rotated. It is taken as 0 degrees. The fifth and sixth argument specifies the range of angles for which ellipse needs to be drawn. It is taken as 0 to 180 degrees. So only half ellipse will be drawn. The next argument specifies the color of an ellipse which is specified only as 255. It is same as (255,0,0) which is blue color. The final argument specifies the thickness of border. It is taken as -1 so ellipse will be filled with blue color.

Writing text on Image

If you want to write some text on an image then OpenCV provides a function for that which is putText. The syntax for the funtion is shown below:

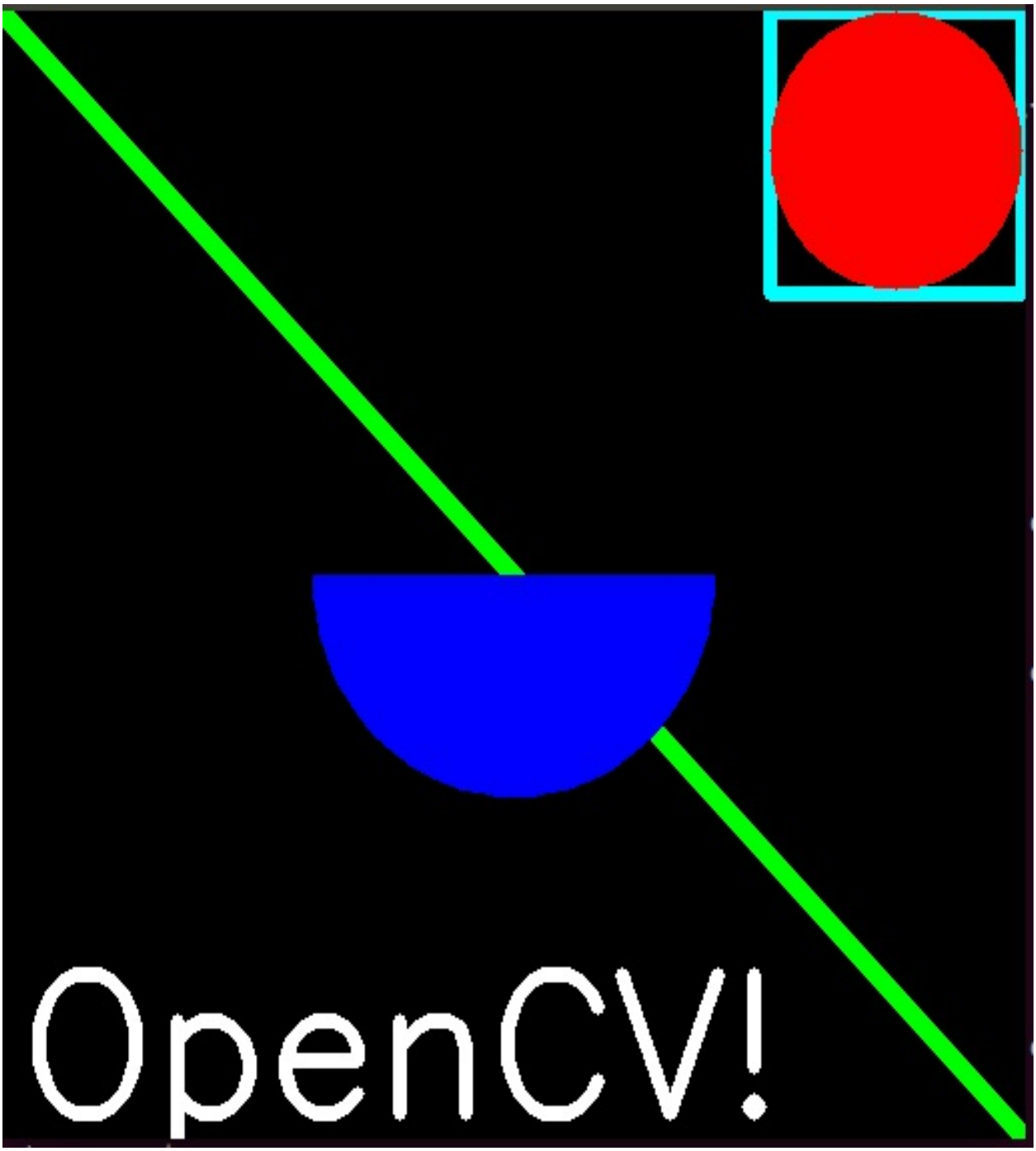
```
putText( img, "OpenCV!", Point(10,500), FONT_HERSHEY_SIMPLEX, 3,Scalar(255, 255, 255), 5, 8 );
```

The putText function has many arguments. The first argument is the image on which text is to be written. The second argument is the text as String datatype, which we want to write on image. The third argument specifies the bottom left corner of text. The fourth argument specifies the type of font. There are many font types available in OpenCV. You can check OpenCV documentation for more font types. The fifth argument specifies the scale for the font. The sixth argument is the color for text. It is taken as (255,255,255) which makes white color. The seventh argument is a thickness of the text which is taken as 5 and the last argument specifies linetype which is taken as 8.

We have seen separate functions to draw shapes on an empty black image. The code below shows the combination of all the functions discussed above.

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{

 Mat img(512, 512, CV_8UC3, Scalar(0,0,0));
 line(img,Point(0,0),Point(511,511),Scalar(0,255,0),7);
 rectangle(img,Point(384,0),Point(510,128),Scalar(255,255,0),5);
 circle(img,Point(447,63), 63, Scalar(0,0,255), -1);
 ellipse(img,Point(256,256),Point(100,100),0,0,180,255,-1);
 putText( img, "OpenCV!", Point(10,500), FONT_HERSHEY_SIMPLEX, 3,Scalar(255, 255,  255), 5, 8 );
 String win_name = "Shapes on blank Image";
 namedWindow(win_name);
 imshow(win_name, img);
 waitKey(0);
 destroyWindow(win_name);
 return 0;
}
```

The output image for the above code is shown below:


OpenCV!

Saving Image to a file

The images can also be saved to a disk from OpenCV program. It is helpful when we want to store our processed image to a disk on a computer. OpenCV provides imwrite function to do this operation. The syntax of this function is shown below:

```
bool flag = imwrite("images/save_image.jpg", img);
```

The imwrite function takes two arguments. The first argument is the name of the file you want to give along with its path. The second argument is the image variable which you want to save. This function returns a Boolean value which indicates the file is saved successfully or not on a disk.

In this section, we have worked with images using OpenCV. In the next section, we will work with videos which are nothing but a sequence of images using OpenCV.

Working with videos in OpenCV

This section will show the process for reading videos from a file and webcam using OpenCV. It will also describe the process to save videos to a file. It can work with USB cameras attached to computers also. Videos are nothing but a sequence of images. Though OpenCV is not optimized for video processing applications, it does a decent job at it. OpenCV is not able to capture audio so we have to use some other utilities with OpenCV to capture both audio and video.

Working with video stored in Computer

This section describes the process for reading a video file stored on a computer. All the frames from a video will be read one by one, operated upon and displayed on the screen in all video processing applications using OpenCV.

The code for reading and displaying video is shown below and then it is explained line by line:

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;
using namespace std;
int main(int argc, char* argv[])
{
 //open the video file from PC
 VideoCapture cap("images/rhinos.avi");
 // if not success, exit program
 if (cap.isOpened() == false)
 {
  cout << "Cannot open the video file" << endl;
  return -1;
 }
cout<<"Press Q to Quit" << endl;
String win_name = "First Video";
namedWindow(win_name);
 while (true)
 {
  Mat frame;
  // read a frame
  bool flag = cap.read(frame);

  //Breaking the while loop at the end of the video
  if (flag == false)
  {
   break;
  }
  //display the frame
  imshow(win_name, frame);
  //Wait for 100 ms and key 'q' for exit
  if (waitKey(100) == 'q')
  {
    break;
  }
 }
destroyWindow(win_name);
return 0;
}
```

After including libraries, the first thing that needs to be done inside the main function to process video is to create an object of VideoCapture. VideoCapture class has many constructors available to work with videos. When we want to work with video files stored on computer, we need to provide the name of the video along with its path as an argument to constructor while creating an object of VideoCapture.

This object provides many methods and properties which gives information related to a video. We will see those as and when they are required. It provides isopened property which indicates whether the object creation was successful and video is available or not. It returns a boolean value. If cap.isopened is false then the video is not available so there is no need to go ahead in the program. So that is handled by if loop which exits the program after notifying the user when a video is not available.

VideoCapture class provides a read method which captures the frame one by one. To process the entire video we have to start a continuous loop which runs until the end of a video. The infinite while loop can do this job. Inside the while loop, the first frame is read using read method. This method has one argument. It is a mat object in which we want to store the frame. It returns a boolean value which indicates whether the frame has been read successfully or not. When the loop will reach the end of video then this boolean will return false indicating there is no frame available. This flag is checked continuously in the loop for the end of the video, if it is detected then we come out of the while loop using the break statement.

The frame is a single image so displaying process for that is same as what we have seen earlier. In the code above, the waitKey function is used inside an if statement. It is waiting for 100ms after every frame for a keystroke. The if statement is checking whether the keystroke is 'q' or not. If it is 'q' then it means that user wants to quit the video so break statement is included inside if.

This code will terminate displaying video either when the whole video is finished or user presses 'q' from a keyboard. We will use this coding practice throughout this book while processing videos. The output of above program is shown below. It is one of the frames from a video.

We have used 100 ms delay between every frames. What do you think iwill happen when you decrease this value to suppose 10 ms? The answer is, frames will be displayed faster. It does not mean frame rate of the video is changing. It just that delay between frames is reduced. If you want to see the actual frame rate of video then it can be view by using CAP_PROP_FPS property of cap object. It can be displayed by following code:

```
double frames_per_second = cap.get(CAP_PROP_FPS);
cout << "Frames per seconds of the video is : " << frames_per_second ;
```

cap object also has other properties like CAP_PROP_FRAME_WIDTH

and CAP_PROP_FRAME_HEIGHT which indicates the width and height of the frames. It can also be fetched by get method. These properties can be set by using a set method of cap object. The set method has two arguments. The first argument is the name of the property and the second argument is the value we want to set.

This section described the method to read a video from a file. The next section will show the process to work with videos from either webcam or USB camera.

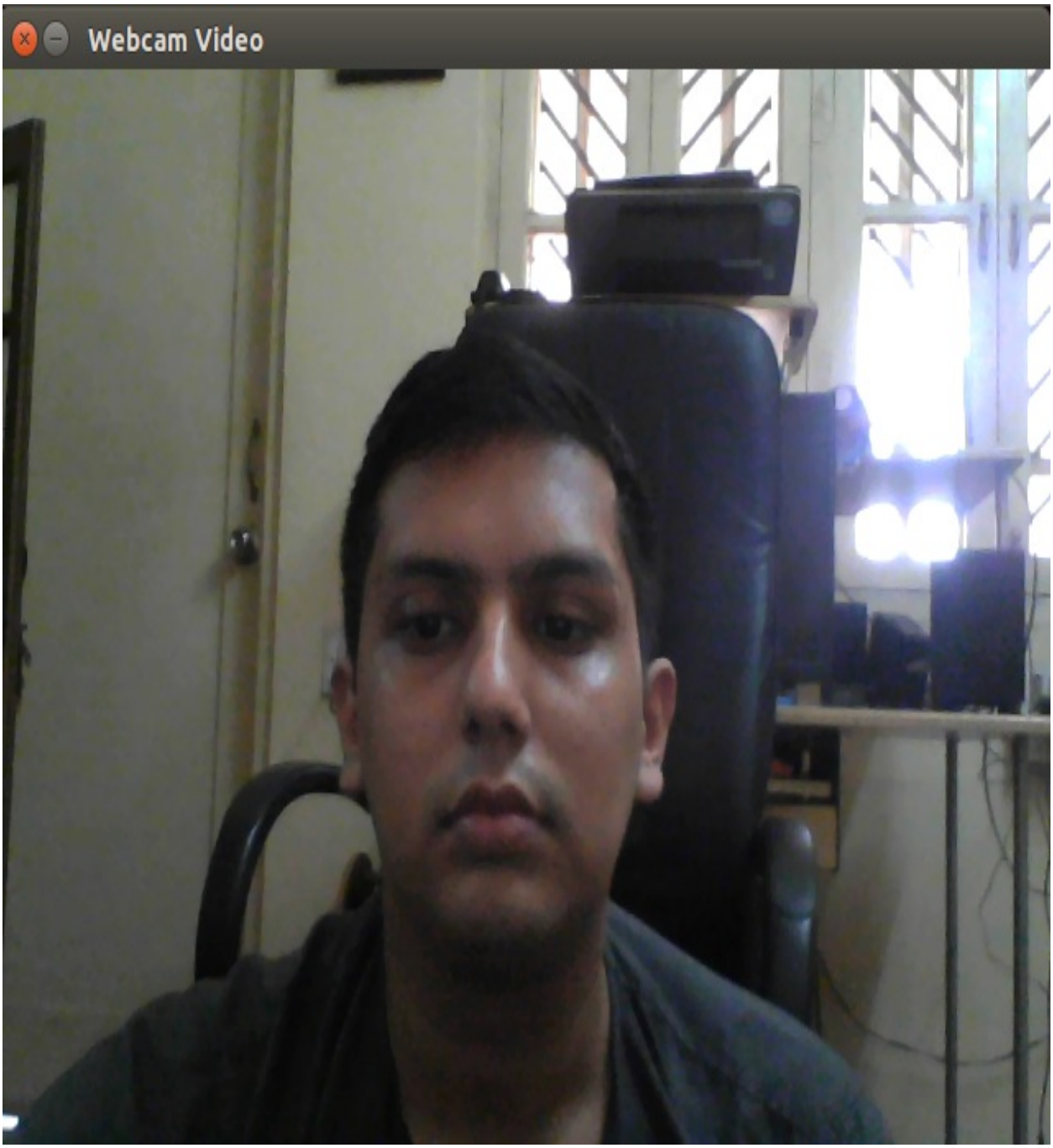
Working with videos from webcam

This section describes the process to capture video from a webcam or USB cameras attached to a computer. The good part of OpenCV is that this same code will work for laptop and any embedded system that can run C/C++. This helps in deploying computer vision applications on any hardware platforms. The code for capturing video and displaying it is shown below:

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int argc, char* argv[])
{
 //open the Webcam
 VideoCapture cap(0);
 // if not success, exit program
 if (cap.isOpened() == false)
 {
  cout << "Cannot open Webcam" << endl;
  return -1;
 }
 //get the frames rate of the video from webcam
 double frames_per_second = cap.get(CAP_PROP_FPS);
 cout << "Frames per seconds : " << frames_per_second << endl;
 cout<<"Press Q to Quit" <<endl;
 String win_name = "Webcam Video";
 namedWindow(win_name); //create a window
 while (true)
 {
  Mat frame;
  bool flag = cap.read(frame); // read a new frame from video
  //show the frame in the created window
  imshow(win_name, frame);
  if (waitKey(1) == 'q')
  {
      break;
  }
 }
return 0;
}
```

While capturing video from webcam or USB camera, the device id for that camera needs to be provided as an argument to the constructor of VideoCapture object. The primary camera connected will have a device id zero. The webcam of laptop or USB camera (when there is no webcam) will have device id zero. If there are multiple cameras connected to a device then there device id will be 0,1 and so on. In the code above zero is provided that indicates that primary camera will be used by code to capture video.

The other code is more or less similar to the code of reading video from a file. Here, the frame rate of the video is also fetched and displayed. The frames will be read one by one at a 1 ms interval and displayed on the window created. You have to press 'q' to terminate the operation. The output of video captured using webcam is shown below:


Webcam Video

Saving video to a disk

To save video from OpenCV program, we need to create an object of VideoWriter class. The code to save a video to a file is shown below:

```
Size frame_size(640, 640);
int frames_per_second = 30;

VideoWriter v_writer("images/video.avi", VideoWriter::fourcc('M', 'J', 'P', 'G'), frames_per_second,
frame_size, true);

//Inside while loop
v_writer.write(frame);

//After finishing video write
v_writer.release();
```

While creating an object of VideoWriter class, the constructor takes five arguments. First argument is the name of the video file you want to give along with the absolute or relative path. Second argument is the four character code used for video codec. It is created using VideoWriter::fourcc function. Here we are using motion JPEG codec so four character code for it is 'M','J','P' and 'G'. There are other codec that can be used depending on the requirement and operating systems. The third argument is frames per second. It can be specified as integer variable previously defined or integer value directly in the function. In the code above, 30 frames per second is used. The fourth argument is the size of the frame. It is defined using Size keyword with two arguments which is frame_width and frame_height. It is taken as 640x640 in the code above. The fifth argument specifies whether the frame to be stored is color or grayscale. If its true then frame savea d as color frame.

To start writing frames using the VideoWriter object , it provides a write method. This method is used to write frames in to video one by one so it is included inside an infinite while loop. This method takes only one argument which is the name of the frame variable. The size of the frame should be same as size specified while creating VideoWriter object. It is important to flush and close the video file created after writing is finished. This can be done by releasing the created VideoWriter object using release method.

To summarize, in this section we have seen the process of reading video from a file or camera attached to the device. We have also seen the code for writing the video to a file. From next section onwards, we will see how we can operate on images or videos using OpenCV with CUDA acceleration.

Basic Computer Vision Applications using OpenCV CUDA module

We have seen in the earlier chapters that CUDA provides an excellent interface to utilize the parallel computing capability of GPU to accelerate complex computing applications. In this section, we will see how we can utilize the capability of CUDA alongside OpenCV for Computer Vision applications

Introduction to OpenCV CUDA module

OpenCV has a CUDA module which has hundreds of functions which can utilize GPU capabilities. It is only supported on Nvidia GPUs because it uses Nvidia CUDA runtime in the background. OpenCV has to be compiled with WITH_CUDA flag set to ON for using CUDA module.

The great feature of using CUDA module of OpenCV is that it provides a similar API to regular OpenCV API. It also does not require detail knowledge of programming in CUDA although knowledge of CUDA and GPU architecture will not do any harm. The research have shown that using functions with CUDA acceleration can provide 5x-100x speedup over similar CPU functions.

In the next section, we will see how to use CUDA module along with OpenCV in various computer vision and image processing applications which operates on individual pixels of an image.

Arithmetic and Logical Operations on images

In this section, we will see how to perform various arithmetic and logical operations on Images. We will use functions defined in CUDA module of OpenCV to perform this operations.

Addition of Two images

The addition of two images can be performed when two images are of the same size. OpenCV provides a add function inside cv::cuda namespace for addition operation. It performs pixel-wise addition of two images. Suppose in two images, the pixel at (0,0) has intensity values 100 and 150 respectively then intensity value in the resultant image will be 250 which is the addition of two intensity values. OpenCV addition is a saturated operation which means that if an answer of addition goes above 255 then it will be saturated at 255. The code to perform addition is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main (int argc, char* argv[])
{
    //Read Two Images
    cv::Mat h_img1 = cv::imread("images/cameraman.tif");
    cv::Mat h_img2 = cv::imread("images/circles.png");
    //Create Memory for storing Images on device
    cv::cuda::GpuMat d_result1,d_img1, d_img2;
    cv::Mat h_result1;
    //Upload Images to device
    d_img1.upload(h_img1);
    d_img2.upload(h_img2);

    cv::cuda::add(d_img1,d_img2, d_result1);
    //Download Result back to host
    d_result1.download(h_result1);
    cv::imshow("Image1 ", h_img1);
    cv::imshow("Image2 ", h_img2);
    cv::imshow("Result addition ", h_result1);
    cv::imwrite("images/result_add.png", h_result1);
    cv::waitKey();
    return 0;
}
```

When any Computer vision operations need to be performed on GPU, the images have to be stored on device memory. The memory for it can be allocated by gpumat keyword which is similar to mat type used for host memory. The images are read in the same way as earlier. Two images are read for addition and stored in host memory. These images are copied to device memory using upload method of device memory variable. The host image variable is passed as a parameter to this method.

The function in GPU CUDA module is defined in cv::cuda namespace. It requires images on device memory as its arguments. The add function from CUDA module is used for image addition. It requires three arguments. The first two arguments are two images that are to be added and the last argument is the destination in which result will be stored. All three variable used should be defined using gpumat.

The resultant image is copied back to host using download method of device variable. The host image variable in which result will be copied is provided as an argument to download method. Then this image is displayed and stored on the disk using the same functions explained in the last section. The output of the program is shown below:

Image 1

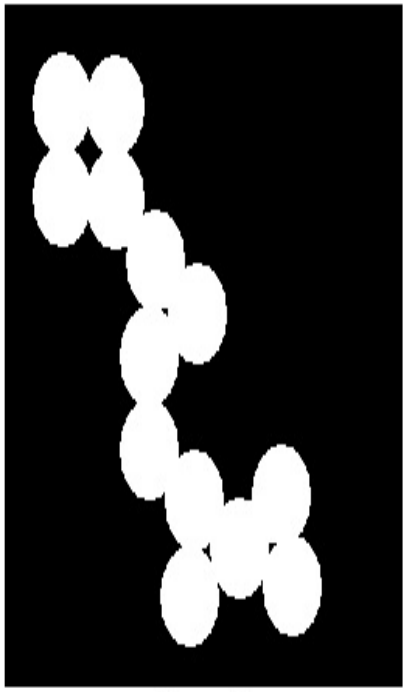


Image 2

Result after addition

Subtracting Two images

Other arithmetic operations can also be performed on images using OpenCV and CUDA. subtract function is provided by OpenCV to subtract two images. It is also a saturated operation which means that when the answer of subtraction goes below zero then it will be saturated to zero. The syntax of subtract command is shown below:

```
//d_result1 = d_img1 - d_img2
cv::cuda::subtract(d_img1, d_img2,d_result1);
```

Again two images to be subtracted are provided as first two arguments and the resultant image is provided as the third argument. The result of subtraction between two images is shown below:

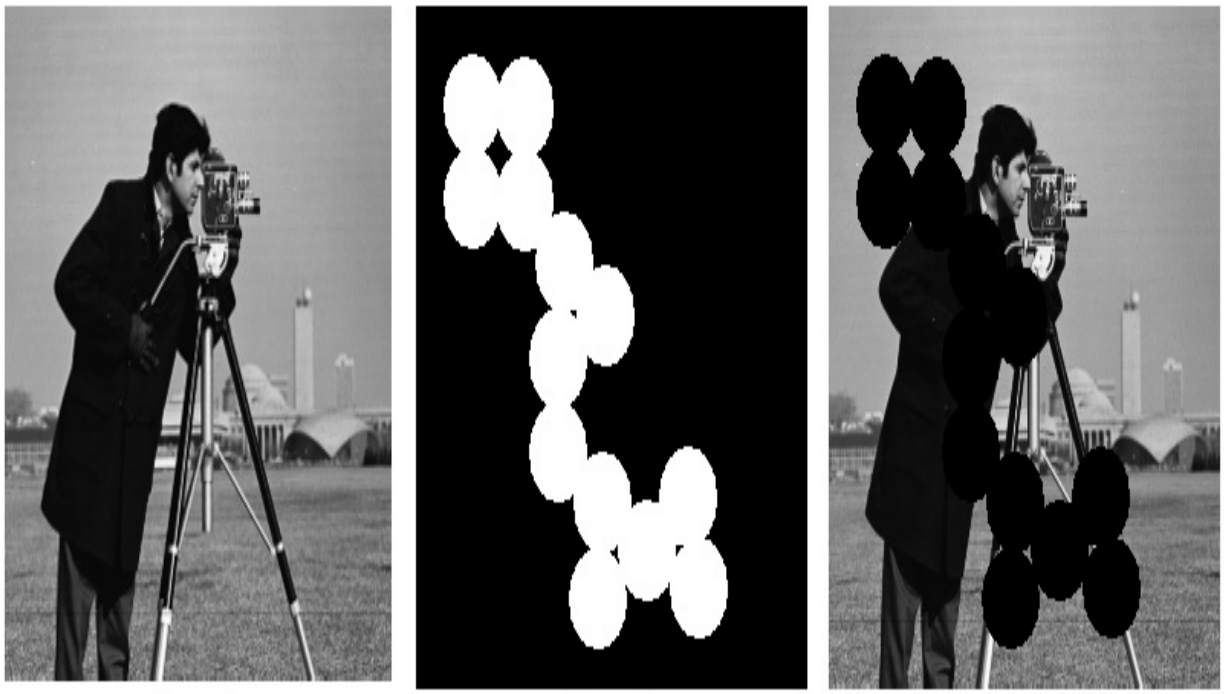


Image 1 Image 2 Result after Subtraction

Image Blending

Sometimes there is a need to blend two images with different proportion instead of directly adding two images. An image blending can be represented mathematically by follwing equation.

```
result = α * img1 + β * img2 + γ
```

This can be easily accomplished by addWeighted function inside OpenCV. The syntax of the function is shown below:

```
cv::cuda::addWeighted(d_img1,0.7,d_img2,0.3,0,d_result1)
```

The function has six arguments. The first argument is first source image, the second argument is a weight of the first image for blending, the third argument is second source image, the fourth argument is a weight of the second image for blending and the fifth argument is the constant gamma to be added while blending. The final argument specifies the destination in which result needs to be stored. The function above takes 70% of img1 and 30% of img2 for blending. The output for this function is shown below:

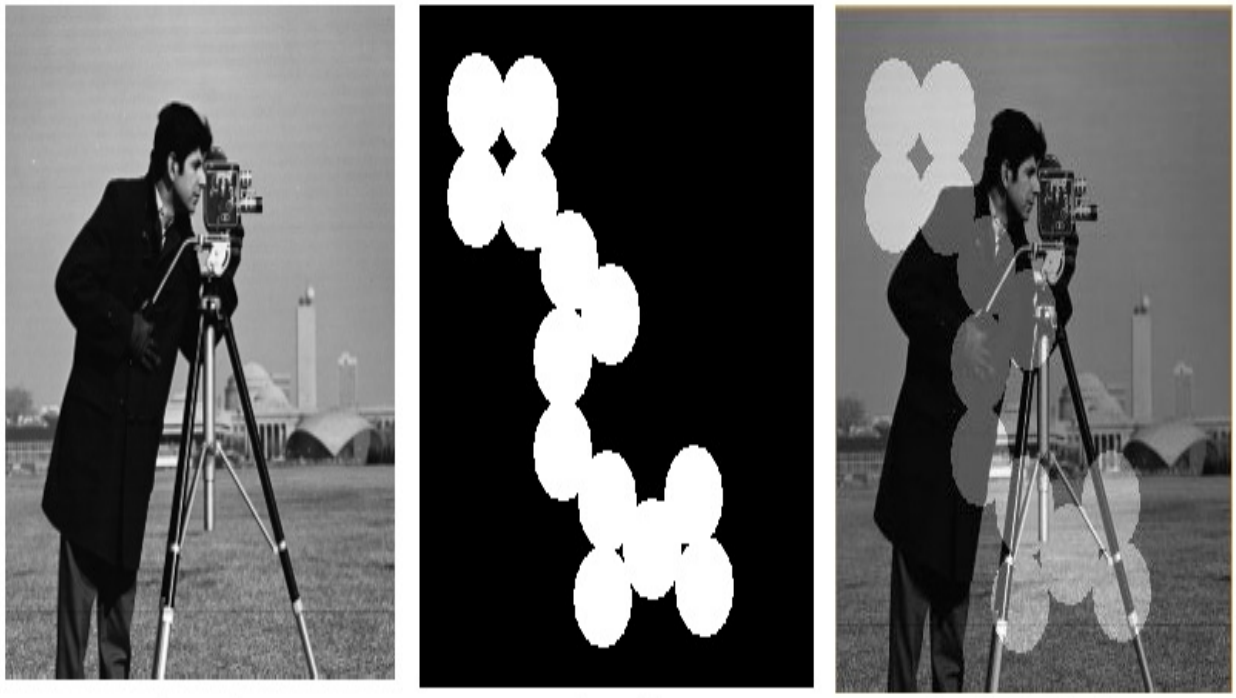


Image 1 Image 2 Result after Blending

Image Inversion

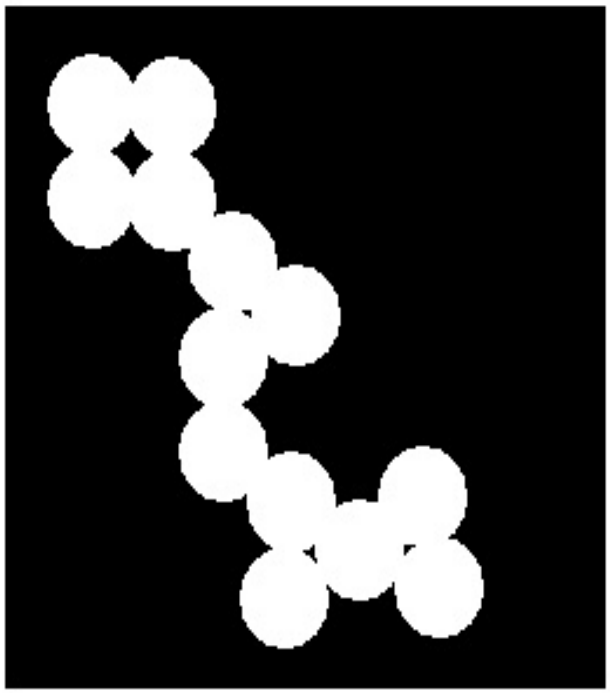
Apart from arithmatic operations, OpenCV also provides boolean operations which works on indiviual bits. It includes AND, OR, NOT etc. AND and OR are very useful for masking operations which we will see later on. NOT operation is used for inverting an image where black is converted to white and white is converted to black. It can be represented by the following equation:

```
result_image = 255 - input_image
```

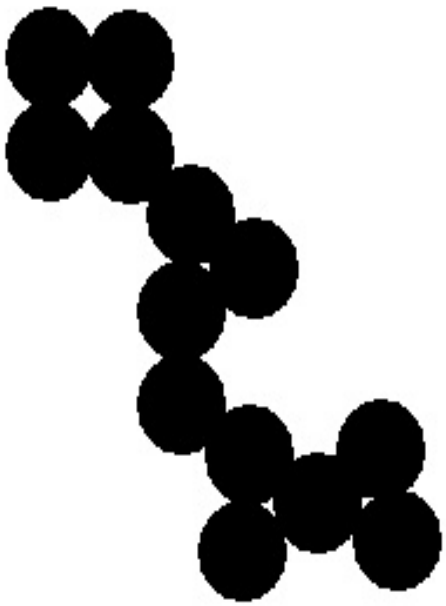
In the equation, 255 indicate maximum intensity value for an 8-bit image. The program for doing image inversion is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main (int argc, char* argv[])
{
    cv::Mat h_img1 = cv::imread("images/circles.png");
    //Create Device variables
    cv::cuda::GpuMat d_result1,d_img1;
    cv::Mat h_result1;
    //Upload Image to device
    d_img1.upload(h_img1);

    cv::cuda::bitwise_not(d_img1,d_result1);

    //Download result back  to host
    d_result1.download(h_result1);
    cv::imshow("Result inversion ", h_result1);
    cv::imwrite("images/result_inversion.png", h_result1);
    cv::waitKey();
    return 0;
}
```

The program is similar to the program for Arithmetic operation. bitwise_not function is used for image inversion. The image should be a grayscale image. It takes two arguments. The first argument indicates the source image to be inverted and the second argument indicates the destination in which inverted image is to be stored. The output of the bitwise_not operation is shown below:



Source Image



Inverted Image

As can be seen that by doing an inversion, white color is converted to black and black color is converted to white.

To summarize, in this section we have seen various arithmetic and logical operations using OpenCV and CUDA. In the next section, we will see some more computer vision operations that are widely used in Computer Vision applications.

Changing Color-Space of an Image

As described earlier, OpenCV can read an image as a grayscale image or as a color image with three channels green, blue and red which is called BGR format. The other image processing software and algorithms worked on RGB images where Red channel is first followed by green and blue. There are many other color formats which can be used for certain applications. These include HSV color space where three channels are Hue, Saturation, and Value. Hue represents color value, saturation indicates the gray level in color and value represents the brightness of the color. The other color space is YCrCb which is also very useful. The system represents colors in an image in terms of one luminance component/luma (Y), and two chrominance components/chroma(Cb and Cr).

There are many other color spaces available which are supported by OpenCV like XYZ, HLS, Lab etc. OpenCV supports more than 150 color conversion methods. The conversion from one color space to the other can be accomplished by using cvtColor function available in OpenCV. The example of using this function for changing between various color space is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main (int argc, char* argv[])
{
        cv::Mat h_img1 = cv::imread("images/autumn.tif");
        //Define device variables
        cv::cuda::GpuMat d_result1,d_result2,d_result3,d_result4,d_img1;
        //Upload Image to device
        d_img1.upload(h_img1);

        //Convert image to different color spaces
        cv::cuda::cvtColor(d_img1, d_result1,cv::COLOR_BGR2GRAY);
        cv::cuda::cvtColor(d_img1, d_result2,cv::COLOR_BGR2RGB);
        cv::cuda::cvtColor(d_img1, d_result3,cv::COLOR_BGR2HSV);
        cv::cuda::cvtColor(d_img1, d_result4,cv::COLOR_BGR2YCrCb);

        cv::Mat h_result1,h_result2,h_result3,h_result4;
        //Download results back to host
        d_result1.download(h_result1);
        d_result2.download(h_result2);
        d_result3.download(h_result3);
        d_result4.download(h_result4);

        cv::imshow("Result in Gray ", h_result1);
        cv::imshow("Result in RGB", h_result2);
        cv::imshow("Result in HSV ", h_result3);
        cv::imshow("Result in YCrCb ", h_result4);

        cv::waitKey();
        return 0;
}
```

The imshow function expects color images in BGR color format so the output of other color formats using imshow might not be visually attractive. The output of the above program with the same image in different color format is shown below:

Original Image

Grayscale Image

RGB Image

HSV Image

YCrCb Image

Image Thresholding

Image thresholding is a very simple Image segmentation technique used to extract important regions from a grayscale image based on certain intensity values. In this technique if the pixel value is greater than a certain threshold value then it is assigned one value else it is assigned another value.

The function used for Image thresholding in OpenCV and CUDA is cv::cuda::threshold. This function has many arguments. First argument is the source image which should be a grayscale image. The second argument is the destination in which result is to be stored. The third argument is the threshold value which is used to segment the pixel values. The fourth argument is the maxVal constant which represents the value to be given if the pixel value is more than the threshold value. OpenCV provides different types of thresholding techniques and it is decided by the last argument of the function. These thresholding types are:

- cv:.THRESH_BINARY:

  If the intensity of the pixel is greater then threshold then set that pixel intensity equal to maxVal constant else set that pixel intensity to zero.

- cv::THRESH_BINARY_INV

  If the intensity of the pixel is greater then threshold then set that pixel intensity equal to zero else set that pixel intensity to maxVal constant.

- cv::THRESH_TRUNC

  It is basically a truncation operation. If the intensity of the pixel is greater then threshold then set that pixel intensity equal to the threshold else keep intensity value as it is.

- cv::THRESH_TOZERO

  If the intensity of the pixel is greater then threshold then keep pixel intensity as it is else set that pixel intensity to zero.

- cv::THRESH_TOZERO_INV

  If the intensity of the pixel is greater then threshold then set that pixel intensity equal to zero else keep pixel intensity as it is.

The Program to implement all this thresholding techniques using OpenCV and CUDA is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main (int argc, char* argv[])
{
        cv::Mat h_img1 = cv::imread("images/cameraman.tif", 0);
```

```
        //Define device variables
        cv::cuda::GpuMat d_result1,d_result2,d_result3,d_result4,d_result5, d_img1;
        //Upload image on device
        d_img1.upload(h_img1);

        //Perform different thresholding techniques on device
        cv::cuda::threshold(d_img1, d_result1, 128.0, 255.0, cv::THRESH_BINARY);
        cv::cuda::threshold(d_img1, d_result2, 128.0, 255.0, cv::THRESH_BINARY_INV);
        cv::cuda::threshold(d_img1, d_result3, 128.0, 255.0, cv::THRESH_TRUNC);
        cv::cuda::threshold(d_img1, d_result4, 128.0, 255.0, cv::THRESH_TOZERO);
        cv::cuda::threshold(d_img1, d_result5, 128.0, 255.0, cv::THRESH_TOZERO_INV);

        cv::Mat h_result1,h_result2,h_result3,h_result4,h_result5;
        //Copy results back to host
        d_result1.download(h_result1);
        d_result2.download(h_result2);
        d_result3.download(h_result3);
        d_result4.download(h_result4);
        d_result5.download(h_result5);
        cv::imshow("Result Threshhold binary ", h_result1);
        cv::imshow("Result Threshhold binary inverse ", h_result2);
        cv::imshow("Result Threshhold truncated ", h_result3);
        cv::imshow("Result Threshhold truncated to zero ", h_result4);
        cv::imshow("Result Threshhold truncated to zero inverse ", h_result5);
        cv::waitKey();

    return 0;
}
```

In the cv::cuda::threshold function for all thresholding techniques 128 is taken as a threshold for pixel intensity which is a midpoint between black (0) and white (255). The maxVal constant is taken as 255 which will be used to update pixel intensity when it exceeds the threshold. The other program is similar to other OpenCV programs seen earlier. The output of the program is shown below which displays input image along with the output of all five thresholding techniques:

Original Image

Result of binary Threshold

Result of inverse binary Threshold

Result of Thresholding by truncating

Result of Thresholding by truncating to zero

Result of Thresholding by truncating to zero inverse

Performance comparison of OpenCV applications with and without CUDA support

The performance of image processing algorithms can be measured in terms of time it takes to process a single image. When algorithms work on video then performance is measured in terms of frames per second which indicates the number of frames it can process in a second. When algorithm can process more than 30 frames per second then it can be considered to work in real time. We can also measure the performance of our algorithms implemented in OpenCV which will be discussed in this section.

As we have discussed earlier, When OpenCV is built with CUDA compatibility it can increase the performance of algorithms drastically. OpenCV functions in CUDA module are optimized to utilize GPU parallel processing capability. OpenCV also provides similar functions that only run on CPU. In this section, we will compare the performance of thresholding operations built in the last section with and without using GPU. We will compare the performance of the thresholding operation in terms of time taken to process one image and frames per second. The code to implement thresholding on CPU and measure performance is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
using namespace cv;
using namespace std;

int main (int argc, char* argv[])
{
cv::Mat src = cv::imread("images/cameraman.tif", 0);
cv::Mat result_host1,result_host2,result_host3,result_host4,result_host5;

//Get initial time in miliseconds
int64 work_begin = getTickCount();
cv::threshold(src, result_host1, 128.0, 255.0, cv::THRESH_BINARY);
cv::threshold(src, result_host2, 128.0, 255.0, cv::THRESH_BINARY_INV);
cv::threshold(src, result_host3, 128.0, 255.0, cv::THRESH_TRUNC);
cv::threshold(src, result_host4, 128.0, 255.0, cv::THRESH_TOZERO);
cv::threshold(src, result_host5, 128.0, 255.0, cv::THRESH_TOZERO_INV);

//Get time after work has finished
int64 delta = getTickCount() - work_begin;
//Frequency of timer
double freq = getTickFrequency();
double work_fps = freq / delta;
std::cout<<"Performance of Thresholding on CPU: " <<std::endl;
std::cout <<"Time: " << (1/work_fps) <<std::endl;
std::cout <<"FPS: " <<work_fps <<std::endl;
return 0;
}
```

In the code above, threshold function from cv namespace is used which only uses CPU for execution instead of cv::cuda module. The performance of the algorithm is measured using gettickcount and gettickfrequency functions. The gettickcount function returns the time in milliseconds that have passed after starting of the system. We measured the time ticks before and after the execution of code which operates on Image. The difference between this time ticks indicates the ticks passed during an execution of the algorithm to process an Image. This time is measured in delta variable. The gettickfrequncy function returns the frequency of the timer. Total time taken to process an image can be measured by dividing time ticks by the frequency of

the timer. The inverse of this time indicates frames per second (FPS). Both this performance measures are printed on the console for thresholding application on CPU. The output on the console is shown below:

```
Performance of Thresholding on CPU:
Time: 0.169766
FPS: 5.89046
```

As can be seen from the output, the CPU takes 0.169766 seconds to process one image which is equal to 5.89046 FPS. Now we will implement the same algorithm on GPU and try to measure the performance of the code. As per discussion earlier, this should increase the performance of algorithm drastically. The code for GPU implementation is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main (int argc, char* argv[])
{
 cv::Mat h_img1 = cv::imread("images/cameraman.tif", 0);
cv::cuda::GpuMat d_result1,d_result2,d_result3,d_result4,d_result5, d_img1;
//Measure initial time ticks
int64 work_begin = getTickCount();
d_img1.upload(h_img1);
cv::cuda::threshold(d_img1, d_result1, 128.0, 255.0, cv::THRESH_BINARY);
cv::cuda::threshold(d_img1, d_result2, 128.0, 255.0, cv::THRESH_BINARY_INV);
cv::cuda::threshold(d_img1, d_result3, 128.0, 255.0, cv::THRESH_TRUNC);
cv::cuda::threshold(d_img1, d_result4, 128.0, 255.0, cv::THRESH_TOZERO);
cv::cuda::threshold(d_img1, d_result5, 128.0, 255.0, cv::THRESH_TOZERO_INV);

cv::Mat h_result1,h_result2,h_result3,h_result4,h_result5;
d_result1.download(h_result1);
d_result2.download(h_result2);
d_result3.download(h_result3);
d_result4.download(h_result4);
d_result5.download(h_result5);
//Measure difference in time ticks
int64 delta = getTickCount() - work_begin;
double freq = getTickFrequency();
//Measure frames per second
double work_fps = freq / delta;
std::cout <<"Performance of Thresholding on GPU: " <<std::endl;
std::cout <<"Time: " << (1/work_fps) <<std::endl;
std::cout <<"FPS: " <<work_fps <<std::endl;
 return 0;
}
```

In the code, the functions are used from cv::cuda module which are optimized for GPU parallel processing capabilities. The images are copied to device memory, operated upon on GPU and copied back to host. The performance measures are calculated in a similar way as above and printed on the console. The output of the program is shown below:

```
Performance of Thresholding on GPU:
Time: 0.000550593
FPS: 1816.22
```

As can be seen that GPU implementation only takes 0.55 ms to process a single image which is equal to 1816 FPS. This is a drastic improvement over a CPU implementation. Though it must be kept in mind that this is a very simple application and not ideal for performance comparison

between CPU and GPU. This application was just shown to make you familiar with how one can measure the performance of any code in OpenCV.

More realistic comparison of CPU and GPU performance can be made by running the example codes provided in OpenCV installation in samples/gpu directory. One of the code hog.cpp calculates the histogram of oriented(HoG) features from an image and classifies it using Support Vector Machine (SVM). Though details of algorithms are out of the scope of this book, it gives you an idea about performance improvement while using GPU implementations. The performance comparison on Webcam video is shown below:

CPU Performance

GPU Performance

As can be seen that while we use only CPU, the performance of the code is around 13 FPS and if we use GPU it increases to 24 FPS which is almost double of CPU performance. This will give you an idea about the importance of using CUDA with OpenCV.

To summarize, in this section we have seen the comparison between the performance of OpenCV with using CUDA (GPU) and without using CUDA (CPU). It reemphasizes the notion that use of CUDA will improve the performance of the computer vision applications drastically.

Summary

In this chapter, we have started with the introduction of computer vision and image processing. We have described OpenCV library which is specifically made for computer vision applications and how it is different than other computer vision software. OpenCV can leverage the parallel processing capability of GPU by using CUDA. We have seen the installation procedure for OpenCV with CUDA in all operating systems. We have described the process to read an image from disk, display it on screen and save it back to disk. The videos are nothing but a sequence of images. We have learned to work with videos from disk as well as videos captured from the camera. We have developed several image processing applications which do different operations on images like Arithmetic operations, Logical Operations, Color Space conversions and Thresholding. In the last section, we have compared the performance of the same algorithm on CPU and GPU in terms of time taken to process an image and FPS. So at the end of this chapter, you have an idea of the usefulness of OpenCV with CUDA in computer vision applications and how to write simple code using it. In next chapter, we will build upon this knowledge and try to develop some more useful computer vision applications like filtering, edge detection and morphological operation using OpenCV.

Questions

1. State the difference between terms Computer Vision and Image Processing.
2. Why is OpenCV ideal for deploying Computer Vision applications on Embedded Systems?
3. Write an OpenCV command to initialize 1960 x 1960 color image with red color.
4. Write a program to capture frames from a webcam and save it to disk.
5. Which color format is used by OpenCV to read and display a color image?
6. Write a program to capture video from webcam, convert it to grayscale and display on the screen.
7. Write a program to measure the performance of add and subtract operation on GPU.
8. Write a program for bitwise AND and OR operation on Images and explain how it can be used for masking.

Basic computer vision Operations using OpenCV and CUDA

Introduction

The last chapter described the process of working with Images and Videos using OpenCV and Cuda. We have seen the code for some basic Image and Video processing applications and compared the performance of OpenCV code with and without CUDA acceleration. In this chapter, we will build on this knowledge and try to develop some more computer vision and Image processing applications using OpenCV and CUDA. This chapter describes the method to access individual pixel intensities in color and gray-scale images. A histogram is a very useful concept for image processing. This chapter describes the method for calculating histogram and how histogram equalization can improve the visual quality of Images. This chapter will also describe how different geometric transformation can be performed using OpenCV and CUDA. Image filtering is a very important concept which is useful in Image preprocessing and feature extraction. This is described in detail in this chapter. The last part of the chapter describes different morphological operations like erosion, dilation, Opening and closing on Images.

The following topics will be covered in this chapter:

- Accessing individual pixel intensities in OpenCV
- Histogram calculation and Histogram Equalization
- Image Transformation
- Filtering Operations on Images
- Morphological Operations on Images

Technical Requirements

This chapter requires a basic understanding of image processing and computer vision. It needs familiarity with the basic C or C++ programming language, CUDA and all the codes explained in previous chapters. All the code used in this chapter can be downloaded from following github link: https://github.com/PacktPublishing/Hands-On-GPU-Accelerated-Computer-Vision-with-OpenCV-and-CUDA. The code can be executed on any operating system though it has only been tested on Ubuntu 16.04.

Accessing Individual Pixel Intensities of an Image

Sometimes there is a need to access pixel intensity value at a particular location when we are working with Images. This is very useful when we want to change the brightness or contrast of a group of pixels or we want to perform some other pixel level operations. For an 8-bit gray-scale image this intensity value at a point will be in a range of 0 t0 255 while for a color image there will be three different intensity values for the blue, green and red channel with all having the values between 0 to 255.

OpenCV provides a cv::Mat::at<> method for accessing intensity values at a particular location for any channel images. It needs one argument which is the location of Point at which the intensity is to be accessed. The point is passed using Point class with row and column values as arguments. For a gray-scale image, the method will return a scalar object while for a color image it will return a vector of three intensities. The code for accessing pixel intensities at a particular location for gray-scale as well as color image is shown below.

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  //Gray Scale Image
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::Scalar intensity = h_img1.at<uchar>(cv::Point(100, 50));
  std::cout<<"Pixel Intensity of gray scale Image at (100,50) is:"  <<intensity.val[0]<<std::endl;
  //Color Image
  cv::Mat h_img2 = cv::imread("images/autumn.tif",1);
  cv::Vec3b intensity1 = h_img1.at<cv::Vec3b>(cv::Point(100, 50));
  std::cout<<"Pixel Intensity of color Image at (100,50) is:"<<intensity1<<std::endl;
  return 0;
}
```

The gray-scale image is read first and 'at' method is called on this image object. The intensity value is measured at point (100,50) which indicates pixel at the 100^{th} row and 50^{th} column. It returns a scalar which is stored in intensity variable. The value is printed on the console. The same procedure is followed for a color image but the return value for it will be a vector of three intensities which is stored in Vec3b object. The intensity values are printed on the console. The output of the above program is shown below:

```
bhaumik@bhaumik-Lenovo-ideapad-520-15IKB:~/Desktop/opencv/Chapter 6$ g++ -std=c+
+11 individual_pixel.cpp `pkg-config --libs --cflags opencv` -o pixel
bhaumik@bhaumik-Lenovo-ideapad-520-15IKB:~/Desktop/opencv/Chapter 6$ ./pixel
Pixel Intensity of gray scale Image at (100,50) is:9
Pixel Intensity of color Image at (100,50) is:[175, 179, 177]
```

As can be seen that, pixel intensity for a gray-scale image at (100,50) is 9 while for a color image it is [175,179,177] which indicates blue intensity is 175, the green intensity is 179 and red

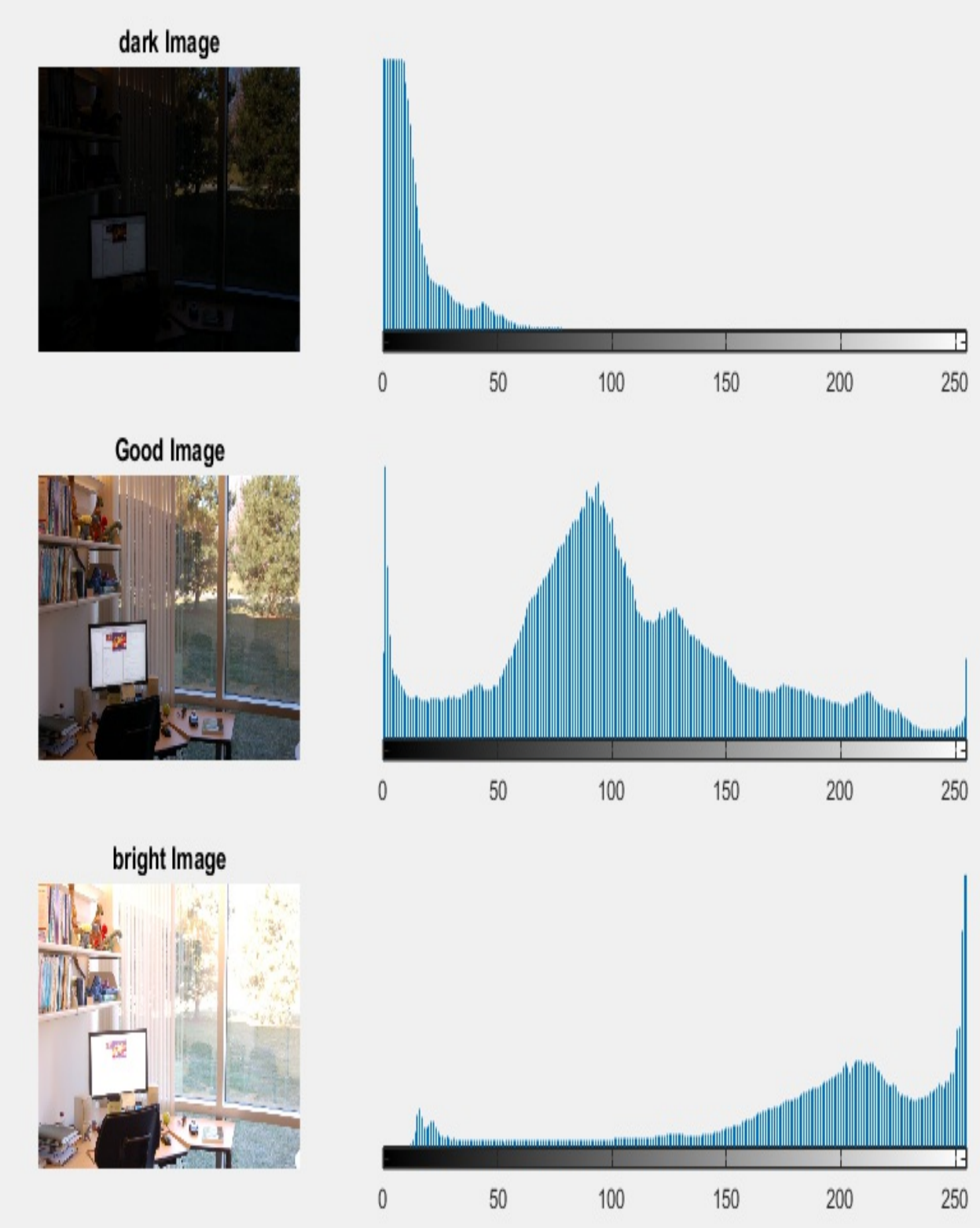
intensity is 177. The same method is used to modify pixel intensity at a particular location. Suppose, you want to change pixel intensity at (100,50) location to 128 then you can write:

```
h_img1.at<uchar>(100, 50) = 128;
```

To summarize, in this section we have seen a method to access and change intensity values at a particular section. In the next section, we will see the method to calculate the histogram in OpenCV.

Histogram calculation and Equalization in OpenCV

The histogram is a very important property of image as it provides a global description of the appearance of an Image. An enormous amount of information can be obtained from the histogram. It represents relative frequency of occurrence of the gray levels in an Image. It is basically a plot of gray levels on X-axis and number of pixels in each gray levels on Y-axis. If the histogram is concentrated on the left side then the image will be very dark and if it is concentrated on the right side then the image will be very bright. It should be evenly distributed for a good visual quality of an image. The following figure demonstrates histogram for dark ,bright and normal image.



OpenCV provides a function to calculate the histogram of an Image. The syntax of the function is shown below.

```
void cv::cuda::calcHist ( InputArray src, OutputArray hist)
```

The function needs two Array as an argument. The first array is the input image for which histogram needs to be calculated. The second argument is an output array in which histogram will be stored. The output can be plotted to get a histogram like what is shown in the figure above. As described earlier, a flat histogram improves the visual quality of an Image. OpenCV and CUDA provide a function to flatten the histogram which is described in the section below.

Histogram Equalization

A perfect image has an equal number of pixels in all its grey levels. So histogram should have a large dynamic range and an equal number of pixels in the entire range. This can be accomplished by the technique called as histogram equalization. It is a very important preprocessing step in any computer vision application. In this section, we will see how histogram equalization can be performed for grayscale and color image using OpenCV and CUDA.

Grayscale Images

Grayscale images are normally 8-bit single channel images which have 256 different gray levels. If the histogram is not evenly distributed, the image is too dark or the image is too light then histogram equalization should be performed to improve the visual quality of any Image. The following code describes the process for histogram equalization on the grayscale image.

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::cuda::GpuMat d_img1,d_result1;
  d_img1.upload(h_img1);
  cv::cuda::equalizeHist(d_img1, d_result1);
  cv::Mat h_result1;
  d_result1.download(h_result1);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Histogram Equalized Image", h_result1);
  cv::waitKey();
  return 0;
}
```

The image read is uploaded to device memory for histogram equalization. It is a mathematically intensive step so CUDA acceleration will help in improving performance the program. OpenCV provides equalizeHist function for histogram equalization. It needs two arguments. The first argument is the source image and the second argument is the destination image. The destination image is downloaded back to host and displayed on the console. The output after histogram equalization is shown below:

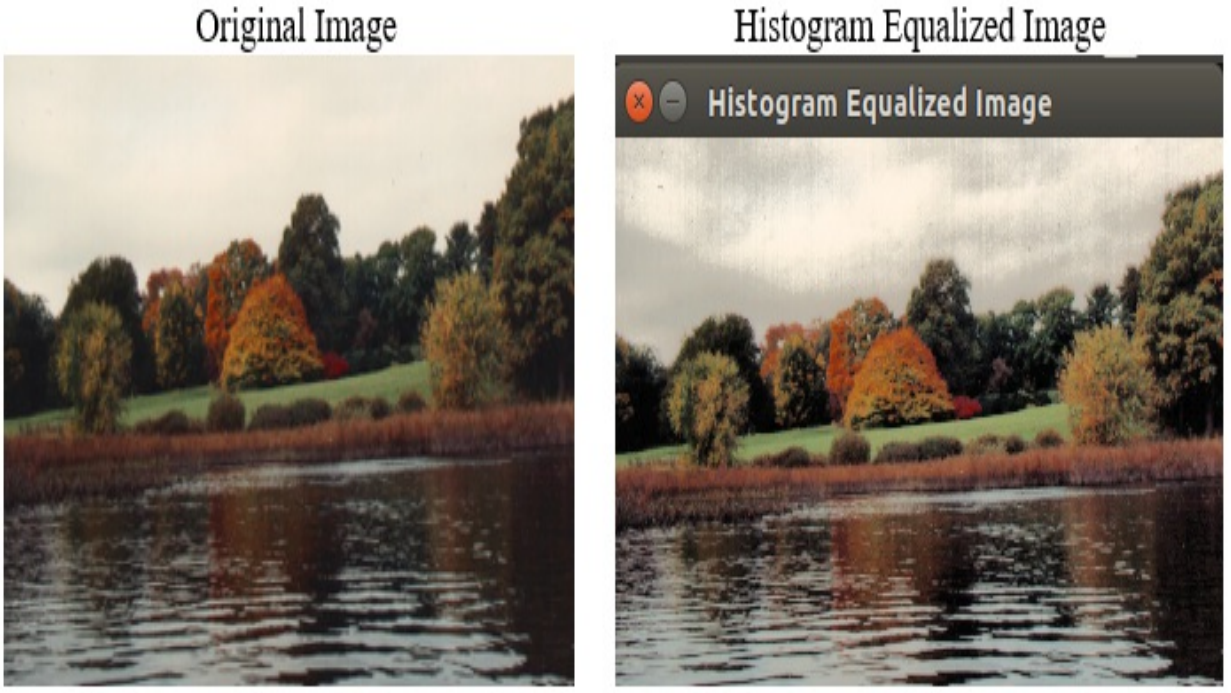
As can be seen, the image after histogram equalization has a better visual quality than the original image. The same operation on color images as described below.

Color Image

Histogram equalization can also be done on color images. It has to be performed on separate channels. So the color image has to be split into three channels. The histogram of each channel is equalized independently and then channels are merged to reconstruct the image. The code for histogram equalization on the color image is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/autumn.tif");
  cv::Mat h_img2,h_result1;
  cvtColor(h_img1, h_img2, cv::COLOR_BGR2HSV);
  //Split the image into 3 channels; H, S and V channels respectively and store it in a std::vector
  std::vector< cv::Mat > vec_channels;
  cv::split(h_img2, vec_channels);
  //Equalize the histogram of only the V channel
  cv::equalizeHist(vec_channels[2], vec_channels[2]);
  //Merge 3 channels in the vector to form the color image in HSV color space.
  cv::merge(vec_channels, h_img2);
  //Convert the histogram equalized image from HSV to BGR color space again
  cv::cvtColor(h_img2,h_result1, cv::COLOR_HSV2BGR);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Histogram Equalized Image", h_result1);
  cv::waitKey();
  return 0;
}
```

The histogram is not normally equalized in BGR color space. HSV and YCrCB color spaces are used for it. So in the code, BGR color space is converted to HSV color space. Then it is split into three separate channels using split function. Now, hue and saturation channels contain the color information so there is no point in equalizing those channels. Histogram equalization is performed only on the value channel. Three channels are merged back to reconstruct the color image using merge function. The HSV color image is converted back to BGR color space for displaying using imshow. The output of the program is shown below:



To summarize, histogram equalization improves the visual quality of an image so it is a very important preprocessing step for any computer vision application. The next section describes the geometric transformation of images.

Geometric Transformation on Images

Sometimes there is a need for resizing an image, translation of images and rotation of images for larger computer vision applications. This kind of geometric transformation is explained in this section.

Image Resizing

Images need to be of specific sizes in some computer vision applications. So there is a need to convert the image of arbitrary size into the specific size. OpenCV provides a function to resize an image. The code for image resizing is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::cuda::GpuMat d_img1,d_result1,d_result2;
  d_img1.upload(h_img1);
  int width= d_img1.cols;
  int height = d_img1.size().height;
  cv::cuda::resize(d_img1,d_result1,cv::Size(200, 200),   cv::INTER_CUBIC);
  cv::cuda::resize(d_img1,d_result2,cv::Size(0.5*width, 0.5*height),   cv::INTER_LINEAR);
  cv::Mat h_result1,h_result2;
  d_result1.download(h_result1);
  d_result2.download(h_result2);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Resized Image", h_result1);
  cv::imshow("Resized Image 2", h_result2);
  cv::waitKey();
  return 0;
}
```

The height and width of an image can be obtained using two different functions as shown in the code. The rows and cols property of the Mat object describes the height and width of an image respectively. The Mat object also has size() method which has height and width property which is used to find the size of an image. The image is resized in two ways. In a first way, the image is resized to a specific size of (200,200) and in the second it is resized to half of its original dimensions. OpenCV provides resize function for this operation. It has four arguments. The first two arguments are source and destination images respectively. The third argument is the size of the destination image. It is defined using the Size object. When images are resized then pixel values have to be interpolated on the destination image from the source image. There are various interpolation methods like bilinear interpolation, bicubic interpolation, area interpolation available for interpolating pixel values. This interpolation method is provided as the fourth argument to the resize function. It can be `cv::INTER_LINEAR (bilinear)`, `cv::INTER_CUBIC (bicubic)` and `cv::INTER_AREA (Area)`. The output of image resizing code is shown below:

Original Image

Image Resize (200,200)

Image Resize downscaled by 2

Image Translation and Rotation

Image translation and rotation are the important geometric transformation that is needed in some computer vision applications. OpenCV provides an easy API to perform this transformation on Images. The code to perform translation and rotation is shown below:

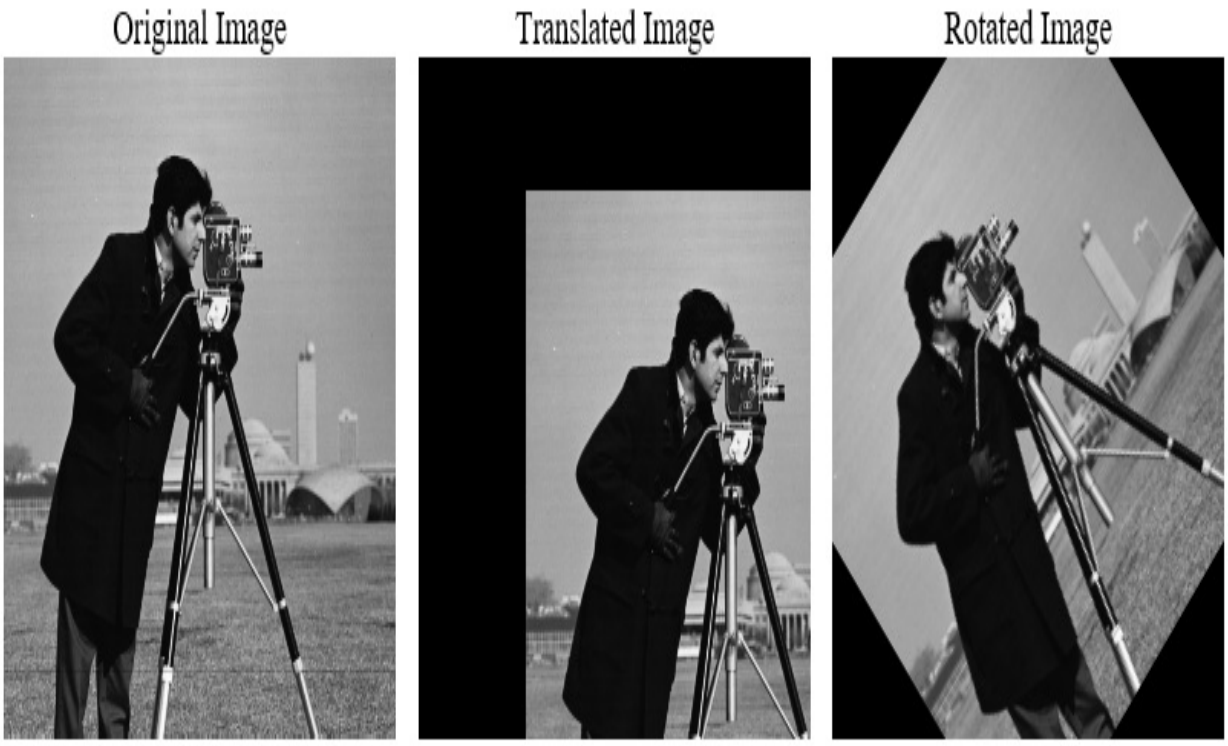
```
#include <iostream>
#include "opencv2/opencv.hpp"

int main ()
{
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::cuda::GpuMat d_img1,d_result1,d_result2;
  d_img1.upload(h_img1);
  int cols= d_img1.cols;
  int rows = d_img1.size().height;
  //Translation
  cv::Mat trans_mat = (cv::Mat_<double>(2,3) << 1, 0, 70, 0, 1, 50);
  cv::cuda::warpAffine(d_img1,d_result1,trans_mat,d_img1.size());
  //Rotation
  cv::Point2f pt(d_img1.cols/2., d_img1.rows/2.);
  cv::Mat rot_mat = cv::getRotationMatrix2D(pt, 45, 1.0);
  cv::cuda::warpAffine(d_img1, d_result2, rot_mat, cv::Size(d_img1.cols, d_img1.rows));
  cv::Mat h_result1,h_result2;
  d_result1.download(h_result1);
  d_result2.download(h_result2);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Translated Image", h_result1);
  cv::imshow("Rotated Image", h_result2);
  cv::waitKey();
  return 0;
}
```

Translation matrix needs to be created which specifies the image translation in horizontal and vertical directions. It is a 2x3 matrix like shown below:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \end{bmatrix}$$

`tx` and `ty` are translation offset in x and y-direction. In the code, this matrix is created using Mat object with 70 as an offset in the x-direction and 50 as an offset in the y-direction. This matrix is passed as an argument to `warpAffine` function for Image translation. The other arguments for the `warpAffine` function are source image, destination image and size of an output image respectively.

Rotation matrix should be created for Image rotation at a particular degree centered at a particular point. OpenCV provides `cv::getRotationMatrix2D` function to construct this rotation matrix. It needs three arguments. The first argument is the point for rotation. The center of the image is used for this. The second argument is the angle of rotation in degrees which is specified as 45 degrees. The last argument is the scale which is specified as 1. The constructed rotation matrix is again passed as an argument to `warpAffine` function for Image rotation. The output of the Image translation and Image rotation code is shown below:



To summarize, this section described various geometric transformation like Image resizing, Image translation and Image rotation using OpenCV and CUDA.

Filtering Operation on Images

The methods described till this point were working on a single pixel intensity which is called as point processing methods. Sometimes it is helpful to look at a neighborhood of pixel rather than only single pixel intensity. It is called as neighborhood processing techniques. The neighborhood can be 3x3, 5x5, 7x7 etc matrix centered at a particular pixel. Image filtering is an important neighborhood processing technique.

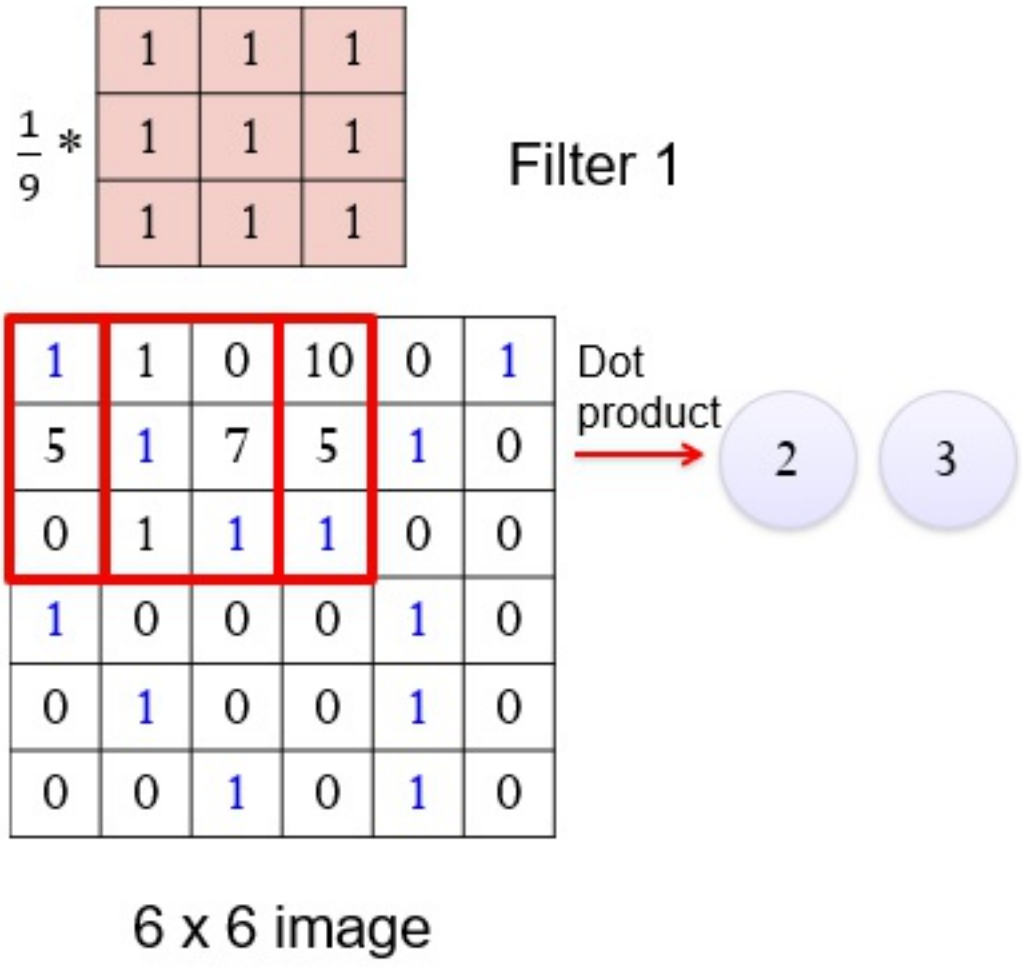
Filtering is an important concept in signal processing where we reject a certain band of frequencies and allow a certain band of frequency to pass. how frequency is measured in Images? If gray levels changes slowly over a region then it is a low-frequency region. If grey levels changes drastically then it is a high-frequency region. Normally background of an image is considered as low-frequency region and edges are high-frequency region. Convolution is a very important mathematical concept for neighborhood processing and Image filtering in particular. It is explained in the section below.

Convolution Operation on Image

The basic idea of convolutions evolved from the similar idea in biology called receptive field where it is sensitive to some part in an image and insensitive to other parts. It can be mathematically represented as:

$$g(x,y)=f(x,y)*h(x,y)= \sum\sum f(n,m)h(x-n,y-m)$$

In simplified form, this equation is a dot product between a filter *h* and a sub-image of image *f* centered around *(x,y)* point. The answer to this product is equal to *(x,y)* point in image *g*. To illustrate the working of convolution operation on an image, an example of 3 x 3 filter applied to an image of size 6 x 6 is shown in the figure below.



A dot product is taken between the leftmost window shown in red with the filter is taken to find a point in the destination image. The answer of the dot product will be 2 ((1*1 + 1*1 + 1*0 + 1*5 + 1*1 +1*7 +1*0 +1*1 + 1*1)/9). The same operation is repeated after moving this window by 1

pixel to the right and answer will be 3. This is repeated for all windows in an image to construct the destination image. Different low pass and high pass filters can be constructed by changing the values of 3x3 filter matrix. This is explained in the next two sections.

Low Pass filtering on Image

Low pass filter removes high-frequency content from an Image. Generally, Noise is a High-frequency content so low pass filter removes noise from an Image. There are many types of noise like gaussian noise, uniform noise, exponential noise, salt and pepper noise etc that can effect Image. Low pass filters are used to eliminate this kind of Noise. There are many types of Low pass filters available:

1. Averaging or Box Filter
2. Gaussian Filter
3. Median Filter

These filters and implementation of them using OpenCV is explained in this section.

Averaging Filter

An averaging filter as the name suggests performance averaging operation on neighborhood pixels. If a Gaussian noise is present in an Image then low pass averaging filter can be used to remove the noise. It will also blur the edges of an Image because of averaging operation. The neighborhood can be 3x3, 5x5, 7x7 and so on. The more the size of the filter window, more blurring of the image will take place. The 3x3 and 5x5 averaging mask is shown below:

Averaging Filter 3x3

$$\frac{1}{9} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Averaging Filter 5x5

$$\frac{1}{25} * \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

OpenCV provides a simple interface to apply many kinds of filter on Image. The code to apply an averaging filter with a different mask is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::cuda::GpuMat d_img1,d_result3x3,d_result5x5,d_result7x7;
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filter3x3,filter5x5,filter7x7;
  filter3x3 = cv::cuda::createBoxFilter(CV_8UC1,CV_8UC1,cv::Size(3,3));
  filter3x3->apply(d_img1, d_result3x3);
  filter5x5 = cv::cuda::createBoxFilter(CV_8UC1,CV_8UC1,cv::Size(5,5));
  filter5x5->apply(d_img1, d_result5x5);
  filter7x7 = cv::cuda::createBoxFilter(CV_8UC1,CV_8UC1,cv::Size(7,7));
  filter7x7->apply(d_img1, d_result7x7);

  cv::Mat h_result3x3,h_result5x5,h_result7x7;
  d_result3x3.download(h_result3x3);
  d_result5x5.download(h_result5x5);
  d_result7x7.download(h_result7x7);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Blurred with kernel size 3x3", h_result3x3);
  cv::imshow("Blurred with kernel size 5x5", h_result5x5);
  cv::imshow("Blurred with kernel size 7x7", h_result7x7);
  cv::waitKey();
  return 0;
}
```

cv:: Ptr which is a templated class for smart pointers is used to store a filter of type cv::cuda::Filter. Then createBoxFilter function is used to create an averaging filter of different window sizes. It requires three mandatory and three optional arguments. The first and second arguments are datatypes for source and destination Images. They are taken as CV_8UC1 which indicates 8-bit unsigned greyscale image. The third argument defines the size of the filter window. It can be 3x3,5x5,7x7 and so on. The fourth argument is the anchor point which has a

default value of (-1,-1) which indicate anchor is at the center point of the kernel. The final two optional arguments are related to pixel interpolation method and border value which are omitted here.

The created filter pointer has an apply method which is used to apply the created filter on any image. It has three arguments. The first argument is the source image, the second argument is the destination image and the third optional argument is Cuda stream which is used for multitasking as explained earlier in the book. In the code, three averaging filters of different sizes are applied on an Image. The result is shown below:

As can be seen from the output, as the size of the filter increases more pixels are used for averaging which introduces more blurring on Image. Though large filter will eliminate more noise.

Gaussian Filter

The Gaussian filter uses a mask which has a Gaussian distribution, to filter an Image instead of a simple averaging mask. This filter also introduces smooth blurring on an Image and widely used to eliminate noise from an Image. The 5x5 Gaussian filter with an approximate standard deviation of 1 is shown below:

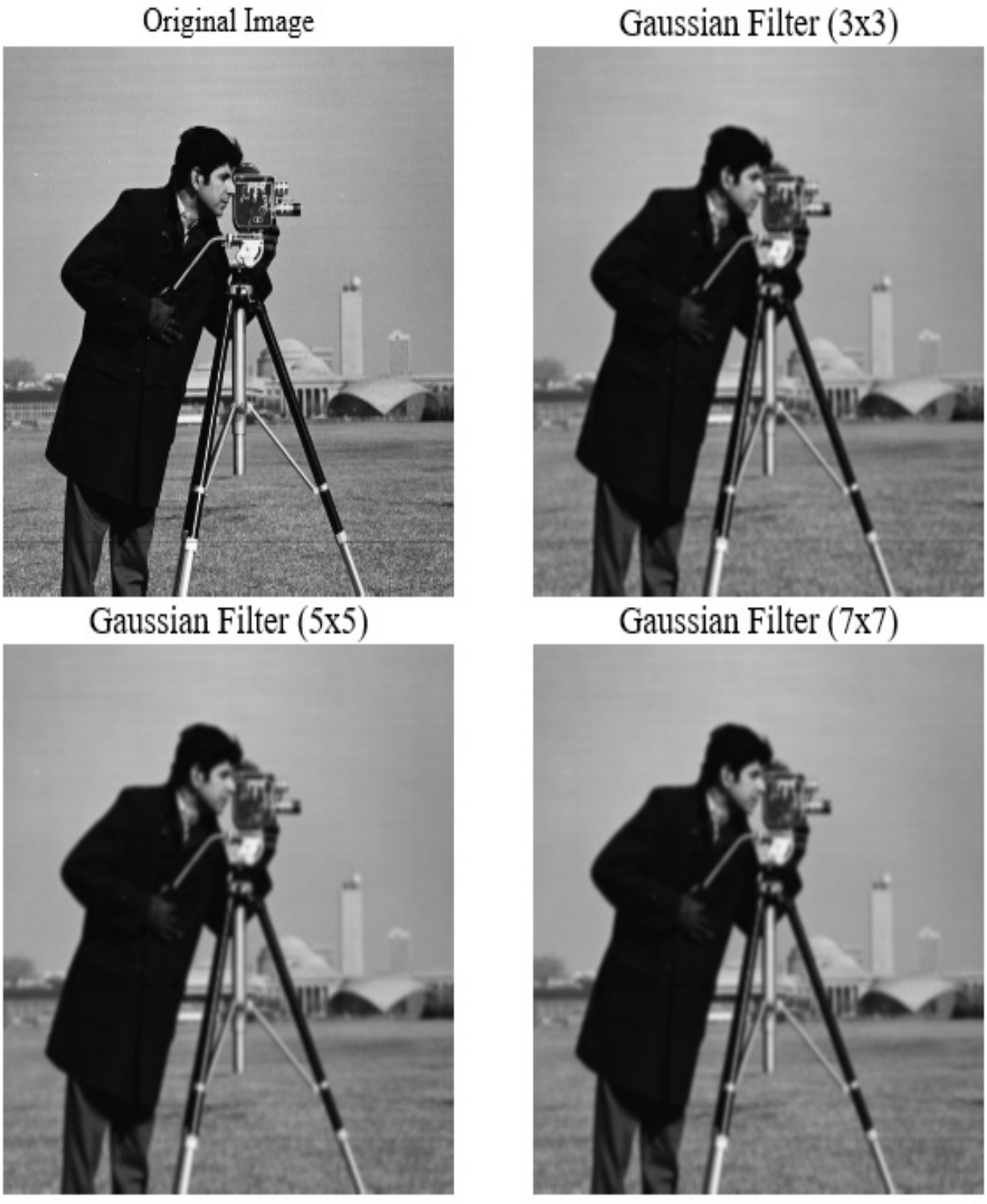
Gaussian Filter 5x5

$$\frac{1}{273} * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

OpenCV provides a function to implement the Gaussian filter. The code for it is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main ()
{
  cv::Mat h_img1 = cv::imread("images/cameraman.tif",0);
  cv::cuda::GpuMat d_img1,d_result3x3,d_result5x5,d_result7x7;
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filter3x3,filter5x5,filter7x7;
  filter3x3 = cv::cuda::createGaussianFilter(CV_8UC1,CV_8UC1,cv::Size(3,3),1);
  filter3x3->apply(d_img1, d_result3x3);
  filter5x5 = cv::cuda::createGaussianFilter(CV_8UC1,CV_8UC1,cv::Size(5,5),1);
  filter5x5->apply(d_img1, d_result5x5);
  filter7x7 = cv::cuda::createGaussianFilter(CV_8UC1,CV_8UC1,cv::Size(7,7),1);
  filter7x7->apply(d_img1, d_result7x7);

  cv::Mat h_result3x3,h_result5x5,h_result7x7;
  d_result3x3.download(h_result3x3);
  d_result5x5.download(h_result5x5);
  d_result7x7.download(h_result7x7);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Blurred with kernel size 3x3", h_result3x3);
  cv::imshow("Blurred with kernel size 5x5", h_result5x5);
  cv::imshow("Blurred with kernel size 7x7", h_result7x7);
  cv::waitKey();
  return 0;
}
```

The `createGaussianFilter` function is used to create a mask for the Gaussian filter. The datatype of the source and destination images, size of the filter and standard deviation in the horizontal direction is provided as an argument to the function. We can also provide a standard deviation in the vertical direction as an argument. if it is not provided then its default value is equal to standard deviation in horizontal direction. The created Gaussian Mask of different size is applied to the image using an apply method. The output of the program is shown below:



Again as the size of Gaussian Filter increases, more blurring is introduced in the image. The Gaussian filter is used to eliminate noise and introduce smooth blurring on an Image.

Median Filtering

When Image is affected by salt and pepper noise then it will not be eliminated by the Averaging or Gaussian filter. It needs a nonlinear filter. Median operation on neighborhood instead of averaging can help in eliminating salt and pepper noise. In this filter, the median of 9-pixel values in the neighborhood is placed at the center pixel. It will eliminate extreme high or law values introduced by salt and pepper noise. Though OpenCV and CUDA provide a function for median filtering but it is slower than regular function in OpenCV so this function is used to implement median filter as shown in code below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main ()
{
  cv::Mat h_img1 = cv::imread("images/saltpepper.png",0);
  cv::Mat h_result;
  cv::medianBlur(h_img1,h_result,3);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Median Blur Result", h_result);
  cv::waitKey();
  return 0;
}
```

medianBlur function in OpenCV is used to implement a median filter. It needs three arguments. The first argument is the source image, the second argument is the destination image and the third argument is the window size for the median operation. The output of the median filtering is shown below:

The source image is affected by salt and pepper noise as can be seen in the figure. This noise is eliminated completely by the median filter of size 3x3 without introducing extreme blurring. So median filter is very important preprocessing step when images for applications are affected by salt and pepper noise.

To summarize, we have seen three types of low pass filter which are widely used in various computer vision applications. The averaging and Gaussian filter is used to eliminate gaussian noise but it will also blur the edges of an Image. The median filter is used to remove the salt and pepper noise.

High Pass Filtering on Image

High pass filter removes the low-frequency components from the image and enhances high-frequency components. So when high pass filter is applied to an Image, it will remove the background as it is low-frequency region and enhances edges which are high-frequency components. So high pass filters can also be called edge detectors. The coefficients of the filter will change otherwise it is similar to the filters seen in the last section. There are many high pass filters available like:

1. Sobel Filter
2. Scharr Filter
3. Laplacian Filter

We will see each one of them separately in this section.

Sobel Filter

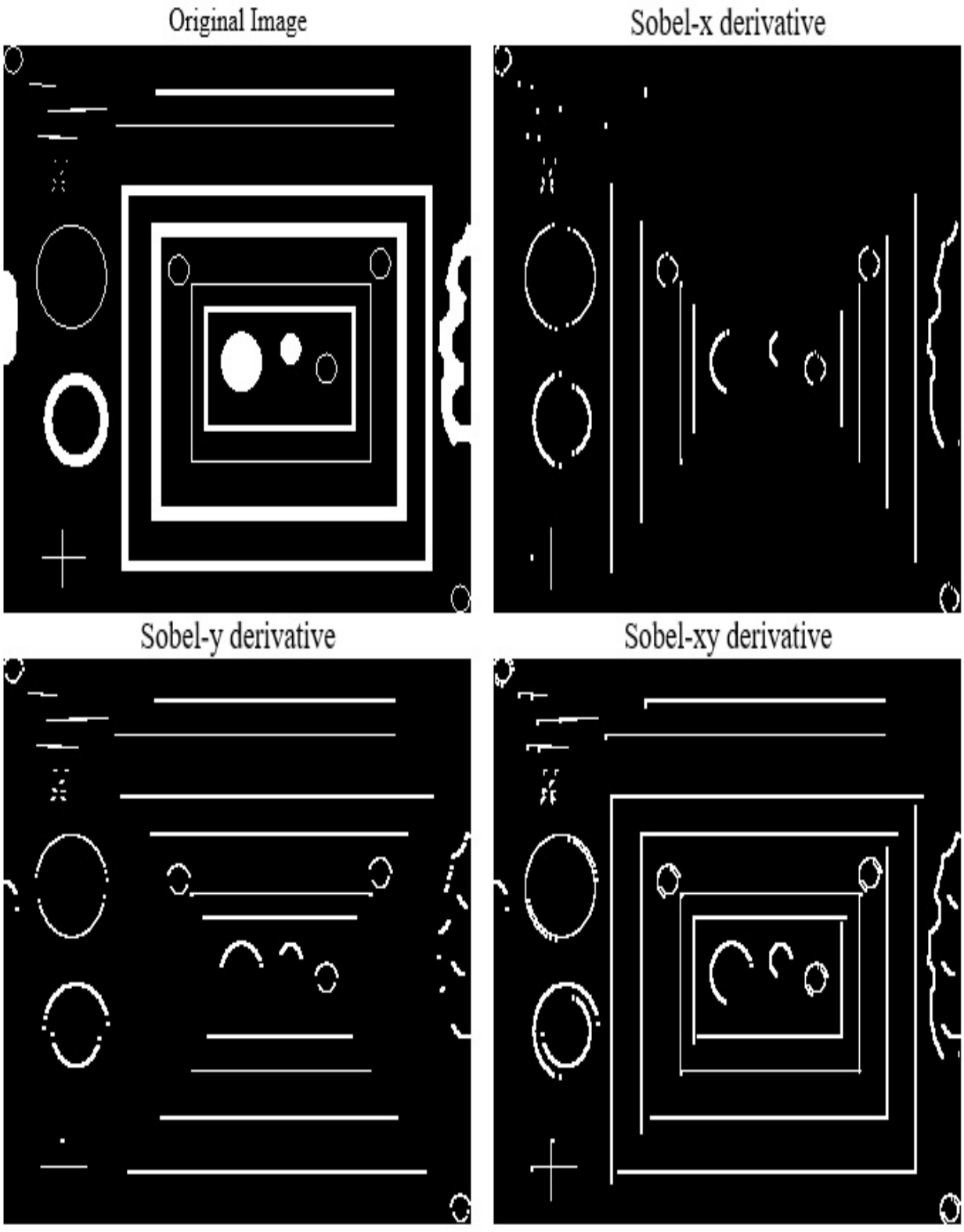
Sobel operator or Sobel filter is a widely used image processing and computer vision algorithm for edge detection applications. It is a 3x3 filter which approximates the gradient of image intensity function. It provides a separate filter to compute gradient in a horizontal and vertical direction. The filter is convolved with the image in a similar way as described earlier in this chapter. The horizontal and vertical 3x3 Sobel filter is shown below:

$$Sx = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad Sy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The code for implementing this Sobel filter is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main ()
{
  cv::Mat h_img1 = cv::imread("images/blobs.png",0);
  cv::cuda::GpuMat d_img1,d_resultx,d_resulty,d_resultxy;
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filterx,filtery,filterxy;
  filterx = cv::cuda::createSobelFilter(CV_8UC1,CV_8UC1,1,0);
  filterx->apply(d_img1, d_resultx);
  filtery = cv::cuda::createSobelFilter(CV_8UC1,CV_8UC1,0,1);
  filtery->apply(d_img1, d_resulty);
  cv::cuda::add(d_resultx,d_resulty,d_resultxy);
  cv::Mat h_resultx,h_resulty,h_resultxy;
  d_resultx.download(h_resultx);
  d_resulty.download(h_resulty);
  d_resultxy.download(h_resultxy);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Sobel-x derivative", h_resultx);
  cv::imshow("Sobel-y derivative", h_resulty);
  cv::imshow("Sobel-xy derivative", h_resultxy);
  cv::waitKey();
  return 0;
}
```

OpenCV provides createSobelFilter function for implementing Sobel filter. It requires many arguments. The first two arguments are datatype of source and destination images. Thr third and fourth arguments are the order of x and y derivative respectively. For computing x derivative or vertical edges 1 and 0 are provided and for computing y derivative or horizontal edges 0 and 1 are provided. The fifth argument is optional which indicate the size of the kernel. The default value is 3. The scale for derivatives can also be provided. To see both horizontal and vertical edges simultaneously, the result of x-derivative and y-derivative are summed. The result is shown below:



Sobel operator provides a very inaccurate approximation of derivative but still, it is quite useful in computer vision applications for edge detection. It does not have rotation symmetry. To

overcome that Scharr operator is used.

Scharr Filter

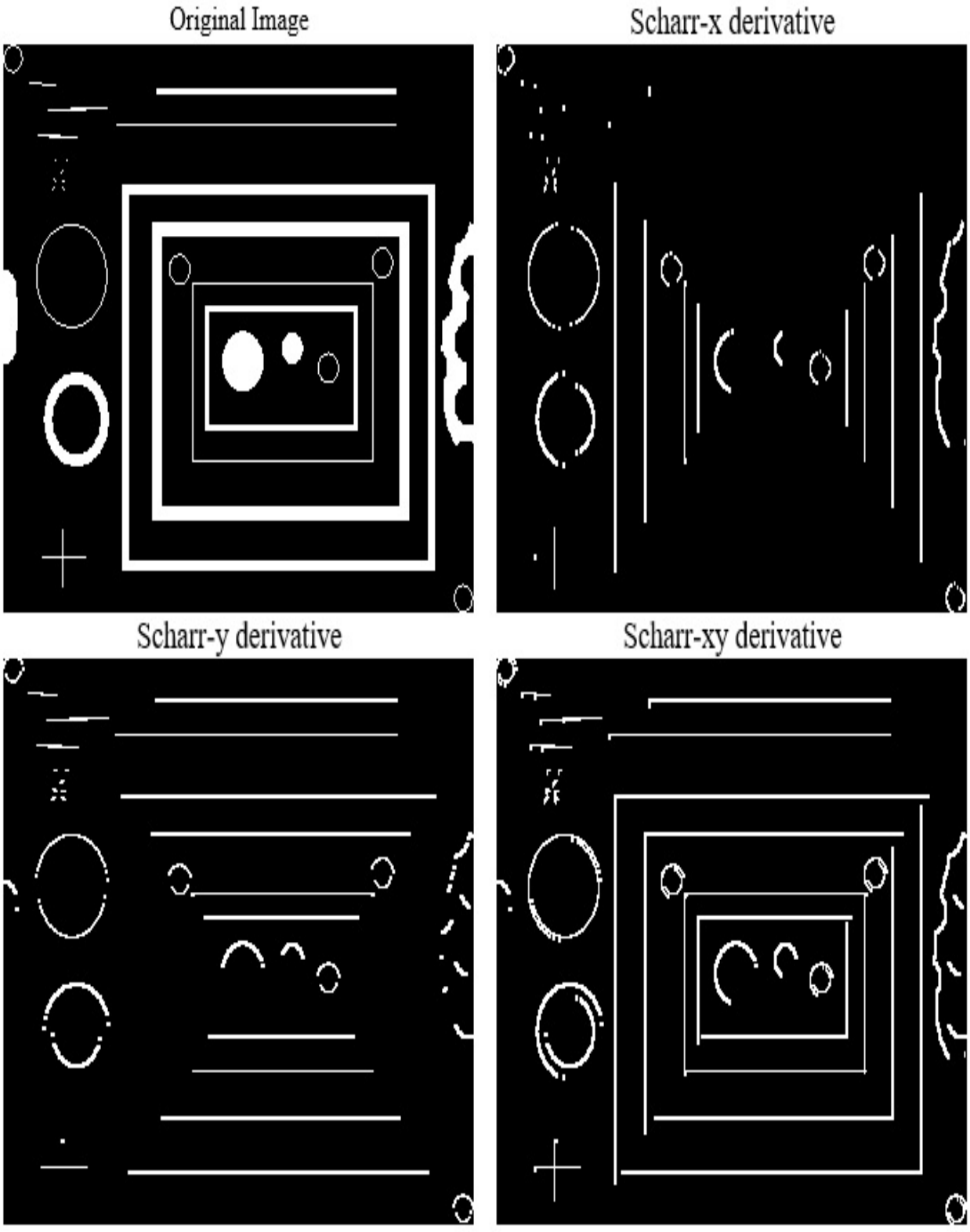
As Sobel does not provide rotation symmetry, Scharr operator is used to overcome that by using different filter mask as shown below:

$$Sx = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \qquad Sy = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

As can be seen from the mask, the Scharr operator gives more weight to a central row or central columns to find edges. The program to implement Scharr filter is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/blobs.png",0);
  cv::cuda::GpuMat d_img1,d_resultx,d_resulty,d_resultxy;
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filterx,filtery;
  filterx = cv::cuda::createScharrFilter(CV_8UC1,CV_8UC1,1,0);
  filterx->apply(d_img1, d_resultx);
  filtery = cv::cuda::createScharrFilter(CV_8UC1,CV_8UC1,0,1);
  filtery->apply(d_img1, d_resulty);
  cv::cuda::add(d_resultx,d_resulty,d_resultxy);
  cv::Mat h_resultx,h_resulty,h_resultxy;
  d_resultx.download(h_resultx);
  d_resulty.download(h_resulty);
  d_resultxy.download(h_resultxy);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Scharr-x derivative", h_resultx);
  cv::imshow("Scharr-y derivative", h_resulty);
  cv::imshow("Scharr-xy derivative", h_resultxy);
  cv::waitKey();
    return 0;
}
```

OpenCV provides createScharrFilter function for implementing Scharr filter. It requires many arguments. The first two arguments are datatype of source and destination images. Thr third and fourth arguments are the order of x and y derivative respectively. For computing x derivative or vertical edges 1 and 0 are provided and for computing y derivative or horizontal edges 0 and 1 are provided. The fifth argument is optional which indicate the size of the kernel. The default value is 3. To see both horizontal and vertical edges simultaneously, the result of x-derivative and y-derivative are summed. The result is shown below:

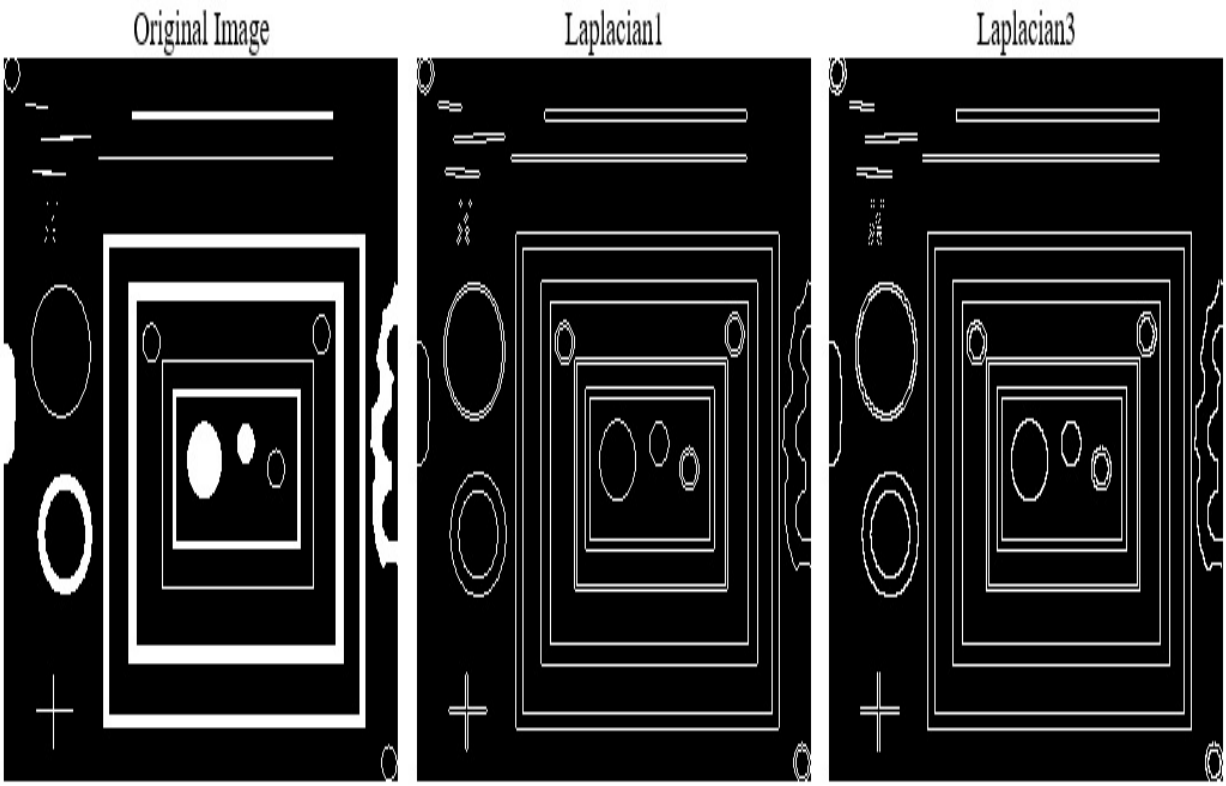

Original Image
Scharr-x derivative
Scharr-y derivative
Scharr-xy derivative

Laplacian Filter

Laplacian filter is also a derivative operator to find out edges in an Image. The difference is that Sobel and Scharr are first derivative Operators while laplacian is second derivative Operator. It also finds out edges in both horizontal and vertical direction simultaneously which is different that Sobel and Scharr operator. The Laplacian is computing the second derivative so it is very sensitive to noise in an Image. So it is desirable to blur the image and remove noise before applying the Laplacian filter. The code for implementing laplacian filter is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"

int main ()
{
  cv::Mat h_img1 = cv::imread("images/blobs.png",0);
  cv::cuda::GpuMat d_img1,d_result1,d_result3;
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filter1,filter3;
  filter1 = cv::cuda::createLaplacianFilter(CV_8UC1,CV_8UC1,1);
  filter1->apply(d_img1, d_result1);
  filter3 = cv::cuda::createLaplacianFilter(CV_8UC1,CV_8UC1,3);
  filter3->apply(d_img1, d_result3);
  cv::Mat h_result1,h_result3;
  d_result1.download(h_result1);
  d_result3.download(h_result3);
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Laplacian filter 1", h_result1);
  cv::imshow("Laplacian filter 3", h_result3);
  cv::waitKey();
  return 0;
}
```

Two laplacian filters with kernel size 1 and 3 are applied on an Image
using createLaplacianFilter function. Along with the size of the kernel, the function also requires datatype of the source and destination image as arguments. The created Laplacian filter is applied to an image using the apply method. the output of the Laplacian filter is shown below:



To summarize, in this section we have described different high pass filters like sobel, scharr and laplacian. Sobel and Scharr are first-order derivative operators used to compute edges and they are less sensitive to noise. laplacian is a second order derivative operator used to compute edges and it is very sensitive to noise.

Morphological operations on Images

Image morphology deals with regions and shapes of Image. It is used to extract image components that are useful to represent shapes and regions. Image morphology treats the image as an ensemble of sets unlike other Image processing operations seen earlier. Image interacts with a small template which is called a structuring element which defines the region of interest or neighborhood in image morphology. There are various morphological operations that can be performed on Images. They are explained one by one in this section.

1. Erosion: Erosion sets a center pixel to the minimum over all pixels in the neighborhood. The neighborhood is defined by the structuring element which is a matrix of 1's and 0's. Erosion is used to enlarge holes in the object, shrink boundary, eliminate the island and get rid of narrow peninsulas that might exist on Image boundary.

2. Dilation: Dilation sets a center pixel to the maximum over all pixels in the neighborhood. The dilation increases the size of a white block and reduces the size of the black region. It is used to fill the holes in the object and expand the boundary of the object.

3. Opening: Image opening is basically a combination of erosion and dilation. Image opening is defined by erosion followed by dilation. Both operations are performed using the same structuring elements. It is used to smooth the contours of the image, break down narrow bridges and isolate objects which are touching one another. It is used in the analysis of wear particles in engine oils, ink particles in the recycled paper etc.

4. Closing: Image oclosing is defined by dilation followed by erosion. Both operations are performed using the same structuring elements. It is used to fuse narrow breaks and eliminate small holes.

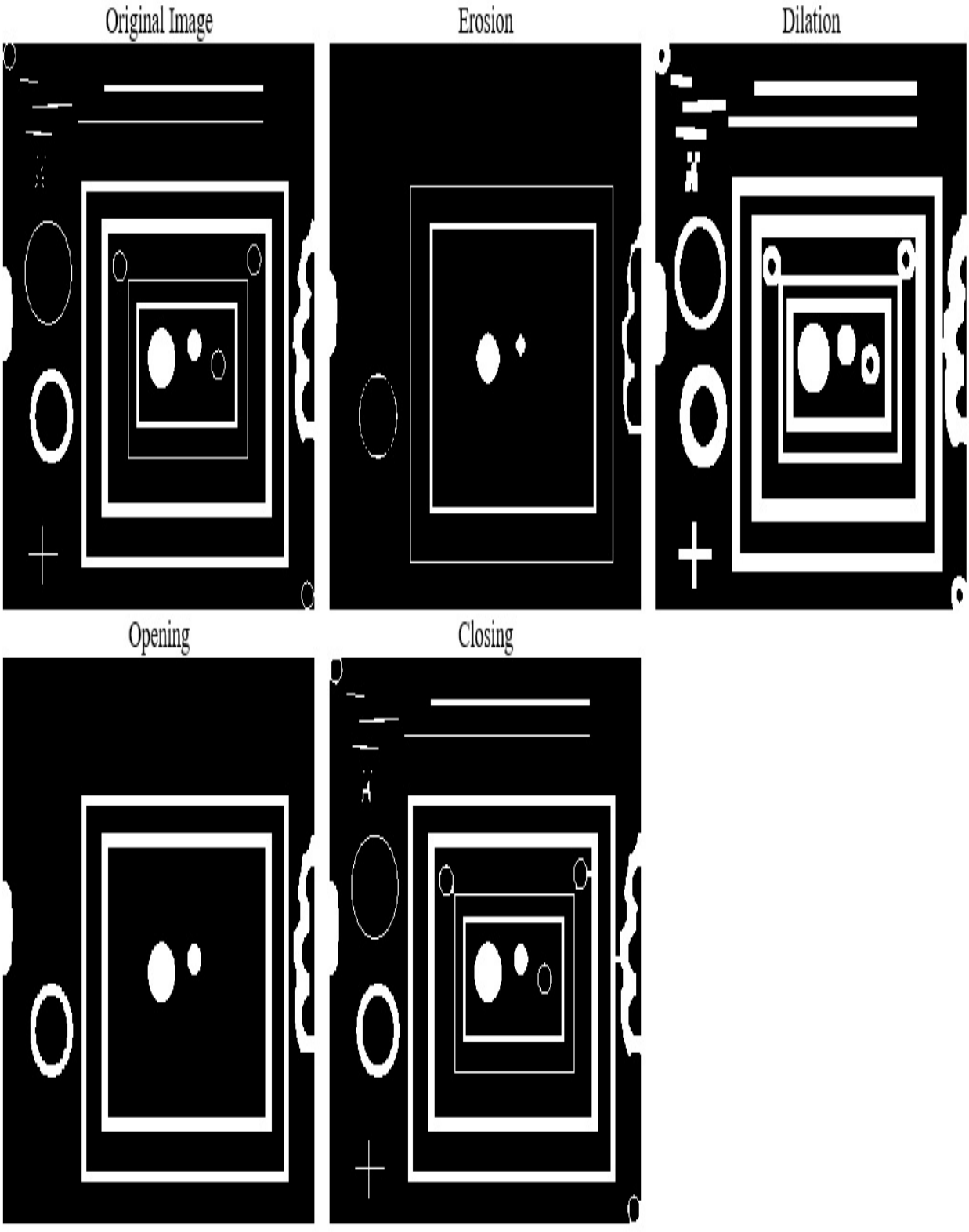
The morphological operators can be easily understood by applying them on binary images which only contains black and white color. OpenCV and CUDA provide an easy API to apply a morphological transformation on Images. The code for it is shown below:

```
#include <iostream>
#include "opencv2/opencv.hpp"
int main ()
{
  cv::Mat h_img1 = cv::imread("images/blobs.png",0);
  cv::cuda::GpuMat d_img1,d_resulte,d_resultd,d_resulto, d_resultc;
  cv::Mat element = cv::getStructuringElement(cv::MORPH_RECT,cv::Size(5,5));
  d_img1.upload(h_img1);
  cv::Ptr<cv::cuda::Filter> filtere,filterd,filtero,filterc;
  filtere = cv::cuda::createMorphologyFilter(cv::MORPH_ERODE,CV_8UC1,element);
  filtere->apply(d_img1, d_resulte);
  filterd = cv::cuda::createMorphologyFilter(cv::MORPH_DILATE,CV_8UC1,element);
  filterd->apply(d_img1, d_resultd);
  filtero = cv::cuda::createMorphologyFilter(cv::MORPH_OPEN,CV_8UC1,element);
  filtero->apply(d_img1, d_resulto);
  filterc = cv::cuda::createMorphologyFilter(cv::MORPH_CLOSE,CV_8UC1,element);
  filterc->apply(d_img1, d_resultc);

  cv::Mat h_resulte,h_resultd,h_resulto,h_resultc;
  d_resulte.download(h_resulte);
  d_resultd.download(h_resultd);
  d_resulto.download(h_resulto);
  d_resultc.download(h_resultc);
```

```
  cv::imshow("Original Image ", h_img1);
  cv::imshow("Erosion", h_resulte);
  cv::imshow("Dilation", h_resultd);
  cv::imshow("Opening", h_resulto);
  cv::imshow("closing", h_resultc);
  cv::waitKey();
  return 0;
}
```

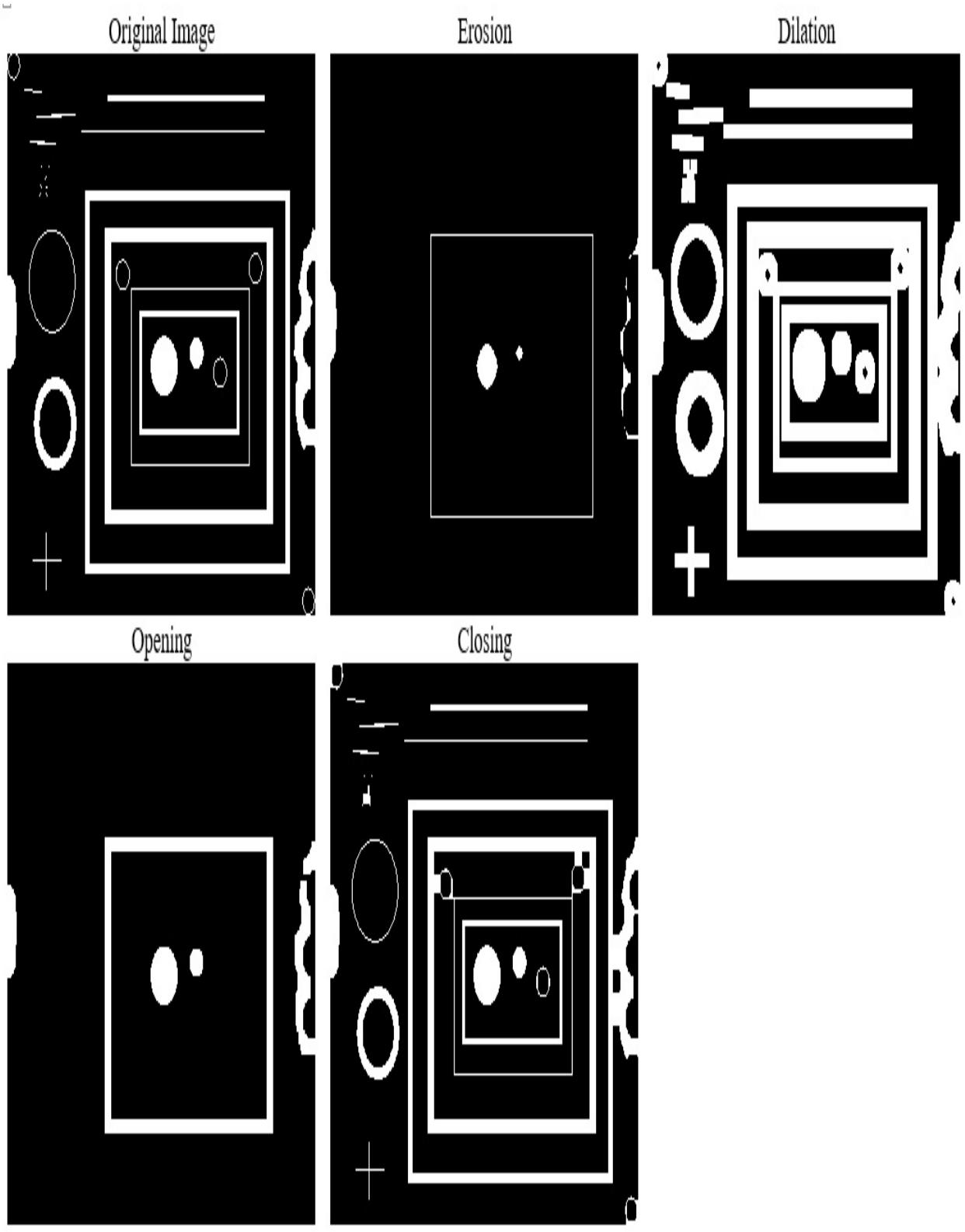
Structuring element which defines the neighborhood for morphological operation needs to be created first. This can be done by getStructuringElement function in OpenCV. The shape and size of the structuring element need to be provided as an argument to this function. In the code rectangular structuring element of size 5x5 is defined.

The filter for morphological operations is created by using `createMorphologyFilter` function. It needs three mandatory arguments. The first argument defines the operation to be performed. `cv::MORPH_ERODE` is used for erosion, `cv::MORPH_DILATE` is used for dilation, `cv::MORPH_OPEN` for opening and `cv::MORPH_CLOSE` is used for closing. The second argument is the datatype of an Image and the third argument is the structuring element created earlier. The apply method is used to apply this filters on an Image. The output of morphological operations on an Image is shown below:



As can be seen from the output, erosion reduces the boundary of an object while dilation thickens it. We are considering the white part as an object and the black part is background. Opening

smooths the contours of the image. The closing is eliminating small holes in an Image. If the size of the structuring element is increased to 7x7 from 5x5 then the erosion of boundary will be more pronounced in erosion and boundary will get thicker in dilation operation. The small circles on the left side which were visible in eroded image with 5x5 are removed when it is eroded with 7x7. The output of morphological operations using 7x7 structuring element is shown below:



To summarize, morphological operations are important to find out components used to define shape and regions of an Image. It can be used to fill holes in an Image and smoothen the contours

of the Image.

Summary

This chapter described the method to access pixel intensities at a particular location in Image. It is very useful when we are performing a pointwise operation on Images. The histogram is a very important global feature used to describe an Image. This chapter described the method to compute the histogram and the process of histogram equalization which improves the visual quality of an Image. A various geometric transformation like Image resizing, rotation and translation are explained in detail. Image filtering is a useful neighborhood processing technique used to eliminate noise and extracting edge features of an Image are described in detail. Low pass filter is used to remove noise but it will also blur out edges of an image. High pass filter removes the background which is low-frequency region while enhancing edges which are high-frequency regions. The last part of the chapter described different morphological operators like erosion, dilation, opening and closing which can be used to describe a shape of the image and fill holes in an Image. In the next chapter, we will use these concepts to build some useful computer vision applications using OpenCV and CUDA.

Questions

1. Write an OpenCV function to print pixel intensity at location(200,200) of any color image on the console.
2. Write an OpenCV function to resize an image to (300,200) pixels. Use bilinear Interpolation method.
3. Write an OpenCV function to upsample an Image by 2. Use Area Interpolation method.
4. State True or False: The blurring decreases as we increase the size of the averaging filter.
5. State True or False: Median filter can remove gaussian noise.
6. What steps can be taken to reduce noise sensitivity of Laplacian operator?
7. Write an OpenCV function to implement top hat and black hat morphological operation.

Object detection and tracking using OpenCV and CUDA

Coming Soon...!

Introduction to Jetson Tx1 development board and installing OpenCV on Jetson TX1

Coming Soon...!

Deploying computer vision applications on Jetson TX1

Coming Soon...!

Getting started with PyCUDA

Coming Soon...!

Working with PyCUDA

Coming Soon...!

Basic Computer vision application using PyCUDA

Coming Soon...!