



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER SYSTEMS**

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

## **SIMULATION OF ULTRASOUND PROPAGATION IN BONES**

SIMULACE ŠÍŘENÍ ULTRAZVUKU V KOSTECH

**MASTER'S THESIS**

DIPLOMOVÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**Bc. KRISTIÁN KADLUBIAK**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**Ing. JIŘÍ JAROŠ, Ph.D.**

BRNO 2017

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačových systémů

Akademický rok 2016/2017

**Zadání diplomové práce**

Řešitel: **Kadlubiak Kristián, Bc.**

Obor: Počítačové a vestavěné systémy

Téma: **Simulace šíření ultrazvuku v kostech**

**Simulation of Ultrasound Propagation in Bones**

Kategorie: Modelování a simulace

**Pokyny:**

1. Seznamte se se současnou implementací simulace šíření ultrazvuku v kostech vytvořenou v prostředí MATLAB.
2. Prostudujte možnosti akcelerace vědeckých aplikací na superpočítačích Anselm a Salomon.
3. Analyzujte současnou implementaci v prostředí MATLAB.
4. Navrhněte postup pro převod kódu do některého HPC jazyka včetně formátu vstupně výstupních souborů.
5. Implementujte simulaci šíření ultrazvuku v kostech v některém HPC jazyce s ohledem na maximální výkon a minimální paměťovou režii.
6. Analyzujte výkon navržené implementace na standardní sadě testovacích úloh.
7. Zhodnoťte dosažené výsledky a diskutujte přínos navržené implementace pro praxi.

**Literatura:**

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).


Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D., UPSY FIT VUT**

Datum zadání: 1. listopadu 2016

Datum odevzdání: 24. května 2017

**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
Fakulta informačních technologií  
Ústav počítačových systémů a sítí  
611 00 Brno, Božetěchova 2

  
prof. Ing. Lukáš Sekanina, Ph.D.  
vedoucí ústavu

## Abstract

It is estimated that mind-boggling 14.1 million new cases of cancer occurred worldwide in 2012 alone. This number is alarming. Although healthy lifestyle may reduce a risk of developing cancer, there is always some probability that cancer would develop even in an absolutely fit individual. There are two main conditions for successful treatment of cancer. Firstly, early diagnostic is absolutely crucial. Secondly, there is a need for suitable surgical methods for affected tissue removal. Ultrasound has a great potential to be used for both purposes as a non-invasive method. Photoacoustic spectroscopy is imaging method for tumor detection of great properties making the use of ultrasound while High-Intensity Focused Ultrasound (HIFU) is non-invasive surgical method. These methods would be impossible without precise ultrasound propagation simulations. The k-Wave is an open source MATLAB toolbox implementing such simulations. So, why are not these methods already deployed in treatment? Unfortunately, the simulation of ultrasound propagation is a very time consuming task, which makes it ineffective for medical purposes. However, there are a few options how to accelerate these simulations. The use of GPU is a very promising way to accelerate simulation. The main topic of this thesis is the acceleration of the simulation of soundwaves propagation in bones and hard tissue. The implementation developed as a part of this thesis was benchmarked on various supercomputers including Anselm in Ostrava and Piz Daint in Lugano. The implemented solution provides remarkable acceleration compared to the original MATLAB prototype. It was able to accelerate the simulation around 160 times in the best case. It means that the simulation, which would otherwise last for 6.5 days, can be now computed in one hour. This acceleration was achieved using an NVIDIA Tesla P100 to run the simulation with the domain size of  $416^3$  grid points. The thesis includes performance benchmarks on different GPUs to provide complex image acceleration capabilities of developed implementation and provides discussion about memory usage and numerical accuracy. Thanks to the implemented solution harnessing the power of modern GPUs, doctors and researchers all around the world have a powerful tool in hands.

## Abstrakt

Odhaduje sa, že v roku 2012 sa objavilo celosvetovo neuveriteľných 14.1 milióna nových prípadov rakoviny. Toto číslo je alarmujúce. Napriek tomu, že zdravý životný štýl môže zredukovať riziko vzniku rakoviny, vždy existuje istá pravdepodobnosť, že sa rakovina objaví aj u úplne zdravého jedinca. Na úspech liečenia rakoviny majú vplyv najmä dva faktory. Po prvé - včasná diagnostika je absolútne nevyhnutná, po druhé - musí existovať vhodná operačná metóda na odstránenie poškodeného tkaniva. V obidvoch prípadoch má ultrazvuk veľký potenciál ako neinvazívna metóda. Fotoakustická spektroskopia je zobrazovacia metóda so skvelými vlastnosťami, založená na princípe ultrazvuku, schopná detegovať tumor. High-Intensity Focused Ultrasound (HIFU) je neinvazívny chirurgický postup. Tieto metódy by však neboli možné bez presnej simulácie šírenia ultrazvuku. Balíček k-Wave je open source toolbox pre MATLAB, ktorý implementuje tieto simulácie. Vystáva otázka, prečo nie sú tieto metódy bežne používané v praxi? Dôvodom je fakt, že simulácia šírenia ultrazvuku je veľmi časovo náročná operácia, čo robí tieto metódy neefektívnymi. Avšak existujú spôsoby akcelerácie takýchto simulácií. Implementácia simulácie na GPU je veľmi perspektívny prístup k akcelerácií. Hlavnou úlohou tejto diplomovej práce je akcelerácia simulácie šírenia ultrazvuku v kostiach a iných tvrdých tkanivách. Implementácia vyvinutá v rámci diplomovej práce bola testovaná na rôznych superpočítačoch ako napríklad Anselm v Ostrave alebo Piz Daint v Lugane. Implementované riešenie dosahuje pozoruhodné zrýchlenie v porovnaní s originálnym prototypom v prostredí MATLAB. V najlepšom prípade bola implementácia schopná urýchliť simuláciu približne 160 násobne. To znamená, že simulácia, ktorá by za iných okolností trvala 6,5 dňa, je dnes dokončená za jednu hodinu. Toto zrýchlenie bolo dosiahnuté počas simulácie s rozmermi  $416^3$  bodov a za použitia karty NVIDIA Tesla P100. Diplomová práca obsahuje porovnanie výkonu na rôznych grafických kartách, aby čitateľovi umožnila komplexnejší náhľad na akceleračné schopnosti vyvinutej implementácie a tiež poskytuje bližší pohľad na pamäťovú náročnosť a numerickú presnosť aplikácie. Vďaka schopnosti aplikácie naplno využiť potenciál grafických kariet, majú lekári a vyskumníci z celého sveta v rukách mocný nástroj.

## Keywords

GPGPU, HPC, CUDA, k-Wave, ultrasound propagation simulation

## Klíčová slova

GPGPU, HPC, CUDA, k-Wave, simulácia šírenia ultrazvuku

## Reference

KADLUBIAK, Kristián. *Simulation of Ultrasound Propagation in Bones*. Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Jaroš Jiří.

# Simulation of Ultrasound Propagation in Bones

## Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Jiří Jaroš, Ph.D. and Dr. Bradley E. Treeby. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....  
Kristián Kadlubiak  
May 23, 2017

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The k-Wave toolbox . . . . .	4
1.2	Wave propagation through elastic medium . . . . .	5
1.3	Assignment tasks . . . . .	5
<b>2</b>	<b>Graphics Processing Unit</b>	<b>7</b>
2.1	Architecture comparison . . . . .	7
2.2	General Purpose Graphics Computing Unit . . . . .	8
2.2.1	GPGPU framework . . . . .	8
2.3	CUDA-capable GPU architectures . . . . .	9
2.3.1	Host interface . . . . .	9
2.3.2	Copy engine . . . . .	9
2.3.3	DRAM adapter . . . . .	10
2.3.4	Device memory . . . . .	10
2.3.5	Streaming multiprocessor . . . . .	10
2.4	CUDA thread execution model . . . . .	11
2.4.1	Kernel . . . . .	12
2.4.2	Grid . . . . .	12
2.4.3	Block . . . . .	12
2.4.4	Thread . . . . .	12
2.4.5	Warp and lane . . . . .	13
2.5	CUDA memory model . . . . .	13
2.5.1	Global memory . . . . .	14
2.5.2	Local memory . . . . .	14
2.5.3	Registers . . . . .	14
2.5.4	Shared memory . . . . .	14
2.5.5	Texture memory . . . . .	15
2.5.6	Constant memory . . . . .	15
<b>3</b>	<b>Reference Implementation</b>	<b>16</b>
3.1	The k-Wave implementation . . . . .	16
3.1.1	Governing equations . . . . .	18
3.2	Implementation of governing equations in MATLAB . . . . .	20
3.2.1	Computation of gradients of stress tensor . . . . .	20
3.2.2	Computation of split-field particle velocity . . . . .	20
3.2.3	Spatial velocity gradient calculation . . . . .	21
3.2.4	Spatial gradients of the time derivative of the velocity . . . . .	21
3.2.5	Computation of stress tensor values in next time step . . . . .	21

3.2.6	Conclusion	21
3.3	Existing CUDA framework	22
3.3.1	Main components of framework	22
<b>4</b>	<b>Implementation</b>	<b>25</b>
4.1	Implementation breakdown into tasks	25
4.2	Technologies used in development	25
4.2.1	The Google test	25
4.2.2	The HDF5 library	26
4.2.3	The cuFFT library	26
4.3	Compute kernels implementation on GPU	27
4.3.1	Computation of gradients of stress tensor	27
4.3.2	Computation of split-field particle velocity	27
4.3.3	Spatial velocity gradient calculation	28
4.3.4	Implementation of stress tensor computation	28
4.3.5	Implementation of velocity and stress sources	29
4.3.6	Unit tests of individual kernels	29
4.3.7	Current limitations of implementation	30
4.4	Modification of I/O file format	30
4.5	Integration with the framework	30
4.6	Integration with the k-Wave	31
4.7	Numerical accuracy testing	31
4.8	Performance testing	32
4.9	Documentation	32
<b>5</b>	<b>Experimental Results</b>	<b>33</b>
5.1	Performance evaluation	33
5.2	Numerical accuracy	37
5.3	Memory consumption	39
5.4	Performance limitations	41
5.4.1	The split-field particle velocity computation	42
5.4.2	The matrix addition computation	45
5.4.3	Conclusion	48
<b>6</b>	<b>Conclusion</b>	<b>50</b>
6.1	Impact	50
6.2	Further improvements	51
	<b>Bibliography</b>	<b>52</b>
	<b>A Performance benchmark data</b>	<b>55</b>
	<b>B Format of the HDF5 input file</b>	<b>57</b>

# Chapter 1

## Introduction

Cancer is nowadays one of the most frightening diseases. It is estimated that in 2012 alone, mind-boggling 14.1 million new cases of cancer occurred worldwide [24]. Without a doubt, research of cancer as well as possibilities of treatment are among highest priority tasks of many health organization around the world. What are actual possibilities of treating cancer? First of all, the most important is prevention. It is obvious that to minimize the probability of developing cancer, one should eliminate all activities and actions which may contribute to cancer development, such as smoking, exposure to certain types of chemicals and so on. It is also important to take action which can reduce the chance, for example exercise regularly. However, there are certain factors that person does not have control of. The genetics or condition of immune system are such factors. There even exist several viral diseases that can directly cause cancer. All things considered, despite our best effort to minimize the possibility of developing cancer, it will never be equal to zero. So, what are the possible steps of treating the patient who has already cancer? Basically there are two important things. Cancer has to be detected in the first place in patient. Detecting cancer in its early stages is absolutely crucial to the success of entire treatment. Most of the time, the stage of detected cancer determines patient prospects of recovery. The other part of cancer treatment is removal or destruction of affected tissue, preferably, with very little impact on surrounding tissue and entire human organism.

The ultrasound has a potential to be used in both aspects of the treatment and in some cases is already being used for certain type of cancer. The photoacoustic spectroscopy is method which uses ultrasound to create internal representation of examined material in non-invasive way. In medicine, photoacoustic spectroscopy allows in vivo visualization of light absorbing structures. It is based on photoacoustic effect and involves illumination of tissue sample with short pulses of light, infra-red in most cases. The difference between light absorption properties of blood and surrounding tissue results in creation of ultrasonic waves by thermoelastic effect. These waves propagate through the tissue to the surface where they are recorded. Images of initial light absorption properties of the tissue (thus image of vascular system) can be obtained via backward simulation of recorded acoustic pressure [23]. The precision of this method strongly depends on used equipment and computational model but, in general, photoacoustic spectroscopy is able to extract very detail pictures of tissue. Abilities of this method are shown in Fig. 1.1 where photoacoustic spectroscopy was used to retrieve images of leukemia cells in abdomen of a mouse.



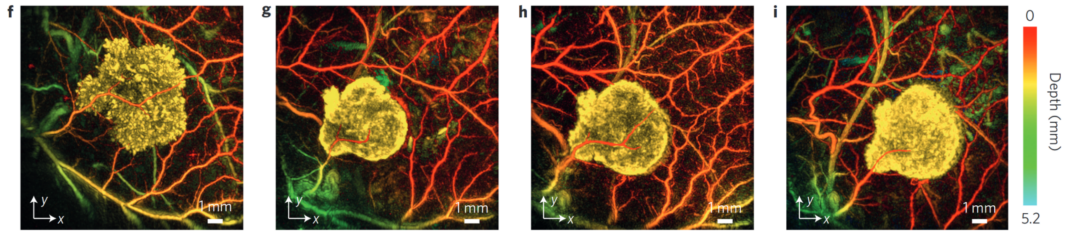


Figure 1.1: In vivo photoacoustic images of Tyr-expressing K562 cells after subcutaneous injection into flank of nude mouse at different time points [6].

The High Intensity Focused Ultrasound (HIFU) is method where several rays of ultrasound are focused into a matter in such way that they create one focal point. Each ray passes trough matter with little effect, but in a focal point energy of all beams is combined together. Ultrasound is basically a mechanical vibration and therefore the HIFU method increases temperature of focal point. The temperature is adjustable by setting the intensity of respective ultrasound rays. The HIFU method has tremendous potential to improve the treatment of certain types of brain cancer. As this modality is non-invasive and accurate, it may be able to ablate only targeted tissue while sparing healthy adjacent tissue. This is especially critical to the brain where any damage to healthy tissue can result in significant loss of function. In addition, focused ultrasound has the potential to reduce the risk of infection and bleeding, lower procedural morbidity by not opening the skull, and avoid the toxicity of radiation [1]. Moreover, this method, thanks to variable energy, can be also used in treatment of many different diseases. For example, lower-intensity HIFU can be used to destruct blood cloths in arteries. There are many others applications. So, why is potential of ultrasound not fully utilized in medicine yet? Firstly, despite the fact that ultrasound and mentioned physical effects are well known for decades, methods as photoacoustic spectroscopy and HIFU are relatively new fields in biomedicine and these methods are subject of intensive research and thus they cannot be used as standard go-to methods for treatment just yet. Moreover, these methods would not be possible without a proper computational model. Unfortunately, these models require great amount of data and computational time to be able to produce desired outcome. One of the projects implementing such a model is k-Wave toolbox for MATLAB and the main goal of this thesis is to accelerate k-Wave simulation of ultrasound wave propagation in elastic medium on graphics processing unit.

## 1.1 The k-Wave toolbox

The k-Wave is an open source third party toolbox for MATLAB developed for simulation and reconstruction of wave fields propagation in either homogeneous or heterogeneous material in one, two or three dimensions. Main goals of creators are creation of fast, precise and easy-to-use solution. Therefore, there is also a lot of interest in the speed of the simulation and reconstruction. In this matter, several significant measures have been taken.

Acoustic wave equations are partial differential equations and they are used in simulation of wave fields in k-Wave. The most common numerical methods for solving partial differential equations are finite-difference, finite-element and boundary-element methods [22]. Common methods achieve unsatisfying results in terms of performance as they require extreme amount of memory. Major disadvantages of traditional methods are a great number of grid points per wavelength, and small time-step size to minimize numerical error. There-

fore, pseudo-spectral and k-space methods were implemented. The pseudo-spectral method is based on Fourier series, which can be efficiently calculated by Fast Fourier Transform (FFT). As little as two grid points per the shortest wavelength occurring in domain are needed when the pseudo-spectral method is used. The pseudo-spectral method brought improvement in spatial domain. The k-space method is used to achieve improvements in time domain by allowing greater time steps while preserving the precision [22]. Besides special methods implemented to improve application's performance, there is also another way of increasing speed the of algorithms. Parallelism and architecture specific optimization techniques could be used to improve the performance.

In practice, k-Wave has been successfully used to evaluate prostate cancer patients' suitability for HIFU treatment [2]. The most common method of prostate cancer treatment is radiotherapy which requires a small metal marker to be inserted into the prostate. Respective position of this metal marker and the tumor affects significantly the effectiveness of the HIFU treatment. The impact on wave propagation by various marker positions was calculated by k-Wave toolbox. Now, patients' suitability can be evaluated based on a few x-ray images without the need for any additional procedure.

## 1.2 Wave propagation through elastic medium

It is a common knowledge that up to 70% of the human body is made out of water, and therefore, the majority of the human tissue can be modeled as fluid during wave propagation simulations. However, there are a few occasions when this model fails. For example, wave propagation in bones. Although, wave propagation model in elastic medium share a lot of similarities with fluid model, there are several additional aspects which have to be taken into account. Bones, in general, have higher acoustic absorption coefficients and more than double compressional sound speed when compared to fluids. This fact leads to the reflection, attenuation and aberration of propagating waves and a reduction in focusing quality [7]. One of the most important difference between fluid and elastic model is that in elastic medium, the propagation of both compressional (longitudinal) and shear (transversal) waves has to be simulated in order to accurately calculate the absorption and related heat generation. The wave propagation itself can be simulated using Hook's law and the momentum preservation equation. For viscoelastic materials where absorption occurs, Hook's law has to be extended so it exhibits time-dependent behavior. Absorption can be modeled in variety of ways. The one of widely used is Kelvin-Voigt model which can be described as damped spring where spring and damper is connected parallel to each other [23].

## 1.3 Assignment tasks

In this section, main tasks which are part of thesis assignment are summed up and for each task a part of document is mentioned where the reader is able to find further information. Assignment has following tasks:

- study the current implementation of the simulation of ultrasound wave propagation in bones created in MATLAB (more information in Sec. 3.2),
- study possibilities of scientific computation acceleration on Anselm and Salomon supercomputers (more information in Sec. 2),

- analyze the current implementation in MATLAB (more information in Sec. 3.2),
- design an approach for the code transformation from MATLAB into one of HPC languages including the format of input and output data (more information in Sec. 3.3),
- implement simulation of ultrasound propagation in bones using selected HPC language with respect to maximal performance and minimal memory requirements (more information in Sec. 4),
- analyze the performance of the implemented application on a set of standard test tasks (more information in Sec. 5),
- discuss achieved results and practical impact of the thesis (more information in Sec. 6).

## Chapter 2

# Graphics Processing Unit

At the beginning of computer graphics, all necessary calculations were done by central processing unit (CPU). As computer graphics became more complex, CPU got overloaded with graphics computation and the performance of CPU declined rapidly. This trend resulted in the development of certain dedicated hardware for accelerating of graphics computation. This kind of specific hardware is today commonly known as a graphics processing unit (GPU).

The GPU is an electronic circuit specially designed to accelerate creation of images in the display buffer consequently displayed on a display. Modern GPUs possess highly parallel architecture, very efficient in calculations of large blocks of data up to certain size. This size is limited by the size of the GPU memory. The computing power is widely used not only in computer graphics, but also in physical calculations, simulations and generally in high-performance computing. The first generation of GPUs was designed as fixed-function accelerators with a limited set of instructions. The need for higher flexibility resulted in the development of programmable GPUs.

This thesis is built on knowledge gained in bachelor's thesis. Since common topic of both theses is acceleration on GPU, this section was taken from bachelor's thesis [8].

### 2.1 Architecture comparison

The main difference between CPU and GPU is in their architecture. Current CPUs are composed of low tens of cores and supports parallel execution of different processes on CPU at the same time. CPU also contains deep hierarchy of caches which makes them optimized for context switching and complex calculations. On the other hand, GPU provides much greater level of parallelism and therefore much greater throughput. For example, GeForce GTX TITAN is equipped with 2688 cores capable of floating-point operations compared to Intel Haswell E5-2699V3 containing eight cores each of which is equipped with AVX2 capable of producing 32 floating-point operations per cycle [5] [13]. We can see that there is a significant difference in maximum number of operations per cycle for each architecture. However, we have to take in consideration that the clock rate of GPU is about one third of CPU, depending on specific models. Despite this fact, GPU can easily outperform CPU in specific type of problems. In fact, the theoretical single-precision performance of GPU GeForce GTX TITAN is about 5 times greater than the theoretical performance of Intel Haswell E5-2699V3 according to Fig 2.1. It is important to mention that GPUs lack many optimizations, such as long pipelines and out of order execution, important for general-

purpose performance. Thus, not all problems are suitable to be accelerated on the GPU.

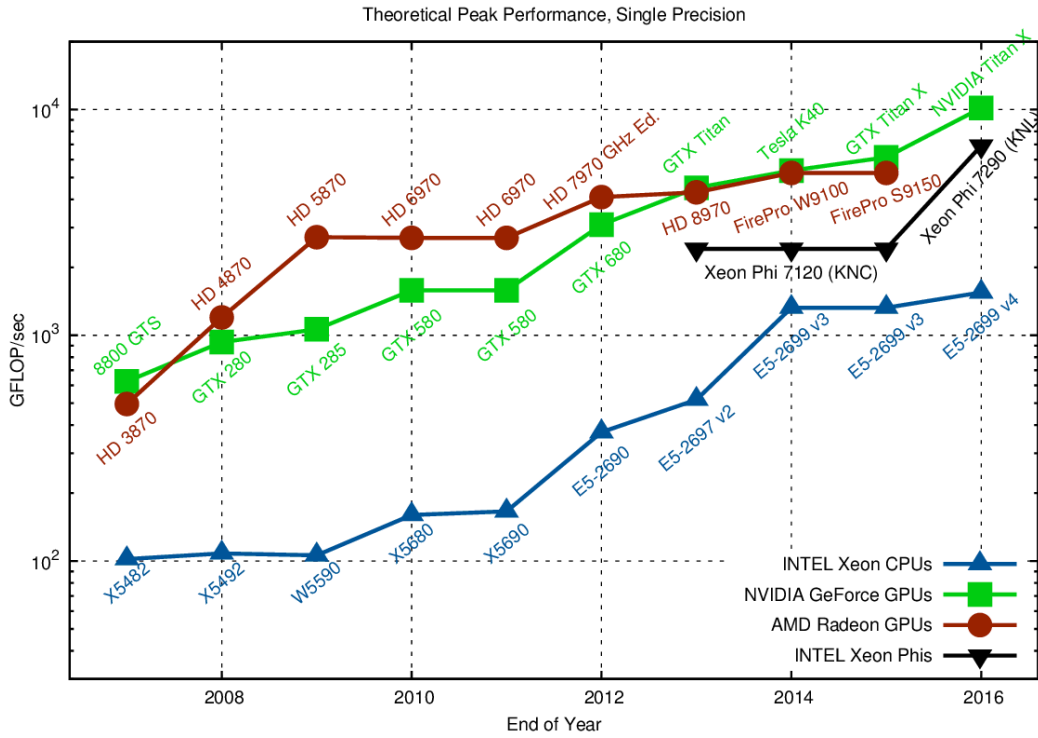


Figure 2.1: Comparison of single-precision peak performance of high-end GPUs, CPUs and accelerators [19].

## 2.2 General Purpose Graphics Computing Unit

The General Purpose Graphics Computing Unit (GPGPU) is a term that refers to a concept in which GPU features are exploited to accelerate computations usually handled by the CPU. This concept is vastly used for an acceleration of calculation involved in fields like bioinformatics, molecular biology, image processing, particle physics and many others [20].

### 2.2.1 GPGPU framework

The GPGPU framework is a platform which contains mechanisms that allows transferring computation on GPU. Two main platforms are open-source OpenCL framework and CUDA framework.

OpenCL is open-source standard for cross-platform parallel programming developed and maintained by Khronos group. Its main purpose is to enable writing applications that can be executed across heterogeneous systems composed of devices such as CPU, GPU, digital signal processor, FPGA and many others. Standard languages as C or C++ can be used for programming purposes with OpenCL. The OpenCL defines API to control and execute code on various devices. It implies that key feature of OpenCL standard is compatibility with various devices created by various vendors [9].

CUDA stands for Compute Unified Device Architecture. It is a proprietary platform for parallel computing and programming model developed by graphics card vendor NVIDIA. It provides mechanisms to write and execute applications exploiting GPU. The main advantage of CUDA comes from the fact that it is a proprietary platform and therefore CUDA is optimized for the use with NVIDIA GPUs [11].

For the purposes of this thesis, CUDA platform has been chosen because CUDA provides better results than OpenCL when used with NVIDIA graphic card present in our testing environment.

## 2.3 CUDA-capable GPU architectures

CUDA is supported by six different microarchitectures [25]:

- Tesla microarchitecture firstly presented in 2006, with GeForce 8800 GTX,
- Fermi microarchitecture firstly presented in 2010, with GeForce GTX 480,
- Kepler microarchitecture firstly presented in 2012, with GeForce GTX 680,
- Maxwell microarchitecture firstly presented in 2014, with GeForce GTX 750,
- Pascal microarchitecture firstly presented in 2016, with Tesla P100,
- Volta microarchitecture expected in 2018,

Although, there is a difference between each architecture, all architectures possesses common hardware features:

- host interface that connects GPU with CPU via PCI-Express bus,
- copy engines,
- DRAM adapter, which interconnect GPU and its device memory,
- device memory and caches,
- certain number of execution units organized in structures called streaming multiprocessors.

Some of mentioned components can be seen in Fig. 2.2

### 2.3.1 Host interface

Host interface is part of the GPU which handles all communication between CPU and GPU. Its only purpose is to receive commands via PCI-Express and decode and delegate these commands further into GPU.

### 2.3.2 Copy engine

Copy engine is hardware capable of performing memory transfers between CPU and GPU while computation is being done on GPU. First microarchitectures do not feature copy engines. Later on, copy engines were only capable of transferring linear device memory. Today, GPUs are equipped with up to two copy engines, which can convert data between CUDA arrays and linear memory. Two copy engines provide full-duplex memory transfers.

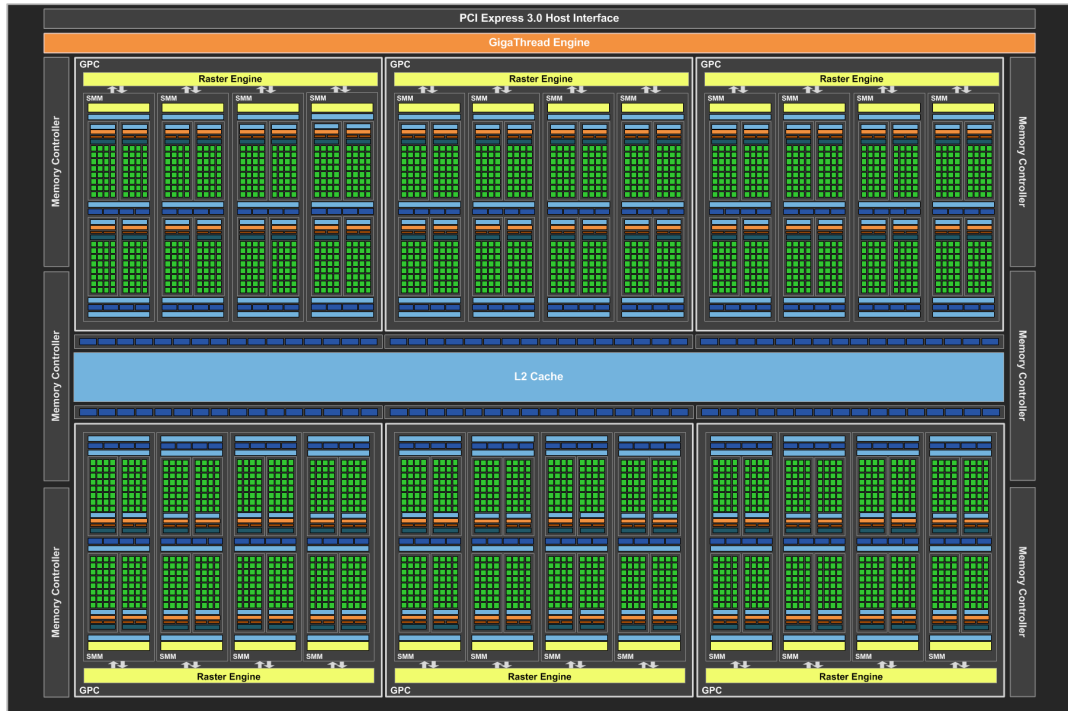


Figure 2.2: Composition of NVIDIA Tesla P100 [14].

### 2.3.3 DRAM adapter

Memory operations bandwidth and latency have a great impact on the GPU performance, therefore GPUs possess powerful DRAM providing hundreds of gigabytes per second of bandwidth and includes hardware support for merging multiple memory operations. Early hardware required contiguous memory addresses and memory alignment. In later versions, requirement for memory alignment was removed. However, there is still a performance penalty.

### 2.3.4 Device memory

The device memory is an equivalent of main memory of CPU. In this memory all data transferred from CPU is stored. For example, NVIDIA Tesla K20 is equipped with GDDR5 memory with a capacity of 5 GB and the throughput of 208 GB/s [10]. The global memory is cumbersome and slow, therefore L2 cache is present in modern GPUs to enhance performance of memory sub-system.

### 2.3.5 Streaming multiprocessor

The main component of GPU is a streaming multiprocessor (SM), which is in charge of all computations. The number of SMs on card is model-specific but the architecture of SM remains in the main the same. Each multiprocessor consists of:

- execution units capable of 32-bit integer, single-precision and double-precision floating-point arithmetic,

- special function units for computing single-precision approximations of mathematical functions (log, exp, sqrt, sin, cos, etc.),
- instruction cache, warp scheduler and dispatch unit for scheduling and dispatching instruction execution by execution units,
- load/store units,
- register field for storing local variables,
- shared memory with L1 cache for communication between threads and storing temporary result,
- constant cache for broadcasting constant variable to each thread,
- cache texture hardware with various functions (1D, 2D, 3D prefetching, interpolation etc.).

## 2.4 CUDA thread execution model

Thread arrangement, when done wrongly, can have a severe negative impact on execution time of application using CUDA and therefore it is good to keep in mind few basic principles of CUDA thread execution model. This section is dedicated to clarification of this model depicted on Fig. 2.3.

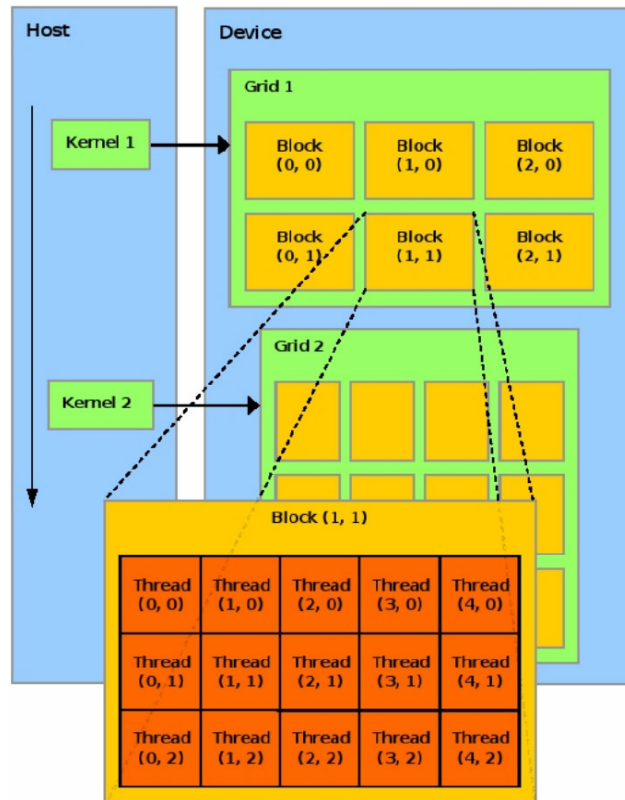


Figure 2.3: CUDA thread execution model [12].



### 2.4.1 Kernel

The kernel is an equivalent of procedures found in common programming languages. Kernels are part of the application that are computed on GPU. Kernel is declared by keyword `__global__`. The launch of the kernel is similar to a traditional function or procedure call, the only difference is the presence of a special triple angle bracket construction (also known as kernel configuration) in which grid size and block size are specified. In most cases, once an optimal size of block is experimentally obtained (value depends on certain GPU and implementation), grid size is adjusted accordingly to the size of problem. Most of the time, the goal is to create ideally the same number of threads in grid as the number of elements in domain.

### 2.4.2 Grid

The grid encapsulates all threads which are invoked by launching a kernel. Its size is specified in the number of blocks in three dimensions. The blocks within the grid tend to be assigned on different SM to maximize performance, although a few different blocks can reside on the same SM. The maximum size of the grid can be up to  $65535^2$  blocks for 1.x computation capable hardware and  $65535^3$  blocks for 2.x computation capable hardware.

### 2.4.3 Block

The block is an abstraction of independent execution unit. It is a group of threads which are executed on the same SM. CUDA only allows per-block resource allocation and therefore overall achieved occupancy of kernel is strongly dependent on size of block and amount of resources required per block. Achieved occupancy is number of active warps compared to maximum number of active warps on GPU and represents overall utilization of GPU. The main reason why GPU cannot be fully utilized is that the block requires too much resource and therefore only a few whole blocks can fit to SM wasting unallocated resources. In such case, a logical step would be decreasing amount of threads per block. However this is only beneficial to certain point because block hides memory latencies by context switching and when the number of threads in block is too low there is chance that block would be not able to find a set of threads which is ready to execute and thus not able to hide memory latencies. It is also good to keep in mind that threads are only able to communicate with each other within block and therefore CUDA provides a mechanisms for inter-block communication and synchronization. The block size can also be specified in three dimensions and the block can contain up to 512 threads in total for 1.x computation capable hardware and 1024 threads in total for 2.x and above computation capable hardware.

### 2.4.4 Thread

The thread is an elementary part of execution. Each thread has its own unique identification within the block. Resolving global identification of the thread is essential, as it is the only mechanism to assign correct portion of the work to the thread. To help resolving global identification of thread, built-in variables of type `dim3` are available for each thread. The `dim3` type consists of 3 integer variables, each for one dimension:

- `gridDim` specifies dimensions of the grid in blocks,
- `blockDim` specifies dimensions of the block in threads,

- blockIdx specifies the index of a certain block within the grid,
- threadIdx specifies the index of a certain thread within the block.

Then statement for computing global index, supposing the grid and the block are defined only in one dimension, should look as follows:

```
globalIdx = (blockIdx.x*blockDim.x) + threadIdx.x;
```

If the grid and the block are both defined in all three dimensions, indices have to be calculated separately for each dimension:

```
globalIdx.x = (blockIdx.x*blockDim.x) + threadIdx.x;
globalIdx.y = (blockIdx.y*blockDim.y) + threadIdx.y;
globalIdx.z = (blockIdx.z*blockDim.z) + threadIdx.z;
```

If there is a need to calculate a flatten 3D index within the block, especially when accessing shared memory, it can be calculated as follows:

```
localIdx = threadIdx.z*blockDim.y*blockDim.x;
localIdx += threadIdx.y*blockDim.x;
localIdx += threadIdx.x;
```

Sometimes, there is more work for the kernel that can all the threads in the grid process in one go. Then the global index have to be recalculated as follows:

```
globalIdx += blockDim.x*blockDim.x;
```

### 2.4.5 Warp and lane

Threads are executed simultaneously (SIMD-like) in 32-thread packs called warps which are atomic element of execution. All 32 threads execute the same instruction, therefore it is recommended that number of threads in block is divisible by 32 otherwise one or more threads of last warp of block would not execute any code. The number of the thread within the warp is called the lane. Both values warp id and lane id can be computed from the local id of the thread.

```
warpIdx = localIdx / 32;
laneIdx = localIdx & 31;
```

Warps are the part of the mechanism of covering memory latencies. When one warp reaches an instruction resulting in, for example, global memory access, which can last for hundreds of clocks cycles, warp scheduler activates different warp until data transfer is over. It is important to say that SM should have enough pending warps ready to be executed in order to effectively overlap latencies with calculation otherwise stalls are inevitable.

## 2.5 CUDA memory model

The CUDA platform offers various types of the memory, whether it is physical or logical memory. Each of these memories serves different purpose and has its own advantages and disadvantages.

### 2.5.1 Global memory

It is a physical memory which creates the main memory pool for GPU. It means that it is accessible from each GPU thread. All data transferred from the CPU onto the GPU resides in this memory, and therefore, each application running on the GPU needs to access global memory at some point. This memory has the greatest capacity and the lowest bandwidth compared to all other physical memories present in the GPU. For example, NVIDIA Tesla K20 is equipped with the GDDR5 memory with capacity 5 GB and throughput of 208 GB/s, as was mentioned in the Sec. 2.3.4.

To reduce the impact of the low bandwidth, global memory is accessed through the L2 cache and SM's private L1 cache. By turning L1 caching on and off, we can influence global memory load granularity. If L1 caching is enabled, the size of memory transaction is 128B, otherwise is 32B. These transactions are aligned to 128B or 32B respectively. When warp executed operation results in the global memory access, memory sub-system tries to merge memory transfers to as few transfers as possible. For example, the warp is requesting 32 consecutive 4B floats aligned to 128B. Supposing L1 caching is turned on, this 128 byte memory request will be satisfied with one global memory transfer. However, if the memory request is not aligned to 128B, two memory transfers are needed. In the worst case, if each of these 32 floats is situated in different 128B block, the transaction is satisfied by 32 global memory transfers. Therefore, it is crucial to ensure a sensible access pattern in the application in order to achieve maximum performance. Setting specific caching options can lower penalty for unpredictable access patterns.

### 2.5.2 Local memory

The local memory is a logical memory used for the local variables when the block requires more space for its local variables than the SM offers. This memory is accessible only by the thread and it is situated in the global memory pool, and therefore, it has the same properties.

### 2.5.3 Registers

Each SM has its own field of registers. These registers are used to store local variables of each thread which resides on the SM. The registers are private to the thread. Registers are the fastest memory type.

### 2.5.4 Shared memory

The shared memory is present on each SM and shares the same memory pool with the L1 cache. Shared memory is accessible for each thread within the same block often used as a mean of inter-block communication. The shared memory is also often used to reduce the penalty caused by scattered access patterns into the global memory. In order to reduce the penalty, a block firstly has to load the values from the global memory with coalesced access pattern, store them in the shared memory, and then access the data in faster shared memory. However, shared memory bandwidth is also affected by scattered access pattern. For devices with the compute capability 2.x and above, shared memory is divided into 32 banks. In the shared memory, 32 consecutive 4-byte values are assigned to 32 banks. If each thread within the block requests a value assigned to the different bank, the transfer is done in one transaction. On the other hand, if we access shared memory with the stride

equal to 2, it leads to a two-way bank conflict. The two-way bank conflict is a situation where two different threads request values at different memory addresses assigned to the same bank, and therefore, the memory transfer is carried out in two transactions. In the worst case, 32-way bank conflict can occur resulting in 32 memory transactions. However, if all 32 threads of the active warp request the value from the same address, this request is satisfied in one transaction in the broadcast fashion.

### **2.5.5 Texture memory**

Although this type of memory is located in the global memory, it is cached and accessed via dedicated hardware present in each SM. This memory is accessible from each thread. The texture hardware has some interesting properties. It is able to perform interpolation in 1D, 2D or 3D and whenever some value is requested, the texture hardware also prefetches surrounding values based on their position in 1D, 2D or 3D array. Due to prefetching, texture memory can be exploited to reduce impact of some access pattern.

### **2.5.6 Constant memory**

The constant memory is special 64KB read-only memory and it is immutable by kernel and therefore has to be initialized prior to the kernel launch. This memory is cached with dedicated cache and it is capable broadcast.

## Chapter 3

# Reference Implementation

### 3.1 The k-Wave implementation

As mentioned earlier, k-Wave is MATLAB toolbox for simulation of acoustic wave propagation. The simulation of wave propagation in elastic medium is also a part of the toolbox. The main goal of this thesis is to accelerate mentioned simulation on GPU. Although, k-Wave implements a lot of different models, basic simulation execution is the same. User has to specify four main parameters: grid, medium, source and sensor. The grid is a structure created by utility function which is part of toolbox. This structure represents domain in which entire simulation takes place. The grid is specified by the number of points in each dimension (3D in this case) and their spacing. Evenly spaced array of time values is also a part of grid structure. Another user defined structure is medium. In elastic simulation, it is possible to specify these properties of medium: compressional sound speed distribution, shear sound speed distribution, density distribution, absorption coefficient for compressional waves and absorption coefficient for shear waves. Either of these parameters can be specified as a matrix with same dimensions as computational grid (heterogeneous medium) or as a scalar (homogeneous medium). When both absorption coefficients are not specified, simulation is lossless. Source structure specifies position within the medium, type of source, mode of source and signal generated by the source. Source can be either stress source or velocity source or even both. Source can have two modes: additive mode adds current value of source signal to calculated value of pressure or velocity in source position and Dirichlet mode forces source value of respective type in source position. It is possible to specify time-varying signal for each source point or have one common for all points. Using sensor structure, users are able to highlight an area of domain which they are interested in and specify quantities which they want to record (for example maximal pressure, minimal pressure, final distribution of particle velocity, etc.). Afterwards, users are able run simulation by calling a simulation function with mentioned parameters. In case of 3D elastic simulation, it is `pstdElastic3D`. As first, input parameters are checked and preprocessing takes place. Afterwards, simulation is executed in iterations according to time array specified in grid structure. Iteration is composed of following operations: computation the gradients of the stress tensor, calculation of particle velocity in next time step according to gradients of stress tensor, application of velocity source if specified, calculation of the velocity gradients, calculation of spatial gradients of time derivative of particle velocity provided medium is absorbing, computation of normal and shear components of the stress tensor in next step using lossless or Kelvin-Voigt model, application of stress source if specified, and finally computation of pressure using normal components of stress. Governing equations of

simulation loop can be seen in next section. After the end of simulation, the data specified by sensor structure are collected, processed and optionally displayed to user (some data may be gathered throughout simulation).

This implementation is using Fourier pseudospectral method to compute gradients. The difference between local and global method (Fourier pseudospectral method) can be seen in Fig. 3.1. This method of calculation of spatial gradients has its benefits (reduction number of needed grid points per wavelength while preserving the same accuracy as local methods), however, it also has some drawbacks. The use of the FFT to calculate spatial gradients implies that the domain is periodic. This causes waves leaving one side of the domain to reappear at the opposite side. The wave wrapping caused by the FFT can be largely eliminated by the use of a Perfectly Matched Layer (PML). This is a thin absorbing layer that encloses the computational domain and cause anisotropic absorption at the domain edges [21]. Another important fact to mention is that k-Wave is using staggered grid to achieve additional accuracy. Staggered grid is a simple way to avoid odd-even decoupling between the pressure and velocity. Odd-even decoupling is a discretization error that can occur on collocated grids and which leads to checkerboard patterns in the solutions [4]. The principles of staggered grid is shown in Fig. 3.2.

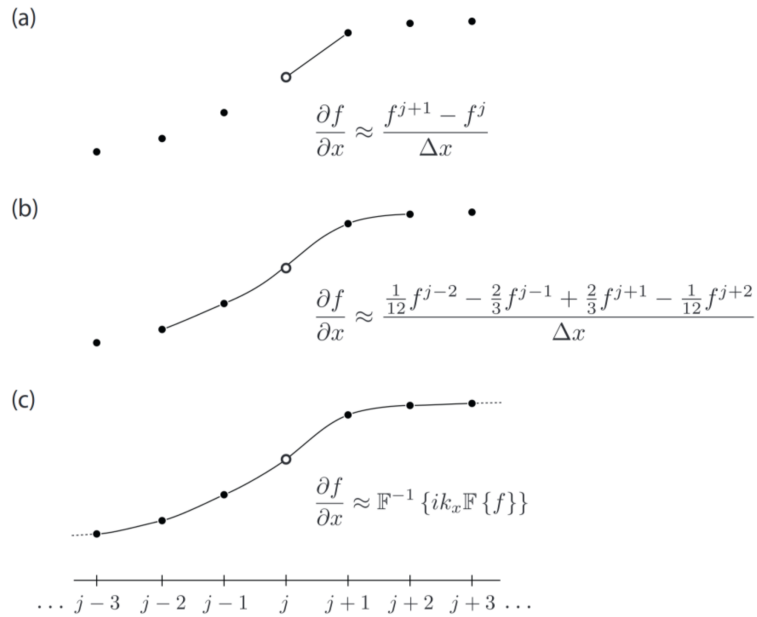


Figure 3.1: Calculation of spatial gradients using local and global methods. (a) First-order accurate forward difference. (b) Fourth-order accurate central difference. (c) Fourier collocation spectral method [21].

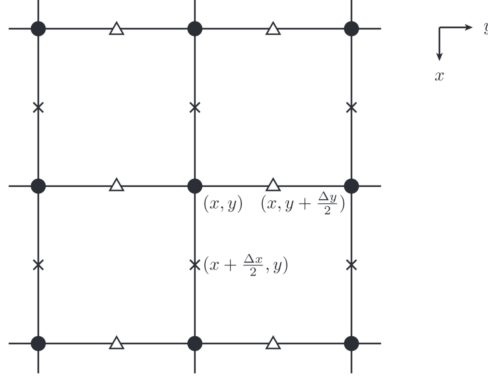


Figure 3.2: Schematic showing the principles of using a staggered spatial grid in 2D. Here  $\partial_x \sigma_{xx}$  is evaluated at grid points staggered in the x-direction (crosses), while  $\partial_x \sigma_{yy}$  evaluated at grid points staggered in the y-direction (triangles) [21].

### 3.1.1 Governing equations

In this section, governing equations (which are used in k-Wave 2D elastic simulation) will be presented and described. This section was taken from a paper by The Biomedical Ultrasound Group at University College London [23].

Following equations represent only 2D model, however, extension into 3D is not complicated. Fundamental equations for studying lossy wave propagation are derived from Hook's law. Provided modeling of viscoelasticity is based on Kelvin-Voigt model, resulting equation can be written using Einstein summation as follows

$$\sigma_{ij} = \lambda \delta_{ij} \epsilon_{kk} + 2\mu \epsilon_{ij} + \chi \delta \frac{\partial}{\partial t} \epsilon_{kk} + 2\eta \frac{\partial}{\partial t} \epsilon_{ij} \quad , \quad (3.1)$$

where  $\sigma$  is stress tensor,  $\epsilon$  is dimensionless strain tensor,  $\lambda$  and  $\mu$  are Lamé parameters. Here  $\mu$  is ratio of shear stress to shear strain. Coefficients  $\chi$  and  $\eta$  represent compressional and shear viscosity respectively. The Lamé parameter are related to compressional and shear sound speed of medium by

$$\mu = c_s^2 \rho_0, \quad \lambda + 2\mu = c_p^2 \rho_0, \quad (3.2)$$

where  $\rho_0$  is mass density. If a relation between strain and particle displacement  $u_i$

$$\epsilon_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \quad (3.3)$$

is used, then Eq. (3.1) can be re-written as function of a particle velocity  $v_i$ , where  $v_i = \partial u_i / \partial t$

$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda \delta_{ij} \frac{\partial v_k}{\partial x_k} + \mu \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) + \chi \delta_{ij} \frac{\partial^2 v_k}{\partial x_k \partial t} + \eta \left( \frac{\partial^2 v_i}{\partial x_j \partial t} + \frac{\partial^2 v_j}{\partial x_i \partial t} \right) \quad (3.4)$$

To be able to simulate wave propagation in elastic medium, Eq. (3.4) is combined with equation representing momentum preservation. The equation is a function of stress and particle velocity and it is given by the relation

$$\frac{\partial v_i}{\partial t} = \frac{1}{\rho_0} \frac{\partial \sigma_{ij}}{\partial x_j} \quad (3.5)$$

Eqs. (3.4) and (3.5) are coupled first-order partial differential equations which model pressure waves propagation in isotropic viscoelastic medium.

A computationally efficient simulation model can be created using coupled differential Eqs. (3.4) and (3.5), and Fourier pseudospectral method. This method uses Fourier collocation spectral method to calculate spatial derivatives and a finite-difference method to integrate in time domain. Using presented model a step of simulation is composed of several operations.

As first, using the Fourier collocation spectral method a spatial gradients of stress field is calculated

$$\begin{aligned} \partial_x \sigma_{xx}^- &= \mathcal{F}_x^{-1} \left\{ i k_x e^{+i k_x \Delta x / 2} \mathcal{F}_x \{ \sigma_{xx}^- \} \right\} \\ \partial_y \sigma_{yy}^- &= \mathcal{F}_y^{-1} \left\{ i k_y e^{+i k_y \Delta y / 2} \mathcal{F}_y \{ \sigma_{yy}^- \} \right\} \\ \partial_x \sigma_{xy}^- &= \mathcal{F}_x^{-1} \left\{ i k_x e^{-i k_x \Delta x / 2} \mathcal{F}_x \{ \sigma_{xy}^- \} \right\} \\ \partial_y \sigma_{xy}^- &= \mathcal{F}_y^{-1} \left\{ i k_y e^{-i k_y \Delta y / 2} \mathcal{F}_y \{ \sigma_{xy}^- \} \right\} \end{aligned} \quad (3.6)$$

Here  $\mathcal{F}_{x,y} \{ \}$  and  $\mathcal{F}_{x,y}^{-1} \{ \}$  are 1D forward and inverse Fourier transformations in  $x$  and  $y$  dimensions,  $i$  is the imaginary unit,  $k_x$  and  $k_y$  are discrete set of wavenumber in  $x$  and  $y$  dimension respectively, and  $\Delta x$  and  $\Delta y$  represents spacing of grid points in uniform Cartesian mesh. In order to achieve higher precision, variables are stored in staggered grid and therefore the exponential terms are spatial shift operators which translate the output by half the grid point spacing.

Then the particle velocity is updated using finite-difference time step  $\Delta t$

$$\begin{aligned} v_x^+ &= v_x^- + \frac{\Delta t}{\rho_0} (\partial_x \sigma_{xx}^- + \partial_y \sigma_{xy}^-) \\ v_y^+ &= v_y^- + \frac{\Delta t}{\rho_0} (\partial_x \sigma_{xy}^- + \partial_y \sigma_{yy}^-) \end{aligned} \quad (3.7)$$

Here superscripts  $-$  and  $+$  denote value at current and next time step respectively.

Afterwards the spatial gradients of updated particle velocity is calculated again using Fourier collocation spectral method

$$\begin{aligned} \partial_x v_x^+ &= \mathcal{F}_x^{-1} \left\{ i k_x e^{-i k_x \Delta x / 2} \mathcal{F}_x \{ v_x^+ \} \right\} \\ \partial_y v_x^+ &= \mathcal{F}_y^{-1} \left\{ i k_y e^{+i k_y \Delta y / 2} \mathcal{F}_y \{ v_x^+ \} \right\} \\ \partial_x v_y^+ &= \mathcal{F}_x^{-1} \left\{ i k_x e^{+i k_x \Delta x / 2} \mathcal{F}_x \{ v_y^+ \} \right\} \\ \partial_y v_y^+ &= \mathcal{F}_y^{-1} \left\{ i k_y e^{-i k_y \Delta y / 2} \mathcal{F}_y \{ v_y^+ \} \right\} \end{aligned} \quad (3.8)$$



Then the spatial gradients of time derivative of particle velocity are calculated using Eq. (3.5)

$$\begin{aligned}
\partial_x \partial_t v_x^- &= \mathcal{F}_x^{-1} \left\{ ik_x e^{-ik_x \Delta x / 2} \mathcal{F}_x \left\{ (\partial_x \sigma_{xx}^- + \partial_y \sigma_{xy}^-) / \rho_0 \right\} \right\} \\
\partial_y \partial_t v_x^- &= \mathcal{F}_y^{-1} \left\{ ik_y e^{+ik_y \Delta y / 2} \mathcal{F}_y \left\{ (\partial_x \sigma_{xx}^- + \partial_y \sigma_{xy}^-) / \rho_0 \right\} \right\} \\
\partial_x \partial_t v_y^- &= \mathcal{F}_x^{-1} \left\{ ik_x e^{+ik_x \Delta x / 2} \mathcal{F}_x \left\{ (\partial_x \sigma_{xy}^- + \partial_y \sigma_{yy}^-) / \rho_0 \right\} \right\} \\
\partial_y \partial_t v_y^- &= \mathcal{F}_y^{-1} \left\{ ik_y e^{-ik_y \Delta y / 2} \mathcal{F}_y \left\{ (\partial_x \sigma_{xy}^- + \partial_y \sigma_{yy}^-) / \rho_0 \right\} \right\}
\end{aligned} \tag{3.9}$$

Finally the stress field is updated using a finite-difference time scheme

$$\begin{aligned}
\sigma_{xx}^+ &= \sigma_{xx}^- + \lambda \Delta t (\partial_x v_x^+ + \partial_y v_y^+) + \mu \Delta t (2\partial_x v_x^+) + \chi \Delta t (\partial_x \partial_t v_x^- + \partial_y \partial_t v_y^-) \\
&\quad + \eta \Delta t (2\partial_x \partial_t v_x^-) \\
\sigma_{yy}^+ &= \sigma_{yy}^- + \lambda \Delta t (\partial_x v_x^+ + \partial_y v_y^+) + \mu \Delta t (2\partial_y v_y^+) + \chi \Delta t (\partial_x \partial_t v_x^- + \partial_y \partial_t v_y^-) \\
&\quad + \eta \Delta t (2\partial_y \partial_t v_y^-) \\
\sigma_{xy}^+ &= \sigma_{xy}^- + \mu \Delta t (\partial_y v_x^+ + \partial_x v_y^+) + \eta \Delta t (\partial_y \partial_t v_x^- + \partial_x \partial_t v_y^-)
\end{aligned} \tag{3.10}$$

## 3.2 Implementation of governing equations in MATLAB

In this section, all functions in main the simulation loop will be mentioned and described on representative example. Only a sample of code is presented due to the fact that simulation, being 3D in nature, uses most of the code repetitively, and therefore, the same operation is repeated for each of the Cartesian components.

### 3.2.1 Computation of gradients of stress tensor

Derived from Eq. (3.6) MATLAB code for calculation of spatial gradient of stress tensor look as follows

```
dsxxdx = real( ifft( bsxfun(@times, ddx_k_shift_pos, ...
    fft(sxx_split_x + sxx_split_y + sxx_split_z, [], 1)), [], 1) );
```

Here `fft(data, [], 1)` represent a 1D fast Fourier transform in the axis given by the third parameter (1 for  $x$ , 2 for  $y$ , 3 for  $z$ ), `ifft()` represent an inverse fast Fourier transform, `bsxfun(@times, vector, matrix)` is a function for element-wise matrix-by-vector multiplication, `ddx_k_shift_pos` refers to an exponential term from Eq. (3.6), and `real(data)` extracts the real part of the complex numbers. From The CUDA implementation point of view, this code requires two kernels. The first kernel to add split-field values together, and second to multiply matrix by vector element-wise.

### 3.2.2 Computation of split-field particle velocity

This code snippet refers to Eq. (3.7). It uses stress calculated by the kernel presented in Sec. 3.2.1 in form of

```
ux_split_x = bsxfun(@times, mpml_z, bsxfun(@times, mpml_y, ...
    bsxfun(@times, pml_x_sgx, bsxfun(@times, mpml_z, ...
    bsxfun(@times, mpml_y, bsxfun(@times, pml_x_sgx, ux_split_x)))...
    + dt.* rho0_sgx_inv .*dsxxdx));
```

This operation involves element-wise matrix-by-vector multiplication, matrix-by-constant multiplication, element-wise matrix multiplication and matrix addition. In implemented kernel values of `mpml_x`, `mpml_y` and `mpml_z` could be most probably reused. This may result in a better data temporal locality than in the former case. Variables `mpml_x`, `mpml_y` and `mpml_z` represent Multi-axial Perfectly Matched Layer mentioned in Sec. 3.1.

### 3.2.3 Spatial velocity gradient calculation

Calculation of spatial velocity gradient implementing Eq. (3.8) is performed by

```
duxdx = real( ifft( bsxfun(@times, ddx_k_shift_neg, ...
                    fft(ux_sgx, [], 1)), [], 1));
```

It is clear that this function is almost identical to the one presented in Sec. 3.2.1, and therefore, the same assumption can be made. However, in this case, no matrix addition is performed so only one kernel is needed. Moreover, there is good chance that the kernel implementing the code from Sec. 3.2.1 can be reused.

### 3.2.4 Spatial gradients of the time derivative of the velocity

This code directly refers to Eq. (3.9) and looks as follows:

```
dduxdxdt = real(ifft( bsxfun(@times, ddx_k_shift_neg, ...
                        fft((dsxxdx + dsxydy + dsxzdz).*rho0_sgx_inv, [], 1 )), [], 1));
```

It is obvious that this function shares a lot of common features with function presented in Sec. 3.2.1. It can be said that for this computation an slightly modified of kernel implementing code in that section can be used. Also the same assumptions, stated in Sec. 3.2.1, are valid for this function.

### 3.2.5 Computation of stress tensor values in next time step

Finally, the computation of stress field for next time step derived from Eq. (3.10) has a form of

```
sxx_split_x = bsxfun(@times, mpml_z, bsxfun(@times, mpml_y, ...
        bsxfun(@times, pml_x, bsxfun(@times, mpml_z, ...
        bsxfun(@times, mpml_y, bsxfun(@times, pml_x, sxx_split_x))) ...
        + dt.*(2*mu + lambda).*duxdx ...
        + dt.*(2*eta + chi).*dduxdxdt));
```

This is the most complex computation, but still composed of the same operations as previous functions.

### 3.2.6 Conclusion

Functions, presented in Sec. 3.2, are not the only computation which will be handled by GPU. There are also functions to simulate source of stress or velocity and also other utility functions needed by other kernels, but these are not mentioned in this section for their simplicity.

It is easy to spot that all functions are composed of several basic operations namely matrix addition, element-wise matrix-by-vector multiplication, matrix-by-scalar multiplication and element-wise matrix multiplication. All this operations are fairly simple and do not involve a lot of computation. Therefore all kernel will probably exhibit low arithmetic intensity. Due to this property it is estimated that implementation will be memory-bounded.

### 3.3 Existing CUDA framework

As was mentioned in Sec. 3.1, simulation of wave propagation in elastic medium is a part of k-Wave toolbox. Another part of this toolbox is a fluid medium simulation with stand-alone CUDA/C++ accelerated code (hereinafter referred to as fluid code) created by Dr. Jaroš. The fluid code is capable of running small to moderate ( $64^3$  to  $512^3$  grid points) simulations. This code is part of the k-Wave project and it is implemented in such way that it can be seamlessly incorporated into the toolbox and ran from MATLAB script using a few additional parameters. In k-Wave, there is sub-script, which is able to create HDF5 input file for fluid code based on user-specified parameters. General use-case is then composed of these steps: User specifies initial parameters of simulation (this is closely discussed in Sec. 3.1), user selects fluid code as an acceleration of simulation by running wrapper function (wrapper function has the same interface as k-Wave simulation), k-Wave sub-script generates HDF5 input file with all necessary data and launches the fluid code, fluid code runs the simulation (providing user have previously correctly installed the fluid code) and produces an HDF5 output file, and k-Wave loads output file and presents user with the results. This process is absolutely transparent to users and requires only prior installation of the module and slight change in simulation script.

Due to similar nature of both simulations (fluid simulation and elastic simulation) and to provide the same ability to include acceleration in k-Wave, it has been decided that for purpose of implementation of elastic simulation, the fluid code will serve as the framework. Keeping the same structure and interface (as existing fluid code) is also one of the requirements of implemented code.

#### 3.3.1 Main components of framework

The most important classes of fluid code are mentioned in this section. Please note, that this is not exhaustive list of all parts of the framework.

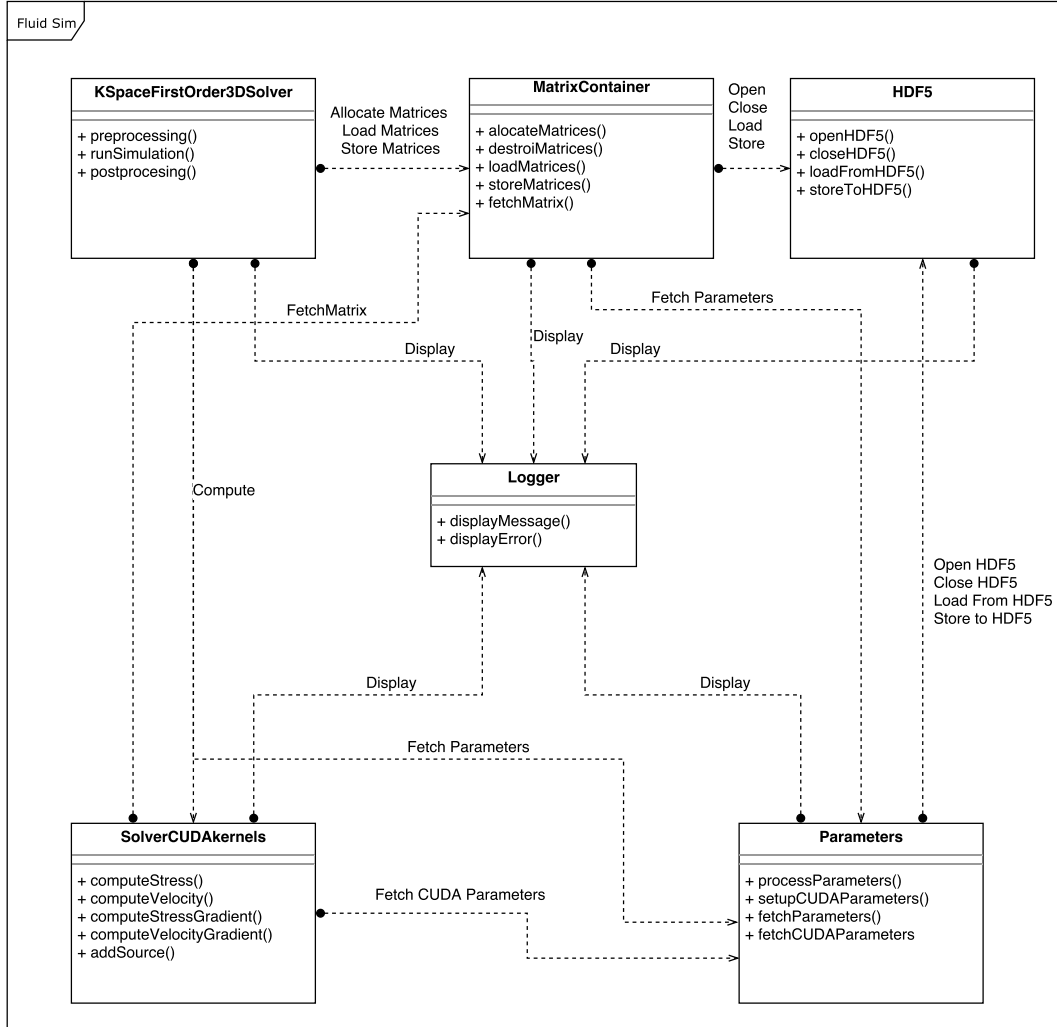


Figure 3.3: Diagram of main classes which are part of existing CUDA framework including: main `KSpaceFirstOrder3DSolver` class, `SolverCUDAkernels` class containing computational kernels, `MatrixContainer` class responsible for matrix operation, `Parameters` class, `HDF5` class which manipulates HDF5 files and `Logger` class to format and display messages

The Fig. 3.3 represents simplified structure of fluid code. The `KSpaceFirstOrder3DSolver` class represents the main class responsible for handling entire simulation. In this class, the preprocessing, the postprocessing as well as simulation loop methods are implemented. In preprocessing phase `KSpaceFirstOrder3DSolver` invokes methods of `MatrixContainer` to allocate the memory and load the data into matrices needed during the computation. Solver class also implements methods to precalculate variables which needs to be available prior to simulation launch. Most of these variables have to be calculated only once and are calculated by CPU. After preprocessing phase is completed, simulation phase begins. In this stage, `KSpaceFirstOrder3DSolver` invokes appropriate methods, implemented in class `SolverCUDAkernels`, one-by-one according to model. This process is then repeated based on time array provided by `Parameters` class. Once simulation is over, main class invokes procedures in `MatrixContainer` to store data specified by parameters (the data storing may also occur during the simulation) and free memory. Afterwards application terminates.

The `SolverCUDAkernels` is class implementing all computations which occur during simulation on GPU. Whenever one of its methods is invoked by main class, `SolverCUDAkernels` calls appropriate computational kernel according to input parameters (linear or nonlinear simulation, homogeneous or heterogeneous medium) with appropriate data provided by `MatrixContainer` class.

The `MatrixContainer` class is responsible for data handling. This class implements methods to create, allocate, destroy and load data into matrices. The create method basically adds matrix records consisting of matrix name, type and size into list. The size of matrices is calculated based on domain size provided by `Parameters` class. Creation of some matrices is even conditioned by values of some parameters. Matrices from the list are then allocated by allocate method. For each matrix, the memory on CPU as well as GPU is allocated. The `MatrixContainer` is responsible for data transfers between CPU and GPU. It also holds references to all the matrices and it is able to serve pointers to CPU or GPU memory of each matrix.

The `Parameters` class process command line parameters, loads parameters from input file, sets up device constants and serves other classes with these parameters. This class is modeled as a singleton and at some point almost every other class in framework obtains a reference to the `Parameters`.

The `Logger` class is responsible for all outputs to standard and error output. It also implements methods for messages formatting.

Class `HDF5` implements the simplified interface to operate with HDF5 files allowing easy manipulation, loading and storing the data.

# Chapter 4

## Implementation

### 4.1 Implementation breakdown into tasks

These steps are necessary to implement acceleration of simulation of sound wave propagation in elastic medium (hereinafter referred to as elastic code):

- implementation of kernels discussed in Sec. 3.2 on GPU,
- modification of Input/Output (I/O) file format,
- integration with the framework
- integration with the k-Wave
- accuracy testing
- performance testing

### 4.2 Technologies used in development

#### 4.2.1 The Google test

Google test is a C++ language unit test framework developed by Google. This framework follows the scheme of xUnit, which is a collective name for unit testing frameworks derived from SUnit developed for Smaltalk and designed by Kent Beck in 1998 [26]. The xUnit frameworks have a common architecture composed of:

- test runner - application which executes xUnit tests and produces test results
- test case - most elemental class implementing a test
- test fixtures - used to set-up the state needed by the test and afterwards returns to normal state. It is useful for memory allocation prior to the test and cleaning after the test
- test execution - execution of the test itself
- test result formatter - transforms the test runner output to specific the format, most commonly, XML

- assertion - function or macro used to verify behavior of a unit under the test. Usually has a form of a logical condition that compares the actual value produced by unit to the expected value. Failing the assertion usually results in an exception being thrown.

The Google test was chosen as the test framework for several reasons. Firstly, the framework is released under the BSD license. This means it is free to use, and therefore, ideal for academic projects. Secondly, it has a great variety of assertions which can be divided into two main groups, asserts and expects. The main difference between assert and expect is that the assertion terminates an execution on failure while expect does not. Google test also supports user defined assertions. The feature that makes the Google test stand out of crowd, is so-called death tests which enable checking the return value and the error message in case of an failure of application.

### 4.2.2 The HDF5 library

In real-life applications, the domains are usually on the order of  $10dm^3$  in size which translates to a Cartesian grid of roughly  $1024^3$  grid points. The number of grid points strongly depends on various properties of simulation, the numbers are just for illustration purpose. To be able to run a simulation, the application needs several matrices representing medium properties, physical quantities, sources, sensors and etc., many of which have the same size as grid. When simulating in MATLAB, all matrices are stored internally and all computation is performed on this data stored in the computer main memory. The problem occurs when data needs to be transferred from MATLAB into file and handed to the accelerated application for processing. The use of standard Unix or Windows files is quit cumbersome. Moreover, those files are incompatible due to file system nuances in many cases. To overcome disadvantages of using the standard files, a HDF5 library have been incorporated into the fluid code. The HDF5 file format is a fifth iteration of hierarchical data format developed by HDF Group [3]. This file format has many advantages just as support for great variety of datatypes, efficient and flexible I/O, portability, extensibility and suitability for high volume and complex data. Another important fact is that MATLAB has a built-in support for HDF5 file format. Thanks to its advantages, HDF5 file format became de-facto industry standard for storing high-volume data, especially in scientific applications.

### 4.2.3 The cuFFT library

The cuFFT library is a Fast Fourier Transform (FFT) library developed by NVIDIA [16]. This library implements a divide-and-conquer approach to effectively compute FFT in parallel on GPU. It is one of the most widely used algorithm to compute such an operation. It is capable of 1D, 2D and 3D forward and inverse transformations on complex or real datasets. The cuFFT library is highly optimized for data sizes that can be written in form of  $2^a \times 3^b \times 5^c \times 7^d$  and the highest performance is on input sizes of power of 2. Another interesting fact is that algorithm exhibits a  $O(n \log n)$  time complexity.

The cuFFT library was chosen as a library to handle FFT due to two main reasons. Firstly, cuFFT is developed by NVIDIA and therefore highly optimized for CUDA graphics cards. The other reason is the implementation of a custom FFT algorithm on GPU is simply out of scope of this thesis.

### 4.3 Compute kernels implementation on GPU

The kernels are the part of framework that is in charge of almost all computations. Kernels implement a physical model of the simulation and are therefore the most important part of the application. The correct implementation of the computation kernels is essential to achieve desired results in terms of performance and precision. Kernels have to be implemented with a deep knowledge of underlying hardware to harness the full potential of modern GPUs and deliver maximum performance.

Almost every implementation was done inside of existing framework. All compute kernels are solely implemented in `SolverCUDAKernel` class (see Sec. 3.3.1 for more information). Due to fact that fluid and elastic code implements different model in terms of required operations, this class has been almost entirely modified. However, some of the kernels was reused, for example kernels for matrix transposition.

#### 4.3.1 Computation of gradients of stress tensor

The computation of the gradient of stress tensor, as presented on Eq. (3.6), uses a pseudo-spectral method to calculate the gradient of a given function. The pseudo-spectral method requires data to be transformed into frequency domain. The whole process can be described in a few independent steps: compute pressure by adding all its components, transform data using the FFT to the frequency domain, perform a shift in frequency domain and multiply the spectrum by the matrix of wavenumbers, and use inverse FFT to transform data back to spatial domain. To be able to calculate gradient of all normal components of stress, described process has to be repeated for each component. Moreover, for each shear component computation has to be done two times. For example,  $\partial_x \sigma_{xy}$  and  $\partial_y \sigma_{xy}$  are calculated from the same shear component of stress tensor  $\sigma_{xy}$ . This computation has to be done 9 times in total to calculate all gradients.

One of the drawback of the cuFFT is that it cannot be called from within another kernel and has to be launched from the host code. It is clear that the computation is divided into 2 FFT and 2 kernel calls. The first kernel is basically a matrix addition with a fairly straightforward implementation. There is no doubt that this kernel has a low arithmetic intensity, and therefore, is bottlenecked by the bandwidth of on-chip memory. The other kernel implements element-wise matrix-by-vector multiplication, where one input is data in spectral domain and other is a precomputed vector of coefficients to perform a shift and wavenumber multiplication in spectral domain.

The cuFFT computes 1D FFT only in  $x$ -axis, therefore before each forward FFT in any other axis, matrix has to be transposed accordingly. This allows to perform element-wise matrix-by-vector multiplication always as if the vector was  $x$ -axis vector. This leads to always optimal pattern when loading data of vector inside the kernel. This fact increase global memory load efficiency. However, matrix transposition itself is very time-consuming operation and its use negates all performance benefits resulting from previously stated fact.

#### 4.3.2 Computation of split-field particle velocity

This kernel is in charge of updating particle velocity according to Eq. (3.7). In this case, no FFT is required to calculate particle velocity, which means that computation is not divided and thus can be implemented using a single kernel. This is desired since kernel launch time is not negligible. On the other hand, computation of one component of velocity tensor (for example  $u_x$ ) is further divided into computation of three Cartesian components,



hence the name split-field particle velocity computation. This decomposition can be seen in Sec. 3.2.2, where `ux_split_x` represents  $x$ -axial Cartesian component of  $u_x$  which is just one part of velocity tensor. Computing in 3D is composed of 9 kernels invocations (3 Cartesian components per velocity vector, 3 velocity vectors in tensor). However, the computations of the velocity components are mutually independent and there is no data dependency, therefore they can be aggregated into fewer kernels. In the actual implementation, computations of all three components of velocity vector were aggregated into one. By doing so, it was possible to create only one CUDA kernel with a reasonable amount of input parameters. The computation of different velocity vectors is then achieved by invoking the kernel with appropriate input parameters. The aggregation of kernels do not just reduced time consumed by the kernels invocation but also the number of memory loads by one third compared to implementation where the kernel would have been invoked for each of the Cartesian components. This optimization may seem to have minor effect on the overall performance of split-field particle velocity computation, but considering the fact that the kernel is composed of a simple mathematical operations with relatively high amount of data, each optimization which would lead to the reduction of memory transfers can be considered significant.

One of the parameters acting in the calculation is density of the medium. However, the k-Wave toolbox allows the definition of the medium density in two variations. It is either a matrix of the same dimensions as a computational grid or a scalar value (heterogeneous or homogeneous medium). To avoid the need for additional kernel version, this kernel is implemented templated. The template variable is a boolean flag signaling the form of the medium density input. Inside the kernel, an ternary operator is used to load a correct value.

### 4.3.3 Spatial velocity gradient calculation

This is implementation of the mathematical operation demonstrated on Eq. (3.8). The implementation is almost identical to the one described in Sec. 4.3.1 with the same assumptions and conclusions. The only difference is the input of an FFT is the same for calculations of all partial derivations of one components of the velocity tensor (for example  $u_x$ ). Then these values can be calculated in advance which saves 6 matrix-addition kernel invocations.

### 4.3.4 Implementation of stress tensor computation

This is the final step of iteration. Here, the values of stress tensor in the next time step are computed based on values computed earlier in the simulation. This implementation refers to Eq. (3.10). The calculation of normal components of a stress tensor is very similar to the implementation discussed in Sec. 4.3.2 with a few differences. The main difference is the computation of a stress field operates with Lamé parameters instead of material density. The k-Wave toolbox allows both Lamé parameters to be defined as matrices with the same dimensions as a computational grid or scalars. This issue is solved in the same way as in Sec. 4.3.2 but this time with two template boolean variables which give four possible variations of the kernel. All these variants are packed into one kernel thanks to templating. Unfortunately, the minor difference between calculation of stress value in different dimension does not allow to use only one kernel with different parameters. Therefore three different kernels has to be implemented.

On the other hand, the calculation of shear components of stress tensor is not that complex and allows to be computed by one kernel. By implementing the same technique

as in Sec. 4.3.2, it is possible to reduce global memory load requests by one third compared to calculation done by separated kernels.

#### 4.3.5 Implementation of velocity and stress sources

So far, the description of the implementation of the whole physical model was presented step by step. However, to be able to simulate a real-life scenarios physical model is insufficient on its own. Users need to have an ability to specify the source of pressure waves. This is why pressure and velocity sources are implemented in the k-Wave version of the elastic simulation. The source is given by a set of indices. These indices defines the placement as well as the shape of sensor. Application of the source signal can be in two modes. The first and default mode is the additive mode meaning the value of signal emitted by source is added to existing value of pressure or velocity at the specific point. The other mode is called Dirichlet. The source in this mode forces its signal value to the specific point. In terms of driving the signal, there are also two different options. The options are: a single time-varying signal applied at all points of source or multiple signals each assigned to a point of source. The implementation of such sources are fairly straightforward. Despite the fact that both kernels implement exactly same functionality, the kernels for velocity and stress source have to be implemented separately due to a difference in the internal data representation. From a performance point of view, it is worth noting that a specifying source position by a set of indices implies double indexing when accessing stress or velocity matrices. This double indexing can have a negative impact on the performance of the kernel in terms of a non-efficient access pattern. This pattern is strongly dependent on the shape of the source. On the other hand, there is one performance optimization implemented for multi-signal sensor. Time-varying signals are not stored in a traditional fashion in  $y$  dimension, where  $x$  dimension is time, but signals are stored in revers order ( $y$  index specifies time,  $x$  index specifies the signal). This way, consecutive threads in warp read signal values that are physically stored consecutively in memory, while updating source points in parallel. This approach balances to some extent disadvantage resulting from the sensor's definition by set of points. However, these kernels consume only a fraction of the computational time and therefore their impact on the overall performance is not significant.

#### 4.3.6 Unit tests of individual kernels

Each kernel is backed up by a unit test to be able to identify errors early on and to support continuous integration. Implementation of unit tests have no relation with any part of existing framework, therefore it was carried out in its own part of project. Implementation includes creation of Google test application into which some of the framework's classes was included. This application allows testing of main components of framework (for example `SolverCUDAkernels`).

Process of creation of typical unit test follows these steps: create input parameters for the kernel, create a reference results using input parameters in the k-Wave simulation, and implement suitable comparison method with the results. The same comparison method was used for most kernel test. As first, the method calculate absolute error of kernel computed value. The absolute error has a form of a matrix. Then for each point, absolute error is normalized by appropriate reference value and checked using Google test assertion to be below a selected value. Sometimes, especially in unit test of function containing FFT calls, absolute error is normalized by maximum value in reference data. This technique allows to spot easily local anomalies in data. To be able to measure the overall precision of the

computation, mean error is calculated and then normalized by mean value of reference data. This error is also tested using assertion.

### 4.3.7 Current limitations of implementation

Due to fact the elastic code is still under development, and is pending to be tested on real-life data, it does not support all features present in the k-Wave. As shown in Sec. 3.2.4, this code snippet computes spatial gradient of time derivatives of particle velocity. This spatial gradient is only needed in lossy Kelvin-Voigt model. However, current version only supports a lossless model, so there is no implementation of this spatial gradient computation. Besides lack of Kelvin-Voigt model, there are few other features k-Wave supports, but the current version elastic code does not. Creating and recovery from checkpoint in the simulation is supported by the framework in fluid code but for elastic code this feature have not been tested and most probably requires additional modifications.

## 4.4 Modification of I/O file format

In the k-Wave framework, there is effort to provide the unified interface for all simulation functions, therefore fluid and elastic code shares a lot of common input variables. However, some changes were inevitable. Simulation of wave propagation in elastic medium requires, compared to fluid simulation, additional 6 PML operators (multi-axial PML operators) and 2 Lamé parameters. On the other hand, elastic code does not need the  $B/A$  nonlinear medium property variable, because only linear propagation is currently supported. Another differences is addition of stress source and absence of transducer source in elastic code. The list of the differences is quit extensive, thus the format of input file is appended to this document (for detail information see Appendix B.1).

The changes in input file have to be reflected in `Parameters` and `MatrixContainer` classes of framework. More on this topic in the next section.

## 4.5 Integration with the framework

After finishing previous implementation, it was necessary to start integrating parts of framework together to form elastic code. It was mentioned that all elastic code compute kernels had been implemented inside `SolverCUDAkernels` class and also that format of the input file underwent some modifications. Therefore, integration with framework consists of: modification of `Parameters` and `MatrixContainer` classes according to new format of the input file, modification of `MatrixContainer` class to handle all variables needed throughout simulation, modification of `KSpaceFirstOrder3DSolver` control class, and other minor modifications.

The `KSpaceFirstOrder3DSolver` class has been modified extensively. A lot of parts of the preprocessing phase has been altered. In elastic code, the preprocessing phase does not include that many operations as in fluid code. The simulation loop method have been almost completely altered. In this method, the computation kernels are invoked one-by-one to compute one simulation step, so the need for modification is obvious. Besides some minor modifications in postprocessing phase, rest of the class remains the same.

In `MatrixContainer` class, the most notable changes have been made in matrices creation method. This method has been modified in such way that class now includes all variables required by elastic simulation and is able to fetch pointers to those variables. Due to clever design, other parts of class have not been modified.

The `Parameters` class have been modified mainly to address new set of parameters occurring in new format of the input file. Also CUDA parameters have been slightly modified.

## 4.6 Integration with the k-Wave

With fully functioning elastic code, there is only one obstacle which prevents simulation to be run by framework. It is a lack of input data. However, as mentioned earlier in Sec. 3.3, k-Wave already includes a sub-script for fluid code input file creation called `kSpaceFirstOrder_SaveToDisk.m`. This file have been slightly modified to provide all data specified in new file format. The other k-Wave integration, creation of wrapper function, have been implemented by Dr. Treeby, co-author of the k-Wave, and is currently only available for testing purposes.

## 4.7 Numerical accuracy testing

Since kernels had got integrated into framework, there was no chance to access framework from the outside and testing with Google test became impossible. Therefore, a MATLAB script was created to calculate error of elastic code compared to the reference implementation on the level of simulation output. It is based on the comparison of the final pressure field. This script also implements a more scientific approach to comparison incorporating  $L_\infty$  and  $L_2$  errors. Method calculates two types of error.

The first one is derived from  $L_\infty$  and it is calculated by

$$L_\infty error = \frac{L_\infty}{\max |x|} = \frac{\max |x - \dot{x}|}{\max |x|}. \quad (4.1)$$

Here the  $x$  and  $\dot{x}$  represent results of elastic code and reference result respectively and  $\max$  function extracts maximum value in data. The standard  $L_\infty$  is then given by  $\max |x - \dot{x}|$ . The  $L_\infty$  is basically the maximal absolute error. This error is then normalized by maximum value of reference data. This is a common practice in signal processing because the signal peaks carry the most energy of signal and therefore the error in this part has a significantly greater impact then the error in parts with relatively low amplitude. The normalization by maximal amplitude reflect this fact.

The other error uses  $L_2$  and it is given by

$$L_2 error = \frac{L_2}{\max |x|} = \frac{\sqrt{\sum_{i=1}^N (x_i - \dot{x}_i)^2}}{N \times \max |x|}. \quad (4.2)$$

This error is calculated very similarly to  $L_\infty error$  with a few differences. The  $L_2$  is basically root-mean-square deviation and represents mean error in every data point. This error is then normalized by the maximum value in reference data. It displays the overall accuracy of the elastic code simulation compared to MATLAB implementation.

Besides the two mentioned errors, the script also calculates raw error, checks the number of Not a Number (NaN) values in data and also provides a unnormalized versions of  $L_\infty$  and  $L_2$ . Another feature of the script is the ability to calculate the number of normalized errors which are greater than a specified limit.

## 4.8 Performance testing

Both k-Wave and framework have built-in timer calculating elapsed time of simulation. These timers was used to measure the time consumed by one time step of the simulation. This value is then used to compare performance of both implementation. Sec. 5 is dedicated to performance evaluation. The NVIDIA Visual Profiler (NVVP) was used to further investigate performance of elastic code. Profiling results and derived conclusions are presented in Sec. 5.4.

## 4.9 Documentation

The process of implementation of the elastic code was documented using git repository. For every feature, a new branch was created with appropriately structured issue to meet the standards. Currently, the git repository is not accessible for public. The doxygen documentation is also a part of project.

## Chapter 5

# Experimental Results

### 5.1 Performance evaluation

Performance is the key aspect of the implementation as it is one of the main goals of the thesis, therefore great amount of effort have been put towards achieving high performance during development. The performance is also a key factor in making the k-Wave suitable for medical application. To be able to measure a performance of the elastic code and get objective picture of performance, elastic code was tested on various hardware configurations ranging from an accessible enthusiast-tier desktop GPUs to latest supercomputer GPUs. The lossless simulation of wave propagation with multi-signal stress source in heterogeneous environment was used as a benchmark. The heterogeneous environment means heterogeneous material properties. By setting non-uniform medium properties, a worst case scenario, in terms of computational time was created. The benchmark was performed using grid sizes starting at  $64^3$  and ending at  $512^3$ . The dimension sizes are gradually increased by 32 one-by-one to ensure sensible FFT plans. Overall performance is expressed by duration of one time step. Both standard CPU MATLAB implementation (M-CPU) and GPU accelerated MATLAB implementation (M-GPU) using built-in acceleration via Parallel Computing Toolbox are taken as reference point. Investigated implementations was tested on following system.

Table 5.1: Hardware configuration of the CPU used for measuring the performance of the reference MATLAB solutions

Processor				Memory	
System	CPU name	Cores	Frequency	Capacity	Speed
Cluster	E5-2660	2x8	2.4 GHz	$16 \times 32$ GB	1600 MHz

Table 5.2: Hardware configuration used for GPU performance benchmarking

GPU name	Processor			Memory		Price
	Architecture	Cores	Peak performance	Capacity	Throughput	
Tesla K20	Kepler	2,496	3.52 Tflops	5 GB	208 GB/s	1,725\$
GTX 980	Maxwell	2,048	4.61 Tflops	4 GB	224 GB/s	700\$
Titan X	Maxwell	3,072	6.14 Tflops	12 GB	336 GB/s	1,250\$
Tesla P40	Pascal	3,840	10 Tflops	24 GB	345 GB/s	5,700\$
Tesla P100	Pascal	3,584	9.3 Tflops	16 GB	720 GB/s	7,400\$

Table 5.3: Overall hardware configuration used for measuring reference MATLAB performance

Dataset name	Processor	Accelerator
M-CPU	Cluster	N/A
M-GPU	Cluster	Tesla K20

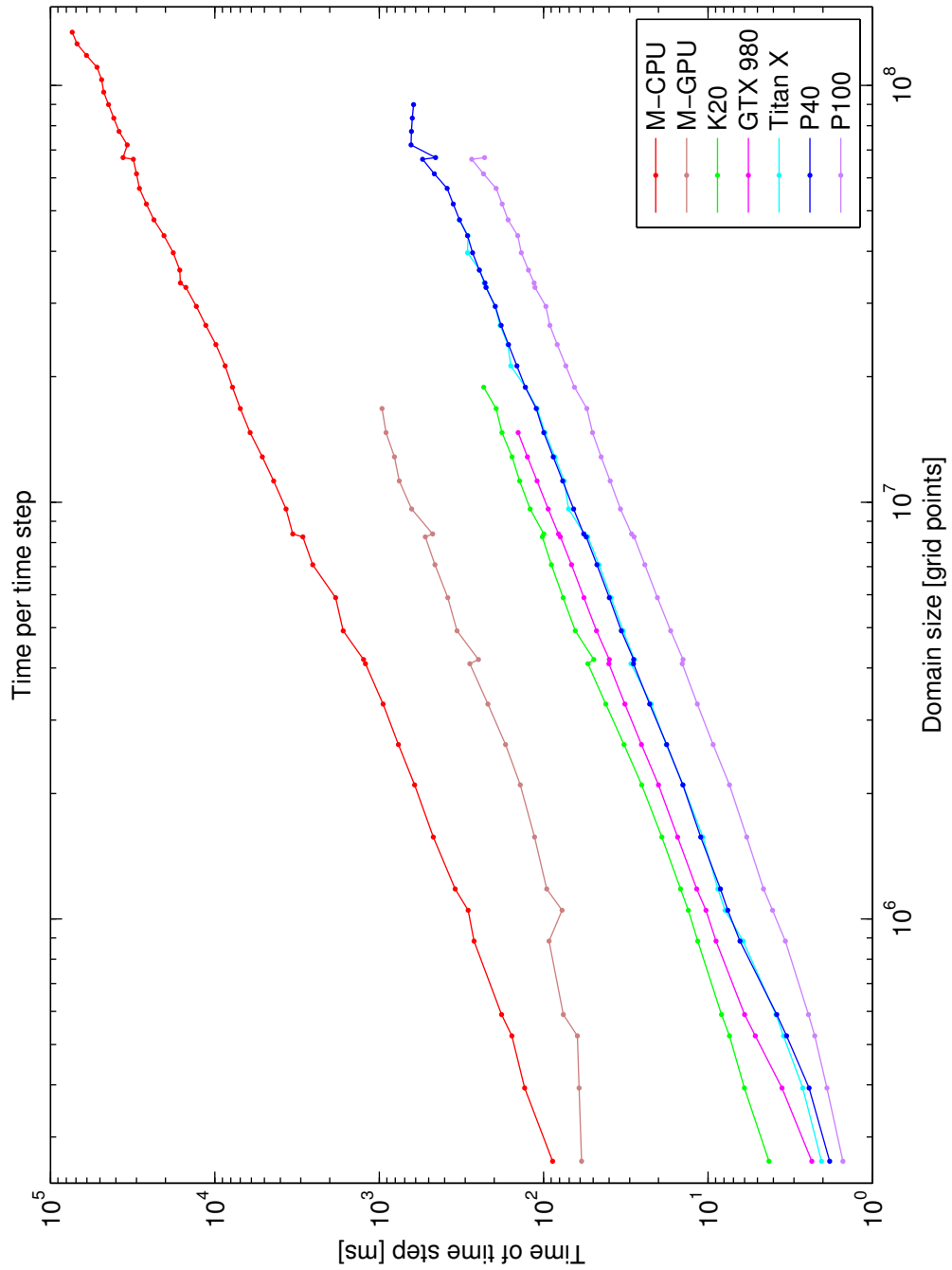


Figure 5.1: Performance benchmark of elastic code on different GPUs and various domain sizes ranging from  $64^3$  to  $512^3$ . There is also data series for reference MATLAB CPU (M-CPU) and MATLAB GPU (M-GPU) solutions.



Fig. 5.1 shows execution times achieved on various GPUs compared to M-CPU (MATLAB's CPU solution) and M-GPU (MATLAB's GPU solution). The figure clearly demonstrates the power of GPU and reasons why it is so often used for acceleration of scientific computations. Even the least effective GPU acceleration, M-GPU represented in Fig. 5.1 by a brown line, is on average 4.6 times faster than standard M-CPU implementation and, considering the peak performance, it is 7.3 times faster. This implementation is least effective because Parallel Computing Toolbox does not have access to problem-specific implementation of kernels, only general purpose implementation of MATLAB functions exists. Therefore, custom implementation can easily outperform M-GPU solution. In fact, the results of M-GPU (brown line) and elastic code (green line) are achieved on the same hardware configuration (cluster node with K20 accelerator). By comparing these two data sets, it is possible to get picture of difference between general implementation and implementation optimized for specific purpose. To be fair though, it is not only the problem-specific optimizations that contributes to this difference. There is also a lot of additional overhead related to M-GPU acceleration (e.g. data transfers between GPU and CPU). Nevertheless, the performance of both approaches differs significantly. The elastic code outperforms M-GPU acceleration by a factor 6.3 on average, and by a factor 13.8 at best. However, the greatest difference in performance occurs for relatively small domain sizes and gradually declines with increasing domain size, so it is safe to say that elastic code is roughly 5 times faster than the M-GPU in general.

The fastest execution was achieved on GPU Tesla P100. The average acceleration factor is 108.3, which is a remarkable achievement, and peak acceleration factor is 158.5. Moreover, the acceleration factor tends to increase with the grid size and, as mentioned earlier, a real-life applications are performed on relatively large domain sizes. In short, the bigger the domain size, the higher the performance benefit. This statement, however, is valid to some extent. The domain size is limited by GPU memory size, but more on this topic in Sec. 5.3.

The Tesla P100 GPU accelerator is the most powerful GPU of all tested without any doubt. On the other hand, it is also the most expensive, and therefore, unreachable for many people and institutions. So it is good to know how implementation behaves on something more accessible. For this purpose, the elastic code performance was measured on GTX 980. This card can be bought for as little as 600 USD (price depends on vendor and retailer). This is very little compared to almost 8,000 USD price tag for Tesla P100. The GTX 980 is in enthusiast tier of graphics cards, but it is still accessible and you can find it in almost every electronics e-shop. Due to the fact that NVIDIA already released a new line of GPU (10th generation), there are also a lot of second hand offers for this card. This GPU was able to accelerate the MATLAB implementation 34 times on average and peak acceleration factor achieved was 42.7. Similarly to the former case, the acceleration factor tends to get higher with increasing domain sizes. From the figure, it is clear that Tesla P100 is roughly 3 times more powerful (considering only this application) but 13 times more expensive than GTX 980. Recalculated on a 1 USD, GTX 980 provides roughly 0.056 acceleration per USD, while Tesla P100 provides only 0.013 acceleration per USD, which makes GTX 980 roughly 4.3 more efficient in terms of invested expenditures. On the other hand, users using GTX 980 would have to settle for smaller domain sizes because elastic code consumes a lot of on-chip GPU memory. In fact, P100 is able to simulate domain of up to  $512 \times 512 \times 256$  points while, for GTX 980, the limit is only  $256 \times 256 \times 224$  points. All things considered, the fluid code provides significant performance boost even on moderate GPUs, however,

users have to take into account slower run of the application and domain size limits as a result of considerably smaller on-chip memory of GPU.

Up on closer inspection of Tab. 5.2 and Fig. 5.1, an interesting question may come to mind. Why acceleration factors of GPUs with comparable performance (Tesla P40 and Tesla P100) differs that much? This question can be quickly answered by looking on memory bandwidth of inspected GPUs. Tesla P100 has almost the double the memory bandwidth compared Tesla P40. This ratio almost perfectly matches with achieved performance difference. This is not accidental. In fact, there is a strong correlation between achieved performance and the memory bandwidth. This relation holds true for every combination of investigated GPUs. It is safe to say that memory throughput has a major impact on, if not determines, GPU performance in elastic code. This only supports the prediction mentioned in Sec. 3.2.6 that most of kernels, and theretofore elastic code itself, will be memory-bound.

The interesting fact is somehow erratic behavior of some GPUs. Most notable example of this behavior can be seen, in Fig. 5.1, as plot of P100 performance approaches last points. It seems the performance of Tesla P100 (and also Tesla P40) suddenly increases for a particular grid size. This behavior is caused by the fact that for this domain size, cuFFT is able to create a very efficient plan and therefore the computation of FFT takes less time. As mentioned in Sec. 4.2.3, the cuFFT library is highly optimized for the grid sizes that can be written in a form of  $2^a \times 3^b \times 5^c \times 7^d$ . In this case, the data size in question is  $512 \times 512 \times 256$  which can be rewritten as  $2^8 \times 2^8 \times 2^7$  and therefore the 1D FFT plans are optimal and performance of cuFFT is high. It may seem the application was only tested under a favorable circumstances, but in real-life scenarios, there is no problem to add padding to data to be able to run simulation more efficiently.

Another fact worth mentioning is seemingly  $O(n)$  time complexity of algorithm despite it was clearly stated, in Sec. 4.2.3, that cuFFT implementation of FFT has  $O(n \log n)$  time complexity. This contraindicative result can be traced to one facts. In the case of time complexity definition of 3D FFT,  $n$  represents the number of grid points in domain. Therefore, it is possible to write it this way  $O(NxNyNz \log NxNyNz)$ . However, the application only uses 1D FFT and therefore the complexity of FFT is  $O(Nx \log Nx)$  or even better written  $\sqrt[3]{n} \log(\sqrt[3]{n})$  (considering the same size in each dimension). Therefore implementation has linear time complexity.

## 5.2 Numerical accuracy

Accuracy is another key aspect of the implementation. As a part of the k-Wave, fluid code is also aspiring to be used as a tool in medicine. In such field the accuracy of simulation is absolutely crucial. However, most of currently used imaging methods has some kind of error in its calculations. The key to success is keeping error of simulation below acceptable level. Transition from a real world into discrete virtual world inevitably introduces an error into simulation. The overall accuracy of simulation of ultrasound waves propagating throughout the elastic medium is affected by variety of factors including: implemented methods, discretization error, time step size, number of grid points, maximum frequency of a simulated signal, level of medium heterogeneity and use of PML just to name a few. For example, the typical error introduced solely by the PML is on the order of  $10^{-3}$  to  $10^{-4}$ , even with optimized parameters [18]. Influence of many of factors can be reduced by applying finer discretization (more grid points and/or smaller time step reduce numerical dispersion). However these actions lead to simulation being more time and memory consuming. In gen-

eral, precision and performance are inversely related to each other. Therefore, in computer simulations, there is a allays need for a acceptable trade-off.

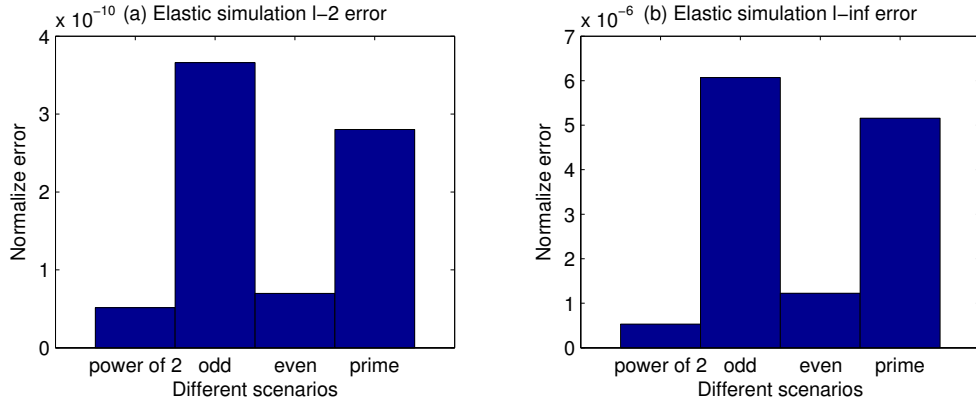


Figure 5.2: Example of numerical accuracy achieved by elastic code on various data sizes. Plot (a) represents the largest normalized error in pressure distribution and plot (b) represents average normalized error.

According to Fig. 5.2, the overall accuracy of fluid code seems to be acceptable at first glance. Presented values are calculated using implemented MATLAB script described in Sec. 4.7 where values  $L_\infty$  and  $L_2$  are calculated based on equations Eqs. (4.1) and (4.2) respectively. The accuracy of the elastic code was measured on pressure distribution in medium. Four scenarios of grid sizes were chosen, namely a power of 2, even, odd and prime which refers to actual sizes  $64^3$ ,  $65^3$ ,  $66^3$  and  $67^3$ . The simulation used for accuracy measurements is the same as the one used to measure performance of the application. Judging from Fig. 5.2, the accuracy of elastic code is dependent on type of domain sizes. The code seems to be quiet accurate for domain sizes which are the power of 2. On the other hand, accuracy drops significantly when simulating domains with odd or prime dimensions. However, this situation can be easily avoided by simple padding. Anyhow, even in the worst case, the maximum error does not go above  $6 \times 10^{-6}$  which is  $6^{-4}\%$ . The error introduce by implementation on GPU is smaller then error caused solely by PML on the order of magnitude and therefore it should not significantly affect the overall accuracy. It is also important to say that the accuracy of elastic code was not tested against the ground truth experiment, so no assumption about overall accuracy of simulation can be made. The displayed data represent differences between k-Wave solution and elastic code only.

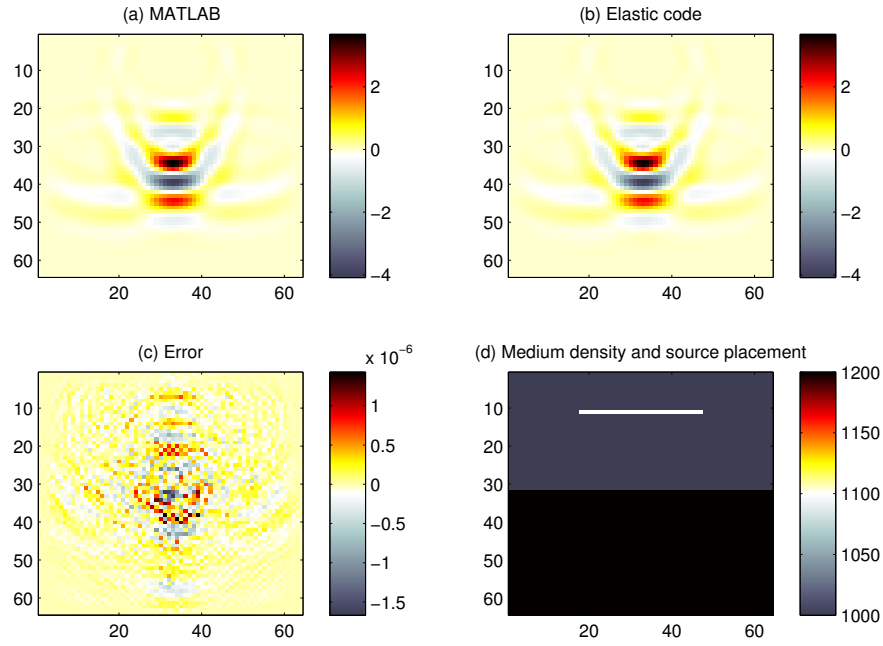


Figure 5.3: Examples of pressure distribution calculated using k-Wave implementation (a) and the fluid code (b). Plot (c) represents error of the fluid code. Plot (d) shows material density and the placement of source (white line).

Fig. 5.3 displays two distributions of acoustic pressure in medium (plots a and b). This two pressure distribution were calculated by k-Wave (plot a) and elastic code (plot b). The final plot represents the error (plot c). The difference between plot a and plot b is impossible to spot with a naked eye, which only supports the claim that error of elastic code is within an acceptable range.

All thing considered, the implementation achieves acceptable level of accuracy while delivering outstanding performance.

### 5.3 Memory consumption

Ultrasound simulations demands a lot of data especially in real-life applications. In such cases, simulation in domain of  $1024^3$  points is nothing unusual. However, single quantity of this domain size represents 4 GB of data in single-precision. It is clear that simulation requires several of such variables to run. Approximate notion of number of variables needed by the simulation can be derived from the equations presented in in Sec. 3.1.1. Nowadays, it is not a problem to run across a cluster node (usually referred to as fat., which has 256 GB or even 512 GB of main memory. However, the situation around GPUs is a bit different. The GPU with most on-chip memory used to test implementation was, according to Tab. 5.2, Tesla P40 which only has 24 GB of memory. Moreover, Tesla P40 is specifically designed to handle large amount of data in scientific computations. More commonly, GPUs have up to 10 GB of memory. Considering the case from the beginning, the Tesla P40 can hold only 6 variables in its on-chip memory. This number is absolutely insufficient for the purpose

of the elastic code and therefore most tested GPUs were able to run simulation only on significantly smaller domains. For GPUs with around 4 GB of memory, the limit size is around  $256^3$ . For Tesla P40, the limit size is around  $448^3$ . Without any doubt, memory consumption is great limitation of elastic code. To be able to predict what is possible and what is beyond capabilities of certain GPU, an equation has been constructed:

$$Memory [GB] \approx \frac{(45 + A)NxNyNz + 6\frac{Nx}{2} + 6Ny + 6Nz + 2GCS + input}{1024^3/4} \quad (5.1)$$

In the Eq. (5.1)  $Nx$ ,  $Ny$  and  $Nz$  represents grid dimensions. Number  $A$  in the first term have value from interval  $< 0, 8 >$  and this number is dependent on the number of material properties that are heterogeneous. For example, when the material density is heterogeneous, it means that 4 additional matrices have to be allocated. The GCS is largest of values  $(\frac{Nx}{2} + 1)NyNz$ ,  $Nx(\frac{Ny}{2} + 1)Nz$  and  $NxNy(\frac{Nz}{2} + 1)$ . Thanks to the implementation of cuFFT, at most  $GCS$  number of elements is needed to store all values computed by 1D FFT in various dimensions. However, values in the frequency domain are complex numbers and therefore the actual number of float elements in this matrix is double the  $GCS$ . The *input* represents the amount of additional user-defined input data, usually sources. The application allows to specify a lot of different sources (`source_p0`, `source_ux`, `source_sxx`, etc.). There could be up to 10 sources that can have sizes up to size of the domain (very unlikely except for `source_p0` which is always the size of domain). Furthermore, each source has time varying signal assigned to it. This means that for each point of source there is time series of values. The final number of bytes needed to store the input can be calculated as  $SizeOfSource \times DurationOfSource \times 4$ . Number 45 represents variables that are mandatory and have to be allocated for every type of simulation. Also, 1D vector variables are taken into account. There are 6 1D vectors needed for each dimension for storing values of shift terms and values of PML. The equation does not account for every memory allocation on GPU, for example device constants are omitted. These kind of allocations are not part of equation because they are either hard to predict or negligible. This equation should provide users with a qualified estimate.

One of the unpredictable memory allocation is related to creation of three FFT plans by cuFFT. The size of plans strongly depends on the domain size. Plan has usually size  $Nx \times Ny \times Nz$  for domains sizes of power of 2, but for sizes which factorization includes big primes, plan can consume 3 or 4 times more memory.

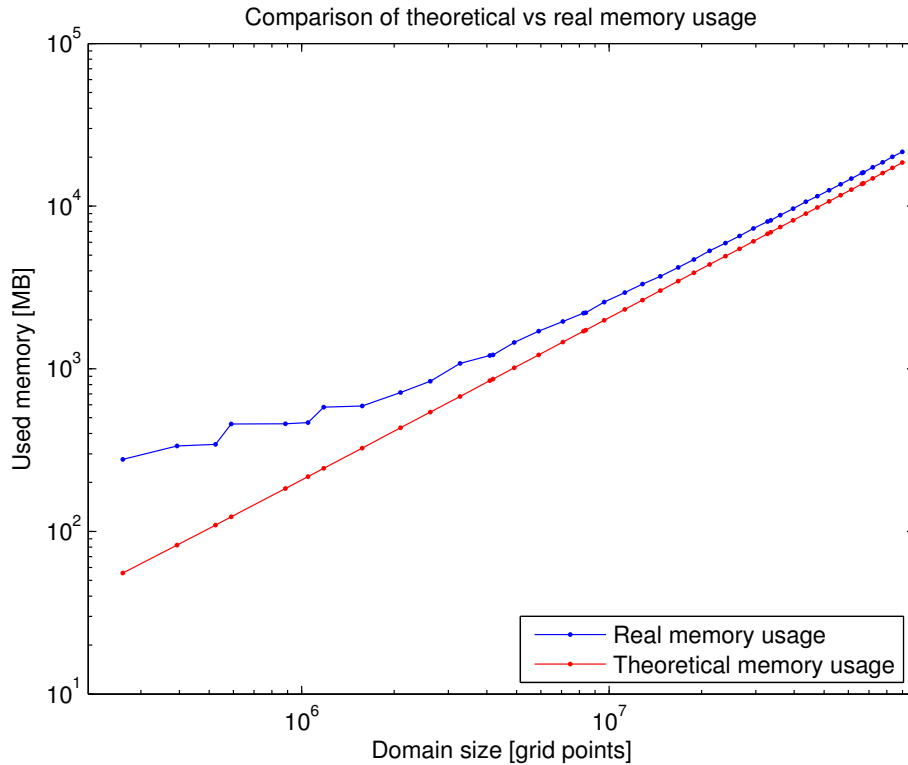


Figure 5.4: Comparison of theoretical memory usage and real memory usage on Tesla P40

Fig. 5.4 shows comparison of the theoretical (red line) and real (blue line) memory usage. The theoretical memory usage was computed using Eq. (5.1) and real usage was measured on Tesla P40 GPU. For small domain sizes, the real memory usage is significantly higher than theoretical. This effect is caused by memory allocations omitted in Eq. (5.1). When the domain is relatively small, this allocations contribute considerably to the overall amount of memory. However, their impact is reduced with increasing domain size. The real memory usage plot also has a some peaks and valleys. This is caused by aforementioned allocation of FFT plans, which are strongly dependent on domain size factorization. It can be said that theoretical memory usage reassembles real memory usage very closely and therefore Eq. (5.1) is a helpful and reliable tool for predicting the memory consumption.

## 5.4 Performance limitations

To be able to improve the algorithm in future, it is good to know what current performance limitations and theoretical limits of optimization of algorithm are. Sometimes, a lot of effort goes into optimization process which eventual does not provide expected results. This could happen because of two reasons. The first reason is that acceleration is simply unreachable due to algorithm itself. Therefore, it is good to keep in mind the theoretical limits of algorithm optimization. The second one is that optimization is focused on other then main reason of bad performance. So, identifying the weak spots of implementation is absolutely crucial for success of optimization. This two aspect are main topics of this section.

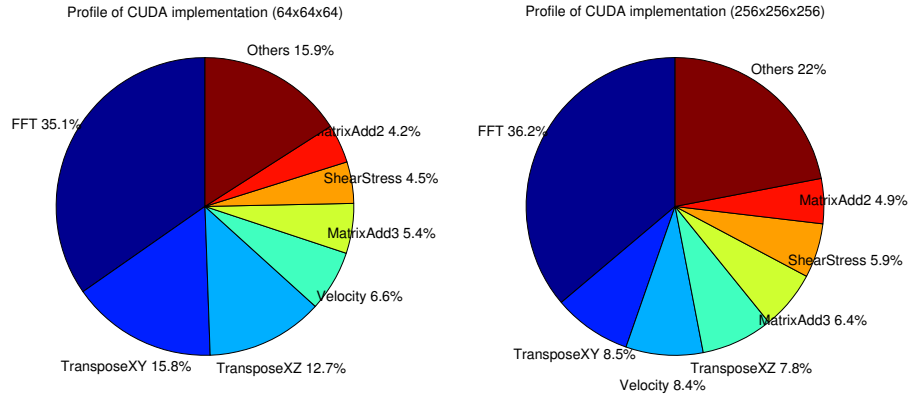


Figure 5.5: The flat profile of elastic code. On left side, values for  $64^3$  domain size. On right, values for  $256^3$  domain size.

Fig. 5.5 shows the flat profile of elastic code produced by NVIDIA Visual Profiler. This values sets approximate order of importance of particular kernels and are useful in determining the theoretical limit of acceleration. Kernels implementing FFT are, as mentioned in Sec. 5.4, highly optimized and therefore there is no space to accelerate these kernels further. Next, matrix transposition kernels have been optimized by Dr. Jaroš as a part of existing framework and these kernels reached their limits in terms of performance. All other kernels are part of the implementation and therefore their optimization represent a way to increase performance. However, these kernels do not take up the vast majority of the overall time. In fact, they take up only 36.9% and 47.5% of time of computation on domain of a size  $64^3$  (left side of figure) and  $256^3$  (right side of figure), respectively. Interestingly, the importance of the implemented kernels is rising with the domain size and so is the theoretical limit of acceleration. However, even considering better case (domain size a  $256^3$ ) and allowing the implemented kernels to be theoretically executed in an instant, the theoretical acceleration limit is only 47.5% given by Ahmdal's law. So it is safe to say that application can be accelerated to achieve maximally the double the performance, although it is only theoretically possible.

To show weak spots and optimization opportunities of the implementation, two implemented kernels with highest impact were chosen and discussed in detail in next sections.

#### 5.4.1 The split-field particle velocity computation

In this section, the velocity kernel is closely analyzed from performance point of view and acceleration opportunities are discussed. The data presented in this section was gathered via NVIDIA Visual Profiler, the program for CUDA code profiling.

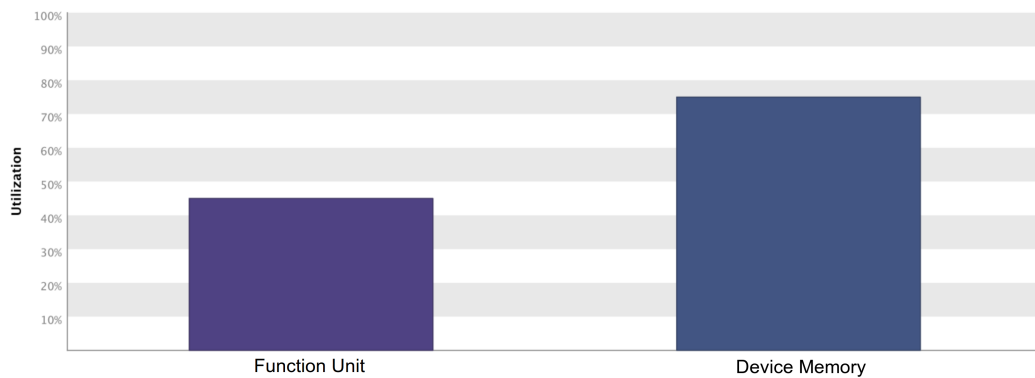


Figure 5.6: Levels of utilization of GPU sub-systems by velocity kernel on Tesla K20 simulating  $256^3$  grid.

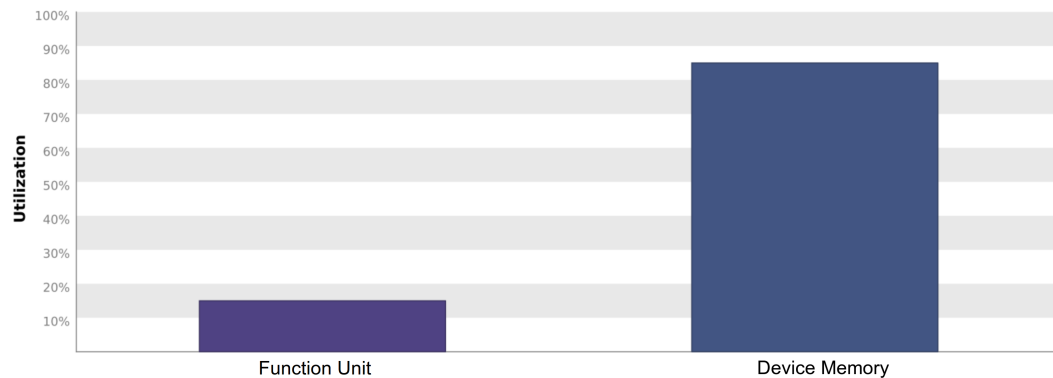


Figure 5.7: Levels of utilization of GPU sub-systems by velocity kernel on Tesla P40 simulating  $256^3$  grid.

Figs. 5.6 and 5.7 present high-level performance analysis. Here, the utilization of the two main sub-systems of GPU, namely function unit and memory, can be seen. At first glance, it is clear that there is a large disproportion between utilization of memory and function unit. The fluid code running on Tesla K20 (Fig. 5.6) was able to utilize memory roughly up to 75% and function unit only up to 45%. This disproportion is even more evident when elastic code is ran on more powerful GPU (Fig. 5.7). Here utilization of memory and function unit reaches roughly 85% and 15%, respectively. This behavior leads to the conclusion that performance of kernel is most likely bound by memory bandwidth.



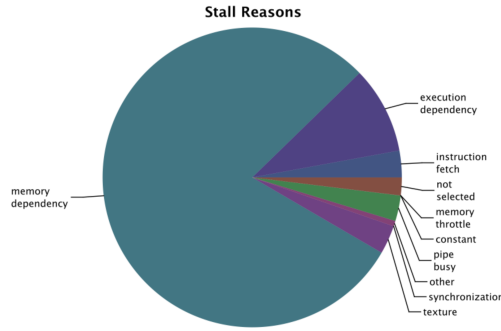


Figure 5.8: Stall reasons of velocity kernel on Tesla K20 simulating  $256^3$  grid.

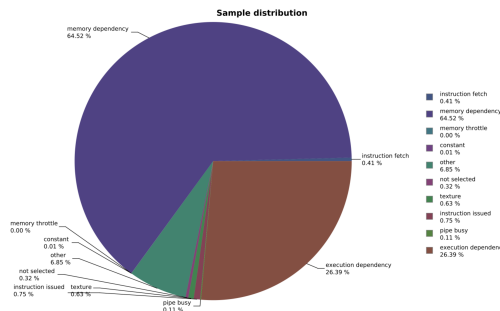


Figure 5.9: Stall reasons of velocity kernel on Tesla P40 simulating  $256^3$  grid.

In previous paragraph, it was stated that velocity kernel is most likely limited by the memory throughput. Figs. 5.8 and 5.9 only support this conclusion. Up on closer inspection of figures, it is clear that the vast majority of stalls is caused by memory dependency. While using Tesla P40 to run elastic code, memory dependencies create 64% of all stall reasons (Fig. 5.9). Unfortunately, older version of NVIDIA Visual Profiler does not display exact values and therefore it can be only estimated that memory dependencies create roughly about 80% of all stall reasons on Tesla K20 (Fig. 5.8). The Memory dependency stall means that load or store operation cannot be completed because required resources are either fully utilized or too many request of a given type is pending. The NVIDIA Visual Profiler suggests resolving data alignment and memory access pattern issues, however the kernel was implemented in such way that it is accessing global memory with best possible pattern.

Device Memory			
Reads	18237991	116.255 GB/s	
Writes	7734440	49.302 GB/s	
<b>Total</b>	<b>25972431</b>	<b>165.557 GB/s</b>	
ECC Overhead	4999176	31.867 GB/s	

Figure 5.10: Global memory throughput achieved by velocity kernel on Tesla K20 simulating  $256^3$  grid.

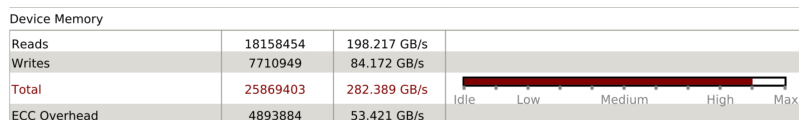


Figure 5.11: Global memory throughput achieved by velocity kernel on Tesla P40 simulating  $256^3$  grid.

In Figs. 5.10 and 5.11, achieved global memory throughput is presented. In both cases, global memory throughput reaches its limits. This fact in combination with kernel having aligned access pattern results in conclusion that kernel performance is definitely limited by the memory throughput.

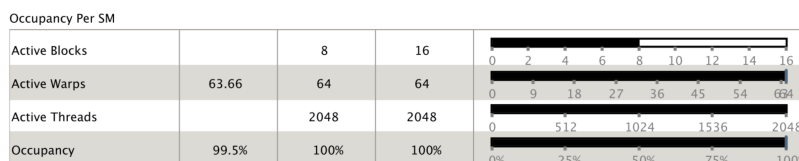


Figure 5.12: The velocity kernel achieved occupancy on Tesla K20 simulating  $256^3$  grid.

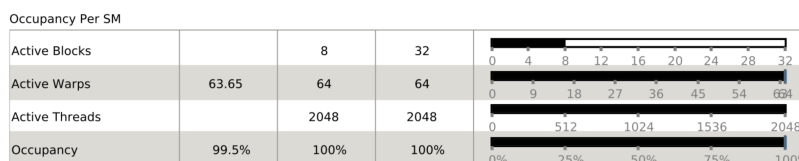


Figure 5.13: The velocity kernel achieved occupancy on Tesla P40 simulating  $256^3$  grid.

Besides memory dependency, execution dependency and memory bandwidth, achieved occupancy can be also limiting performance of the kernel. But according to Figs. 5.12 and 5.13, kernel is able to fully utilize entire GPU in both cases.

#### 5.4.2 The matrix addition computation

In this section, the matrix addition kernel is analyzed from performance point of view and acceleration opportunities are discussed. The data and figures was gathered via NVIDIA Visual Profiler.

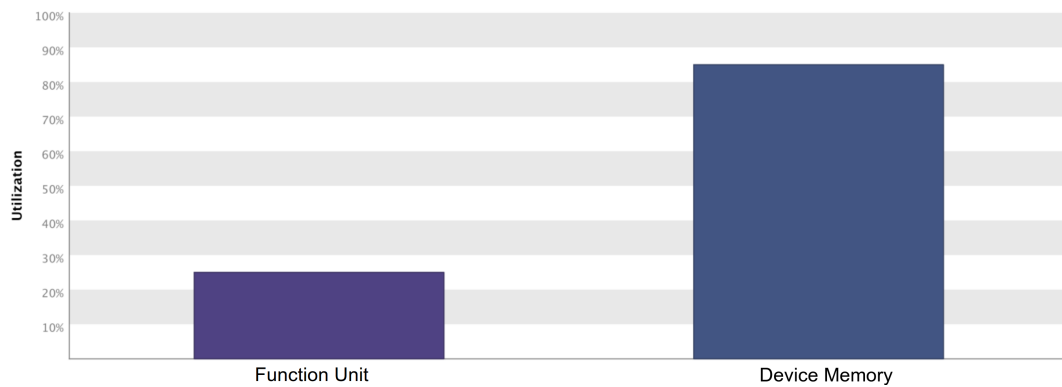


Figure 5.14: Levels of utilization of GPU sub-systems by matrix add kernel on Tesla K20 simulating  $256^3$  grid.

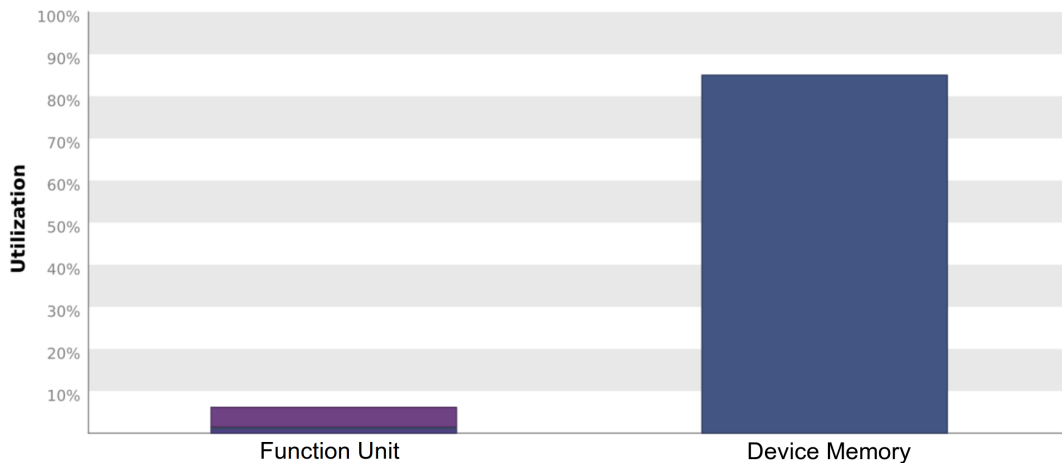


Figure 5.15: Levels of utilization of GPU sub-systems by matrix add kernel on Tesla P40 simulating  $256^3$  grid.

Figs. 5.14 and 5.15 present the utilization of two main sub-systems of GPU, namely function unit and memory unit. It is clear, from these figures, that there is a great difference between utilization level of memory and function unit. This difference is even greater that in case of the velocity kernel. For Tesla K20 (Fig. 5.6) the utilization of memory is roughly 85% and utilization of function unit is about 25%. In case of elastic code being executed on Tesla P40 (Fig. 5.7), the utilization of memory sub-system is also about 85%, however utilization of the function unit drops to only 5%. This is mainly caused by the low arithmetic complexity of the kernel which is only 1/12 floating-point operation per byte. The kernel is wasting performance of the GPU. As in the case of velocity kernel, this kernel is almost certainly limited by the memory throughput.

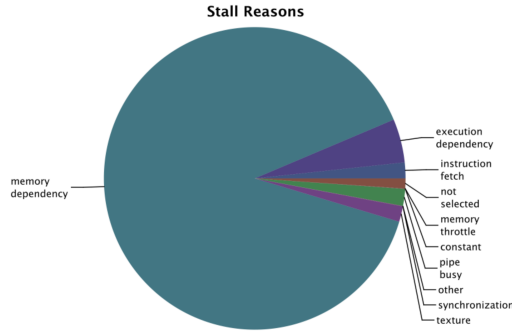


Figure 5.16: Stall reasons of matrix add kernel on Tesla K20 simulating  $256^3$  grid.

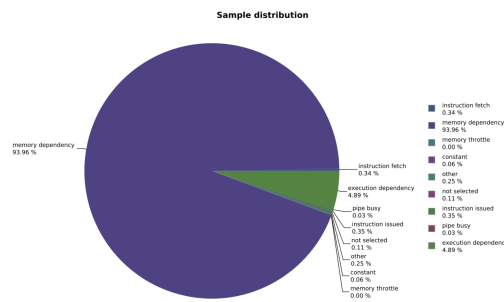


Figure 5.17: Stall reasons of matrix add kernel on Tesla P40 simulating  $256^3$  grid.

Figs. 5.16 and 5.17 support the fact that this kernel is most probably bottlenecked by the memory bandwidth. In fact, matrix addition is prime example of the memory-bound problem on GPU. Therefore, it comes as no surprise that dominant portion of stalls is caused by the memory dependencies. On Tesla P40, memory dependencies create staggering 94% of all stall reasons (Fig. 5.17) and it is estimated that up to 85% of all stalls is caused by memory dependency on Tesla K20 (Fig. 5.16). The usual technique to resolve such an issue is to optimize global memory access pattern and align the data. However, the kernel itself is very straightforward and therefore it was implemented in such way, that there is no way to further optimize accessing pattern or data alignment.

Device Memory			
Reads	7756327	126.182 GB/s	
Writes	2559598	41.64 GB/s	
<b>Total</b>	<b>10315925</b>	<b>167.822 GB/s</b>	
ECC Overhead	1926869	31.347 GB/s	

Figure 5.18: Global memory throughput achieved by matrix add kernel on Tesla K20 simulating  $256^3$  grid.

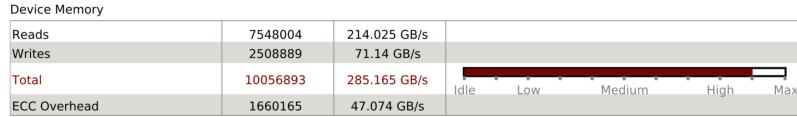


Figure 5.19: Global memory throughput achieved by matrix add kernel on Tesla P40 simulating  $256^3$  grid.

In Figs. 5.18 and 5.19, achieved global memory throughput is presented. In both cases, the kernel is fully utilizing global memory. There are two ways to off-load the main memory. One way is to optimize access pattern and data alignment for global memory load/store operation, and the other way is to use shared memory to pre-load data and then calculate with this data instead. As mentioned in the previous paragraph, the kernel already has the optimized access pattern and aligned data, so this approach is of no avail. Application of the second approach would not be beneficial simply due to logic of computation. Taking into consideration all previously mentioned facts, the performance of this kernel is definitely limited by the memory bandwidth.

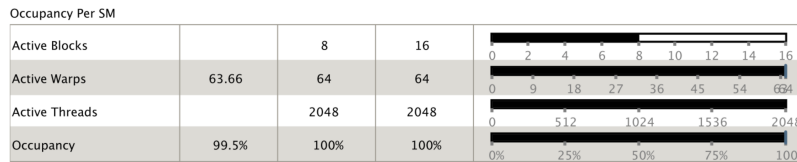


Figure 5.20: The occupancy achieved by matrix add kernel on Tesla K20 simulating  $256^3$  grid.

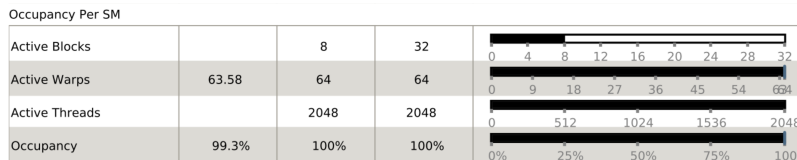


Figure 5.21: The occupancy achieved by matrix add kernel on Tesla P40 simulating  $256^3$  grid.

Figs. 5.20 and 5.21 show that kernel is able to fully occupy the GPU, and therefore, increasing performance by optimizing number of resources or configuration of kernel is not possible.

### 5.4.3 Conclusion

It was clearly shown that performance of both kernels is definitely limited by the memory bandwidth. It was also shown there is almost no room for optimization. It is true that only two implemented kernels was discussed in detail, however, those were the kernels which optimization would have the largest impact on performance. Anyhow, there is the theoretical limit to the acceleration which states that fluid code will never be more than 2 times faster.

All in all, further optimization is not recommended due to fact that cost of the optimization would greatly exceed its benefits.

# Chapter 6

## Conclusion

This thesis can be considered as successful. It fulfills all points of the assignment and in some way exceeds expectations. One of the biggest accomplishment of thesis is the performance of implemented solution. This algorithm can be up to 160 times faster compared to the original MATLAB implementation running on dual 8-core CPU which not only means a great savings of time but also a great reduction of computation cost. Using implemented elastic code, simulation in domain of at most  $448^3$  grid points is possible. This simulation done over 4,655 time steps would last for about 47.9 minutes on Tesla P40. The typical simulation on domain of a  $256^3$  points carried over 2,660 steps would last for 8.6 minutes on less powerful Tesla K20.

The performance of the algorithm on specific GPU is strongly dependent on memory throughput. Therefore, there is a good chance that with the introduction of new line of GPUs with new Volta architecture, the acceleration factor of the elastic code will rise significantly. Upcoming Tesla V100 is said to have memory bandwidth of 900 GB/s [15]. Considering this GPU, the elastic code would accelerate MATLAB solution by a factor around 200. On the other hand, strong performance-bandwidth dependency means that the algorithm is able to achieve relatively good performance even on slightly outdated GPUs with good throughput. The predictable memory usage and almost linear time complexity results in great a scalability. It means that the impact of this thesis will only get greater as the memory capacity of GPUs increases.

All things considers, there is no doubt that this thesis will find use in many fields of science. And hopefully, it will be perceived as benefit to the scientific community.

### 6.1 Impact

In many aspects, the impact of the thesis is a great criterion to measure success. This thesis has a potential to have a significant impact. To back up this claim, let's consider this paper [17]. The paper presents results of using k-Wave for the purpose of transcranial ultrasonic neurostimulation. The team carried out a few experiments, namely a 2D elastic simulation with a domain size of  $3780^2$  over 258,462 time steps accelerated by NVIDIA Titan X which took 10.6 hours to complete and a 3D elastic simulation with domain size of  $1024^3$  over 22,718 timesteps which computation took one node of Salomon cluster 112.3 hours. The 112.3 hours of computation on Salomon cluster represents 2,695.2 corehours which is equivalent of 134.8 USD (according to estimated fair 0.05c per corehour). So, the simulation lasting for several days or even weeks and costing hundreds of dollars is not a

rare. This is a reason why acceleration of such simulations are so important. The latter simulation have not been accelerated by GPU because at that time there was no acceleration of 3D elastic code. Nowadays, it is possible to accelerate this kind of simulation thanks to the code developed in thesis. To demonstrate the benefits of using the accelerated version of the elastic simulation, consider the following example. A researcher needs to compute a simulation and decides to use k-Wave and MATLAB's built-in GPU acceleration on Anselm cluster. The simulation overall time is 96 hours which means 1,536 corehours (allocation of entire node only) and 76.8 USD. Had researcher used my implementation, he would not have only got the result in around 20 hours but also save 80% of the expenses related to simulation using exactly the same machine. This example clearly depicts the benefits of using the elastic code. In medical use, this means that patients do not have to wait for the diagnosis for weeks and the governments do not have to pay hundreds of dollars for such simulations. It would be great if this thesis makes the future medical care better and more accessible. Potential to do it surely possesses.

## 6.2 Further improvements

As discussed in Sec. 5.4, it seems that developed elastic code reaches its limits in terms of performance and so there is a very little space to further improvements. However, the current version does not fully support all features of the elastic simulation present in k-Wave. Therefore the implementation of these features has a high priority. This issue will be very likely resolved in a few months following publishing of this thesis. Moreover, relatively favorable memory consumption allows for implementation of a higher order time stepping scheme such as Runge-Kutta. This would provide ability to use larger time step with the same accuracy further decreasing simulation time. This kind of time stepping scheme of higher order was never implemented in k-Wave elastic code so it will certainly provide a challenge.



# Bibliography

- [1] Fundation, F. U.: *Focused Ultrasound Treatment*. [Online; cited 2.12.2016]. Retrieved from: <https://www.fusfoundation.org/diseases-and-conditions/oncological/brain-cancer>
- [2] Georgiou, P. S.; Jaros, J.; Payne, H.; et al.: Beam distortion due to gold fiducial markers during salvage high-intensity focused ultrasound in the prostate. *Med. Phys.*, vol. 44, no. 2. 2017: page 679–693. doi:10.1002/mp.12044. Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2017-Georgiou-MP.pdf>
- [3] Group, T. H.: *The HDF5*. [Online; cited 2.4.2017]. Retrieved from: <https://support.hdfgroup.org/HDF5/>
- [4] Harlow, F. H.; Welch, J. E.: *Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface*. *The Physics of Fluids*, vol. 8, no. 12. 1965: pp. 2182–2189. doi:10.1063/1.1761178. Retrieved from: <http://aip.scitation.org/doi/abs/10.1063/1.1761178>
- [5] Intel: *Intel Xeon Processor E5-2699 v3*. [Online; cited 3.3.2017]. Retrieved from: [https://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2\\_30-GHz](https://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz)
- [6] Jathoul, A. P.; Laufer, J.; Ogunlade, O.; et al.: *Deep in vivo photoacoustic imaging of mammalian tissues using a tyrosinase-based genetic reporter*. *Nature Photonics*, vol. 9. 2015: pp. 239–246. doi:10.1038/nphoton.2015.22. Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2015-Jathoul-NATPH.pdf>
- [7] Jing, Y.; Meral, C.; Clement, G.: *Time-reversal transcranial ultrasound beam focusing using a k-space method*. *Physics in medicine and biology*, vol. 57. 2015: pp. 901–917. doi:10.1038/nphoton.2015.22. ISSN: 1361-6560. Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2015-Jathoul-NATPH.pdf>
- [8] Kadlubiak, K.: *Fast Tissue Image Reconstruction Using a Graphics Card*. Bakalářská práce. VUT v Brně. 2015.
- [9] Khronos: *The open standard for parallel programming of heterogeneous systems*. [Online; cited 1.12.2014]. Retrieved from: <https://www.khronos.org/opencv1>
- [10] Levites, J.; Jones, S.: *Inside Kepler: world's fastest and most efficient accelerator*. [Online; cited 1.2.2015]. Retrieved from: <http://on-demand.gputechconf.com/gtc-express/2012/presentations/inside-tesla-kepler-k20-family.pdf>

- [11] NVIDIA: *About CUDA*. [Online; cited 12.2.2015].  
Retrieved from: <https://developer.nvidia.com/about-cuda>
- [12] NVIDIA: *CUDA Toolkit Documentation*. [Online; cited 14.11.2016].  
Retrieved from: <http://docs.nvidia.com/cuda/#axzz4VLR2fdRC>
- [13] NVIDIA: *GeForce GTX TITAN specifications*. [Online; cited 20.2.2015].  
Retrieved from: <http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-titan/specifications>
- [14] NVIDIA: *Inside Pascal: NVIDIA's Newest Computing Platform*. [Online; cited 3.10.2016].  
Retrieved from: <https://devblogs.nvidia.com/paralleforall/inside-pascal>
- [15] NVIDIA: *NVIDIA Tesla V100*. [Online; cited 2.5.2017].  
Retrieved from: <https://www.nvidia.com/en-us/data-center/tesla-v100/>
- [16] NVIDIA: *The cuFFT*. [Online; cited 1.4.2017].  
Retrieved from: <https://developer.nvidia.com/cufft>
- [17] Robertson, J. L.; Cox, B. T.; Jaros, J.; et al.: Accurate simulation of transcranial ultrasound propagation for ultrasonic neuromodulation and stimulation. *J. Acoust. Soc. Am.*, vol. 141, no. 3. 2017: pp. 1726–1738. doi:10.1121/1.4976339.  
Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2017-Robertson-JASA.pdf>
- [18] Robertson, J. L.; Cox, B. T.; Treeby, B. E.: Quantifying numerical errors in the simulation of transcranial ultrasound using pseudospectral methods. In *IEEE International Ultrasonics Symposium*. 2014. pp. 2000–2003. doi:10.1109/ULTSYM.2014.0498.  
Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2014-Robertson-IEEEIUS.pdf>
- [19] Rupp, K.: *CPU, GPU and MIC Hardware Characteristics over Time*. [Online; cited 12.10.2016].  
Retrieved from: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time>
- [20] Techtarget: *GPGPU (general purpose graphics processing unit)*. [Online; cited 17.1.2017].  
Retrieved from: <http://whatis.techtarget.com/definition/GPGPU-general-purpose-graphics-processing-unit>
- [21] Treeby, B.; Cox, B.; Jaros, J.: *The k-Wave user manual*. [Online; cited 21.4.2017].  
Retrieved from: [http://www.k-wave.org/manual/k-wave\\_user\\_manual\\_1.0.1.pdf](http://www.k-wave.org/manual/k-wave_user_manual_1.0.1.pdf)
- [22] Treeby, B. E.; Cox, B. T.: *k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields*. *Journal of Biomedical Optics*. 2010: page 15. ISSN: 1083-3668.
- [23] Treeby, B. E.; Jaros, J.; Rohrbach, D.; et al.: *Modelling elastic wave propagation using the k-Wave MATLAB toolbox*. In *IEEE International Ultrasonics Symposium*. 2014. pp. 146–149. doi:10.1109/ULTSYM.2014.0037. [Online; cited 18.12.2016].  
Retrieved from: <http://bug.medphys.ucl.ac.uk/papers/2014-Treeby-IEEEIUS.pdf>

- [24] UK, C. R.: *Worldwide cancer statistics*. [Online; cited 3.2.2017].  
Retrieved from: <http://www.cancerresearchuk.org/health-professional/cancer-statistics/worldwide-cancer>
- [25] Wikipedia: *Category: Nvidia microarchitectures*. [Online; cited 12.4.2017].  
Retrieved from: [https://en.wikipedia.org/wiki/Category:Nvidia\\_microarchitectures](https://en.wikipedia.org/wiki/Category:Nvidia_microarchitectures)
- [26] Wikipedia: *xUnit*. [Online; cited 12.12.2016].  
Retrieved from: <https://en.wikipedia.org/wiki/XUnit>

## Appendix A

# Performance benchmark data

Table A.1: Execution time per time step [ms] of the GPU code for heterogeneous simulations for different 3D grid sizes ranging from  $64^3$  to  $512^3$ .

Domain Size	M-CPU	M-GPU	Tesla K20	GTX 980	Titan X	Tesla P40	Tesla P100
$64 \times 64 \times 64$	88.18	58.74	4.26	2.33	2.03	1.81	1.51
$96 \times 64 \times 64$	130.32	60.81	6.00	3.55	2.65	2.42	1.89
$128 \times 64 \times 64$	156.10	62.35	7.39	5.15	3.48	3.32	2.24
$96 \times 96 \times 64$	180.47	75.81	8.28	5.98	3.86	3.82	2.44
$96 \times 96 \times 96$	265.21	92.66	11.55	8.95	6.11	6.39	3.38
$128 \times 128 \times 64$	287.43	77.16	13.17	10.29	7.87	7.57	4.04
$128 \times 96 \times 96$	345.46	95.89	14.71	11.71	8.69	8.39	4.59
$128 \times 128 \times 96$	468.94	113.52	19.10	15.32	10.74	11.07	5.81
$128 \times 128 \times 128$	609.04	138.94	25.32	20.04	14.19	14.24	7.41
$160 \times 128 \times 128$	763.79	170.57	32.47	25.43	17.95	17.87	9.30
$160 \times 160 \times 128$	946.91	218.39	41.85	32.03	22.20	22.64	11.61
$160 \times 160 \times 160$	1212	281.22	53.72	40.12	29.39	28.38	14.34
$256 \times 128 \times 128$	1245	249.15	49.52	39.83	28.09	28.31	14.19
$192 \times 160 \times 160$	1656	335.90	64.18	47.66	32.78	33.70	16.90
$192 \times 192 \times 160$	1844	382.92	76.12	56.89	38.87	39.89	20.28
$192 \times 192 \times 192$	2542	458.45	89.73	67.76	45.98	47.33	24.21
$224 \times 192 \times 192$	2913	525.04	101.84	79.11	53.78	55.12	28.16
$256 \times 256 \times 128$	3359	472.90	99.93	81.15	56.63	57.13	29.18
$224 \times 224 \times 192$	3692	634.52	120.85	93.83	70.56	65.74	34.08
$224 \times 224 \times 224$	4392	754.54	139.84	109.80	75.12	76.66	39.40
$256 \times 224 \times 224$	5137	808.50	155.49	125.39	85.40	87.51	44.66
$256 \times 256 \times 224$	6100	908.82	179.29	142.91	98.02	99.65	50.49
$256 \times 256 \times 256$	7010	962.49	194.54		109.69	110.86	54.58
$288 \times 256 \times 256$	7816		231.30		128.96	129.24	64.82
$288 \times 288 \times 256$	8651				158.29	145.52	73.25
$288 \times 288 \times 288$	9854				165.05	163.76	82.62
$320 \times 288 \times 288$	11358				184.18	181.50	91.52
$320 \times 320 \times 288$	12965				197.66	196.99	97.05
$320 \times 320 \times 320$	15016				224.40	224.00	113.03
$512 \times 256 \times 256$	16202				229.35	227.16	114.29
$352 \times 320 \times 320$	16351				244.86	246.33	123.78
$352 \times 352 \times 320$	17892				290.60	270.45	136.91
$352 \times 352 \times 352$	20382				288.84	290.11	143.93
$384 \times 352 \times 352$	23502				325.79	324.79	164.76
$384 \times 384 \times 352$	26120					354.58	178.89
$384 \times 384 \times 384$	28758					386.76	194.59
$416 \times 384 \times 384$	29976					461.92	232.33
$416 \times 416 \times 384$	31291					545.57	273.61
$512 \times 512 \times 256$	36215					454.63	228.49
$416 \times 416 \times 416$	34085					640.67	
$448 \times 416 \times 416$	38311					637.41	
$448 \times 448 \times 416$	41221					630.15	
$448 \times 448 \times 448$	44294					617.90	
$480 \times 448 \times 448$	47452						
$480 \times 480 \times 448$	48794						
$480 \times 480 \times 480$	52088						
$512 \times 480 \times 480$	60347						
$512 \times 512 \times 480$	68842						
$512 \times 512 \times 512$	73716						

## Appendix B

# Format of the HDF5 input file

Table B.1: List of datasets that may be present in the input HDF5 file.

Name	Size (Nx, Ny, Nz)	Data Type	Domain Type	Conditions
1. Simulation Flags				
ux_source_flag	(1, 1, 1)	long	real	
uy_source_flag	(1, 1, 1)	long	real	
uz_source_flag	(1, 1, 1)	long	real	
sxx_source_flag	(1, 1, 1)	long	real	
syy_source_flag	(1, 1, 1)	long	real	
szz_source_flag	(1, 1, 1)	long	real	
sxy_source_flag	(1, 1, 1)	long	real	
sxz_source_flag	(1, 1, 1)	long	real	
syz_source_flag	(1, 1, 1)	long	real	
p0_source_flag	(1, 1, 1)	long	real	
nonuniform_grid_flag	(1, 1, 1)	long	real	must be set to 0
absorbing_flag	(1, 1, 1)	long	real	
2. Grid Properties				
Nx	(1, 1, 1)	long	real	
Ny	(1, 1, 1)	long	real	
Nz	(1, 1, 1)	long	real	
Nt	(1, 1, 1)	long	real	
dt	(1, 1, 1)	float	real	
dx	(1, 1, 1)	float	real	
dy	(1, 1, 1)	float	real	
dz	(1, 1, 1)	float	real	

Table B.1: List of datasets that may be present in the input HDF5 file continued ...

Name	Size (Nx, Ny, Nz)	Data Type	Domain Type	Conditions
<b>3. Medium Properties</b>				
<b>3.1 Regular Medium Properties</b>				
rho0	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
rho0_sgx	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
rho0_sgy	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
rho0_sgz	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
mu	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
mu_sgx	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
mu_sgy	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
mu_sgz	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
lambda	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
c_ref	(1, 1, 1)	float	real	
<b>3.2 Absorbing Medium Properties (defined if 'absorbing_flag = 1')</b>				
alpha_coeff_norm	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
alpha_power_norm	(1, 1, 1)	float	real	
alpha_coeff_shear	(Nx, Ny, Nz)	float	real	heterogeneous
	(1, 1, 1)	float	real	homogeneous
alpha_power_shear	(1, 1, 1)	float	real	
<b>4. Sensor Properties</b>				
sensor_mask_type	(1, 1, 1)	long	real	
sensor_mask_index	(Nsens, 1, 1)	long	real	sensor_mask_type = 0
sensor_mask_corners	(Ncubes, 6, 1)	long	real	sensor_mask_type = 1

Table B.1: List of datasets that may be present in the input HDF5 file continued ...

Name	Size (Nx, Ny, Nz)	Data type	Domain Type	Conditions
<b>5. Source Properties</b>				
5.1 Velocity Source Terms (defined if <code>ux_source_flag = 1</code> or <code>uy_source_flag = 1</code> or <code>uz_source_flag = 1</code> )				
<code>u_source_mode</code>	(1, 1, 1)	long	real	
<code>u_source_many</code>	(1, 1, 1)	long	real	
<code>u_source_index</code>	(Nsrc, 1, 1)	long	real	
<code>ux_source_input</code>	(1, Nt_src, 1)	float	real	<code>u_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>u_source_many = 1</code>
<code>uy_source_input</code>	(1, Nt_src, 1)	float	real	<code>u_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>u_source_many = 1</code>
<code>uy_source_input</code>	(1, Nt_src, 1)	float	real	<code>u_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>u_source_many = 1</code>
5.2 Stress Source Terms (defined if <code>sxx_source_flag = 1</code> or <code>syy_source_flag = 1</code> or <code>szz_source_flag = 1</code> or <code>sxy_source_flag = 1</code> or <code>sxz_source_flag = 1</code> or <code>syx_source_flag = 1</code> )				
<code>s_source_mode</code>	(1, 1, 1)	long	real	
<code>s_source_many</code>	(1, 1, 1)	long	real	
<code>s_source_index</code>	(Nsrc, 1, 1)	long	real	
<code>sxx_source_input</code>	(1, Nt_src, 1)	float	real	<code>sxx_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>sxx_source_many = 1</code>
<code>syy_source_input</code>	(1, Nt_src, 1)	float	real	<code>syy_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>syy_source_many = 1</code>
<code>szz_source_input</code>	(1, Nt_src, 1)	float	real	<code>szz_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>szz_source_many = 1</code>
<code>sxy_source_input</code>	(1, Nt_src, 1)	float	real	<code>sxy_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>sxy_source_many = 1</code>
<code>sxz_source_input</code>	(1, Nt_src, 1)	float	real	<code>sxz_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>sxz_source_many = 1</code>
<code>syx_source_input</code>	(1, Nt_src, 1)	float	real	<code>syx_source_many = 0</code>
	(Nsrc, Nt_src, 1)	float	real	<code>syx_source_many = 1</code>
5.3 IVP Source Terms (defined if <code>p0_source_flag = 1</code> )				
<code>p0_source_input</code>	(Nx, Ny, Nz)	float	real	



Table B.1: List of datasets that may be present in the input HDF5 file continued ...

Name	Size (Nx, Ny, Nz)	Data type	Domain Conditions Type
6. k-space and Shift Variables			
ddx_k_shift_pos_r	(Nx/2 + 1, 1, 1)	float	complex
ddx_k_shift_neg_r	(Nx/2 + 1, 1, 1)	float	complex
ddy_k_shift_pos	(1, Ny, 1)	float	complex
ddy_k_shift_neg	(1, Ny, 1)	float	complex
ddz_k_shift_pos	(1, 1, Nz)	float	complex
ddz_k_shift_neg	(1, 1, Nz)	float	complex
x_shift_neg_r	(Nx/2 + 1, 1, 1)	float	complex
y_shift_neg_r	(1, Ny/2 + 1, 1)	float	complex
z_shift_neg_r	(1, 1, Nz/2 + 1)	float	complex
7. PML Variables			
pml_x_size	(1, 1, 1)	long	real
pml_y_size	(1, 1, 1)	long	real
pml_z_size	(1, 1, 1)	long	real
pml_x_alpha	(1, 1, 1)	float	real
pml_y_alpha	(1, 1, 1)	float	real
pml_z_alpha	(1, 1, 1)	float	real
pml_x	(Nx, 1, 1)	float	real
pml_x_sgx	(Nx, 1, 1)	float	real
pml_y	(1, Ny, 1)	float	real
pml_y_sgy	(1, Ny, 1)	float	real
pml_z	(1, 1, Nz)	float	real
pml_z_sgz	(1, 1, Nz)	float	real
mpml_x	(Nx, 1, 1)	float	real
mpml_x_sgx	(Nx, 1, 1)	float	real
mpml_y	(1, Ny, 1)	float	real
mpml_y_sgy	(1, Ny, 1)	float	real
mpml_z	(1, 1, Nz)	float	real
mpml_z_sgz	(1, 1, Nz)	float	real