SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Civil Engineering

Reg. No.: SvF-5343-63353

Parallel computing in image processing using GPU and CUDA architecture

Master thesis

2018 **Bc. Alexander Baťka**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Civil Engineering

Reg. No.: SvF-5343-63353

Parallel computing in image processing using GPU and CUDA architecture

Master thesis

Study programme: Mathematical and Computational Modeling
Study field: 9.1.9. Applied Mathematics
Training workplace: Department of Mathematics and Constructive Geometry
Thesis supervisor: Ing. Jozef Urbán, PhD.

Bratislava 2018 **Bc. Alexander Baťka**

Slovak University of Technology in Bratislava
Department of Mathematics and Constructive Geometry

Faculty of Civil Engineering
Academic year: 2017/2018
Reg. No.: SvF-5343-63353

STU SvF

MASTER THESIS TOPIC

Student: **Bc. Alexander Baťka**

Student's ID: 63353

Study programme: Mathematical and Computational Modeling

Study field: 9.1.9. Applied Mathematics

Thesis supervisor: Ing. Jozef Urbán, PhD.

Topic: **Parallel computing in image processing using GPU and CUDA architecture**

Language of thesis: English

Specification of Assignment:

Téma sa zaoberá paralelizáciou algoritmov spracovania obrazu pomocou grafických procesorov využitím architektúry CUDA (Compute Unified Device Architecture).
Študent naštuduje problematiku využitia GPU pre všeobecné výpočty. Naučí sa pomocou CUDA paralelizovať výpočty používané v spracovaní obrazu
ako BiConjugate Gradient Stabilized Method, filtrácia obrazu lineárnou rovnicou vedenia tepla, transformácia a registrácia obrazu. Dosiahnuté výsledky vhodne prezentuje.

Selected bibliography:

1. Sanders, J. – Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Upper Saddle River : Addison Wesley, 2011. 290 s. ISBN 978-0-13-138768-3.
2. Krivá, Z. – Mikula, K. – Stašová, O. *Spracovanie obrazu: Vybrané kapitoly z prednášok.* Bratislava : Slovenská technická univerzita v Bratislave, 2016. 149 s. ISBN 978-80-227-4535-2.
3. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 631–644
4. CUDA programming: A Developers guide to parallel computing with GPUs, Shane Cook, 2013, ISBN-978-0-12-415933-4

Assignment procedure from: 02. 10. 2017

Date of thesis submission: 10. 05. 2018

Bc. Alexander Baťka
Student

prof. RNDr. Radko Mesiar, DrSc.
Head of department

prof. RNDr. Karol Mikula, DrSc.
Study programme supervisor

POKYNY
na vypracovanie diplomovej práce

Úvodné ustanovenie

V zmysle zákona č. 131/2002 Z. z. o vysokých školách a o zmene a doplnení niektorých zákonov v znení neskorších predpisov je súčasťou štúdia podľa každého študijného programu aj záverečná práca. Jej obhajoba patrí medzi štátne skúšky. Záverečnou prácou pri štúdiu podľa študijného programu druhého stupňa je diplomová práca. Podkladom na vypracovanie diplomovej práce je zadanie diplomovej práce

Štruktúra záverečnej práce

- titulný list,
- zadanie záverečnej práce,
- pokyny na vypracovanie,
- vyhlásenie autora,
- názov a abstrakt v slovenskom a v anglickom jazyku (spolu v rozsahu jednej strany),
- obsah s očíslovaním kapitol,
- zoznam príloh,
- zoznam skratiek a značiek,
- text samotnej práce (odporúčané členenie),
  - úvod,
  - súčasný stav problematiky,
  - ciele záverečnej práce,
  - vlastné riešenie členené na kapitoly podľa charakteru práce,
  - zhodnotenie dosiahnutých výsledkov resp. navrhnutých riešení,
  - záver,
- resumé v slovenskom jazyku v rozsahu spravidla 10 % rozsahu ZP (len pre práce vypracované v cudzom jazyku),
- zoznam použitej literatúry,
- prílohy (výkresy, tabuľky, mapy, náčrty) vrátane postera s rozmermi 1000x700 mm.

Rozsah a forma

1. Obsah a forma záverečnej práce musí byť spracovaná v zmysle vyhlášky MŠVVaŠ SR č. 233/2011 Z. z., ktorou sa vykonávajú niektoré ustanovenia zákona č. 131/2002 Z. z. a v zmysle Metodického usmernenia č. 56/2011 o náležitostiach záverečných prác.
2. Vyžadovaný rozsah diplomovej práce je 30 až 50 strán. Odovzdáva sa v dvoch vyhotoveniach. Jedno vyhotovenie musí byť viazané v pevnej väzbe (nie hrebeňovej) tak, aby sa jednotlivé listy nedali vyberať. Rozsiahle grafické prílohy možno v prípade súhlasu vedúceho práce odovzdať v jednom vyhotovení.

3. Autor práce je povinný vložiť prácu v elektronickej forme do akademického informačného systému. Autor zodpovedá za zhodu listinného aj elektronického vyhotovenia.
4. Po vložení záverečnej práce do informačného systému, predloží autor fakulte ním podpísaný návrh licenčnej zmluvy. Návrh licenčnej zmluvy je vytvorený akademickým informačným systémom.
5. Odporúčaný typ písma je Times New Roman, veľkosť 12 a je jednotný v celej práci. Odporúčané nastavenie strany - riadkovanie 1,5, okraj vnútorný 3,5 cm, vonkajší 2 cm, zhora a zdola 2,5 cm, orientácia na výšku, formát A4.
6. Obrázky a vzorce sa číslujú v rámci jednotlivých kapitol (napr. obr. 3.1 je obrázok č. 1 v kapitole 3). Vzorce sa číslujú na pravom okraji riadku v okrúhlych zátvorkách - napr. (3.1).
7. Všetky výpočty musia byť usporiadané tak, aby bolo možné preveriť ich správnosť.
8. Pri všetkých prevzatých vzorcoch, tabuľkách, citovaných častiach textu musí byť uvedený prameň.
9. Citovanie literatúry vrátane elektronických materiálov sa uvádza podľa STN ISO 690 (01 0197): 2012. *Informácie a dokumentácia. Návod na tvorbu bibliografických odkazov na informačné pramene a ich citovanie.*
10. Príklad zoznamu bibliografických odkazov:
    ABELOVIČ, J. a kol.: *Meranie v geodetických sieťach.* Bratislava, Alfa 1990, ISBN 0-1554-9173.
    MICHALČÁK, O. – ADLER, E.: Výskum stability dunajských hrádzí. In: *Zborník vedeckých prác Stavebnej fakulty SVŠT.* Bratislava: Edičné stredisko SVŠT 1976, s. 17-28. ISBN 0-3552-5214.
    ŠÜTTI, J.: Určovanie priestorových posunov stavebných objektov. *Geodetický kartografický obzor.* 2000, roč. 2, č. 3, s. 8-16. ISSN 0811-6900.
    Article 18. Technical Cooperation. http://www.lac.uk/iso/tc456 (2013-09-28)
11. Za jazykovú a terminologickú správnosť záverečnej práce zodpovedá diplomant.
12. Formu postera (elektronická alebo aj tlačená) určí garant študijného programu.
13. Vzor pre poster je uvedený na dokumentovom serveri v akademickom informačnom systéme univerzity.

.....
podpis garanta študijného programu

Ustanovenia týchto pokynov som vzal na vedomie. Som si vedomý(á), že ak nebude moja diplomová práca vypracovaná v súlade s týmito pokynmi, nebude prijatá na obhajobu.

V Bratislave ....
podpis študenta

Čestné prehlásenie

Prehlasujem, že som diplomovú prácu Parallel computing in image processing using GPU and CUDA architecture vypracoval samostatne, na základe použitej literatúry a s odbornou pomocou vedúceho práce.

Bratislava May 9, 2018 .....

vlastnoručný podpis

Acknowledgment:

I would like to thank my supervisor Ing. Jozef Urbán, PhD. for help, time spent answering my question, understanding in bad times and instant replies to emails regarding the thesis in any hour of the day.

Abstrakt

Cielom práce je porozumieť, porovnať a následne integrovať paralelné algoritmy spracovania obrazu, algoritmických solverov do CUDA. V rámci práce treba vybrať vhodné existujúce softvérové prostriedky, programovacie jazyky, prostredie, knižnicu CUDA a za pomoci správneho hardvéru vytvoriť paralelné algoritmy na grafickej karte, porovnať ich z paralelným algoritmami na CPU, vyhodnotiť nároky potrebné na zníženie časových nárokov a zhodnotiť cenové náklady hardvéru pre jednotlivé prístupy.

Kľúčové slová: CUDA, C++, Paralelný algoritmus

Abstract

The goal of our paper is to understand, compare and integrate parallel algorithms for image processing and algorithmic solvers to CUDA. Within the paper it is needed to chose correct existing software instruments, programming languages, development environment, CUDA library and with the help of correct hardware create parallel algorithms on graphic card, compare them with parallel algorithms on CPUs, evaluate the needs to decrease the time requirements and conclude the price of hardware for either approaches.

Keywords: CUDA, C++, Parallel algorithm

Contents

Acronyms

API Application programming interface

BiCGStab BiConjugate Gradient Stabilized

CIR Central instruction register

CPU Central processing unit

c.c.k. cuda custom kernel

FLOPS Floating point operations per second

GPU Graphics processing unit

HDD Hard drive storage

IPC Instructions per cycle

MAR Memory address register

MPP Massively parallel processors

NUMA Non-Uniform memory access

RAM Random access memory

SIMD Simple instruction, multiple data

SM Shading multiprocessor

SMP Symmetric multiprocessor

SSD Solid state drive

Introduction

From the history of computer development we have observed the need of improving the effectiveness in which they operate, along with improving their computational potential. Unfortunately without improving the software alongside the hardware this never can be achieved. This has become increasingly more evident with the physical barriers of single core processing units, where everything ran sequentially. Therefore, processing units started to increase in numbers within a single system and parallelism became available. Tasks that have before taken days, could be divided into multiple tasks and ran on multiple compute units to reduce this time to insignificant intervals. This potential allowed software engineers to iteratively develop more complex software tools.

Visualization of computer results and instructions known as user interface, led into development of graphics cards which are today an essential part of every computer. The increasingly more powerful graphics cards, with increasingly more cores gave the potential to not only improve the complex visualization calculation, but use this computational potential for general computations and allow software developers and scientists to offload or rather transfer the computations from other computational unit to graphic cards. Thanks to the computational potential of graphics cards, this has led into reducing the time of calculations by a significant margin.

Multiple approaches and software techniques were developed over time for this purpose. In our analysis we will look over these approaches, how they work, what are the necessities one needs for them and we will look how the implementation of general algorithms and image processing algorithms onto the graphics card with the help of CUDA can benefit over implementing them on a CPU, as well as the limitations and price comparisons of each approach.

As the first step to understanding the software, we will look into the hardware-side of a computer, more accurately into CPU, GPU, memory, their involvement in computations and how they can be used together in parallel architectures to further increase performance.

After understanding the basic principles of hardware and how it is used, in second chapter we look into the software implementation of parallel programming. Specifically the problems that need to be overcome, different approaches to overcoming these problems and the differences of parallel programming with the use of different processing units. This under-

standing leads to a comparison in performance, divided into the aspects that affect it as well as the cost of performance when choosing between the different approaches.
As the next step after understanding the theory, implementation is described. Firstly by choosing the correct programming language with development environment. Secondly by correctly implementing the algorithms at hand, familiarizing ourselves with working with CUDA and the different course of implementation when doing general purpose programming on GPU with the ambition to eliminate the drawbacks of doing general purpose computing on a GPU.
At last we look into the different general algorithms used in mathematics as well as image processing algorithms, their multiple implementations, either on GPU or CPU, and the time comparison and computational time decrease when running these algorithms on GPU.

Chapter 1

Hardware

In this chapter we will look over physical components of a computer, their purpose and simplified inner workings. All calculations and comparisons were made on the same personal computer (Asus Zenbook ux501vw) using:

- Windows 10 Pro
- Intel I7-6700Q
- NVidia GTX 960M 4GB
- 16GB RAM
- Samsung 512gb NVME m.2 SSD
- Microsoft C/C++ Compiler - Visual C++ 14.0
- CUDA 9.0

With the power cord always plugged in to eliminate Windows energy saving optimization that reduces components power consumption and thus their computing potential.

1.1 CPU - central processing unit

The Central Processing Unit, or "brains" of the computer, is one of the most important processing units inside a computer. It is the part of a computer that performs actions, calculations and runs programs. It takes inputs, referred to as instructions from the computer's RAM, decodes them, processes them, delivers and outputs them. Vast variety of devices uses CPUs, such as laptops, smartphones, fridges, etc. Their function can be divided into three steps: fetch, decode and execute (Figure 1.1).

Fetch

Fetching data involves receiving instructions represented as series of numbers that are passed to the CPU from the RAM. Each instruction is only a small part of any operation, that means that the CPU needs to know which instruction comes next. The current instruction address is held by a program counter. The contents of the program counter are then loaded to memory address register (MAR), after which the program counter length is increased to reference the next instructions address. The data required is loaded into memory buffer register (MBR), that is followed by loading MAR contents into central instruction register(CIR).

Decode

Once an instruction is fetched and stored in the CIR, the CPU passes the instruction to a circuit called the instruction decoder. This converts the instruction into signals to be passed through to other parts of the CPU for action.

Execute

In the final step, the decoded instructions are sent to the relevant parts of the CPU to be completed. The results are usually written to a CPU register, where they can be referenced by later instructions.

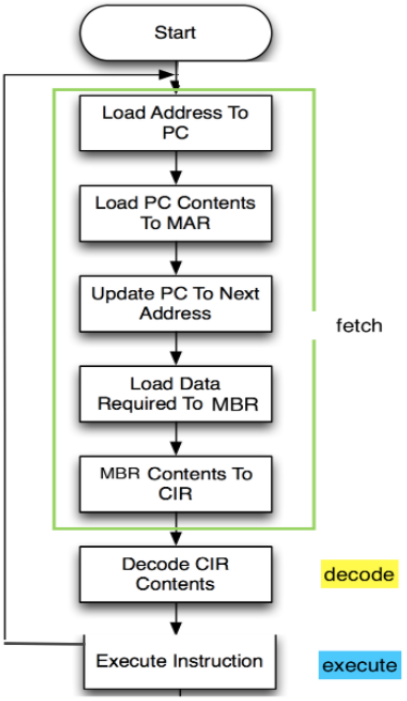

Figure 1.1: Instruction cycle [19]

At first CPUs had only one single core, this meant that the CPU was limited to just single set of tasks in multiple threads which were executed sequentially, which in its essence

was a type of parallelism. After pushing the single core performance to its limits, the drive for performance improvements led to creation of multi-core processors, thus splitting the workload into multiple channels. This is where real parallelism began.

Further improvements were made to the cache of the processor, the way processor communicates with other components, frequency under which the CPU operates and amount of data it could process at one time. About these topics please refer to [3].

1.2 GPU - graphical processing unit

Similarly to CPU, the GPU receives data, decodes them and then executes them, however, the GPU is designed specifically to perform complex mathematical and geometric calculations that are necessary for graphics rendering. Every pixel that is rendered onto the screen is calculated by the GPU and then sent to the monitor as a map of an image to be shown. Unlike CPU where every task may be completely different to the other and thus has to be processed fast one after another by a single core. GPUs were initially used to accelerate the memory-intensive work of texture mapping and polygon rendering, later adding units to accelerate geometric calculations such as the rotation and translation of vertices into different coordinate systems. Because most of these calculations involves matrix and vector operations, these tasks are easily divided into multiple smaller identical tasks that can be executed at the same time. This lead into the GPU having significantly more cores running at a lower frequency than a CPU to allow for parallelism.
Usually there are multiple types of GPUs with different uses, more about them please refer to [12].

1.3 Memory

Core part of each computer are the instructions it executes, without them the processor would not know what to do. These instructions are saved as data and saved in memory. Although each processing unit has its own low level memory (e.g. L2, L3 cache in CPU), which is extremely fast and can be accessed faster than any other type of memory and is used for the instructions and data that the processing unit is implementing in each moment, it is also very expensive and therefore used scarcely. Memory used for other purposes is divided into multiple parts, RAM or random access memory which is used as a temporary storage for all the data that needs to be accessible fast and be available to insure smooth working of a program. This memory is usually different for each computing unit, meaning that CPU uses different RAM than the GPU. The difference between these two types of memory will be discussed in further sections. Then there is offline storage or hard storage, which contains

data that is to be stored for longer periods of time. Offline storage is usually very slow compared to RAM or cache, but has the advantage that it keeps the memory written without any need of power supply. It is because data is physically written on a physical medium (like storage disks in HDD).

1.3.1 RAM

Random access memory, known as primary storage, is called and considered random access because any memory address can be accessed directly at any time if you know the address of the data your are looking for while the computer is running. This is possible only while the memory has sufficient power, after the power has been cut of, all the memory stored on it, is lost.

Main purpose of RAM is to hold data that are currently in use so they can be quickly reached by the processor. If there is more data than the RAM is capable of storing, the overflow is then written to offline storage (HDD, SDD), significantly impacting the performance in negative way. More in [28].

1.3.2 GPU RAM

Every processing unit needs storage for data that can be accessed quickly and without slowdowns. For this purpose GPUs, have their own separate part of RAM, called vRAM (video RAM). This memory is either taken from RAM (case of integrated chips) or is a separate extra memory located and directly connected to the GPU. This memory is used to hold textures, 3D meshes, images and any type of data that the GPU uses. The more memory is available the more data can be used for computation, thus more complex it can be. AS a workaround for not having enough vRam is sending the overflow to the RAM or HDD/SDD which causes significant performance decrease. More in [22].

1.4 Architecture of parallel computers

When it comes to parallel computing, very important aspect of improving performance is how different programs access memory. This can dramatically increase performance or cause absolute ineffectiveness of the program. The most basic implementation is sequential computing, every block of memory has only one processing unit that executes the data, as in Figure 1.2.

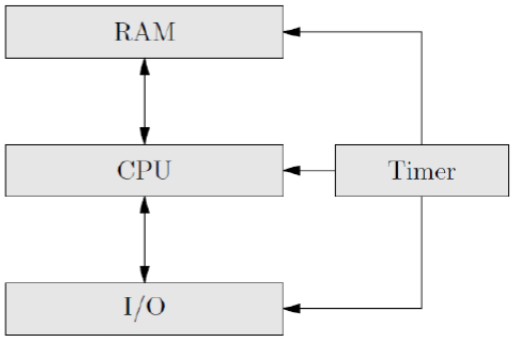

Figure 1.2: Von Neumann architecture of a sequential computer [19]

Whereas parallel computers are defined by having multiple processing units for data processing, as in Figure 1.3.

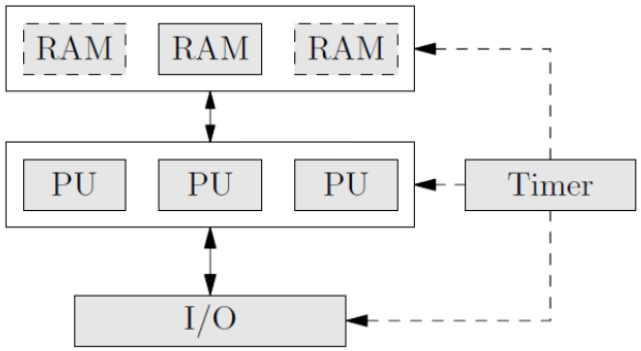

Figure 1.3: Example of parallel computer architecture[19]

1.4.1 Examples of parallel architectures

Parallel computers are divided into categories by:

- the way they access memory (local memory)
- the way they share memory (global memory)
- the way they communicate with each other

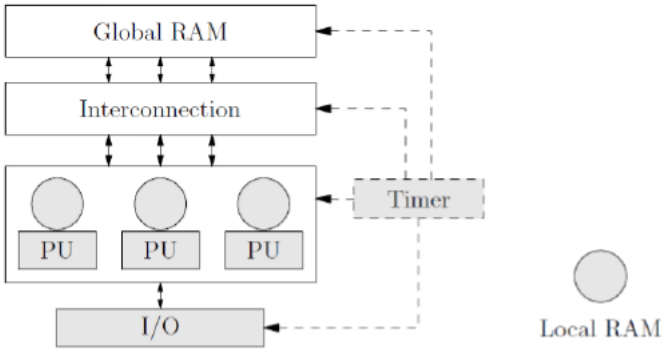

Figure 1.4: Graphical representation of global and local memory [19]

More than one category can be seen in one computer, e.g. in Figure 1.4, all processing units have their own local memory, which cannot be accessed from other processing unit, shared global memory which is accessible by all, though this access is slower than accessing local memory.
Further we will discuss the most used architectural types.

Processor arrays

Type of integrated circuit with parallel array of CPUs and RAM memories controlled by a single front end computer. Front end computers purpose is to distribute the workload between the CPUs in the processor array. Every CPU in the array has its own memory in which it holds instructions for computations, as in Figure 1.5. This architecture is used mainly as ASIC (application specific integrated circuits) machines.

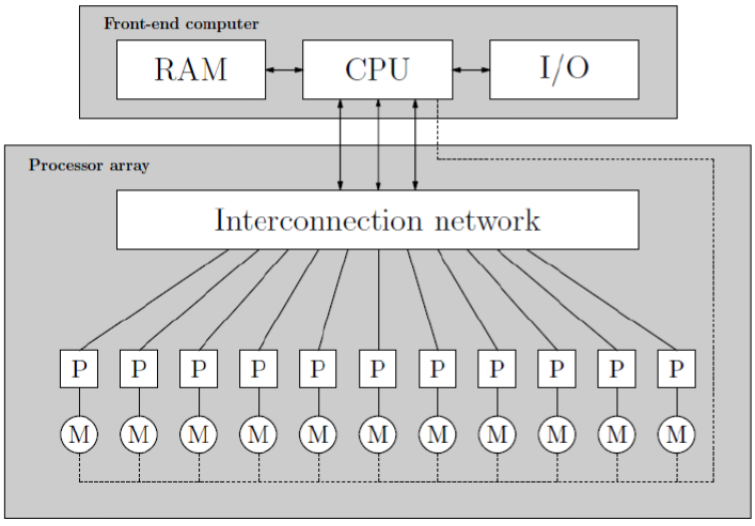

Figure 1.5: Processor array architecture model [19]

Multiprocessors

Contains multiple full fledged CPUs with shared and local memory within a single computer. Mostly used in clusters. They are differentiated by the type of memory they use and how they share memory.

SMP - Symmetric Multiprocessors

Mostly used in personal/server computers with the support for up to 8 CPUs. Every CPU contains its own cache (local memory) and shared memory, to which each CPU has the same access (Figure 1.6), known as uniform memory access (UMA). This means that an different address on each processor can point to the same address in the physical memory.

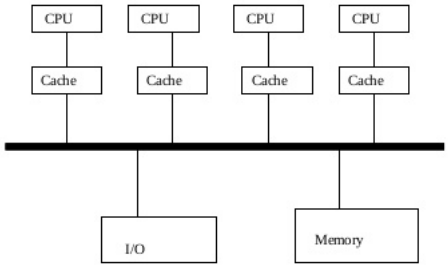

Figure 1.6: Symmetric multiprocessor architecture model [19]

NUMA - Non-Uniform Memory Access

Similarly to SMP, but with shared memory using hardware/software directory-based protocol. Memory is shared between all processors, but is divided into local memories for each processor (Figure 1.7). Every local memory is accessible by other processor at a cost of slower bandwidth.

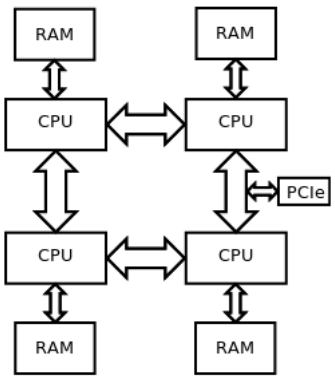

Figure 1.7: Non-uniform memory access architecture model [19]

MPP - Massively Parallel processors

MPP represents a combination of SMP and NUMA nodes interconnected via a high speed network. Every node has its own processor, memory, I/O and operating system (Figure 1.8). This approach allows for easily upgradeable systems. By adding a node to the network we can expand and increase the computational potential. This approach is used in Data warehouses to handle the processing of very large amounts of data.

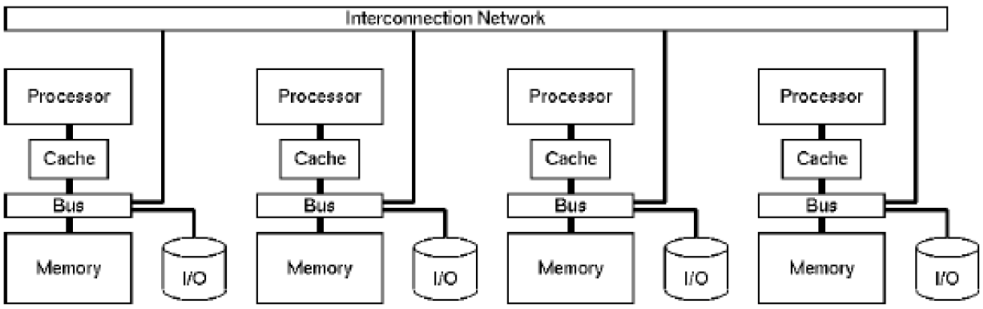

Figure 1.8: Massively parallel processor architecture model [19]

Grid

One of the biggest drawbacks of all the approaches we have talked about is the physical need of having the system on the same network, usually in one location. Grid computing is an approach for a distributed system with non-interactive workloads. Every node/computer in the grid is set to perform a different task, together trying to solve/achieve a common goal. Example of a working grid is PrimeGrid a worldwide grid, searching for prime numbers, that anyone in the world can join with their personal PC.

General-purpose computing on GPU

Use of GPUs, which are typically used only for computing computer graphics, to perform computations usually handled by a CPU. Thanks to video cards being separate integrated circuits with their own memory and processing units, multiple video cards can be connected in one computer which further parallelizes the already parallel nature of GPUs. As discussed in section 1.2 and further in [12], one GPU can have thousands of cores, where as commercial CPUs only have up to 28-cores. This means that if CPU has 220 GFLOPS (section 2.5.4), a similarly priced GPU can have 4500 GFLOPS in raw arithmetic computational potential.

Chapter 2

Parallel programming

Large problems, such as matrix operations, can be often divided into smaller tasks, which can be executed at the same time without affecting the result. Creating parallel programs became broader interest due to physical constraint preventing increase in processor frequency. Along with increase in power consumption (and heat generation) led to focus on parallel architecture of computers, in form of multi-core systems.
Different approaches in code execution and memory usage allow for further optimization and maximizing the raw computational potential a system.

2.1 Models of parallel programming

Different approaches of parallel programming are classified according to the level at which the hardware supports parallelism. Either having multiple cores, processors or processing elements (e.g. graphics card) withing a single machine, or by using multiple computers working on the same task[20].

SMP

Mainly used for workloads where all tasks running concurrently need to share memory. While some tasks has to run in sequence, other can be forked, executed and afterwards joined together in one process to form fork-join model.

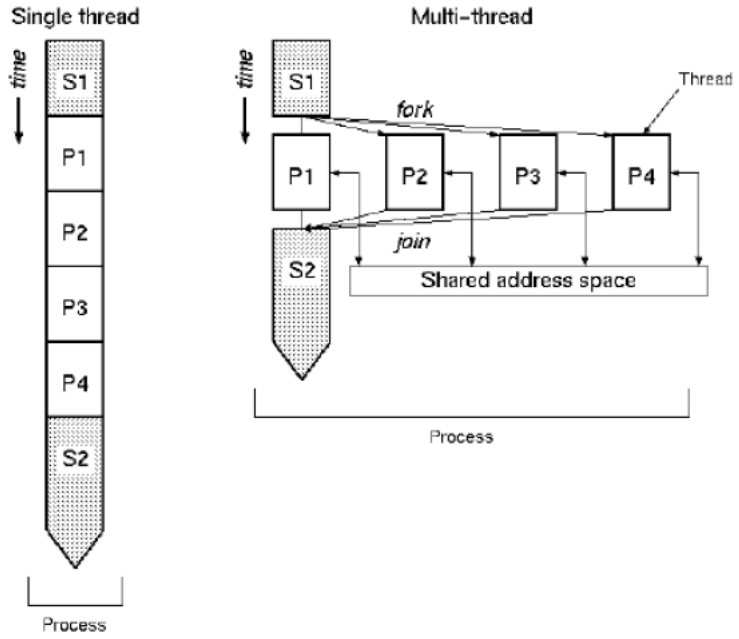

Figure 2.1: Difference between sequential and fork-join process [19]

Every fork creates its own thread in the same process. Widely used API (application programming interface) that focuses on shared memory multiprocessing is OpenMP (more in section 2.3.1).

Simple MPP

Simple MPP is characterized as a single process running on one node of the system while they communicate with other nodes through a network (e.g. Ethernet connection). Normally, the processes running on every node run longer on their own as if they ran in sequence on one node, but the ability to run these processes sequentially allows up to 4x decrease in computational time.

Hybrid MPP

By connecting multiple SMP nodes together, we get hybrid multiprocessor systems. This approach maximizes the effectiveness, availability and expandability, therefore it is mostly used today.

2.2 Communication

Memory is either shared or distributed. Access to local memory is typically faster than access to distributed memory. Physical communication between different types of memory is done through crossbar switch, a shared bus or an interconnected network. But before we

talk about two main approaches to distributed memory communication[20], an important question of all communication is the speed at which data gets from one end to the other.

2.2.1 Communication speed

Two aspects affect communication speed, latency and bandwidth. Latency is time from receiving the instruction, to retrieving the data, to return of the value. It varies by the amount and dispersion of data in the memory and physical interference from other sources, e.g. magnets. It can be reduced by increasing the frequency of the memory. Bandwidth is further discussed in section 2.5.3.

2.2.2 Collective communication

As the name suggests, the base aspect of collective communication is receiving or sending data from one or multiple sources to all other sources. As this communication requires the participation of multiple sources, to prevent losses in data, it also acts as an synchronization point between nodes/processes.

2.2.3 Point-to-point communication

Second type of communication is used to share data between two specific processes. It is important for patterned and irregular communication, e.g. when each process routinely swaps regions of data with specific other processors between calculation steps, or master-slave architecture in which the master sends new data to a slave whenever the previous task is completed.

2.3 Load balancing

Load balancing improves the distribution of workloads across multiple resources, such as computers, clusters, network links, CPUs etc. It aims to optimize resource use, minimize response time, avoid overload of any single resource but most importantly maximize throughput. Thanks to using multiple components with load balancing instead of a single one, we may increase reliability and availability. When we talk about scheduling algorithms, we mean persistence of computation and memory usage. Further we will discuss the different load balancing APIs on CPUs and GPUs.

2.3.1 Multi-processing CPU APIs

As discussed previously, multi-processing depends on whether the load is divided within a single node or into multiple nodes withing the system. Different APIs were created for this

purpose. For our needs we will talk about OpenMP and MPI.

OpenMP

OpenMp is an API supporting multi-platform shared memory multiprocessing programming in C, C++ and Fortran, instruction set architectures and operating systems (e.g. Windows, macOS, Linux). It consinsts of a set of compiler directives, library routines, and environment variables that influence run-time behavior[18].
Managed by the nonprofit technology consortium OpenMP Architecture Review Board, which includes companies like AMD, Intel, Nvidia, HP, Texas instruments, and more.
It is an implementation of multi-threading, where a master thread forks into a number of slave threads and tasks are divided among them. Threads then run concurrently, while threads are allocated to different processors. Each thread has an id attached to it, through which the programmer distinguishes between the executions. After the execution of the parallelized code, it joins back into the master thread, which continues with the execution of the program. More in [18]. An realistic expectation is up to N-times speedup on N processor platform. However this is rarely the case as a large part of the program may not be parallelized, memory bandwidth does not scale up N times and the need for the process to wait until the data it depends on are computed.

MPI

Message passing interface is a standardized and portable message-passing standard. It defines syntax and library routines for writing portable message-passing programs in C, C++, and Fortran. Multiple implementations of MPI are available, many of which are open-source and run on multiple operating systems (e.g. Open MPI on Windows, Linux, macOS). MPI implementations consists of specific set of routines that are directly callable from C, C++ and more. One of the advantages of MPI over other message-passing libraries is its portability and speed.
The interface provides essential virtual topology, communication functionality between processes and synchronization. Programs utilizing MPI always work with processes, that are commonly referred to by programmers as processors. For maximum performance, a single process is assigned to each CPU/core (depending on the amount of cores a certain CPU has). MPI also specifies thread safe interface. More info in [27].

2.3.2 Multi-processing GPU APIs

For multi-processing on a GPU multiple APIs can be used. Some are focused mainly on 3D graphics such as Vulkan, Mantle, DirectX, OpenGL and more. Others like CUDA and

OpenCL are focused on allowing software developers to do general purpose processing on GPUs. These are the APIs we will talk about.

CUDA

CUDA is a parallel computing architecture from NVIDIA. As said before it allows software developers to do general purpose processing on GPUs that are CUDA-enabled. This API has been designed to work with programming languages such as C, C++ and Fortran. CUDA also supports working with other programming frameworks such as OpenACC and OpenCL. CUDA provides both low level API and a higher level API. It contains multiple libraries specified for certain tasks, e.g. Random number generation library, fast Fourier transform library, graph analytics library, basic linear algebra subroutines library and more. Advantages of doing general purpose programming with CUDA are:

- Scattered reads
- Unified virtual memory
- Unified memory
- Shared memory - fast shared memory region that can be shared among threads without communication
- Full support for integer and bit wise operations
- Support for multiple operating systems (Windows, Linux, macOS)

These advantages decrease the learning curve when programming on GPUs, reduce the need for constant data passing between threads when parallelizing code and increase the effectiveness of code. Unfortunately disadvantages include:

- CUDA-enabled GPUs are only available from NVIDIA
- Copying between host and device may cause a performance hit due to the system bus bandwidth and latency
- all newer versions of CUDA Source code is now processed according to C++ syntax rules, it is therefore possible that old C-style CUDA source code will either fail to compile or will not behave as originally intended.
- Interoperability with rendering languages such as OpenGL is one way only, with OpenGL having access to registered CUDA memory but CUDA not having access to OpenGL memory.

cuBLAS

cuBLAS, acronym for basic linear algebra subroutines, is fast GPU accelerated implementation of the standard basic linear algebra. It consists of basic functions that perform scalar and vector based operations, matrix-vector operations and matrix-matrix operations. Further information in [6].
In our testing we will use:

- cublasSgemm - matrix multiplication
- cublasSgemv - matrix vector multiplication

OpenCL

Open computing language is a framework for writing programs that execute across platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors, field-programmable gate arrays and other processors. It is open, royalty-free standard for cross-platform parallel programming. One of the biggest advantages of OpenCL is its portability between platforms, this advantage comes at a cost of performance. Without the direct effort of a programmer the code won't run optimally on all platforms without making changes. Another advantage is of OpenCL being standard, meaning it will pass of the information entirely, using the GPU as a separate general purpose peer processor.

2.4 Memory limitations

One restriction for all algorithms is the data they operate with, when the need for precision is high, the amount of memory that is needed increases. While a processor can take hours, days even months to process the data it is only a "tube" through which the data passes. This data has to be stored somewhere and when the storage runs out of space there can be no more data. This limitation is a problem all programmers face. Further we will go over differences between the memory limitations when programming on a CPU and GPU as well as techniques that allow to compress data into smaller chunks.

2.4.1 Programming on CPU

When programming on CPU the overall limits are theoretically as big as the storage space on the computer. Most operating systems have safety measures to continue working when the system memory limit is reached. Mainly because without any system memory the computer would not be able to work properly. The safety measure when applied starts to

write the data that would normally be written to RAM, to storage devices. Storage memory has more than 10 times lower bandwidth than RAM.

In theory this works, in practice when the memory limit is reached and the safety measure is applied, the calculation speed is slowed more than only by the bandwidth gap. Operating system has to manage writing, reading and replacing the system memory, and storage memory it has allocated for this reason and the computer becomes almost impossible to use until the computation is finished.

2.4.2 Programming on GPU

Due to specific use of graphic cards in regards to CPU, the memory limitations while programming on GPU are much bigger than on CPU. One reason, why the memory limitations are higher, is cost. Memory for GPUs is pricier due to the fact that it cannot be simply replaced or added when more memory is needed. When memory is needed the whole graphics card has to be replaced which does not mean that the graphics card with more memory will also be more powerful, fortunately that is usually the case.

Generally the GPU can process more data than the CPU (when the code is optimized to work on the GPU). Although this only applies when the problem does not have very big memory requirements which cannot be bypassed. One technique of bypassing the GPU memory limitation is use of streams when working with CUDA (section 3.5.2).

To better visualize the limitation, e.g. Nvidia GTX 960M with 4GB of memory is able to work with 4096 million bits, that means 512 million numbers of type "double", in perspective this amount of memory is able to calculate matrix multiplication of 2 matrices of about 15000 x 16000 when both matrices are copied over to the GPU. One way to increase the ability to use bigger arrays is e.g. use of hierarchical matrices[14].

Another memory limitation is copying data from RAM to GPU memory(see section 2.5.3). Software-wise this limitation can be easily determined by 3rd party software such as CUDA-Z.

2.5 Comparison of computing performance

Computing performance is very specific fact that talks about how fast can a component process and calculate specified code. Though this seems as an easy task to determine, it is connected to multiple factors that are associated with it. There are four major factors, architecture, frequency, number of cores and memory bandwidth over which we will talk about[4]. Factors like power delivery problems, not enough storage space and similar limitations will be ignored.

2.5.1 Architecture

The goal of computer architectures is to trade of standards, power versus performance (power needed for 1 GFLOP), cost, memory capacity, latency and throughput. One way to measure computers performance is in IPC (instructions per cycle). This shows the efficiency of the architecture at any clock frequency. While older processors had IPC count as low as 0.1, new processors easily reach near 1 and higher. Therefore a good indicator of a processor performance is to look at combination of multiple factors, one of which is IPC.

2.5.2 Frequency and number of cores

Frequency or clock rate refers to the frequency at which a chip like CPU, or core, is running. It is measured in clock cycles per second. Basic unit of frequency is hertz(Hz), the speed of today's processors is commonly advertised in gigahertz(GHz). This metric is most useful when comparing processors from the same architecture.
While multi-core processors means that a single computing component has more than one independent processing unit called core, which reads and executes programs. Multiple cores allow for higher level of parallelism, therefore faster computations. Having more cores doesn't not mean directly better performance, because not all software can take advantage of higher number of cores.

2.5.3 Bandwidth

On its own bandwidth can mean more than one thing, a range withing a band of frequencies or wavelengths or in our case amount of data that can be transmitted in a fixed amount of time. In digital devices bandwidth is usually expressed in bits or bytes per second. For CPUs and GPUs we talk about memory bandwidth. Knowing how much data can flow through a CPU or GPU is a useful information but this is known as absolute memory bandwidth. When we look closer the picture is a little different.

For CPUs we take one of the newer architectures of processors (7th gen.), the Intel Core i7-7700k. This CPU has about 50GB/s bandwidth with its 4 cores. That is about 12.5GB/s per core. With core frequency of 4.2GHz that comes just under 3 bytes per cycle of memory bandwidth. The modern architecture allows the CPU core to execute multiple (up to three) 256-bit SIMD(simple instruction, multiple data) operations in one cycle. If we treat the CPU like a GPU and divide this 256-bit SIMD into 32-bit vectors and treat them as separate threads, we get 24 instructions executed per cycle and memory bandwidth of about 0.125 bytes per cycle per simulated thread which adds up to one byte every 8 instructions[15].

This gets even worse with GPUs, if we take a modern high-end GPU, the Nvidia GTX 1080Ti with bandwidth of 484GB/s as an example it seems that there is no comparison between the CPU and the GPU, but the story is completely different. Even though with core frequency of 1.48 Ghz, 327 bytes/cycle for the whole GPU, the bandwidth is higher, it has 28 Shading Multiprocessors (comparable to CPU cores) and 3584 CUDA cores we get about 11.7 bytes/cycle per SM (Shading multiprocessor), that is 4x more what the 7700k gets, but each SM has 128 CUDA cores corresponding to a thread. That means we get one byte every 11 instructions meaning the GPU is even worse in than the 7700k. Fortunately it has a big advantage and that is the number of compute units on the GPU, meaning it can handle multiple small tasks at once which results in the much higher bandwidth[15].

One more thing to take into consideration when running a computation on a GPU is copying the data over to the GPU and back. That means that the task has to be large enough, that the data allocation and copy process is significantly lower that the computation of the said problem itself. E.g. for matrix multiplication with I7 - 6700HQ and Nvidia GTX 960M and matrix size of 10x10 the CPU is able to multiply two matrices in 0.000001 seconds while the data copy to the GPU and back along with the calculation takes 0.000769 seconds which is mroe than 700-times slower than the CPU.

2.5.4 FLOPS

When it comes to computing performance it is possible and very practical to represent component performance by one number. This number is called FLOPS (floating point operations per second). FLOPS were first introduced by Frank H. McMahon to measure and compare computing performance of supercomputers.

This value can be calculated by a simple equation:

$$FLOPS = \frac{cores}{sockets} * \frac{cycles}{sec} * \frac{FLOPs}{cycle} \tag{2.1}$$

Where the number of FLOPs per cycle is determined by the manufacturer. It is not wise to build and work with particular components only by the amount of FLOPS. Compatibility and especially price is a big factor about which we will talk next.

2.5.5 Price/performance comparison

When it comes to comparing of raw potential of a processing unit, multiple factors have to be taken into consideration. Power consumption, raw compute performance (given by equation in section 2.5.4), raw compute performance per watt, memory bandwidth and the benefits of adopting CPU or GPU computing approach.

As time passed the raw compute performance difference between CPUs and GPUs became evident as seen in Figure 2.2 and 2.3. In our comparisons we will look over these devices:

- Intel Xeon CPUs - professional / workstation grade CPUs
- Nvidia GeForce GPUs - consumer grade GPUs
- Nvidia Tesla GPUs - professional / workstation grade GPUs
- AMD Radeon GPUs - consumer grade (up to year 2013), professional / workstation from 2014-2016
- Intel Xeon Phis - x86 CPU based add-on card designed for massive parallelism and vectorization

between years 2009 and 2016.

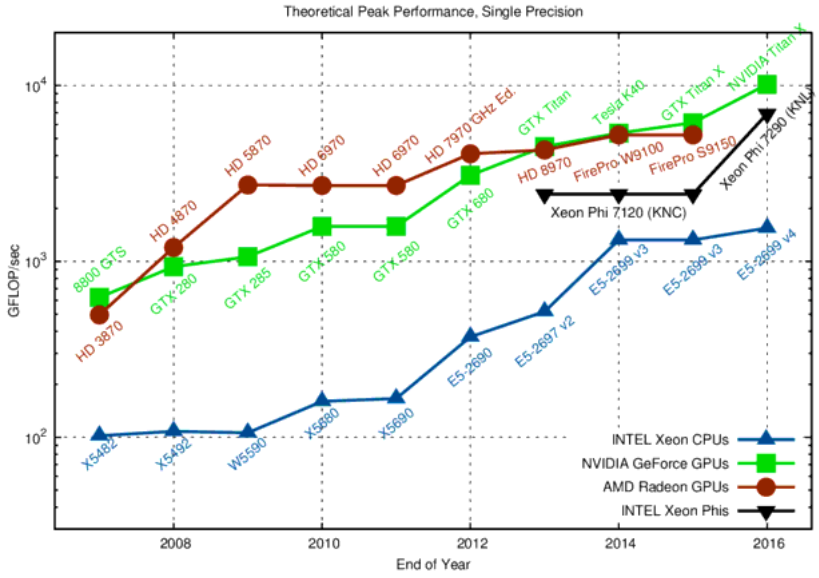

Figure 2.2: Theoretical peak performance comparison over time, single precision $(GFLOP = 10^9 FLOPS)$ [5]

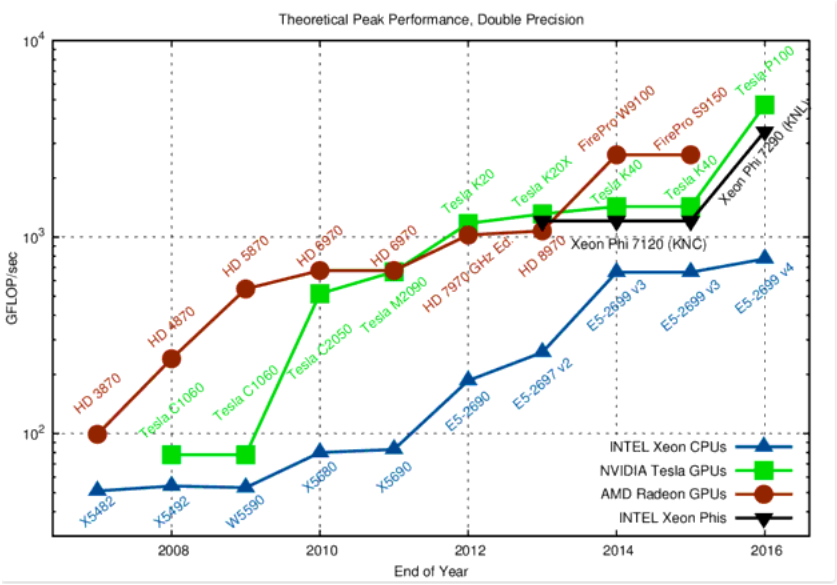

Figure 2.3: Theoretical peak performance comparison over time, double precision $(GFLOP = 10^9 FLOPS)$ [5]

This performance increase wasn't without its drawbacks. As the performance rose, so did the energy consumption.

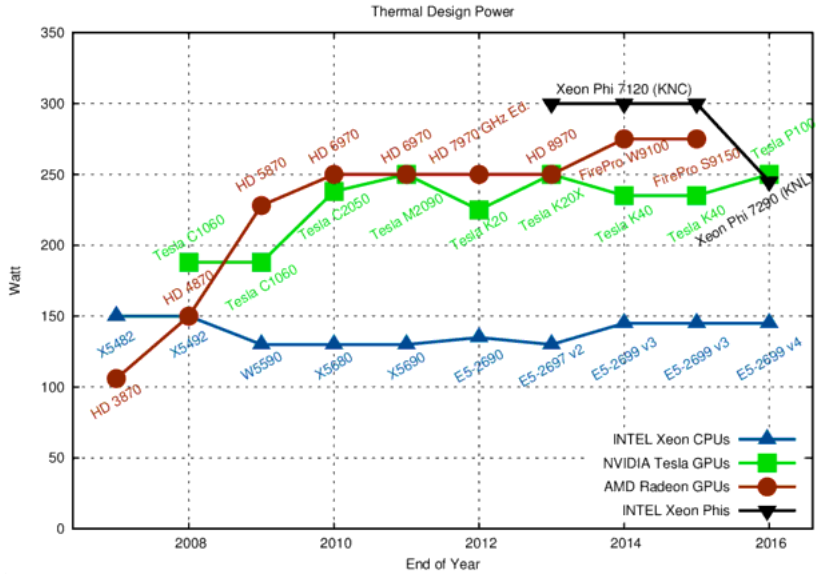

Figure 2.4: Thermal design power consumption comparison over time [5]

Similarly to theoretical peak performance, the GPUs are better optimized per watt of energy than the CPUs.

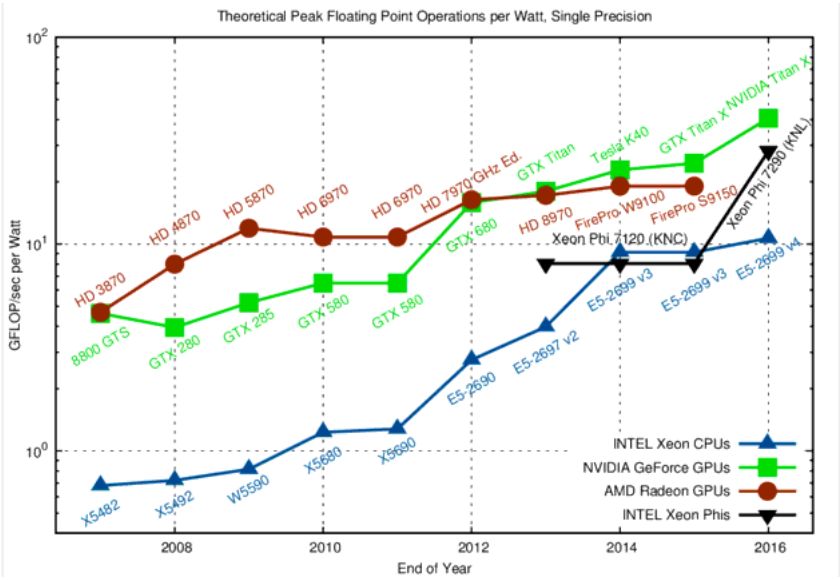

Figure 2.5: Theoretical peak Floating point operations per watt compatison over time, single precision [5]

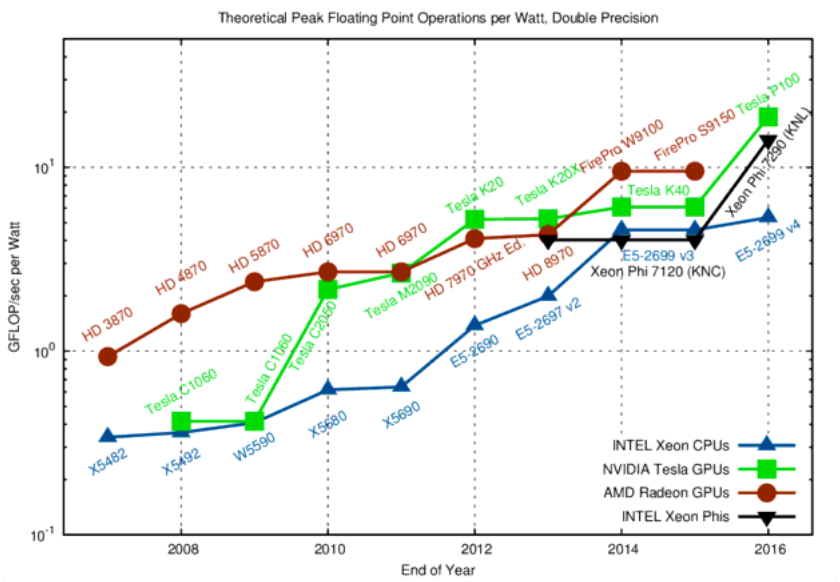

Figure 2.6: Theoretical peak Floating point operations per watt compatison over time, double precision [5]

In order to use all arithmetic units efficiently, the respective data must be available in registers. Unless the data is already available to be used in registers or cache, they need to loaded and written back at some point. These loads and stores are a bottleneck that slows down many operations. Therefore memory bandwidth is one of the things to take into consideration.

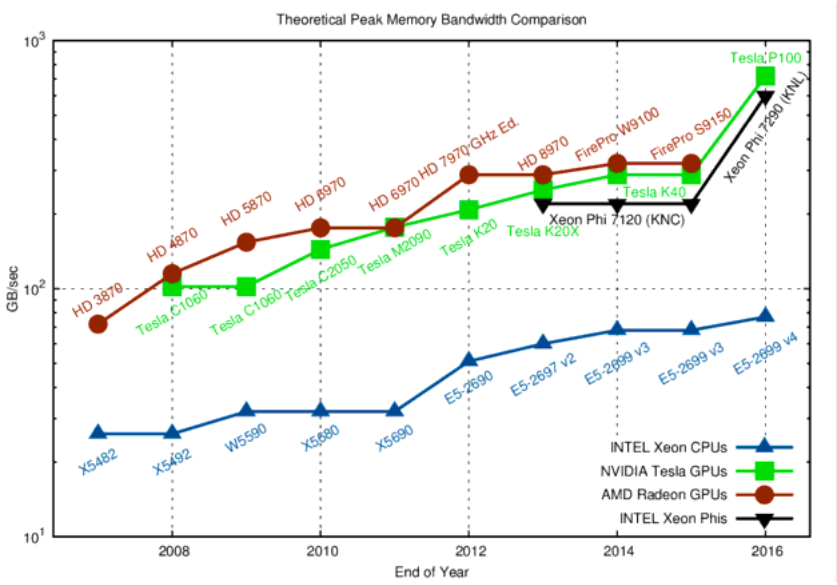

Figure 2.7: Theoretical peak memory bandwidth comparison over time [5]

And at last the biggest consideration is the price. As an example we will take the most powerful CPU + the most powerful GPU for single and double precision from our comparisons.

Single precision

While the Xeon E5-2699v4 is priced at 4115$ the Nvidia Titan X is priced at 1100$ and offers about 10-times higher raw performance than the Xeon while consuming only about 150 watts more.

Double precision

When it comes to double precision computation the consumer grade cards are not match a to the professionally oriented cards which are priced at much higher premiums. This causes that at double precision computations, while the Xeon is priced at 4115$ the Tesla P100 is priced at around 6000$. Even though the price is 1.5-times higher, the theoretical performance increase is almost 10-times higher.

Another important part of the comparison is the hardware/space limitation when it comes to motherboards. While server motherboards can have 4 sockets with one CPU in each of them. Depending on the CPU and the number of memory lanes it supports, one motherboard with one CPU can have up to 10 GPUs. For comparison a server where each server computer has 4 CPUs and has space for 9 of these server computers, a server that uses mainly GPUs can have 3 computers where every computer has 10 GPUs which take the same amount of space as the CPU based system, while offering much higher performance. This approach while more expensive since there are 30 GPUs + 3 CPUs, which, when taking the best of the

best, adds up to 192345$ (in processing unit costs only), while the CPU based system with 36 CPUS, adds up to 148140$ (in processing unit costs only), is space effective compared to performance per m^3 of space.

2.5.6 Conclusion

When we take all the aspect that were presented in this chapter into consideration, the final conclusion is, that using a combination of one CPU plus multiple GPUs can theoretically extremely increase the performance potential of a system (by up to 10 or more times) and decrease the need for another computer connected into a grid. This advantage comes at a price of learning new techniques of working with GPUs oriented libraries and closely managing the memory.

Chapter 3

Implementation

In this chapter we will go over the process and troubles we faced during implementation of our algorithms, the programming language we used, environment we worked in and things to look out for when working with CUDA and implementing algorithms along practical examples.

3.1 Language C++

From many different programming languages we chose C++, because of its ability to create classes, libraries for further use and compatibility with CUDA, OpenMP and MPI which are tied to use of C, C++, and a few of other programming languages. Biggest advantages of C++ are it being an object oriented language, that can be compiled and used irrespective of operating system as well as Hardware. To our purpose it also provides performance and memory efficiency as well as good re-usability of code. Despite it being an high-level programming language, it is useful for the low level programming language and very efficient for general purpose.

3.2 Development environment

Experiments were made with Visual Studio 2015 Community, which is free to use for everyone, as their development environment. With easy access to all the projects in one solution, simple debugging and user friendly library management Visual Studio 2015 Community was the choice while working with C and C++. Not only it has a great documentation and support, but also includes Intellisense, the so called intelligent sense, that automatically completes the command that was being written and increases the speed of writing the code. Visual Studio has pre-loaded project templates to which one can program directly without creating additional files when starting a project from scratch. It allows for direct memory usage monitoring. This feature is an essential tool when working with GPUs and big data

sets, because it gives the programmer instant idea of the data set memory size without a direct calculation.
Part of working with Visual Studio and CUDA together is installing the CUDA toolkit which automatically integrates CUDA support into Visual Studio.

3.3 Unit testing

Unit testing is a part of software testing where individual components of a software are tested. In our case individual parts of code, such as matrix addition, image transformation, etc. The purpose is to validate each component, so that it performs as designed. In procedural programming, a unit can be an individual program, function, etc. In object-oriented programming it is a method.
For this purpose we created two libraries, one with methods running on a CPU and the other with methods running on a GPU. Each library contains the same methods, with the same inputs and the same outputs.

3.4 Differentiation between programming on CPU/GPU

Writing a program using CUDA and C++ is a little bit more complicated than writing a simple code in C++ only as seen in Figure 3.2. In this section we will go over the aspects that are needed for a code to be able to run on a GPU using CUDA. Further information about things mentioned in this section can be found in [6] or directly from NVIDIA [7]. In further reading we will refer to the CPU as host and GPU as device.

3.4.1 Memory allocation and copying on device

Same as programming without a GPU, the memory has to be allocated before it can be used. At first the memory on the device has to be allocated, after which the data needed for the calculation must be copied to the device. At this point the calculation on the GPU can be run, after which the data is copied back to the host and freed on the device to not cause any memory leeks.

```
// Allocation of host specific array
        float * hostMap;
        cudaHostAlloc((void **)&hostMap,
           sizeof(float)*width*height, cudaHostAllocDefault);
// Allocating pointers on host to be passed to device
        float * deviceMap;
```

```
// Allocating GPU memory
        cudaMalloc((void **)&deviceMap,
           sizeof(float)*map->height*map->width);
// Copy memory to the GPU
        cudaMemcpy(deviceMap, data, sizeof(float)*height*width,
           cudaMemcpyHostToDevice);
        ...
        Calculate new pixel values on the GPU
        ...
// Copy the results in GPU memory back to the CPU
        cudaMemcpy(hostMap, deviceMapNew,
           sizeof(float)*height*width, cudaMemcpyDeviceToHost);
// Free the GPU memory
        cudaFree(deviceMap);
        cudaFreeHost(hostMap);
```

Listing 3.1: Memory allocation on device

3.4.2 Kernel

Kernels are small functions defined in C with support for C++ features (in C++11):

- auto
- lambda functions
- range base for loops
- std::initializer_list
- variadic templates
- static_asserts
- constexpr
- rvalue references

and features not supported:

- thread_local
- standard std:: libraries

that, when called are executed in parallel N times by N different CUDA threads where N is the number of threads on the GPU. Kernel is differentiated from other code by using the `__global__` specifier and the number of CUDA threads, blocks and grids, that should execute the kernel is specified using the `<<<...>>>` execution configuration syntax, followed by the arguments needed for the task within the kernel to be executed. With CUDA new variables are introduced for the device, such as `__constant__` (constant for the device), `shared` (shared variables between blocks), `tex2D` (textures used in graphics to create visual appearance of a surface e.g. picture),`__device__` (functions defined to run on GPU, which are callable from kernel) etc. Each thread is given a unique thread ID, that can be called withing the kernel by `threadIdx` variable.

```
__global__ void myKernelFunction(kernel arguments)
{
        int indexOfThread = threadIdx.x;
}

int main()
{
        ...
        myKernelFunction <<<1,N>>> (kernel arguments) // N is the
           number of threads
        ...
}
```

Listing 3.2: Kernel implementation example

It is possible to call another kernel from within the kernel, called dynamic parallelism (up to 3x), that is supported from CUDA 3.5 and higher.

3.4.3 Blocks and grids

Let us a consider a GPU with 4 multiprocessing units within the main processing unit, from which each can run 768 threads, which is dependent on graphics card specification and can be found in CUDA profiler. That means at any given moment no more than $4 * 768$ threads will be really running in parallel. If we differentiate the tasks at hand into more than the amount of threads the GPU has available, they will be waiting until the threads before them are finished with their task. In CUDA, threads are organized into blocks, which are then executed by a multiprocessing unit. The threads of a block are indexed using 1D(x), 2D(x,y) or 3D(x,y,z) indexes, but in any case the dimensions must be $xyz <= 768$ (other restriction apply according to the device capabilities). If the data is bigger than $4 * 768$, you obviously need more than 4 blocks. These can be also indexed in 1D, 2D or 3D into grids. There is a

queue of blocks waiting to be run on GPU. In our case, since the GPU has 4 multiprocessing units, only 4 blocks are being executed simultaneously.
As an example, we will use processing of an image with dimensions 256 x 256. If we want one pixel $\{a_{i,j}\}$ with space coordinates (i, j) to be processed in one thread we will need $256 * 256$ threads. Lets consider using blocks of 64 threads each. Then we need $256 * 256/64 = 1024$ blocks. For easier manipulation we will organize the threads in 2D blocks 8×8,

```
dim3 threadsPerBlock(8,8);
```

and 2D grid of 32 x 32 blocks (the 1024 needed).

```
dim3 dimBlocks(imageWidth/threadsPerBlock.x,
   imageHeight/threadsPerBlock.y);
```

As the last part we have to tell the GPU how it should divide its resources through kernel parameters

```
kernelCall <<<dimBlocks,threadsPerBlock>>> (function parameters)
```

This means that there will be a queue of 1024 blocks, from which, each is waiting to be assigned to one multiprocessor to get its 64 threads executed.
As the threads are running simultaneously, only information they know is their index in the block, the block id and the block dimensions, therefore to get the global index of the pixel in picture that threads represent, it has to be calculated:

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

Correctly dividing threads into blocks and further into grids can severely increase or decrease the performance potential of a GPU.
From hardware perspective, a GPU consists of multiple streaming multiprocessors, which consist of CUDA cores. Connecting hardware and software perspective (Figure 3.1) means that one thread, is executed by one CUDA core, one block is executed by one streaming multiprocessor and one grid is executed by the whole GPU unit (e.g. graphics card).

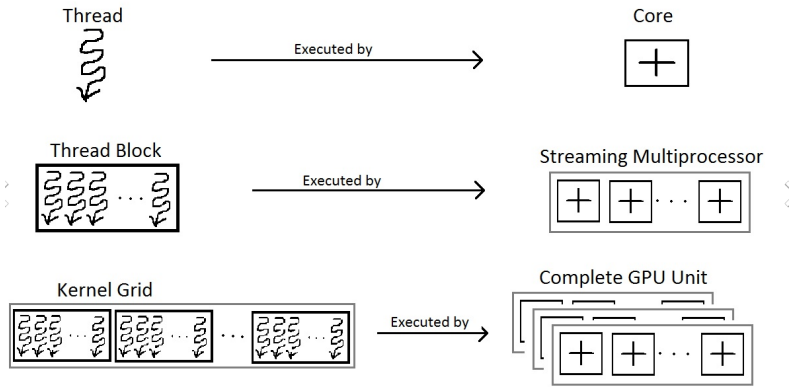

Figure 3.1: Correlation of a programmers perspective versus a hardware perspective [6]

Another thing to bear in mind are so called "warps", physical part of a GPU so called "warp scheduler" can schedule certain amounts of threads per warp. This means that the scheduler can give tasks only to a certain amount of threads per cycle. How warps affect performance can be seen in Table 4.2. In our case the warp size was 32. Information about warp sizes is available from the manufacturer for every graphics card. Knowing the number of threads in a warp and taking it into consideration becomes important when trying to maximize the performance of an algorithm.

3.4.4 Error handling

While debugging a code running on the host, our development environment provides an easy way to go through each line of code step by step, to see if any error has occurred. Unfortunately this capability is not available for device code, thus another approach has to be implemented to compensate for this. Every function in CUDA returns `cudaError_t` statement. This statement consists of information if the kernel were run successfully or not and the error message that may point the developer in the direction of the problem.
For this purpose we have written a simple function that evaluates this statement and either stops all calculations and returns the error or continues on.

```
void funcCheck(cudaError_t stmt)
{
        cudaError_t err = stmt;
        if (err != cudaSuccess)
        {
                printf("Got CUDA error ...  %s \n",
                   cudaGetErrorString(err));
                cudaDeviceReset();
                exit(-1);
        }
}
```

Listing 3.3: Cuda error checking function

3.4.5 Comparison of code on host vs. code on device

When we combine all of the above we can write an example and show the differences between code on running host and on device. Other than defining kernel, allocating memory and error handling, important command to use are:

- cudaSetDevice(0) - used to select on which graphics card the following code is supposed to be run
- cudaDeviceSynchronize() - needed to run after kernel launch, to guarantee the kernel to finish, before the application is allowed to continue
- cudaDeviceReset() - a good practice to call at the end of a program, to free all the memory on the device and stop all calculation running on it

```
#define N 1000
__global__ void timesTwo(int *a,
    int *b) {
 int i = blockIdx.x;
 if (i<N) \\ to ensure we are
    withing the range of our
    vector
 {
  b[i] = 2*a[i];
 }
}

int main() {
 int *hosta,
 int *hostb;
 cudaHostAlloc((void **)&hosta,
    N*sizeof(int),
    cudaHostAllocDefault);
 cudaHostAlloc((void **)&hostb,
    N*sizeof(int),
    cudaHostAllocDefault);
 int *devicea, *deviceb;
 cudaMalloc((void **)&devicea,
    N*sizeof(int));
 cudaMalloc((void **)&deviceb,
    N*sizeof(int));
 for (int i = 0; i<N; ++i)
 {
  hosta[i] = i;
 }
 cudaMemcpy(devicea, hosta,
    N*sizeof(int),
    cudaMemcpyHostToDevice);
 add<<<N, 1>>>(devicea, deviceb);
 cudaMemcpy(hostb, deviceb,
    N*sizeof(int),
    cudaMemcpyDeviceToHost);
 cudaFree(devicea);
 cudaFree(deviceb);
 return 0;
}
```

Listing 3.4: Sample code running on device

```
#define N 1000
int main() {
 int hosta[N], hostb[N];
 for (int i = 0; i<N; ++i) {
  hosta[i] = i;
 }
 for (int i = 0; i<N; ++i) {
  hostb[i] = 2*hosta[i];
 }
 return 0;
}
```

Listing 3.5: Sample code running on host

Figure 3.2: Example of a code on host and device with the same purpose and result

We can see that simple code that calculates $b = 2 * a$ where a, b are of dimension $N \times 1$ can be written either on the device or on the host. The code on the host is much easier to write, but does not offer the parallel capabilities of the GPU. Please bear in mind that this example is too simple, to see any performance increase, rather the opposite is happening, because the data allocation and transfer from host to device and back takes more time than the calculation itself on the CPU as is further described in section **??**. The performance increase is evident on larger problems as seen in the next chapter.

3.5 Data passing and performance decrease

There are three approaches we will go over when passing data to and from the GPU, their benefits over other methods and their drawbacks. Unfortunately since we have to move the memory whether we want to or not when doing general purpose computing on a GPU, we often encounter performance hits, which cause our code to run longer than on a CPU. This is due to the time it takes to copy data from memory that is available to the CPU to memory that is available to the GPU.

3.5.1 GPU memory types

CUDA-enabled devices have different types of memory spaces. Each type of memory on the device has its advantages and disadvantages. Incorrectly using available memory can cause performance decreases. In terms of speed and types they are:

- Register file - available only to the thread that wrote it and lasts only for the lifetime of that thread
- Shared memory - visible to all threads within a block and lasts for the duration of the block (used for communication between threads)
- Constant memory - read-only memory used for data that wont change during the execution
- Texture memory - read-only memory specially designed for reading physically adjacent memory (e.g. Linear filtration with heat transfer, where every new pixel is calculated from nearby pixels)
- Local and Global memory - local memory has the same rules as register memory but performs much slower, while global memory is visible to all threads with the application and lasts for the duration of the host allocation

3.5.2 Direct copy

The most basic way of passing data to the device from host is creating a hard copy and passing it as a whole at one time (Listing 3.4.1). This means, that the device has to wait until all the data is transferred to continue with code execution. This is known as serial data passing.

Figure 3.3: Serial data passing

3.5.3 Data streaming and concurrency

Concurrency of CUDA, is the ability to perform multiple CUDA operations simultaneously. This is another type of parallelism, specifically different components of a system working on one task simultaneously. Four tasks can run in parallel:

- CUDA kernel
- HostToDevice copy
- DeviceToHost copy
- operations on the CPU

Concurrent data passing is done through streams. Stream is a sequence of operations that executes in issue-order on the GPU. The simplest way of concurrency is 2-way concurrency, which can speed the process up to 2x in comparison to Figure 3.3.

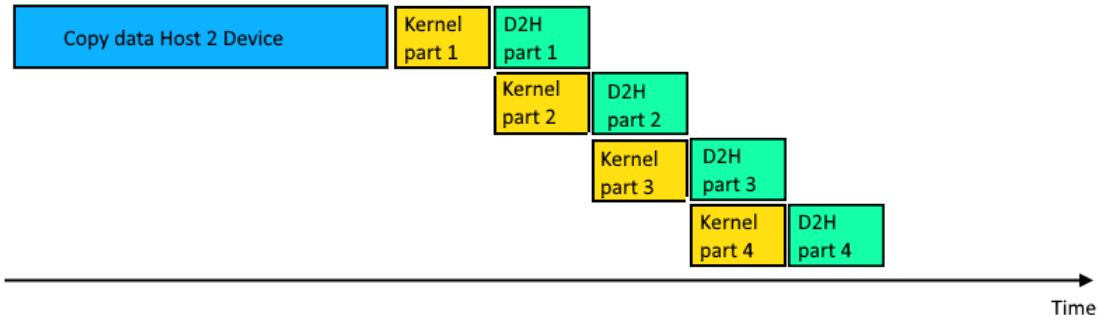

Figure 3.4: 2-way concurrency

The idea behind 2-way concurrency as seen in Figure 3.4, is running kernel alongside the data transfer from device to host. The data that has been already calculated is sent from

device to host. This approach can be further improved by also concurrently sending data from host to device in 3-way concurrency (Figure 3.5), which can speed up the process up to 3x in comparison to serial approach.

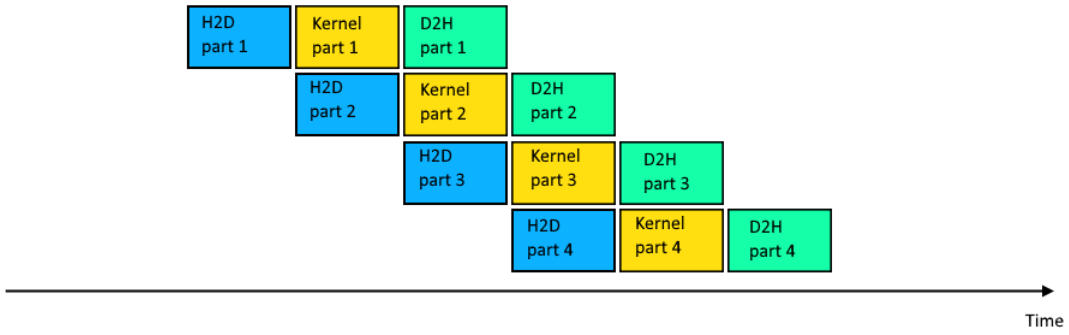

Figure 3.5: 3-way concurrency

Until this point the CPU has been waiting for the calculation to finish on the GPU. For further improvement we can appoint one part of the calculation to run simultaneously on the CPU to achieve 4-way concurrency as seen in Figure 3.6, which can speed up the process up to 3x+ in comparison to serial approach.

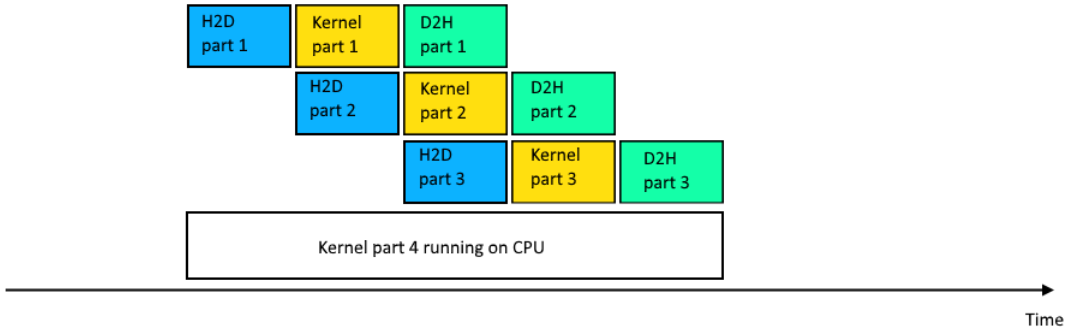

Figure 3.6: 4-way concurrency

4-way concurrency can be further broadened to 4+ way concurrency by further division of the kernel runs. For examples and tutorials about dividing your code can be found in [13].

3.5.4 Unified memory

Unified memory is a single memory address space accessible from any processor in a system. This technology allows applications to allocate data that can be used either on CPUs or GPUs (Figure 3.7). Allocating this memory is done by replacing `malloc()` or `new` with `cudaMallocManaged()`. Cuda system software and the hardware takes care of migrating memory pages to the memory of the accessing processor. Important point is that older GPUs based on the Kepler and Maxwell architecture also support unified memory, but

in a more limited form. Newer Pascal GPU architecture is the first with hardware support for virtual memory page faulting and migration, via its Page Migration Engine.
This means that on pre-Pascal GPUs, all data must be resident on the GPU when the kernel is running, therefore the performance increase next to direct copy approach is negligible and the GPU cannot use more data than its own memory can hold. Main goal of unified memory is improving ease of GPU programming[21]. More in [26].

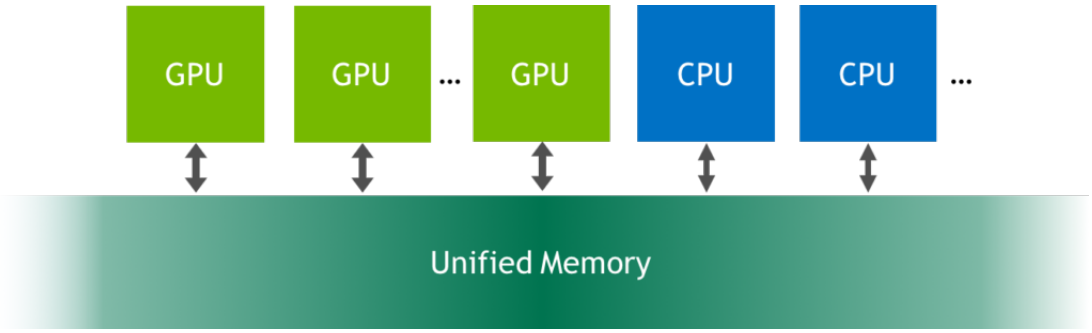

Figure 3.7: Unified memory representation

Another way of allocating memory that is accessible to the device is `cudaHostAlloc`. Which allocates size bytes of host memory that is page-locked. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than page-able memory obtained with functions such as `malloc()`. Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

Chapter 4

Experiments

To test the improvements in computational speed on a GPU we have concluded few tests that should give the reader broadened picture about when it is good to incorporate GPU into computing and when it is better not to risk the time loss of copying data from one processing unit to the other (CPU -> GPU). All experiments were done in Release mode without debugging. All calculation times of CUDA implementations are including data transfer to and from GPU and GPU array allocations. For most of the algorithms either on the CPU or a GPU, unit testing was done to ensure proper functionality, for algorithms that unit testing was not implemented the comparison of result between CPU and GPU was done to ensure correct results.

Specifications of used GPU:

- Nvidia GTX 960m 4GB
- 640 cuda cores
- 1096 Mhz processor clock speed
- 588 Mhz graphics clock speed
- 128-bit memory interface width
- 80GB/s memory bandwidth
- 1024 - max_threads_per_block
- 1024 - max_block_dim_X
- 1024 - max_block_dim_Y
- 64 - max_block_dim_Z

- 2048 - max_threads_per_multiprocessor
- $2147483647 \times 65536 \times 65536$ - max_num_of_blocks_in_grid $(x \times y \times z)$

These specifications can be retrieved from Nsight - system info tool withing Visual Studio.

4.1 Matrix operations

Working with matrices is the basics to most iterative methods. Therefore basic matrix operations are a target of continuous optimization and computing speed improvement. These operations are easily parallelised and thus can be implemented on a GPU to further decrease the computational time. In the next sections we will talk only about matrix multiplication, matrix additions and why some operations are good to be parallelised and some are not.

4.1.1 Matrix additions

The most basic matrix operation, matrix addition, can be easily parallelised on a GPU since the cores don't need to know anything about each other to calculate the solution.

$$\begin{aligned} A + B &= C \\ A &\in R^{m \times n}, \\ B &\in R^{m \times n}, \\ C &\in R^{m \times n} \end{aligned} \tag{4.1}$$

In computing this means adding first element of matrix A with first element of matrix B and so on to get the full matrix C. Therefore the number of additions that have to be done is $n \times m$ while they are each independent of one another. This fact allows it to be easily parallelised into $n \times m$ operations.

n x m	1-core CPU	cuda custom kernel
10 x 10	0.000001	0.000905
500 x 500	0.000181	0.002071
5000 x 5000	0.023387	0.083861
10000 x 10000	0.096804	0.328139

Table 4.1: Computational time matrix addition (seconds) using different implementation methods and different matrix dimensions

As seen in the Table 4.1, the drawback of copying data from one processing unit to the other causes the computational time to be longer than without the need of data transfer.

In simple operations where the complexity of the operation is not bigger than O(N), we don't see improvement in computation on GPU. The computational time can be affected by correctly choosing the amount of threads in a block as seen in Table 4.2 with constant matrix size of 10000 x 10000 and the times are an average of 10 runtimes.

Number of blocks	Block size	cuda custom kernel
100000000	1	0.9629981
12500000	8	0.3698929
6250000	16	0.3357488
3125000	32	0.3189522
1562500	64	0.3148245
781250	128	0.315618
390625	256	0.3154788
195313	512	0.3157323
97657	1024	0.3148105

Table 4.2: Computational time matrix addition (seconds) using different block sizes

4.1.2 Matrix-vector multiplication

Matrix-vector multiplication is a great example of an algorithm, that shows where the GPU is starting to take advantage of its more cores.

$$\begin{aligned} A.b &= c \\ A &\in R^{n \times m}, \\ b &\in R^{n}, \\ c &\in R^{n} \end{aligned} \tag{4.2}$$

Which means that each element c_i of the newly calculated vector c is a dot product of the vector b and row A_i of A. This fact allows the division of this problem into n threads, where every thread calculates one element c_i of c.

$A^{nxm}.b^{n}$	1-core CPU	cuda custom kernel
$A^{10x10}.b^{10}$	0.000001	0.001490
$A^{500x500}.b^{500}$	0.000416	0.004817
$A^{5000x5000}.b^{5000}$	0.039974	0.035103
$A^{10000x10000}.b^{10000}$	0.148150	0.126872

Table 4.3: Computational time matrix-vector multiplication (seconds) using different implementation methods and different matrix dimensions

As we can see from Table 4.3, the bigger the matrix and vector is, the GPU is slowly starting to take advantage of its cores. Unfortunately this problem is still small enough, that the advantage over CPU is not very significant.

4.1.3 Matrix multiplication

Many algorithms suffer with long computational times because of matrix multiplications, where for matrix $m \times n$ you need $m * n * n$ operations.

$$\begin{aligned} A.B &= C \\ A &\in R^{n \times m}, \\ B &\in R^{n \times m}, \\ C &\in R^{n \times m} \end{aligned} \tag{4.3}$$

Thanks to the fact that this operation can be divided into multiple, where each thread needs only one row of matrix A and one column of matrix B to calculate an element in matrix C, which are independent of each other, makes this a perfect example where GPU computation is much faster than on the CPU.

n x m	1-core CPU	4-core MPI	cuda custom kernel	cublas
10 x 10	0.000001	0.000001	0.000769	0.267797
100 x 100	0.001681	0.0004952	0.001808	0.288092
500 x 500	0.194761	0.056821	0.005838	0.276858
1000 x 1000	1.777029	0.48565	0.023176	0.264300
5000 x 5000	1306.742310	357.2365	1.853558	0.533037

Table 4.4: Computational time matrix multiplication (seconds) using different implementation methods and different matrix dimensions

The times tell us, that for very small matrices, the time to initialize and transfer the data onto the GPU, takes much longer than the computation itself. Fortunately when the dimensions grow past a certain point, the data transfer is not the major part of the computation anymore, therefore it is effective to solve this problem on the GPU. It is very important to consider the amount of memory the GPU has, when the size of the data is higher than the memory of the GPU, each matrix has to be divided into smaller submatrices and computed alone.
We have chosen matrix multiplication to demonstrate the differences between using sequential code (Listing 4.5), MPI (Listing 4.5), CUDA (Listing 4.5) and cuBLAS (Listing 4.5).

4.2 Iterative methods

The term "iterative method" refers to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step. Usually they are divided into two categories, stationary methods that are much older, simpler to understand and implement but less effective and nonstationary methods, harder to understand but can be highly effective. We will discuss nonstationary methods based on the idea of sequences of orthogonal vectors.

For an iterative method to be effective it cannot improve its approximation forever. Therefore there must be a stopping criterion, tolerance at which the approximation is within a set error. The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix. This leads to usually transformation of the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a preconditioner. Not only does a good preconditioned improves the convergence of the iterative method but without a preconditioner the iterative method may even fail to converge.

Examples of stationary iterative methods are Jacobi, Gauss-Seidel, SOR, SSOR, etc. For nonstationary methods Conjugate Gradient, Minimum residual method, Genera minimal residual method and more. More information in [25]. We have chosen to implement and focus on Biconjugate Gradient Stabilized (BiCGStab) method.

4.2.1 Preconditioned BiConjugate Gradient Stabilized method (BiCGStab)

This method was developed to solve nonsymmentric linear systems while avoiding the often irregular convergence patterns on Confugate Gradient Squared method (see [1]). BiCGStab computes $i- > Q_i(A)P_i(A)r^0$ where Q_i is an ith degree polynomial describing a steepest descent update. To reduce and save time by a few operations this method uses two stopping criteria. At the start if the method has already converged at the first test on the norm of s, to continue updating would be irrelevant.

Pseudo-code for the BiCGStab with preconditioner M is given in the next Figure.

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}$ (for example, $\tilde{r} = r^{(0)}$)
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$
    $p^{(i)} = r^{(i-1)}$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1}v^{(i-1)})$
  endif
  solve $M\hat{p} = p^{(i)}$
  $v^{(i)} = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T v^{(i)}$
  $s = r^{(i-1)} - \alpha_i v^{(i)}$
  check norm of s; if small enough: set $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p}$ and stop
  solve $M\hat{s} = s$
  $t = A\hat{s}$
  $\omega_i = t^T s / t^T t$
  $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
  $r^{(i)} = s - \omega_i t$
  check convergence; continue if necessary
  for continuation it is necessary that $\omega_i \neq 0$
end

Figure 4.1: The Preconditioned BiConjugate Gradient Stabilized Method

Challenges

As seen in the Figure 4.1 to successfully use the BiCGStab method it is required to do two matrix-vector products and four inner products. Because of the computational complexity of O(n m) when working with matrix-vector products it would be highly inefficient to calculate them sequentially.

Possible implementations

One approach to solve the BiCGStab method is to completely parallelise it on the CPU. Not only is this approach very fast, but also easy to implement. But the fact that matrix operations like matrix-vector products, matrix multiplication etc. have a very high ability to divide it into smaller tasks. CPU is very fast for tasks that can be done only in sequence, but it lacks the possibility to divide an algorithm into thousands of smaller tasks computed in parallel on its own. Fortunately that is the inner workings of a GPU, f.e. matrix-vector products can be divided into at least m-tasks (m is the number of rows) running simultaneously.

Our implementation

As talked in section 2.5.3 it is important to use the GPU/CPU to its full potential and for tasks that have small data-sets it is counterproductive to copy the data over to the GPU and copy it back to for the algorithm to continue. A balance of CPU and GPU usage must be found. Operations like sums, dot products, number multiplications are implemented on the CPU and more time demanding tasks e.g. matrix-vector products, matrix multiplications are done on the GPU.

Dataset	1-core CPU	4-core CPU MPI	4-core CPU OpenMp	cuda custom kernel	cuda custom kernel opt
902	0.3145	0.152	0.234	0.435785	0.409
1298	1.062	0.378	0.625	0.853864	0.871107
3602	54.203	17.538	25.063	16.7105	15.064

Table 4.5: Computational time comparison (seconds) using different implementation methods and different data-set sizes

After running BiCGStab on small data-sets we can see that the speed difference is not that significant. Though after running the algorithm on bigger data-sets we have gotten a measurable improvement in performance. As the data-set got bigger and copying became less costly than the calculation itself the improvement became evident. One way to further improve the performance would be to use a combination of these approaches and use the GPU only on the most demanding tasks of the algorithm.

4.3 Image processing algorithm - Linear filtration

Filtration is an important part of image processing and image analysis. Its goal is to reduce image noise, where noise is a set of random variation of pixel values. Further we will go over few examples of noise.

Additive noise

Is specified as adding or subtracting random values from some distribution(most commonly $N(0,1) * constant$) to original pixel values. For reduction of such noise we use methods that don't change the picture average function e.g. linear diffusion.

Salt and pepper noise

Salt and pepper noise is characterized by taking a random set of pixels and assigning them values from range 0,...,Q-1 (where Q is maximal possible intensity of a pixel).

4.3.1 Linear filtration using transient heat transfer

In image Ω, with pixels x with the iteration time of σ, we are looking for a function $u(x,t)$, where $x \in \Omega, t \in [0,\sigma]$,

$$\frac{\partial u(x,t)}{\partial t} = \triangle u(x,t), \Omega \times [0,\sigma] \tag{4.4}$$

$$\frac{\partial u(x,t)}{\partial \boldsymbol{n}} = 0 \qquad , \partial\Omega \times [0,\sigma] \tag{4.5}$$

$$u(x,0) = u^0(x) \qquad , x \in \Omega, \tag{4.6}$$

n is normal on edge $\partial\Omega$ and $u^0(x)$ represents the image intensity of initial image.

Explicit scheme for solving transient heat transfer

At first, time ($[0,T]$), where T is the end-time, is discretized into N parts, with time step τ. Solution in time step $n = u^n$. Time difference is replaced with an approximation, time difference in our case. We will use explicit time difference,

$$\frac{u^{n+1} - u^n}{\tau},$$

after which, explicit time discretization,

$$\frac{u^{n+1} - u^n}{\tau} = \triangle u^n = \bigtriangledown.(\bigtriangledown u^n), \tag{4.7}$$

where $N(p)$ is a set of surrounding pixels that share an edge with the pixel p. Next step is space discretization of 4.7, with

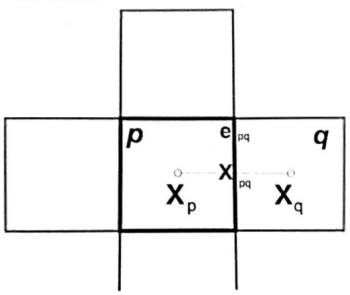

Figure 4.2: Pixel structure[24]

using finite volume method as seen in [24] we get explicit scheme for solving transient heat transfer.

$$u_p^{n+1} = (1 - \frac{\tau}{h^2} \sum_{q \in N(p)} 1) u_p^n + \frac{\tau}{h^2} \sum_{q \in N(p)} u_q^n. \tag{4.8}$$

Neumann boundary conditions $\frac{\partial u}{\partial \boldsymbol{n}} = 0$ on $\partial\Omega$, in 4.8 can be represented as use of mirror image of one row of boundary pixels.

4.3.2 Implementation comparison

With given parameters $\tau = 0.2, h = 1$, where h is the height and width of each pixel and pictures with dimensions 385 x 288 and 2400 x 1204 pixels. We continually increase the number of iterations τ.

τ	1-core CPU	cuda custom kernel
385x288		
10	0.008418	0.005174
500	0.390428	0.036612
5000	3.608867	0.299972
10000	7.149872	0.587601
2400x1204		
10	0.197871	0.005804
500	9.486399	0.38331
5000	93.991882	0.695056
10000	187.109116	0.980782

Table 4.6: Computational time of Explicit scheme for solving transient heat transfer (seconds) using different implementation methods and different number of iterations

With very generous estimate of linear reduction of computational time when increasing number of cores on a CPU, we see that as the number of iterations increases the computing efficiency of GPU is more and more visible.

4.4 Distance function in image processing

Let us consider a digital image A described as a discrete function in domain D of size $N \times M$ with binary values from set $\{0, 1\}$. We consider the image A as a set of pixels $\{A_{i,j}\}$ with values $\{a_{i,j}\}$ where every pixel is defined by spatial coordinates (i, j).
Distance function or distance map gives each pixel value with the distance to the nearest obstacle pixel. In case of a binary image a boundary pixel of an object.

$$f(x) = d(x, \partial\Omega) \quad \text{if } x \in \Omega \tag{4.9}$$

Where $\partial\Omega$ denotes the boundary of Ω and for any $x \in D$

$$d(x, \partial\Omega) = \inf_{y \in \partial\Omega} d(x, y) \tag{4.10}$$

Another type of distance function is signed distance function. It is defined as:

$$f(x) = \begin{cases} d(x, \partial\Omega) & \text{if } x \in \Omega \\ -d(x, \partial\Omega) & \text{if } x \notin \Omega \end{cases} \tag{4.11}$$

One more way to differentiate distance functions is to introduce different types of metrics. In our calculations we will use Euclidean metric.

4.4.1 Brute force distance function

For ease of implementation and purpose of computational improvement we have chosen brute force distance function with slight modification to image scanning.
Brute force algorithm is based on looping through the whole image for each pixel and calculating distance to each pixel containing a value of 1. By comparing all the calculated distances we get the lowest distance which is then assigned to the given pixel.

```
define mapD (N,M)
for each pixel X
        for each pixel Y
                where Y = 1
                        d = distance(X,Y)
        end for
        mapD(X) = min(d(Y))
end for
```

Listing 4.1: Brute force distance function pseudo code

The need of looping through the image one time for each pixel gives the complexity of O(n^2). We can reduce this need of looping through the image n^2 times, where n is $N \times M$, by introducing a different approach to image scanning. We take each pixel as the center of a circle s with gradually expanding diameter r. Upon encountering a pixel with a value of 1 the r is the distance for given pixel s. In the worst scenario, when the image is empty, this approach has no effect on calculation time, in the best scenario, it is more than three times faster. By using midpoint circle algorithm [16], we ensure to cover all the pixels in the image.

Implementation comparison

Using the same binary image with dimension 256 x 256 and 512 x 512 pixels.

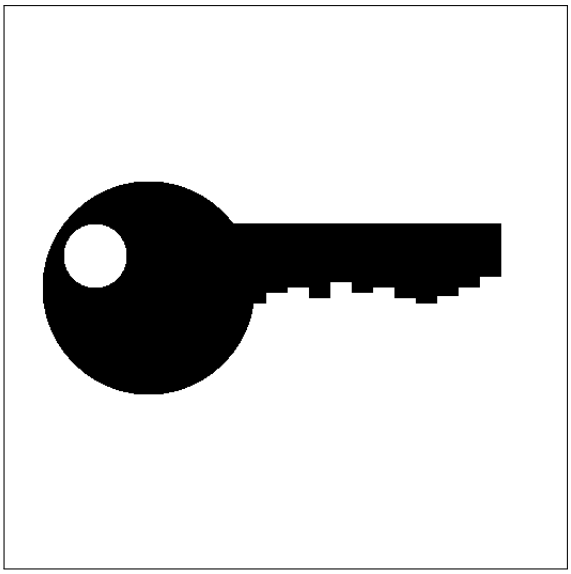

Figure 4.3: Binary image used

N x N	1-core CPU	cuda custom kernel
Brute force algorithm		
256 x 256	7.718576	0.134052
512 x 512	27.302315	2.227006
Brute force algorith with modified image scanning		
256 x 256	0.840206	0.028177
512 x 512	12.012230	0.327531

Table 4.7: Computational time of brute force distance function (seconds) using different implementation methods and different image sizes

4.5 Shape registration

A process of transforming different sets of data into one coordinate system is called shape registration. Often used in computer vision, military automatic target recognition, medical imaging, etc. Another use case is in astrophotography, where multiple images of a faint objects are registered to one another to increase the signal from the object and reduce the noise.

In our use case we consider two images D and S and we are looking for a transformation

$$A = \begin{bmatrix} cos\Theta & sin\Theta & 0 \\ -sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{s} & 0 & 0 \\ 0 & \frac{1}{s} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 0 \end{bmatrix}, \tag{4.12}$$

that minimizes the difference between the transformed image S and the image D. Where Θ is the degree of rotation, s is the scaling factors and T_x and T_y are the translation coefficients. We define Φ_D as distance function of the image D, then we define a functional E, which is represented by the sum of squares of the source image and destination image distance function, which we are trying to minimize.

$$\begin{aligned} E(s,\Theta,\boldsymbol{T}) &= \int_D (\Phi_D(x,y) - \Phi_S(A^T(x.y))^2 dxdy, \text{where} \\ \boldsymbol{T} &= \begin{bmatrix} T_x \\ T_y \end{bmatrix} \end{aligned} \tag{4.13}$$

translation factor. By following [17] we get modified optimization criterion

$$\begin{aligned} E(s,\Theta,\boldsymbol{T}) &= \int_D N_\delta(\Phi_D,\Phi_S)(\Phi_D(x,y) - \Phi_S(A^T(x,y)))^2 dxdy, \text{ where} \\ N_\delta(\Phi_1,\Phi_2) &= \begin{cases} 0, min(|\Phi_1|,|\Phi_2|) > \delta \\ 1, min(|\Phi_1|,|\Phi_2|) \leq \delta \end{cases} . \end{aligned} \tag{4.14}$$

As stated in [17], this modification accelerates the calculation and focus optimization just on the area that interests us the most. For registering the two images we used gradient method[11] and calculated new distance functions after each iteration. Where particular components of the gradient are

$$\begin{aligned} \frac{d}{dt}\Theta &= 2\int_D N_\delta(\Phi_D,\Phi_S)(\nabla\Phi_S.\nabla_\Theta A^T)(\Phi_D(x,y) - \Phi_S(A^T(x,y))dxdy \\ \frac{d}{dt}s &= 2\int_D N_\delta(\Phi_D,\Phi_S)(\nabla\Phi_S.\nabla_s A^T)(\Phi_D(x,y) - \Phi_S(A^T(x,y))dxdy \\ \frac{d}{dt}\boldsymbol{T} &= 2\int_D N_\delta(\Phi_D,\Phi_S)(\nabla\Phi_S.\boldsymbol{\nabla_T} A^T)(\Phi_D(x,y) - \Phi_S(A^T(x,y))dxdy. \end{aligned} \tag{4.15}$$

To approximate $\nabla\Phi_S$ we used central difference

$$\nabla\Phi_S(x,y) = \begin{pmatrix} \nabla_x\Phi_S(x,y) \\ \nabla_y\Phi_S(x,y) \end{pmatrix} = \begin{pmatrix} \frac{\Phi_S(x+1,y)-\Phi_S(x-1,y)}{2h} \\ \frac{\Phi_S(x,y+1)-\Phi_S(x,y-1)}{2h} \end{pmatrix}, \tag{4.16}$$

where h is the pixel size and forward difference

$$\nabla\Phi_S(x,y) = \begin{pmatrix} \nabla_x\Phi_S(x,y) \\ \nabla_y\Phi_S(x,y) \end{pmatrix} = \begin{pmatrix} \frac{\Phi_S(x+1,y)-\Phi_S(x,y)}{h} \\ \frac{\Phi_S(x,y+1)-\Phi_S(x,y)}{h} \end{pmatrix} \tag{4.17}$$

for beginnings of ∂D and ∂S and backward difference

$$\nabla\Phi_S(x,y) = \begin{pmatrix} \nabla_x\Phi_S(x,y) \\ \nabla_y\Phi_S(x,y) \end{pmatrix} = \begin{pmatrix} \frac{\Phi_S(x,y)-\Phi_S(x-1,y)}{h} \\ \frac{\Phi_S(x,y)-\Phi_S(x,y-1)}{h} \end{pmatrix} \tag{4.18}$$

for endings of ∂D and ∂S. Where particular components of $\nabla_\Theta A^T, \nabla_s A^T$ and $\nabla_T A^T$ are

$$\begin{aligned}
\nabla_\Theta A^T &= \begin{pmatrix} \frac{ycos(\Theta)}{s} - \frac{xsin(\Theta}{s} \\ -\frac{xcos(\Theta}{s} - \frac{ysin(\Theta)}{s} \end{pmatrix} \\
\nabla_s A^T &= \begin{pmatrix} -\frac{xcos(\Theta)}{s^2} - \frac{ysin(\Theta)}{s^2} \\ -\frac{ycos(\Theta)}{s^2} + \frac{xsin\Theta}{s^2} \end{pmatrix} \\
\nabla_{T_x} A^T &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \nabla_{T_y} A^T = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.
\end{aligned} \tag{4.19}$$

Our testing was done using binary image (Figure 4.3) with dimensions 512 x 512. Where the transformation we were looking for was $\Theta = -0.2, s = 0.8, T_x = 40$ and $T_y = -100$. As our initial approximation of transformation parameters for gradient method we used $\Theta = 0.1, s = 1.1$ and the difference of image centroids between the transformed image and the original as translation parameters. Where image centroid was calculated as the weighted average of pixel intensities and their position. As a stopping criterion for gradient method we used maximum number of iterations of 1000 and $E < tol$ where $tol = 1000$.

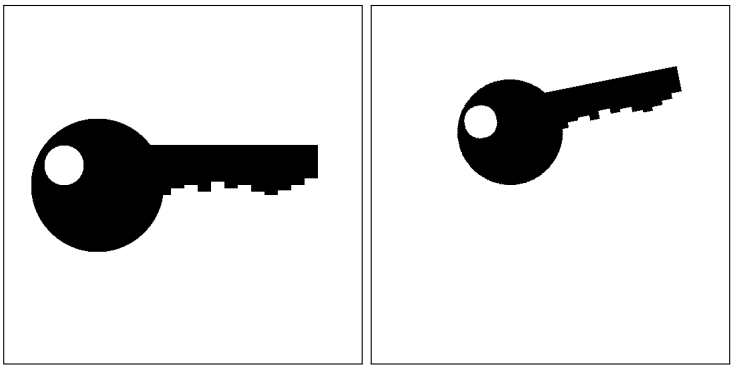

Figure 4.4: Image D and transformed image S

Initial distance functions of image D and transformed image S (Figure 4.5)

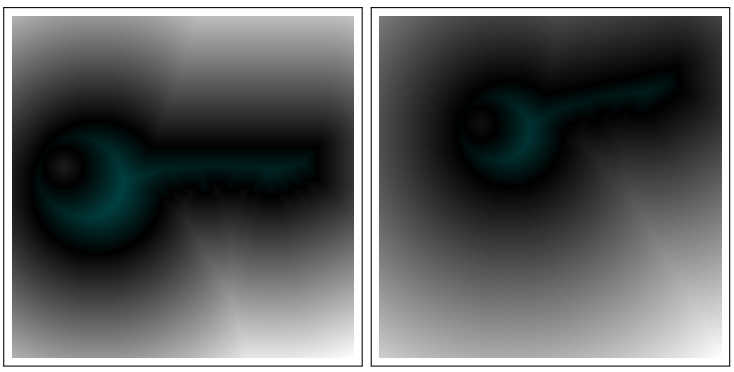

Figure 4.5: Distance functions of image D and transformed image S

The registration algorithm finished after 139 iterations with $E = 993.715149$ and registered parameters

- $\Theta = -0.200210$
- $s = 0.799858$
- $T_x = 39.987698$
- $T_y = -100.005966$

with less than 1% error from original parameters and $max(\Phi_S - \Phi_P) = 2.82843$ (maximal difference of 3 pixels), where P is the registered image. As expected from the parameters approximation, the registered image and distance function look identical to the transformed image S and its distance function (Figure 4.6).

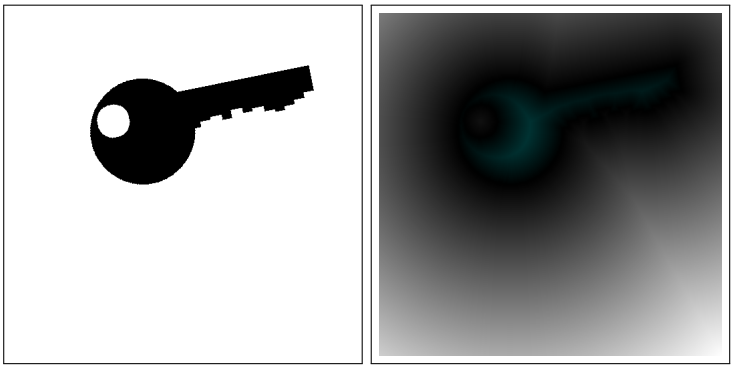

Figure 4.6: Registered image and its distance function

As a way to improve the time of the calculation, we have resorted in better approximation of initial parameters, by first down-sampling the image 4-times, registering the image to approximate the rotation and scaling parameters, next we used these parameters on an image down-sampled 2-times. Registered it and the new approximation were used for the final registration. Even though we had to run the registration multiple times, with better approximation of initial parameters, we have achieved that the final registration took only 85 iterations with $E = 998.236$ and registered parameters

- $\Theta = -0.200237$
- $s = 0.799873$
- $T_x = 39.985424$
- $T_y = -100.007820.$

The most impressive is the computational increase when using GPU as seen in 4.8.

1-core CPU 1-iter	1-core CPU all iter	1-core CPU with down-sampling 1-iter (final regist.)	1-core CPU with down-sampling all iter
22.4186889	3116.197754	18.74703	2091.466064
cuda custom kernel 1-iter	cuda custom kernel all iter	c.c.k. with down-sampling 1-iter (final regist.)	c.c.k. with down-sampling all iter
0.6102348	84.822639	0.615236	62.351429

Table 4.8: Time comparison in seconds of 1-core CPU vs GPU runtime of shape registration algorithm

The next step in improving the speed of the algorithm, is to combine down-sampling with increased tolerance to $tol = 12000$, from experiments we have found that this approximation is good enough, with translation parameters within half of a pixel from the transformation we are looking for and rotation with scaling within the 3% margin in an image with dimensions 512 x 512. This tolerance should increase with larger image and decrease with a smaller one.
This modification leads to approximation (on GPU) of parameters after 582 iterations for first down-sample by 4:

- $\Theta = -0.1774377$
- $s = 0.821956$
- $T_x = 10.447988$
- $T_y = -24.996664,$

in 3.6947 seconds. For the next down-sample by 2, we take translations parameters from last down-sample multiplied by 2, which finishes in 99 iterations and 4.785946 seconds. Approximated parameters of second down-sample:

- $\Theta = -0.200292$
- $s = 0.802109$
- $T_x = 20.008118$
- $T_y = -50.053726.$

Translation parameters are again taken and multiplied by 2 for our final registration on the original image, resulting in 10 more iterations and 6.750599 seconds and approximated parameters:

- $\Theta = -0.199721$
- $s = 0.800957$
- $T_x = 40.181030$
- $T_y = -100.012573.$

As is evident from these approximations, they are within the 1% tolerance from the sought after parameters, with the time of whole registration of 15.231 seconds, which is 4-times faster than running it on the original image from the start. This optimization increase would be the same with higher calculation time for the code running on the CPU.

Conclusion

In our work we familiarized ourselves with using CUDA library, its advantages, problems and seen the performance increase it brings to general purpose computing. During our testing we were able to implement multiple image processing algorithms with CUDA, as well as some other general algorithms, compared them with other parallel focused APIs. We have given the reader instruments to minimize the drawbacks of working with CUDA, like learning new syntax and memory management by using unified memory or computational time increase due to memory passing by using concurrency. We have seen that even though GPUs in general have much higher peak performance potential than CPUs, using them is not always the best solution. When it comes to very simple tasks, the memory passing is too costly, as to tasks that are more complex, can be easily parallelised and require more work, we have seen in some cases almost 20x improvement on a GPU using CUDA over a CPU using MPI. In image processing where a lot of the algorithms is iterative, the improvement is even more significant, because when every iteration is running X times faster on a GPU, the consequence is exponential improvement.

In the past using GPUs was not as beneficial as today, not only from a performance standpoint but also from financial standpoint and space saving standpoint. It has become a trend that GPU with the same peak performance potential as a CPU can be significantly cheaper. In some cases even GPUs that are 4x cheaper than a CPU can be 10x faster, which is all dependent on the task at hand. Since usually an add-on card to the computer, there can be multiple GPUs in one computer which in the end saves space while increasing compute potential.

All source code and tests used in this diploma project is available on bitbucket for further use.

Resumé

Cieľom našej práce je integrovať algoritmy spracovania obrazu za pomoci CUDA na GPU. Porovnať časové zlepšenie behu algoritmov oproti implementáciam na CPU. Taktiež porovnať cenové náklady a časové náklady integrácie jednotlivých prístupov. V rámci našej práce vysvetlíme, výpočtové časti počítača, softvérové komponenty potrebné k paralelizácii algoritmov a následne si prejdeme proces implementácie, od výberu programovacieho jazyka, problémov, ktoré nastali až po experimenty vedúce k výsledku. Diplomová práca je zložená z troch častí.

V teoretickej časti sa zaoberáme výpočtovými súčasťami počítača, ako fungujú, ako a odkial dostávajú informácie a ich kombinácie na zvýšenie výpočtového výkonu v rôznych paralelných architektúrach. Po hardvérovej stránke sa venujeme softvérovej stránke problému. Dôležitou súčasťou porozumenia paralelných algoritmov je ich implementácia v počítači a teda si spomenieme modely paralelného programovania, ktoré sa nezaobídu bez svojich obmedzení a výhod. V rámci paralelného programovania sa dozvieme akou formou spolu jednotlivé vetvy komunikujú, čím je táto komunikácia ovplivnená a prostriedky dostupné pre programovanie na CPU a GPU. Ako prípravu pre praktickú časť našej práce sú vysvetlíme faktory ovplivňújúce výpočtový výkon a porovnáme finančnú výhodnosť jednotlivých prístupov.

Praktickú časť sme rozdelili na dve časti. V implementačnej časti sa zameriame na dôvody výberu programovacieho jazyka C++, ako aj detaily o tomto jazyko ako takom. Ďalej sa pozrieme na spomenuté vývojové prostredie Visual Studio 2015 Community, ktoré je voľne dostupné pre verejnosť a pomocou ktorého sme všetky experimenty vykonali. Bližší pohľad venujeme programovania pomocou CUDA, jej syntaxi, rozdielom oproti iným implementáciam a hlavne častiam, na ktoré si programátor musí dať pozor pri práci s CUDA. Opíšeme spôsoby ako minimalizovať negatívne stránky práce na grafickej karte a vysvetlíme akým spôsobom paralelné algoritmy bežia na počítači. Následne v experimentálnej časti sa pustíme do postupného budovania a časového porovnanie behu algoritmov potrebných pre použité metódy spracovania obrazu. Každý experiment je vysvetlený po teoretickej stránke, spolu s výsledkami praktickej stránky. Ako prvé su maticové operácie, ktoré su nevyhnutnou súčastou algoritmou na spracovania obrazu, taktiež su perfektným príkladom na ukázanie výhod a nevýhod práce s grafickými kartami. Ukážeme si dopad zle implementovaných

programov na grafickej karte spolu s porovnaním s implementáciou sekvenčne, MPI alebo s využitím už existujúcich prostriedkov ako cuBLAS. Ďalším krokom je iteratívna metóda BiCGStab pre riešenie sústav linearných rovníc, ktorú je možno použiť pri riešení lineárnej filtrácie. V tejto metóde sa zvýraznia nevýhody programovania na grafických kartách no i napriek týmto nevýhodám sa potvrdí hlavná výhoda pri programovaní na grafickej karte. Následne sa zameriame na algoritmus spracovania obrazu a to lineárnu filtráciu, konkrétnejšie filtrácie obrazu pomocou explicitnej metódy vedenia tepla, kde výhoda použitia grafických kariet začne byť evidentná. Predposledým experimentom je vzdialenostná funkcia v spracovaní obrazu, ktorá je následne použitá pri registrácii tvarou. Ako najpoučnejší príklad sme si zobrali "brute force" dištančnú funkciu, ktorá je časom výpočtu najnáročnejšia, následne sme ju modifikovali a podarilo sa nám znížiť dĺžku výpočtu skoro 10-násobne oproti originálnemu algoritmu. Takto upravenú dištančnú funkciu sme použili v poslednom experimente a to registrácii tvarou, kde sme si odvodili celý algoritmus registrácie, experimentálne ukázali jeho fungovanie a po jeho vylepšení pomocou prevzorkovania sme dospeli skoro k 30-násobnému zlepšeniu výpočtového času na GPU oproti jednému jadru na CPU. Čím sme potvrdili a ukázali hlavnú a velikú výhodu programovania a robenia všeobecných výpočtou na GPU.

Bibliography

[1] Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 631–644.

[2] C++ programming language, <https://tekslate.com/c-explain-advantages-disadvantages/>

[3] Central processing unit, <https://en.wikipedia.org/wiki/Central_processing_unit/>

[4] Computer Architecture Performance Evaluation Methods, Lieven Eeckhout, 2010, ISBN-978-1-608-45467-9

[5] CPU, GPU and MIC Hardware Characteristics over Time, <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

[6] CUDA programming: A Developers guide to parallel computing with GPUs, Shane Cook, 2013, ISBN-978-0-12-415933-4

[7] Cuda toolkit documentation, <https://docs.nvidia.com/cuda/>

[8] Current FLOPS prices, <https://aiimpacts.org/current-flops-prices/>

[9] Distance-based functions for image coparison, Vito di Gesu, Valery Starovoitov, Institute of Engineering Cybernetics, National Academy of Sciences, Belarus, 21. September 1998

[10] FLOPS, <https://en.wikipedia.org/wiki/FLOPS>

[11] Gradient method, <https://web.stanford.edu/w̃fsharpe/mia/opt/mia_opt1.htm>

[12] Graphics processing unit, <https://en.wikipedia.org/wiki/Graphics_processing_unit/>

[13] How to Overlap Data Transfers in CUDA C/C++ , <https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/>

[14] Introduction to hierarchical matrices with applications, Steffen Borm, Lars Grasedyck, Wolfgang Hackbusch, University Kiel, Germany, 2002

[15] Memory bandwidth in computers, <https://fgiesen.wordpress.com/2017/04/11/memory-bandwidth/>

[16] Midpoint circle algorithm, <https://en.wikipedia.org/wiki/Midpoint_circle_algorithm>

[17] Nové zlepšenia atlasom riadených segmentácií obrazu, Ing. Jozef Urbán, PhD., Stavebná fakulta STU Katedra matematiky a deskriptívnej geometrie, 01.07.2016, pp. 47-50

[18] OpenMP, <http://www.openmp.org/>

[19] Paralelné algoritmy, lectures, Ing. Róbert Čunderlík, PhD.

[20] Parallel Programming For Multicore and Cluster Systems, Thomas Rauber, Gudula Runger, 2010, ISBN 978-3-642-04817-3

[21] Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications, Lukasz Jarzabek, Pawel Czarnul, Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Gdansk, 2017

[22] Random access memory (RAM),<https://searchstorage.techtarget.com/definition/RAM-random-access-memory>

[23] Spracovanie obrazu, lectures, doc. RNDr. Zuzana Krivá, PhD.

[24] Spracovanie obrazu, Zuzana Krivá, Karol Mikula, Oľga Stašová , 2016, ISBN 978-80-227-4535-2, pp. 51-53

[25] Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods; Richard Barret, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine and Henk Van der Vorst

[26] Unified Memory for CUDA Beginners, <https://devblogs.nvidia.com/unified-memory-cuda-beginners/>

[27] Using MPI, William Gropp, Ewing Lusk, Anthony Skjellum, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1999, ISBN-0-262-57132-3

[28] What Every Programmer Should Know About Memory, <http://futuretech.blinkenlights.nl/misc/cpumemory.pdf>

Appendix

```
#define N 100
void MatrixMultiplicationOnHost(float * A, float * B, float * C,
   int numARows,
        int numAColumns, int numBRows, int numBColumns,
        int numCRows, int numCColumns)
{
        for (int i = 0; i < numARows; i++){
          for (int j = 0; j < numAColumns; j++){
           C[i*numCColumns + j] = 0.0;
           for (int k = 0; k < numCColumns; k++){
            C[i*numCColumns + j] += A[i*numAColumns + k] *
                B[k*numBColumns + j];
           }
          }
        }
        return;
}
int main(int argc, char ** argv) {
        float * hostA; // The A matrix
        float * hostB; // The B matrix
        float * hostC; // The output C matrix
        hostA = (float *)malloc(sizeof(float)*N*N);
        hostB = (float *)malloc(sizeof(float)*N*N);
        for (int i = 0; i < N*N; i++){
         hostA[i] = 1;
         hostB[i] = 1;
        }
        hostC = (float *)malloc(sizeof(float)*N*N);
        MatrixMultiplicationOnHost(hostA, hostB, hostC, N, N, N,
           N, N, N);
        return 0;
}
```

Listing 4.2: Sample code running on host sequentially

```
#define N 100
int main(int argc, char *argv[]){
    int i, j, rank, size, sum = 0;
    int a[N][N];
    int b[N][N];
    int c[N][N];
    int aa[N],cc[N];
        for (int i = 0; i < N; i++){
         for (int j = 0; j < N; j++){
          a[i][j] = 1; b[i][j] = 1;
         }
        }
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //scatter rows of first matrix to different processes
    MPI_Scatter(a, N*N/size, MPI_INT, aa, N*N/size,
       MPI_INT,0,MPI_COMM_WORLD);
    //broadcast second matrix to all processes
    MPI_Bcast(b, N*N, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    //perform vector multiplication by all processes
    for (i = 0; i < N; i++){
     for (j = 0; j < N; j++){
      sum = sum + aa[j] * b[j][i];
     }
     cc[i] = sum; sum = 0;
    }
    MPI_Gather(cc, N*N/size, MPI_INT, c, N*N/size, MPI_INT, 0,
       MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Listing 4.3: Sample code running on host with the use of MPI

```
#define N 100
__global__ void MatrixMultiplicationShared(float * A, float * B,
    float * C, int numARows, int numAColumns, int numBRows, int
    numBColumns, int numCRows, int numCColumns, int blockLength){
 extern __shared__ float blocks[];
 float *sA = blocks;
 float *sB = &sA[blockLength*blockLength];
 int Row = blockDim.y*blockIdx.y + threadIdx.y;
 int Col = blockDim.x*blockIdx.x + threadIdx.x;
 float Cvalue = 0.0;
 sA[threadIdx.y*blockLength + threadIdx.x] = 0.0;
 sB[threadIdx.y*blockLength + threadIdx.x] = 0.0;
 for (int k = 0; k < (((numAColumns - 1) / blockLength) + 1); k++){
  if ((Row < numARows) && (threadIdx.x + (k * blockLength)) <
     numAColumns){
   sA[threadIdx.y*blockLength+threadIdx.x] = A[(Row*numAColumns) +
      threadIdx.x + (k * blockLength)];
  }
  else{
   sA[threadIdx.y*blockLength + threadIdx.x] = 0.0;
  }
  if (Col < numBColumns && (threadIdx.y + k * blockLength) <
     numBRows){
   sB[threadIdx.y*blockLength + threadIdx.x] = B[(threadIdx.y + k
      * blockLength)*numBColumns + Col];
  }
  else{
   sB[threadIdx.y*blockLength + threadIdx.x] = 0.0;
  }
  __syncthreads();
  for (int j = 0; j < blockLength; ++j){
   Cvalue += sA[threadIdx.y*blockLength + j] * sB[j*blockLength +
      threadIdx.x];
  }
  __syncthreads();
 }
 if (Row < numCRows && Col < numCColumns{
  C[Row*numCColumns + Col] = Cvalue;
 }
}
```

```
void MatrixMultiplicationWithCuda(float * A, float * B, float * C,
   int numARows, int numAColumns, int numBRows, int numBColumns,
   int numCRows, int numCColumns){
 float * deviceA;
 float * deviceB;
 float * deviceC;
 int blockLength = 32;
 cudaSetDevice(0); // Choose which GPU to run on, change this on a
    multi-GPU system.
 // Allocating GPU memory
 cudaMalloc((void **)&deviceA, sizeof(float)*numARows*numAColumns);
 cudaMalloc((void **)&deviceB, sizeof(float)*numBRows*numBColumns);
 cudaMalloc((void **)&deviceC, sizeof(float)*numCRows*numCColumns);
 // Copy memory to the GPU
 cudaMemcpy(deviceA, A, sizeof(float)*numARows*numAColumns,
    cudaMemcpyHostToDevice);
 cudaMemcpy(deviceB, B, sizeof(float)*numBRows*numBColumns,
    cudaMemcpyHostToDevice);
 // Initialize the grid and block dimensions
 dim3 dimBlock(blockLength, blockLength, 1);
 dim3 dimGrid((numCColumns / blockLength == 0) ? numCColumns /
    blockLength : numCColumns / blockLength + 1, (numCColumns /
    blockLength == 0) ? numCColumns / blockLength : numCColumns /
    blockLength + 1, 1);
 // Launch the GPU Kernel here
 size_t SharedMemorySize = 2 * blockLength*blockLength *
    sizeof(float);
 MatrixMultiplicationShared << <dimGrid, dimBlock
    ,SharedMemorySize>> > (deviceA, deviceB, deviceC, numARows,
    numAColumns, numBRows, numBColumns, numCRows, numCColumns,
    blockLength);
 cudaDeviceSynchronize();
 // Copy the results in GPU memory back to the CPU
 cudaMemcpy(C, deviceC, sizeof(float)*numCRows*numCColumns,
    cudaMemcpyDeviceToHost);
 cudaFree(deviceA); // Free the GPU memory
 cudaFree(deviceB);
 cudaFree(deviceC);
}
int main(int argc, char ** argv) {
 float * hostA; // The A matrix
```

```
 float * hostB; // The B matrix
 float * hostC; // The output C matrix
 int numARows = N; // number of rows in the matrix A
 int numAColumns = N; // number of columns in the matrix A
 int numBRows = N; // number of rows in the matrix B
 int numBColumns = N; // number of columns in the matrix B
 int numCRows; // number of rows in the matrix C
 int numCColumns; // number of columns in the matrix C
 cudaMallocHost((void**)&hostA,
    sizeof(float)*numARows*numAColumns);
 cudaMallocHost((void**)&hostB,
    sizeof(float)*numBRows*numBColumns);
 for (int i = 0; i < numARows*numAColumns; i++){
  hostA[i] = 1; hostB[i] = 1;
 }
 numCRows = numARows; // Setting numCRows and numCColumns
 numCColumns = numBColumns;
 cudaMallocHost((void**)&hostC,
    sizeof(float)*numCRows*numCColumns);
 MatrixMultiplicationWithCuda(hostA, hostB, hostC, numARows,
    numAColumns, numBRows, numBColumns, numCRows, numCColumns);
 cudaFreeHost(hostA);
 cudaFreeHost(hostB);
 cudaFreeHost(hostC);
 return 0;
}
```

Listing 4.4: Sample code running on device with the use of CUDA

```
#define N 100
void MatrixMultiplicationWithCublas(float *A, float *B, float*C,
   int m, int n, int k){
 float *d_A, *d_B, *d_C;// Allocate 3 arrays on GPU
 cudaMalloc(&d_A, m * k * sizeof(float));
 cudaMalloc(&d_B, k * n * sizeof(float));
 cudaMalloc(&d_C, m * n * sizeof(float));
 cudaMemcpy(d_A, A, m * k * sizeof(float), cudaMemcpyHostToDevice);
 cudaMemcpy(d_B, B, k * n * sizeof(float), cudaMemcpyHostToDevice);
 const float alf = 1;
 const float bet = 1;
 const float *alpha = &alf;
 const float *beta = &bet;
 cublasHandle_t handle;// Create a handle for CUBLAS
 cublasCreate(&handle);
 cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, alpha,
    d_A, m, d_B, k, beta, d_C, m);
 cublasDestroy(handle);// Destroy the handle
 cudaMemcpy(C, d_C, m * n * sizeof(float),
    cudaMemcpyDeviceToHost));
 cudaFree(d_A);//Free GPU memory
 cudaFree(d_B);
 cudaFree(d_C);
}

int main(int argc, char ** argv) {
 float * hostA; // The A matrix
 float * hostB; // The B matrix
 float * hostC; // The output C matrix
 int numARows = N; // number of rows in the matrix A
 int numAColumns = N; // number of columns in the matrix A
 int numBRows = N; // number of rows in the matrix B
 int numBColumns = N; // number of columns in the matrix B
 int numCRows; // number of rows in the matrix C
 int numCColumns; // number of columns in the matrix C
 cudaMallocHost((void**)&hostA,
    sizeof(float)*numARows*numAColumns);
 cudaMallocHost((void**)&hostB,
    sizeof(float)*numBRows*numBColumns);
 for (int i = 0; i < numARows*numAColumns; i++){
```

```
  hostA[i] = 1; hostB[i] = 1;
 }
 numCRows = numARows;// Setting numCRows and numCColumns
 numCColumns = numBColumns;
 cudaMallocHost((void**)&hostC,
    sizeof(float)*numCRows*numCColumns);
 MatrixMultiplicationWithCublas(hostA, hostB, hostC, numAColumns,
    numAColumns, numBColumns);
 cudaFreeHost(hostA);
 cudaFreeHost(hostB);
 cudaFreeHost(hostC);
 return 0;
}
```

Listing 4.5: Sample code running on device with the use of CUDA and cuBLAS

All source code can be found here:
https://bitbucket.org/xbatka/cuda-diploma-project-source-code/src