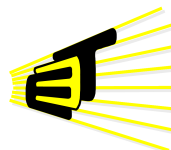


ANALYTICAL MODEL TO ESTIMATE THE EXECUTION TIME OF A 3D ACOUSTIC WAVE EQUATION IMPLEMENTATION USING FDTD IN A GPU

Dorfell Leonardo Parra Prada

Electronic Engineer



Escuela de Ingenierías
Eléctrica, Electrónica
y de Telecomunicaciones



UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE
TELECOMUNICACIONES
BUCARAMANGA

2016

ANALYTICAL MODEL TO ESTIMATE THE EXECUTION TIME OF A 3D ACOUSTIC WAVE EQUATION IMPLEMENTATION USING FDTD IN A GPU

Dorfell Leonardo Parra Prada

Electronic Engineer

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree of Master in
Electronic Engineering

Advisor:

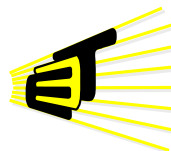
William Alexander Salamanca Becerra

Master in Electronic Engineering

Co-advisor:

Ana Beatriz Ramírez Silva

Ph.D. in Electrical Engineering



Escuela de Ingenierías
Eléctrica, Electrónica
y de Telecomunicaciones



UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE
TELECOMUNICACIONES
BUCARAMANGA

2016

Agradecimientos

Agradecimientos a Dios por darme una hermosa familia, la mejor! A mis padres Felix y Elisa quienes siempre han creído en mí y nunca han dicho no a mis ideas locas, mis experimentos raros o a los proyectos que he emprendido. A mis hermanas Dori y Nubia por aguantar mis cansonerías. A mi esposa Laura y su fé en mí, que me llena de tranquilidad y paz para seguir adelante. A mis nonas Nubia y Bibiana y a las asamblea Divino Niño y Virgen del Carmen por todos sus rosarios. A don Lorenzo y doña Elsin por permitirme trabajar aún estando de visita, gracias por su comprensión. A William y a la profesora Ana por todo su tiempo, por escucharme, aconsejarme y guiarme en el desarrollo del trabajo, sin su ayuda no hubiera terminado este proyecto. Y por supuesto a los muchachos de CPS por sus momentos de disertación, sus preguntas y puntos de vista, valiosos a la hora de diseñar las pruebas que iba a realizar.

A todos muchas gracias.

Contents

	Pag.
Introduction	13
1 Seismic Modelling	15
1.1 Structural Geology	15
1.2 Seismic Reflection Images	16
1.3 Seismic Survey	16
1.4 Synthetic Velocity Models	18
1.5 Seismic Modelling	20
2 Graphics Processing Units	22
2.1 Definition	22
2.2 Brief History of GPUs	23
2.3 GPU Architecture	26
2.4 Memory Hierarchy	27
2.5 GPU Programming	28
2.6 Programming Model	28
2.7 Heterogeneous Programming	29
3 Computational Implementation	31
3.1 Wave Equation Discretization	31
3.2 Finite Difference Time-Domain (FDTD)	31
3.3 Taylor series	32
3.4 Acoustic Wave Equation using Centered FDTD	33
3.5 Stencils	34

3.6 Properties of the FDTD implementation	34
3.6.1 Numerical dispersion	35
3.6.2 Stability	35
3.6.3 Precision	36
3.7 Two Layers Synthetic Velocity Model	36
3.8 CPU Implementation	37
3.9 GPU Implementation	40
4 Modelling of the Implementation	45
4.1 Methodology	45
4.2 Related Work	46
4.3 Memory Warp Parallelism-Compute Warp Parallelism model	47
4.4 Adapted Analytical Model and Parameter Identification for the 3D-Stencil Case	50
4.4.1 $\mathbf{W}_{\text{serial}}$ equation	51
4.4.2 \mathbf{AMAT} equation	51
4.4.3 Departure Delay (Δ)	51
4.4.4 \mathbf{ILP}	52
4.5 Parameters Identification and Extraction	52
4.5.1 Hardware Specifications	53
4.5.2 Parameters from CUDA Binary Tools	54
4.5.3 Parameters from Micro-benchmarks	55
4.5.4 Parameters from the source code	70
4.6 Validating the Adapted Model	71
4.7 Discussion	76
4.8 Conclusions	77
References	79
Bibliography	85

List of Figures

1.1 a) San Andreas Fault at Palmdale in California. Taken from [3]. b) Normal faults in volcanic ashes and paleo-soils, El Salvador, photo by Chuck DeMets. Taken from [1]. . . .	15
1.2 Seattle Fault Zone seismic model. Adapted from [4].	16
1.3 Seismic Survey process. Taken from [7].	17
1.4 Examples of energy sources for marine and on land surveys: a) Mini G airgun suspended from buoys. Taken from [8]. b) HEMI 60, a 61,000 Pound Vibrator. Taken from [9]. . . .	17
1.5 Seismic Trace. Taken from [12].	18
1.6 Seismic Modelling on the Marmousi Model [14]. a) Original Marmousi model. b) Energy source applied to the medium. c) and d) Energy propagation step.	19
1.7 Ricker wavelet. Adapted from [11].	20
1.8 Example of the initial velocity model and the final velocity model: a) Initial synthetic velocity model from Seattle Fault Zone. Taken from [4]. b) Final synthetic velocity model from Seattle Fault Zone. Taken from [4].	21
2.1 Graphics card with NVIDIA GPU GeForce GTX660. Adapted from [24].	22
2.2 1970s Video chips and games. Taken from [21].	23
2.3 1986 ATI Graphics Solution Revision 3. Taken from [22].	24
2.4 1997 Intel i740 AGP Graphics Board [19].	24
2.5 GPU NVIDIA GeForce 256 and VisionTek graphic card. Taken from [17].	24
2.6 AMD and NVIDIA GPUs. Taken from [23] [24].	25
2.7 NVIDIA Kepler GPU Streamming Multiprocessor (SMX). Adapted from [25].	26
2.8 Memory Hierarchy.	27
2.9 GPU Programming Model. Adapted from [26], [27].	29
2.10 Heterogeneous Programming. Adapted from [26], [27].	30

3.1 4 th order stencils in 1D, 2D, 3D. a) 1D stencil. b) 2D stencil. c) 3D stencil.	34
3.2 Numerical dispersion effect. a) Wave front with dispersion. b) Wave front without dispersion. Taken from [35].	35
3.3 3D two layers SVM. Visualization in Paraview.	37
3.4 CPU implementation flow diagram.	38
3.5 Seismic modelling implemented on CPU. All the wave propagation data is stored in “.vtk” format and visualized in Paraview. In a) and b) an Ricker energy source had been applied. c) Energy has reached the top boundary of the field. d) Energy is transmitted and reflected after entering in contact with the 2nd layer of the SVM. e) to f) Energy continue spreading through the medium.	40
3.6 GPU implementation flow diagram.	41
4.1 Methodology.	45
4.2 Control Flow Graph.	54
4.3 Clock special register latency micro-benchmark.	56
4.4 Clock function latency micro-benchmark.	57
4.5 Control flow graph detail of the register latency micro-benchmark kernel.	58
4.6 Register benchmark.	59
4.7 Addition floating point operation latency micro-benchmark.	60
4.8 Subtraction floating point operation latency micro-benchmark.	61
4.9 Multiplication floating point operation latency micro-benchmark.	62
4.10 Division floating point operation latency micro-benchmark.	63
4.11 a) Reading global memory latency. b) Writing global memory latency. Global memory access latency micro-benchmark.	65
4.12 a) Reading shared memory latency. b) Writing shared memory latency. Shared memory access latency micro-benchmark.	66
4.13 Reading L1 Memory Latency.	67
4.14 Reading L2 Memory Latency.	68
4.15 Theoretical latencies versus experimental latencies.	75
4.16 Theoretical latencies versus experimental latencies.	76

List of Tables

3.1 Backwards, Centered and Forward FDTD methods.	32
3.2 Stability limits for $1D$, $2D$ and $3D$ using $2nd$ and $4th$ order Finite Differences. Taken from [32].	36
3.3 Seismic Modelling Parameters.	39
4.1 Parameters used for the adapted model. Adapted from [45].	52
4.2 Hardware specifications used directly or indirectly to compute the set of parameters. . . .	53
4.3 Micro-benchmarks results.	69
4.4 Micro-benchmarks uncertainty.	69
4.5 Theoretical latencies of the implementation of the wave equation propagation for different stencil orders and different number of warps per SM.	74
4.6 Experimental latencies of the implementation of the wave equation propagation for different stencil orders and different number of warps per SM.	75
4.7 Theoretical latencies for different stencil orders and different number of warps per SM considering the $\#total_warps$	76

RESUMEN

Título:

MODELO ANALÍTICO PARA ESTIMAR EL TIEMPO DE EJECUCIÓN DE UNA IMPLEMENTACIÓN DE LA ECUACIÓN DE ONDA ACÚSTICA 3D USANDO FDTD EN UNA GPU [1]

Autor: Dorfell Leonardo Parra Prada [2]

Palabras Claves: GPU, modelado sísmico, modelo analítico, Micro-benchmark

Conocer la estructura de la tierra es importante en varios campos de la industria y la academia. Para obtener una representación de la subsuperficie, datos sísmicos de las adquisiciones son procesados y analizados usando algoritmos complejos como la Migración Reversa en Tiempo (RTM), y la Inversión de Onda Completa (FWI). La etapa fundamental de estos algoritmos es el modelado sísmico. En el modelado sísmico la propagación de la energía en la subsuperficie es simulada usando la ecuación de la onda acústica en términos del método FDTD. Debido a que el método FDTD es computacionalmente costoso, el uso de arquitecturas *many-cores*, (e.g. Unidades de Procesamiento Gráfico (GPU)), se ha vuelto atractivo para problemas de gran escala. El tiempo de ejecución de una implementación de GPU depende de las especificaciones de hardware, parámetros de la implementación, preferencias del usuario, y en el volumen de los datos de entrada entre otros. Para encontrar la configuración de parámetros que permita gastar el menor tiempo en la GPU, una estrategia tradicional consiste en comenzar con una implementación inicial y ejecutar varias pruebas cambiando los parámetros para encontrar el mejor conjunto de parámetros. Sin embargo, para implementaciones de gran escala (e.g. modelado sísmico) esta estrategia es inviable. En este trabajo, una implementación de el modelado sísmico en CPU y GPU es presentada. También se propone una metodología que permite adaptar el modelo analítico MWP-CWP para estimar el tiempo de ejecución de la implementación, así como el proceso de identificación de extracción de parámetros (i.e. binarios de CUDA, Micro-benchmarks, etc.). La validación del modelo compara el tiempo de ejecución estimado con el tiempo de ejecución medido de la implementación del modelado sísmico en una GPU Kepler K40 de NVIDIA.

[1] Tesis de Maestría

[2] Facultad de Ingenierías Físico Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y Director: PhD(c). William A. Salamanca Becerra. Co-director: PhD. Ana Beatriz Ramírez Silva

ABSTRACT

TITLE:

ANALYTICAL MODEL TO ESTIMATE THE EXECUTION TIME OF A 3D ACOUSTIC WAVE
EQUATION IMPLEMENTATION USING FDTD IN A GPU ^[1]

AUTHOR: Dorfell Leonardo Parra Prada ^[2]

KEYWORDS GPU, seismic modelling, analytical model, micro-benchmark

Knowing the earth's subsurface structure is important for several fields in both industry and academy. To obtain a representation of the subsurface, seismic data acquired in surveys are processed and analyzed using complex algorithms such as the Reverse Time Migration (RTM) and the Full Wave Inversion (FWI). The fundamental stage of those algorithms is the seismic modelling. In the seismic modelling the energy propagation through the subsoil can be simulated using the FDTD acoustic wave equation. Because the FDTD method is a highly computationally expensive method, the use of many-core architectures such as Graphic Processor Units (GPU) for large scale problems has become attractive. The execution time of GPU implementation depends on the hardware specifications, implementation parameters, user preferences and input data volume. In order to find the parameters configuration that allows to spend the less time on the GPU, a traditional strategy is to start from a scratch implementation and run several tests varying the parameters to figure out the best set of parameters. However, for large scale implementations (e.g. seismic modelling) this strategy is infeasible. In this work, an implementation of the seismic modelling in both CPU and GPU is presented. Also, a methodology to adapt the MWP-CWP analytical model to estimate the execution time that implementation is proposed, as well as the parameters identification and the extraction process (i.e. CUDA binaries, micro-benchmarks, etc). The validation compares the estimated execution time with the measured execution time of the seismic modelling implementation on a NVIDIA Kepler K40 GPU.

[1] Master Thesis

[2] Faculty of Physics Mechanics Engineering. Electrical, Electronics Engineering and Telecommunications School. Advisor: PhD(c). William A. Salamanca Becerra. Co-advisor: PhD. Ana Beatriz Ramírez Silva

Introduction

Seismic Data Processing (SDP) filters, stacks and migrates traces from the seismic survey to generate Seismic Reflection Images (SRI) of the subsurface. The SRI represents the geological structures of the subsurface and can be used to determine the presence of water, minerals and hydrocarbons. One of the most relevant stages in the SDP is the migration.

In the migration process, the calculated layers are moved to their estimated real position in Earth. There are several migration methods such as the Kirchhoff migration and the 3D Reverse-Time Migration (RTM). Kirchhoff migration offers a high contrast and a low computational cost but due to the complex Colombian geography this migration method does not accomplish a good representation of the subsurface structure. To sort out this problem another migration method such as the RTM 3D must be applied. The RTM 3D is based on the Seismic Modelling which simulates the energy propagation through the subsurface. Propagation can be modeled using the acoustic wave equation with constant density, but the implementation of these type of algorithms is computationally expensive and make them impractical for traditional platforms such as CPUs.

The development of new parallel processing architectures such as Graphic Processing Units (GPUs) and the increasing of memory capacity, have made feasible the implementation of such computational expensive algorithms.

One of the most useful performance measurements of these implementations is the time spent in their execution. The performance of an algorithm implementation on GPU architectures is affected by several factors: bottlenecks, volume data, available resources on-chip, etc. Some techniques used to improve the performance of an algorithm implementation are based on programming expertise, such as the Assess, Parallelize, Optimize and Deploy (APOD) method, proposed by NVIDIA [49]. In particular, APOD technique proposes some guidelines to be followed iteratively until an adequate performance is reached. Those techniques, however, are not suitable to solve the problem of performance improvement of Seismic Modelling implementations, because of the high computational cost of the algorithm.

A different approach that can be used to improve the performance of a GPU implementation is to find an analytical model for the implementation. Commonly analytical models employ implementation parameters (e.g. number of resources used) and hardware specifications to estimate the implementation execution time.

Several works that attempt to estimate the execution time of an application have been proposed in the literature. In [39] the execution time of a program that consists of several kernels is calculated using the MAX(SUM) model. Also, the use of Control Flow Graphs (CFG) and the Instruction-Level Parallelism (ILP) to calculate the execution time is proposed in [40]. In [44] and [45], an analytical model known as Memory Warp Parallelism-Compute Warp Parallelism (MWP-CWP) has been proposed and validated for the GPU implementations of tiled matrix multiplication algorithms, and the Fast Multipole Method (FMM) algorithm. Furthermore, a different model for 2.5D stencil implementation in a GPU was proposed in [46].

Although all this works try to estimate an implementation execution time by modelling the GPU data transfer, and the amount of application parallelism among others factors, those models do not consider algorithms using the acoustic wave equation with Finite Difference in Time-Domain (FDTD) 3D essential for simulating geophysical phenomena. According to [47] one the most accepted analytical models in the literature is the MWP-CWP model [44],[45]. This would be the model used in the development of these work.

The purpose of this work is to implement the solution of the 3D acoustic wave equation with constant density using FDTD in a GPU. Then, propose a model that estimates the execution time of a GPU implementation and finally validate the proposed model. By using this work the understanding of the GPU can be improved, as well as the efficient use of GPU resources leading to decrease an application execution time.

The document is organized as follows: in chapter 1 some of the fundamentals concepts related to the seismic modelling are described. Then, in chapter 2 the architecture of a GPU, the CUDA programming model and a brief history of the GPU is presented. After that, the characteristics of the computational implementation and some of their results are explained in chapter 3. Finally in chapter 4, the use of a general analytical model to evaluate the performance of the 3D stencil-based GPU kernel is proposed. Furthermore, the process of extracting parameters from hardware specifications, source code and CUDA binary utilities, as well as the designed micro-benchmarks and the model validation process.

Seismic Modelling

1.1 Structural Geology

The Earth's structure is composed by several layers formed million years ago. Examples of the earth's structure are shown in Figure 1.1. Figure 1.1a shows a road cut of a pressure ridge along the San Andreas Fault at Palmdale (CA) [3], and Figure 1.1b shows normal faults created by volcanic ashes and paleo-soils in El Salvador [1]. As can be seen, each layer has different properties due to the characteristics of the rocks and the physical conditions like temperature and pressure that allow the layer to be created.



(a)



(b)

Figure 1.1: a) San Andreas Fault at Palmdale in California. Taken from [3]. b) Normal faults in volcanic ashes and paleo-soils, El Salvador, photo by Chuck DeMets. Taken from [1].

In addition to the geological faults shown above, there are other structures such as intrusions, where a body of igneous rocks get through existing formations, and fractures, which are separations of geological formations. The study of the origin, properties and deformation of these structures is known as Structural Geology [2].

1.2 Seismic Reflection Images

Seismic Reflection Images (SRI), also known as Seismic Reflection Profiles, are representations of the Earth's subsurface structure based on physical properties such as density, velocity and anisotropy. SRI are used to study and reproduce geophysical phenomenons by academia and industry [4]. In Fig 1.2 a SRI of the Seattle Fault Zone is shown [4]. This image represents a slice of the zone measured in distance versus depth. The structure of different layers can be seen, as well as the layer velocities in colors.

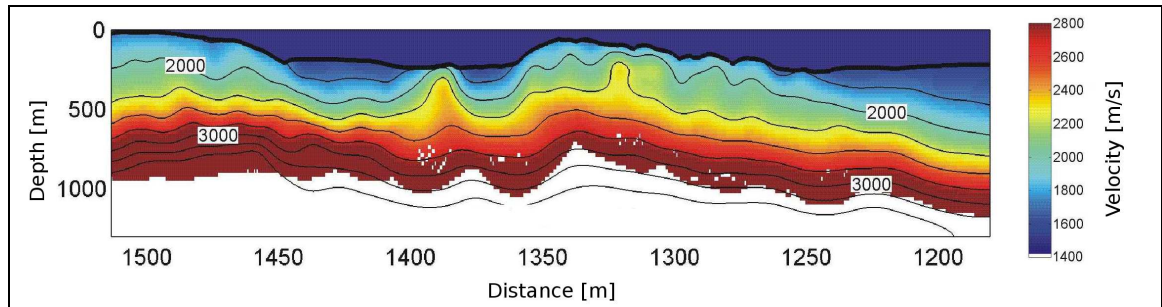


Figure 1.2: Seattle Fault Zone seismic model. Adapted from [4].

1.3 Seismic Survey

Knowing the Earth's subsurface structure is important to several fields such as seismology, geology, mineral industry and oil industry. The process that allows to obtain that information is called Seismic Survey. Seismic surveys use the reflection and refraction of energy through the subsurface in order to infer its structures and physical properties. Figure 1.3 shows the stages involved in this process [5]. Seismic surveys start with the design and planning stage. In this stage characteristics of the energy source (see Figure 1.4) used in the survey are selected, as well as the number of devices used to capture the reflected energy. These devices are named receivers and they can be geophones for on land surveys or hydrophones for marine surveys. Then the survey zone is prepared: receivers are placed in surface in the plan-ahead position using the Global Position System (GPS). Receivers are then connected to a recording station where information such as date, weather, equipment, position and the energy of reflections is stored [7].

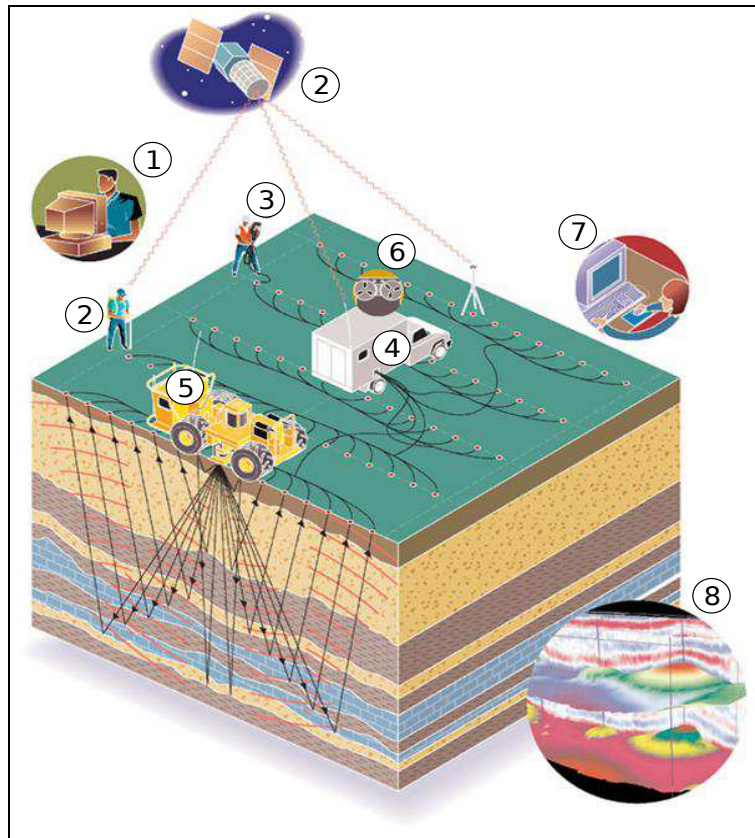
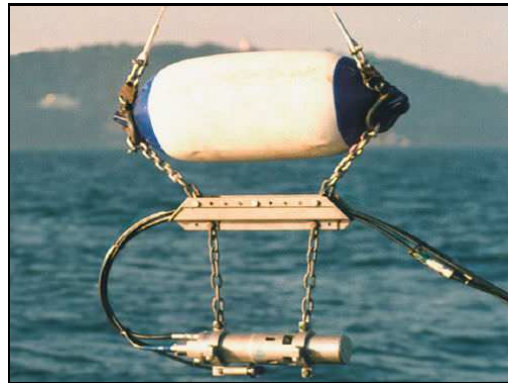


Figure 1.3: Seismic Survey process. Taken from [7].



(a)



(b)

Figure 1.4: Examples of energy sources for marine and on land surveys: a) Mini G airgun suspended from buoys. Taken from [8]. b) HEMI 60, a 61,000 Pound Vibrator. Taken from [9].

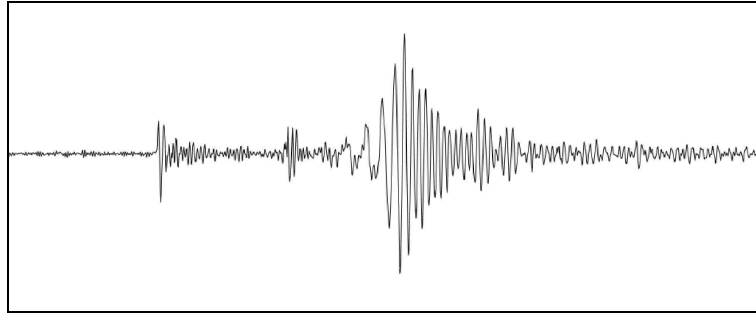


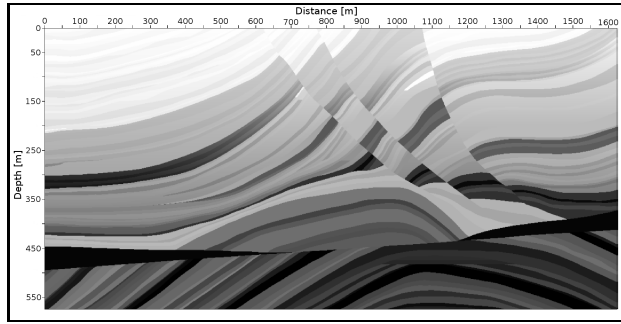
Figure 1.5: Seismic Trace. Taken from [12].

In the next stage mechanical perturbations are created through an energy source. Depending on the location, energy sources can be air cannons for marine surveys (see Figure 1.4a) or special trucks such as the Hemi-60 vibrator for on land surveys (see Figure 1.4b) [7]. Perturbation energy is propagated through the subsurface being reflected, transmitted and refracted when entering in contact with the geological structures. Energy is captured by the receivers and sent to the recording station, where it is recorded in seismic traces. A seismic trace is shown in Figure 1.5. Seismic traces from the survey are processed in post-survey stages to create density profiles, seismic reflection images, etc, that would later be analyzed by experts.

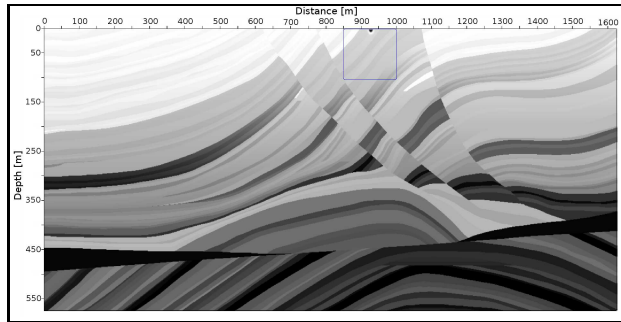
1.4 Synthetic Velocity Models

Synthetic Velocity Models (SVM) represent the velocities in the Earth's subsurface. The number of properties that the model includes, and the level of detail, can lead to really complex models and hence, costly simulations. For this reason, simple models that only consider a few properties are rather used in simulations where no more information is needed.

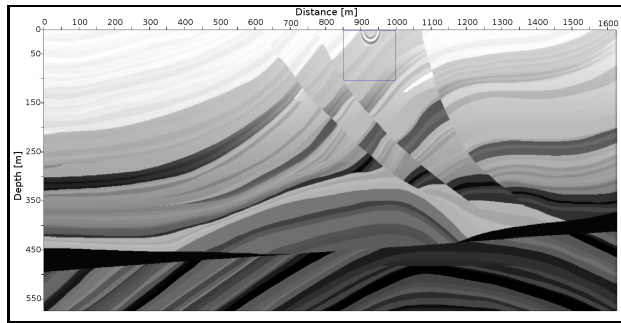
One of the most known synthetic velocity model is the Marmousi Model. Marmousi model was created in 1988 by L'Institut Français du Pétrole (IFP), based on the geologic structure through the Cuanza basin in the North Quenguela area in Luanda, Angola [14]. Several years later elastic properties were added to the Marmousi Model in [13]. In Figure 1.6a, the original Marmousi model is shown. An example of seismic modelling using the Marmousi model is shown in Figure 1.6. First, an energy source is added to the medium in Figure 1.6b. Energy propagation is then modeled with the acoustic wave equation (see equation 1.2) as shown in Figure 1.6c. As the simulation steps move forward, the energy continue spreading through the Marmousi model, as shown in Figure from 1.6d.



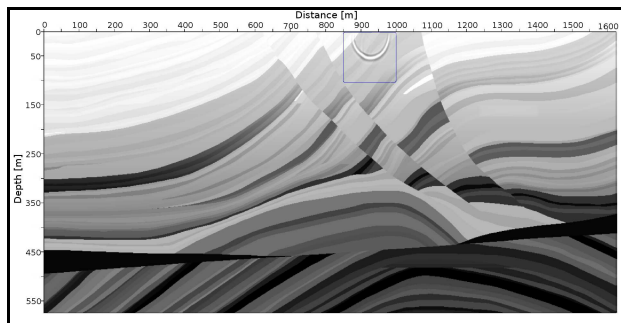
(a)



(b)



(c)



(d)

Figure 1.6: Seismic Modelling on the Marmousi Model [14]. a) Original Marmousi model. b) Energy source applied to the medium. c) and d) Energy propagation step.

1.5 Seismic Modelling

In the Seismic Modelling the energy propagation is simulated using the wave equation, a synthetic velocity model (SVM) and an energy source. Usually the initial SVM is designed from the expertise of previously made seismic surveys.

One of the most used mathematical expressions to simulate the energy source is the Ricker wavelet, proposed by the United States geophysicist Norman H. Ricker (1896 - 1980) (see equation 1.1) [10]. In that equation the wavelet is represented by $f(t)$, f_M is the peak frequency and t is the time. Figure 1.7 shows the Ricker wavelet. Energy source is simulated in the (s_x, s_y, s_z) source position in the SVM.

$$f(t) = (1 - 2\pi^2 f_M^2 t^2) e^{-\pi^2 f_M^2 t^2} \quad (1.1)$$

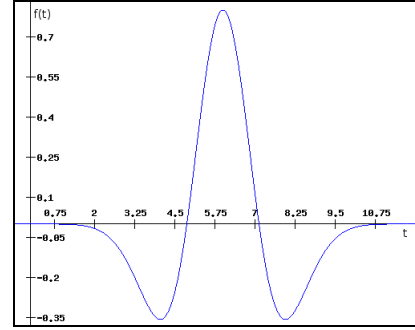
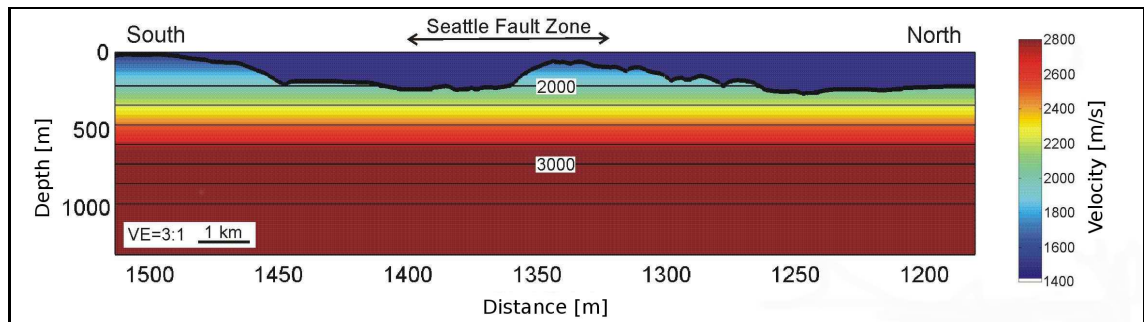


Figure 1.7: Ricker wavelet. Adapted from [11].

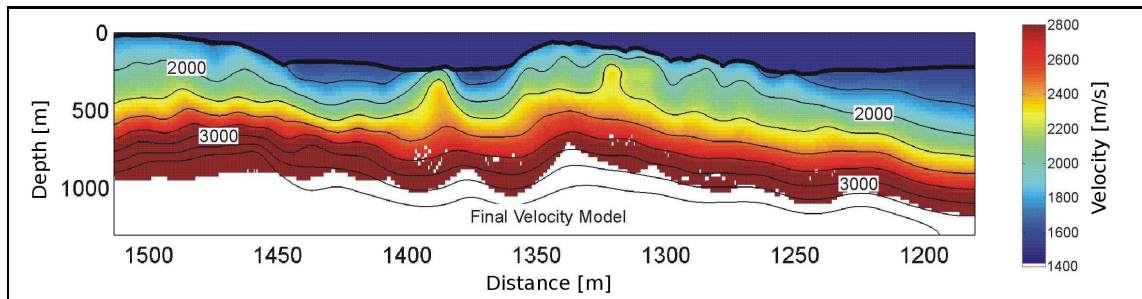
The energy propagation is then modeled with the acoustic wave equation shown in equation 1.2. Where p represents the wave field, v the seismic reflection image, t the time and x, y, z the three-dimensional (3D) spatial coordinates.

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} = \frac{1}{v^2} \frac{\partial^2 p}{\partial t^2} \quad (1.2)$$

Apart from reproducing geophysical phenomenons (e.g. subsurface energy propagation) seismic modelling is also used in algorithms such as Reverse Time Migration (RTM) and Full Wave Inversion (FWI). Moreover synthetic seismic traces can be generated by recording the energy that reaches the SVM surface in the simulation. These traces can be compared with those from the seismic survey to adjust the initial SVM. An example of an initial synthetic velocity model versus the final synthetic velocity model is shown in Figure 1.8.



(a)



(b)

Figure 1.8: Example of the initial velocity model and the final velocity model: a) Initial synthetic velocity model from Seattle Fault Zone. Taken from [4]. b) Final synthetic velocity model from Seattle Fault Zone. Taken from [4].

Graphics Processing Units

2.1 Definition

Graphics Processing Units (GPUs) are integrated circuits designed to be in charge of generating images and accelerating imaging processes, such as rendering, that take place in any computational system. For this work two NVIDIA GPUs of similar architecture generation were used: the GeForce GTX660, and the Tesla K40.

In a graphics card the GPU chip, device RAM banks and others integrated circuits are soldered in a printed circuit board (PCB). Connection between the GPU and the system is made via a communication port such as Peripheral Component Interconnect (PCI), Accelerated Graphics Port (AGP) or Peripheral Component Interconnect-Express (PCI-E) port. Figure 2.1 shows the GeForce GTX660 GPU and the main components of the graphics card.

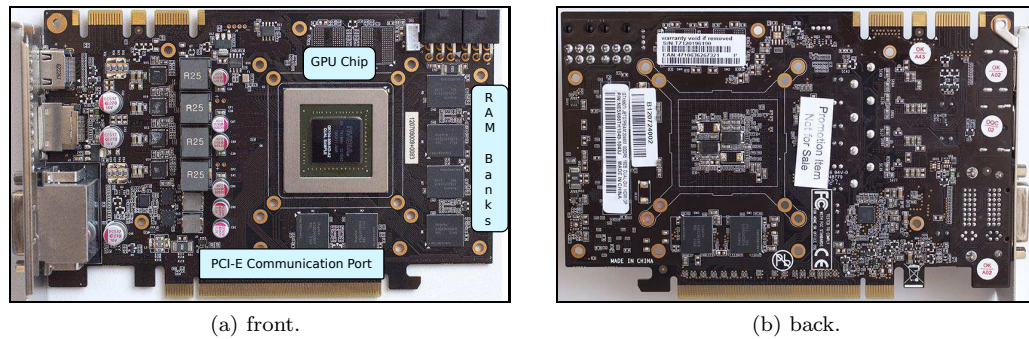
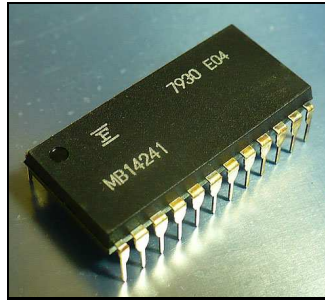


Figure 2.1: Graphics card with NVIDIA GPU GeForce GTX660. Adapted from [24].

2.2 Brief History of GPUs

The high computational cost of using RAM for frame buffers led to the beginning of the GPUs history. In the 1970s the use of Video Display Controllers (VDC) to generate color, luminance, and vertical and horizontal synchronization signals started. Some types of VDC include video shifters, video interface controllers, and video co-processors among others [18]. The Fujitsu's MB14241 (shown in Figure 2.2a) was a video shifter used in several arcade games such as Gun Fight (1975) in Figure 2.2b, Sea Wolf (1976) and Space Invaders (1978) [17].



(a) Fujitsu's MB14241.



(b) Gun Fight (1975).

Figure 2.2: 1970s Video chips and games. Taken from [21].

In the 1980s video chips increased their capabilities, supporting resolutions of 256×256 with 8-bit color data and 512×512 with 1-bit color (i.e. monochrome) data. Also, 2D display instructions to generate graphics for background, title screens and scoring display were added. In 1986 ATI releases the Original Equipment Manufacturer (OEM) Color Emulation Card starting with 16 [kB] of memory (see Figure 2.3). By 1988 ATI's Wonder series offered 16-bit color VGA emulation, 800×600 resolution and 256 [kB] of memory [19].

In the 1990s resolution supported increased from 800×600 to 1600×1200 and Application Programming Interfaces (APIs) such as OpenGL 1.0 (1992) and Direct3D (1995) were launched. Also, the level of integration of the graphics chips varied from more than 500 [nm] to 350 [nm]. 3D features began to be into the chip. Graphic memory reached the 16 [MB] and chip frequency operation was around the 166 [MHz]. AGP communication port was introduced by newer boards such as the 1997 Intel i740 AGP graphics board, shown in Figure 2.4 [19].

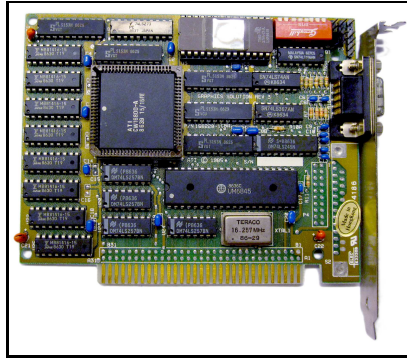


Figure 2.3: 1986 ATI Graphics Solution Revision 3. Taken from [22].

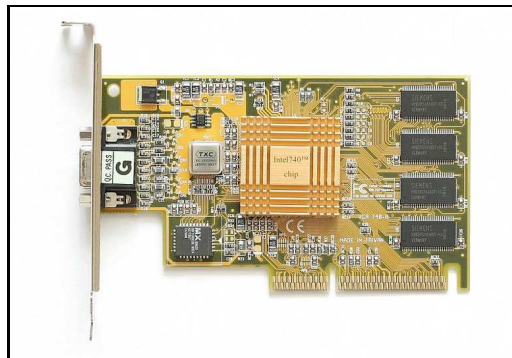
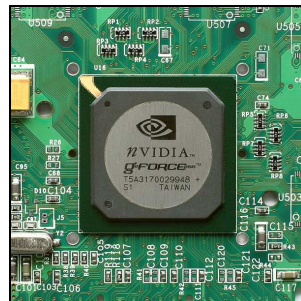


Figure 2.4: 1997 Intel i740 AGP Graphics Board [19].



(a) NVIDIA GeForce 256.



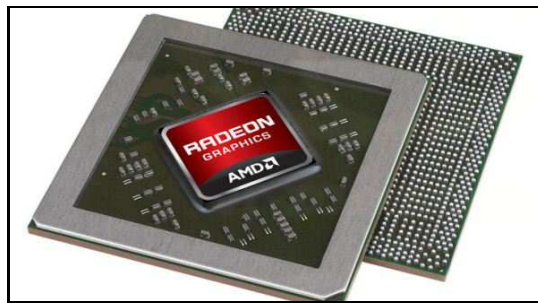
(b) VisionTek Graphic Card.

Figure 2.5: GPU NVIDIA GeForce 256 and VisionTek graphic card. Taken from [17].

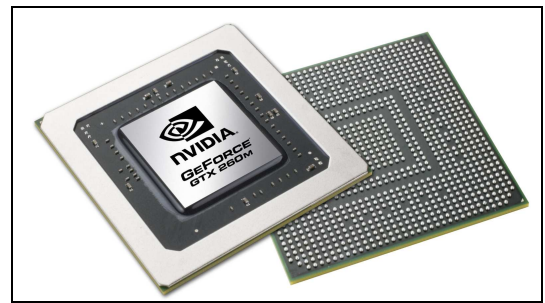
Although a number of video chip designers incorporated programmable pixels shader in prototype version, the first graphics chip that offered hardware-based Transform and Lighting (Tn&L) to consumers was the NVIDIA GeForce 256 (NV10) chip (see Figure 2.5), to which the term Graphics Processing Units (GPUs) was coined.

In the following years more capabilities were added to the GPUs. The ATI Radeon 9700 (R300) added pixel and vertex shader. Scale integration shrank from 250 [nm] to 130 [nm]. 1600×1200 resolution was now available for two screens at the same time. Device RAM memory increased from 32 [MB] to 512 [MB]. To 2006 three builders leaded the market Intel, NVIDIA and AMD (which had bought ATI recently).

GPU designers started to add more floating point units, cores, and cache levels to the GPUs chips. Also, memory in the graphics cards was increased, making the GPU a strong processing unit in fields such as data stream processing and High Performance Computing (HPC) [17]. In Figure 2.6 two GPUs from brands that lead the market are shown: AMD and NVIDIA.



(a) AMD GPU HD7000M.



(b) NVIDIA GPU GeForce GTX 256M.

Figure 2.6: AMD and NVIDIA GPUs. Taken from [23] [24].

Nowadays, GPUs can be found in many embedded systems such as last generation televisions, games consoles, laptops and desktop computers, HPC clusters and even smart phones. This means that GPUs have evolved to the point that there are GPUs for the requirements of many applications (power efficiency, number of element processed per unit of time, battery energy saving, etc.).

2.3 GPU Architecture

The GPU architecture defines how the hardware resources are distributed inside the die and can provide an idea of the behaviour between them. GPU designers improve the capabilities for each new GPU generation by adding, deleting or modifying those hardware resources. The number of processing units, the operation frequency and the memory units available are examples of those changes.

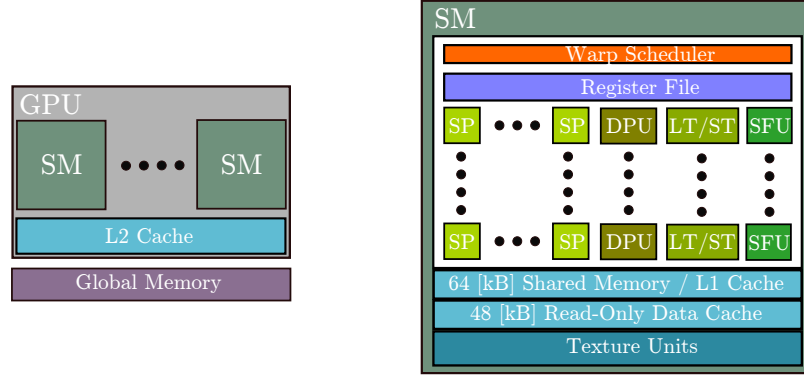


Figure 2.7: NVIDIA Kepler GPU Streaming Multiprocessor (SMX). Adapted from [25].

In Figure 2.7 the architecture of the GPUs used in this work (codenamed Kepler) are shown. The Streaming Processors (SP) or cores are represented by green squares. Tasks programmed are executed by those cores. Registers inside the SM are grouped in the Register File and are used along the cores to execute the programmed tasks. GPU cores are less complex than CPU cores. Additionally, Double Point Units (DPU) can also be used for computations requiring double precision and Special Function Units (SFU) to make special math operations such as trigonometric functions. The load and storage memory transactions are in charge of the LT/ST units. All the tasks to be executed are programmed in groups called warps, the warp scheduler unit manages the organized execution of these groups. All these resources are grouped in Streaming Multiprocessors (SM). Others resources available inside the SM include the cache L1, onstant memory, Read Only Data Cache and Texture Units.

Inside the GPU all the SMs shared a L2 Cache. All the data load from the off-chip global memory (i.e. RAM banks installed in the graphics card) is passed through the L2 cache to the SMs.

2.4 Memory Hierarchy

Memory resource units inside the GPU can be grouped according to their size and the number of clock cycles spend in their access, known as Latency. Figure 2.8 shows a hierarchy organization of those units.

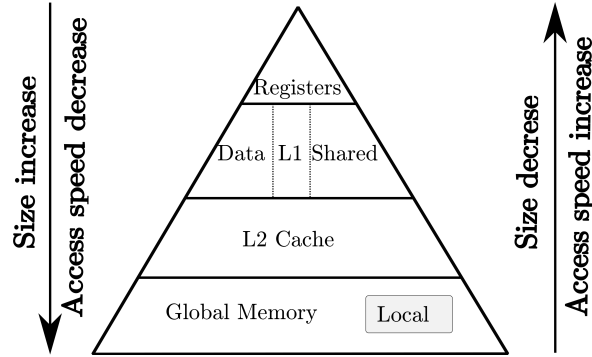


Figure 2.8: Memory Hierarchy.

From top to base of the pyramid the size of the memory units is increased (i.e. from registers to global memory), but also the number of clock cycles used to access those memory units.

A good practice to decrease the amount of clock cycles used for loading input data is to use the GPU available cache levels (L1, L2). Also, the user can enable the Shared Memory unit or choose to use the Constant Memory. To give an idea of the differences between sizes of the memory units, memory specs for the GTX660 and its graphics card are given: 64 [kB] for Shared and L1 Cache memory, 48 [kB] for the Read-Only Data Cache, 1536 [kB] of L2 Cache memory and 1.5 [GB] of off-chip global memory [25]. As can be seen, sizes in cache levels are really small in comparison with global memory, for this reason it is necessary to apply fundamental locality concepts to improve the managing of cache memory. Two of these concepts most widely recognized are the Principle of Spatial Locality and the Principle of Temporal Locality.

Principle of Spatial Locality: This principle says that there is more probability of using datum that is near to a recently used one, that datum that is far to that one.

Principle of Temporal Locality: states that data that was used before had more of probability of being used again, that data that had been never used.

2.5 GPU Programming

GPUs programming cannot be done using only traditional languages such as FORTRAN, C, C++. For this reason the development of Application Programming Interfaces (APIs) for the GPUs has become a necessity. APIs are extensions to traditional programming languages that provide a framework that include libraries, functions and documentation for developing parallel code. Examples of the most known GPUs APIs are CUDA and OpenCL.

Compute Unified Device Architecture (CUDA) is an API created by NVIDIA and launched in 2006 for parallel computing programming in GPUs. This API is available for C , FORTRAN and Python among others languages. This API integrates Software Developing Kits (SDK) such as Nsight.

Open Computing Language (OpenCL) is a development framework initially developed by Apple inc. and passed to the non-profit technology consortium Khronos, its actual developers. Khronos team has members from the companies Intel, Marvel, AMD, Qualcomm, NVIDIA Corporation, Texas Instruments, Apple, Inc., MediaTek Inc., Altera Corporation, Vivante Corporation, Xilinx, Inc., ARM Limited, Imagination Technologies, STMicroelectronics International NV, IBM Corporation, Creative Labs, Samsung Electronics. With this framework parallel code for heterogeneous systems composed by CPUs, GPUs, FPGAs, DSPs, and specific designed hardware, can be developed.

2.6 Programming Model

GPUs Programming models are representations of the tasks to be executed in the GPU. They allow the programmer to manage and organize the resources used in implementations to improve the execution. Although there are several GPU programming APIs, the concepts behind them are the same. In the following paragraphs some basic concepts are described.

The minimum task that can be executed in a GPU is named *Thread*. Inside the GPU, one core is assigned to each thread to be executed. Threads can be grouped in *Thread Blocks* as shown in Figure 2.9, and blocks can be group in a *Grid of Blocks*.

The *Kernels* can be C functions that follow the *Single Instruction Multiple Thread (SIMT)* execution model: “one instruction is programmed to be executed in parallel for many threads” [26]. Before launching the kernel, the programmer must specify the number of threads, blocks and grids to be used. These parameters are limited by the GPU hardware resources.

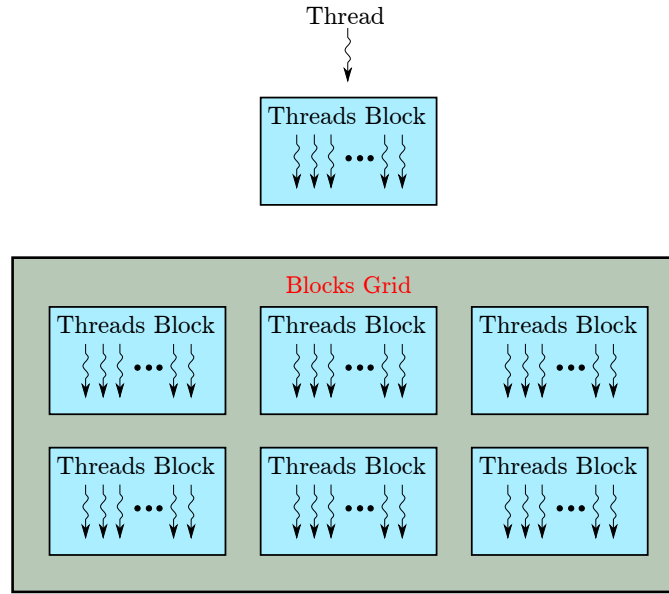


Figure 2.9: GPU Programming Model. Adapted from [26], [27].

Inside the GPU each streaming multiprocessor (SM) take the threads launched for the kernel and creates, manages, schedules and executes them in groups of 32 threads named *warps* [26].

2.7 Heterogeneous Programming

Heterogeneous programming represents the execution flow of implementations in heterogeneous systems. An example of those systems could be a system composed by a host (CPU) and many devices (GPUs). In these systems the host is in charge of the execution control; launching the kernels to be executed in the device, allocating memory and saving data in disk. Device is used for intensive compute task, and parallel tasks.

In Figure 2.10 the heterogeneous programming flow is shown. First the main code is executed in the host and the memory needed is allocated in the device. Then, host copies the input data to the device and launches a kernel specifying the number of threads per block and the number of blocks for grid. After the kernel is executed in GPU the output data is copied back to the host. In this point the host can launch another kernel or save the data and exit the execution flow. This flow can be applied to heterogeneous systems composed of CPU and FPGA, CPU and GPU, CPU and Specific Hardware and any system that uses devices for intensive compute operations.

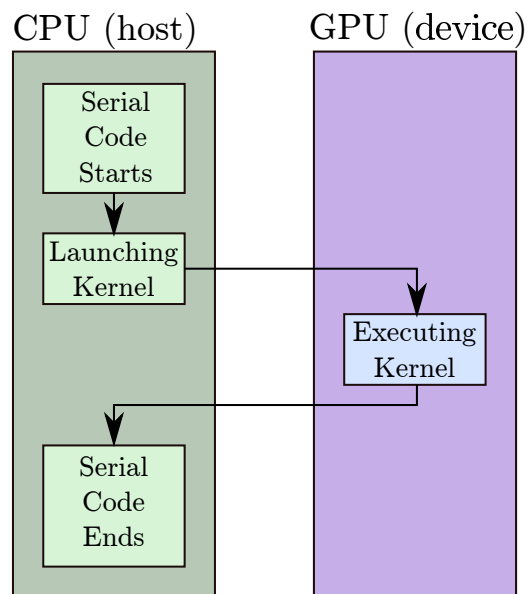


Figure 2.10: Heterogeneous Programming. Adapted from [26], [27].

Computational Implementation

3.1 Wave Equation Discretization

In chapter 1, the 3D acoustic wave equation with constant density was described (see equation 1.2). This equation can be used to simulate the propagation of a wave through a medium. To implement the solution of equation 1.2, a discretization method is needed. The discretization process consists in representing the continuous equation with a discretized expression. Discretization can be made using known numerical methods such as the Finite Elements Method (FEM) or the finite difference method, among others.

FEM was introduced by Hrennikoff in 1941 [30]. Finite differences method was proposed by Yee in 1966 [28] as a staggered grid scheme, and named Finite-Difference in Time-Domain (FDTD) by Taflov in 1980 [29]. Implementing FDTD in a GPU is an excellent alternative due to the “*parallel nature*” of the stencil (see section 3.5). For this reason the FDTD numerical method was chosen for the development of this work.

3.2 Finite Difference Time-Domain (FDTD)

Let $f(x)$ be a continuous function and suppose we are interested on finding the first derivative in $x = x_0$. If the values of the function $f(x)$ are known in $x_0 - h$, x_0 , and $x_0 + h$, where h is step size, then the $f'(x_0)$ can be approximated using the backwards, centered, or forward FDTD methods. Table 3.1 shows the graphic representations and the equations for each of these FDTD methods.

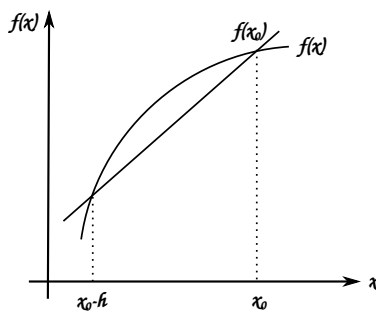
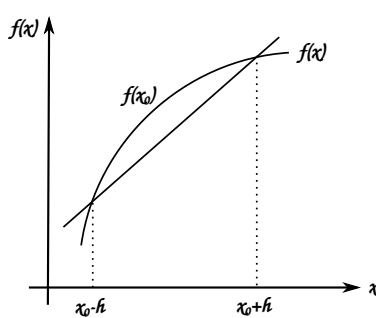
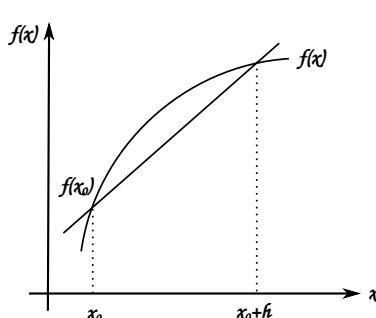
FDTD Method	Graphical Representation	Equation
Backwards		$f'(x_0) = \frac{f(x_0) - f(x_0 - h)}{h} \quad (3.1)$
Centered		$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} \quad (3.2)$
Forward		$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} \quad (3.3)$

Table 3.1: Backwards, Centered and Forward FDTD methods.

3.3 Taylor series

Taylor Series can be used to find the expressions of the FDTD for the derivative. According to Taylor's series, a function $f(x)$ can be expanded around the point x_0 with delta $\pm\delta/2$ as is shown in equations 3.4 and 3.5 [31].

$$f\left(x_0 + \frac{\delta}{2}\right) = f(x_0) + \frac{\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0) + \frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots, \quad (3.4)$$

$$f\left(x_0 - \frac{\delta}{2}\right) = f(x_0) - \frac{\delta}{2}f'(x_0) + \frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0) - \frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots \quad (3.5)$$

Let's subtract equation 3.5 from equation 3.4:

$$f\left(x_0 + \frac{\delta}{2}\right) - f\left(x_0 - \frac{\delta}{2}\right) = \delta f'(x_0) + \frac{2}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0) + \dots \quad (3.6)$$

Then, divide equation 3.6 by δ :

$$\frac{f\left(x_0 + \frac{\delta}{2}\right) - f\left(x_0 - \frac{\delta}{2}\right)}{\delta} = f'(x_0) + \frac{2}{3!}\left(\frac{\delta}{2}\right)^2 f'''(x_0) + \dots \quad (3.7)$$

Arranging equation 3.7, the representation of the centered FDTD can be obtained for the 2nd order derivative, as shown in equation 3.8.

$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \frac{f\left(x_0 + \frac{\delta}{2}\right) - f\left(x_0 - \frac{\delta}{2}\right)}{\delta} + O(\delta^2). \quad (3.8)$$

This demonstration is also valid for the Backwards and the Forward FDTD approximations.

3.4 Acoustic Wave Equation using Centered FDTD

Discretization of the acoustic wave equation (see equation 1.2) can be made using the centered FDTD scheme, as shown in equation 3.9. In that equation the spatial derivatives have been discretized with a 4th order approximation and the time derivative with a 2nd order approximation. $P_{(x,y,z)}^n$ represents the wave field in the position given by the (x, y, z) coordinates for the n time step. Coefficients for each FDTD term appear as C_i , where i is the FDTD term index. Δx , Δy , Δz are the spatial steps and Δt is the time step. At last the v term corresponds to the input Synthetic Velocity Model (SVM).

$$\begin{aligned}
& \frac{C_{-3}P_{-3,0,0}^0 + C_{-2}P_{-2,0,0}^0 + C_{-1}P_{-1,0,0}^0 + C_0P_{0,0,0}^0 + C_1P_{1,0,0}^0 + C_2P_{2,0,0}^0 + C_3P_{3,0,0}^0}{\Delta x^2} + \\
& \frac{C_{-3}P_{0,-3,0}^0 + C_{-2}P_{0,-2,0}^0 + C_{-1}P_{0,-1,0}^0 + C_0P_{0,0,0}^0 + C_1P_{0,1,0}^0 + C_2P_{0,2,0}^0 + C_3P_{0,3,0}^0}{\Delta y^2} + \\
& \frac{C_{-3}P_{0,0,-3}^0 + C_{-2}P_{0,0,-2}^0 + C_{-1}P_{0,0,-1}^0 + C_0P_{0,0,0}^0 + C_1P_{0,0,1}^0 + C_2P_{0,0,2}^0 + C_3P_{0,0,3}^0}{\Delta z^2} = \\
& \frac{1}{v^2} \frac{P_{0,0,0}^{-1} - 2P_{0,0,0}^0 + P_{0,0,0}^1}{\Delta t^2}.
\end{aligned} \tag{3.9}$$

3.5 Stencils

The FDTD method can be represented as computing templates known as *stencils*. In Figure 3.1, the 4th order stencils for 1D, 2D and 3D are shown. The central sphere is the output element and the surrounding spheres are the neighbouring input points. The order of the stencil is given by the number of input points per dimension. Input points such as SVM data and spatial points from present data and past outputs are used to compute a future output point. Since there is not dependency among future output elements, stencil computations could be made at the same time, leading to the idea that “an stencil is parallel in nature”.

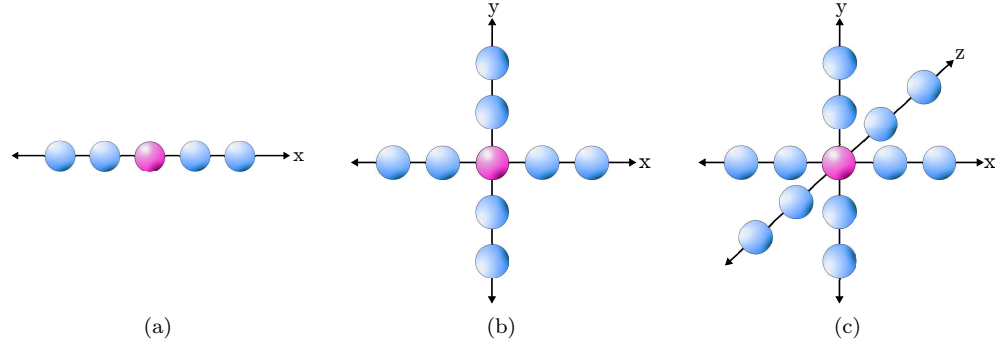


Figure 3.1: 4th order stencils in 1D, 2D, 3D. a) 1D stencil. b) 2D stencil. c) 3D stencil.

3.6 Properties of the FDTD implementation

In this section some of the basic properties of FDTD implementations are described. Properties include precision, numerical dispersion and stability.

3.6.1 Numerical dispersion

Due to the wave equation discretization (see equation 3.9) several frequency components were created, this caused waves traveling at different speeds. This undesirable effect is known as *numerical dispersion* [35]. Figure 3.2a shows this effect; several waves are traveling at different velocities. In Figure 3.2b numerical dispersion has been damp out. To decrease the numerical dispersion effect the number of terms considered in the Taylor's series should be increased. That is, the number of the FDTD terms, also known as order, should be raised.

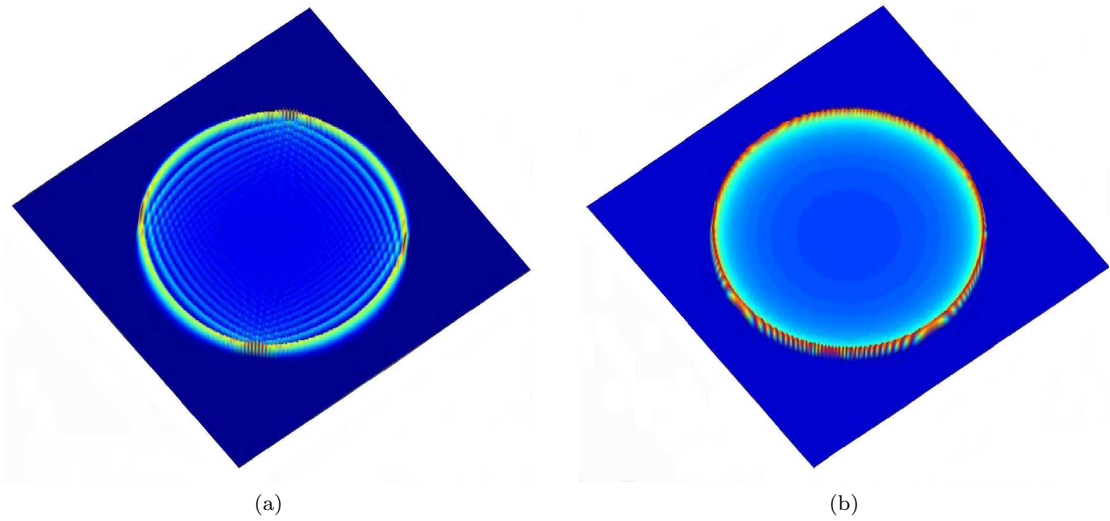


Figure 3.2: Numerical dispersion effect. a) Wave front with dispersion. b) Wave front without dispersion. Taken from [35].

3.6.2 Stability

Numerical stability is a property of numerical schemes such as finite differences. An implementation is said to be stable when error in one computation step doesn't cause an increasing error in the subsequent steps [33], [34]. According to the error's final state, an implementation can be: *Neutrally stable*, errors are constant during all computations carried out. *Stable*, errors decay or may disappear through all the computations. *Unstable*, errors increase in each computation step [34].

Several works have analyzed the stability of FDTD-based implementations. In 1999, Lines et al. [32] proposed a recipe to calculate the maximum temporal resolution δt , that guarantee numerical stability according to a previously selected spatial resolution h . Equation 3.10 shows that recipe for a 2nd order finite differences.

$$\frac{v\delta t}{h} = \frac{1}{\sqrt{2}} \quad (3.10)$$

In Table 3.2 the stability limits ($\frac{v\delta t}{h}$) for 1D, 2D and 3D using 2nd and 4th order finite differences are summarized.

<i>Dimension</i>	<i>2nd Order</i>	<i>4th Order</i>
1D	1	$\frac{\sqrt{3}}{2}$
2D	$\frac{1}{\sqrt{2}}$	$\sqrt{\frac{3}{8}}$
3D	$\frac{1}{\sqrt{3}}$	$\frac{1}{2}$

Table 3.2: Stability limits for 1D, 2D and 3D using 2nd and 4th order Finite Differences. Taken from [32].

3.6.3 Precision

Precision is a property that allows to evaluate if the amount of relevant numbers used to represent the data of a computational problem are enough to the problem scope. Two of the most used data types are *float* for single precision, and *double* for double precision. Float data is stored in 32-bits registers: 1 bit for sign, 23 bits for mantissa and 8 bits for exponent. On the other hand, double data uses 64-bits registers: 1 bit for sign, 52 for mantissa and 11 for exponent [31]. Generally, processing units, such as CPUs or GPUs, have a finite number of modules for single and double precision, usually more single modules than double. As a consequence the performance can be affected when using double precision. Rounding data or using few terms of the Taylor's series to keep some level of precision might lead to local errors, for this reason, knowing the data type that best-fit the computational problem is fundamental to decrease errors.

3.7 Two Layers Synthetic Velocity Model

For the sake of simplicity in the CPU and GPU implementations of this work, (see sections 3.8, 3.9), a 3D synthetic velocity model with two layers (*Layer 1* = 1500 [m/s], *Layer 2* = 4700 [m/s]) was created. The SVM size is given by Nx , Ny , Nz for each coordinate x , y , z . The velocity model was defined using C language (see listing 3.1) and the Visualization Toolkit (VTK) [15] was used to store data as a VTK Image Data (.VTI) file.

Visualization is made using an open-source, multi-platform data analysis application named Paraview [16]. In Figure 3.3 the 3D synthetic seismic reflection image is shown.

Listing 3.1: Two layers synthetic velocity model C code.

```

1  for(ix=0; ix<Nx; ix++) {
2    for(iy=0; iy<Ny; iy++){
3      for(iz=0; iz<Nz; iz++){
4        if (iz > (Nz/2 - 1) ){
5          c(ix,iy,iz) = 4700;}
6        else{
7          c(ix,iy,iz) = 1500;}
8      }
9    }

```

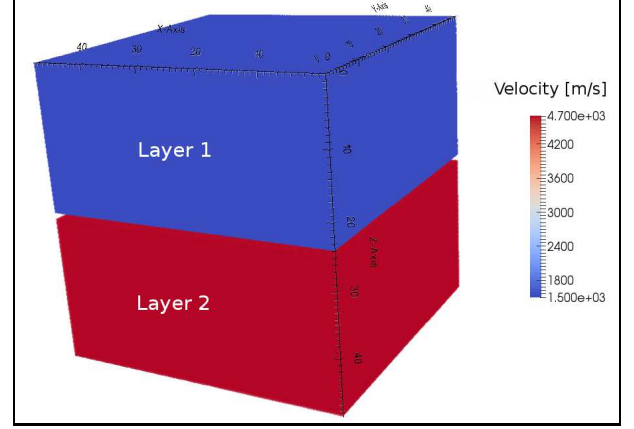


Figure 3.3: 3D two layers SVM. Visualization in Paraview.

3.8 CPU Implementation

Figure 3.4 shows the diagram flow for the CPU implementation. First a SVM and Seismic Modelling parameters are read. Specified parameters cover the number of terms of the FDTD approximation (FDTD order) Ord ; the x , y , z sizes of the modelling process Nx , Ny , Nz respectively; the x , y , z spatial resolutions dx , dy , dz ; the Ricker source position, sx , sy , sz ; Ricker source frequency fq , and the number of time steps $itmax$. Then, the stability analysis is carried out to check if the parameters accomplish with the stability conditions described in section 3.6.2 [32]. If the conditions do not meet the stability criterion, spatial resolution dx , dy , dz is adjusted. Next, the Ricker source data is computed, using the equation 1.1 described in section 1.5, for the number of time steps specified in $itmax$ and the fq frequency. Following a time step is executed $itmax$ times. The time step is composed mainly of spatial propagation, energy source addition and the updating of the computed field.

In Algorithm 1, the pseudo-code of a time iteration is shown. The spatial loops sweep through the Nz , Ny , Nx modelling space evaluating the FDTD wave equation for the approximation order selected (e.g. equation 3.9 for a 4th order approximation).

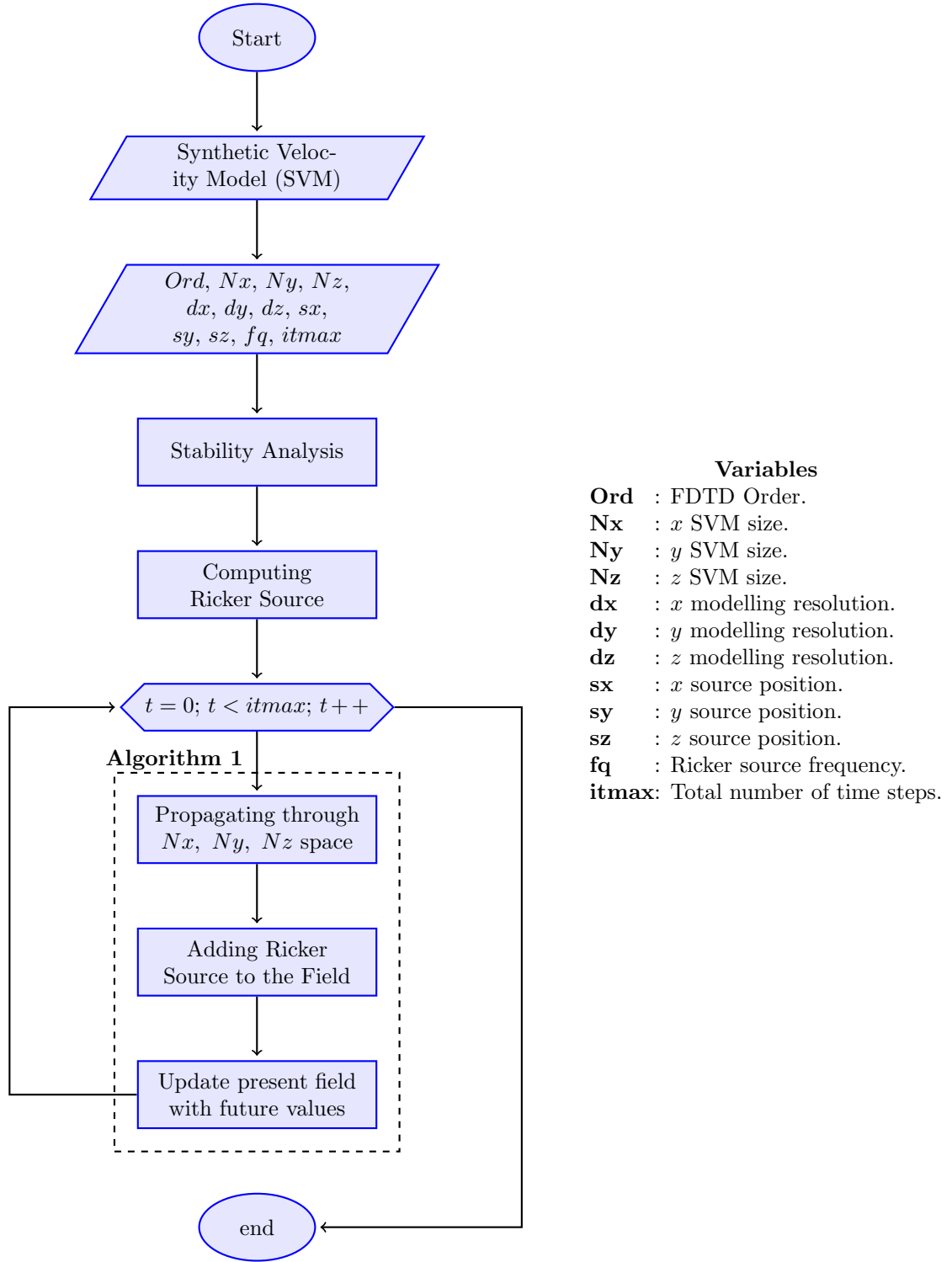


Figure 3.4: CPU implementation flow diagram.

Algorithm 1 Time Step CPU

```
1: procedure TIME LOOP
2:   Spatial loops:
3:     for ( $k = 1; k < Nz - 1; k++$ ) do
4:       for ( $j = 1; j < Ny - 1; j++$ ) do
5:         for ( $i = 1; i < Nx - 1; i++$ ) do
6:           FDTD wave equation;
7:         end for
8:       end for
9:     end for
10:  Adding Ricker Source:
11:    present field += sourcedata;
12:  Updating field:
13:    present field = future field;
14:  Saving data:
15:    writting vtk data field.
16: end procedure
```

Hence, energy from the Ricker source for each step is added and the present field is updated. Finally, the computed field is stored.

In Figure 3.5 an example of the seismic modelling implementation in CPU is shown. The computed field is stored in *vtk* format and it is visualized using the Paraview application [16]. Some of the measurements used are presented in Table 3.3.

Parameter	Description
Synthetic Velocity Model	Two layers SVM ($Layer\ 1 = 1500\ [m/s]$, $Layer\ 2 = 4700\ [m/s]$).
SVM size	$Nx = 50$, $Ny = 50$, $Nz = 50$.
Spatial resolution	$dx = 10\ [m]$, $dy = 10\ [m]$, $dz = 10\ [m]$.
Energy source position	$sx = 250\ [m]$, $sy = 250\ [m]$, $sz = 100\ [m]$.
Energy source frequency	$f_q = 20\ [Hz]$.
FDTD approximation order	16th-order (17 FDTD terms).

Table 3.3: Seismic Modelling Parameters.

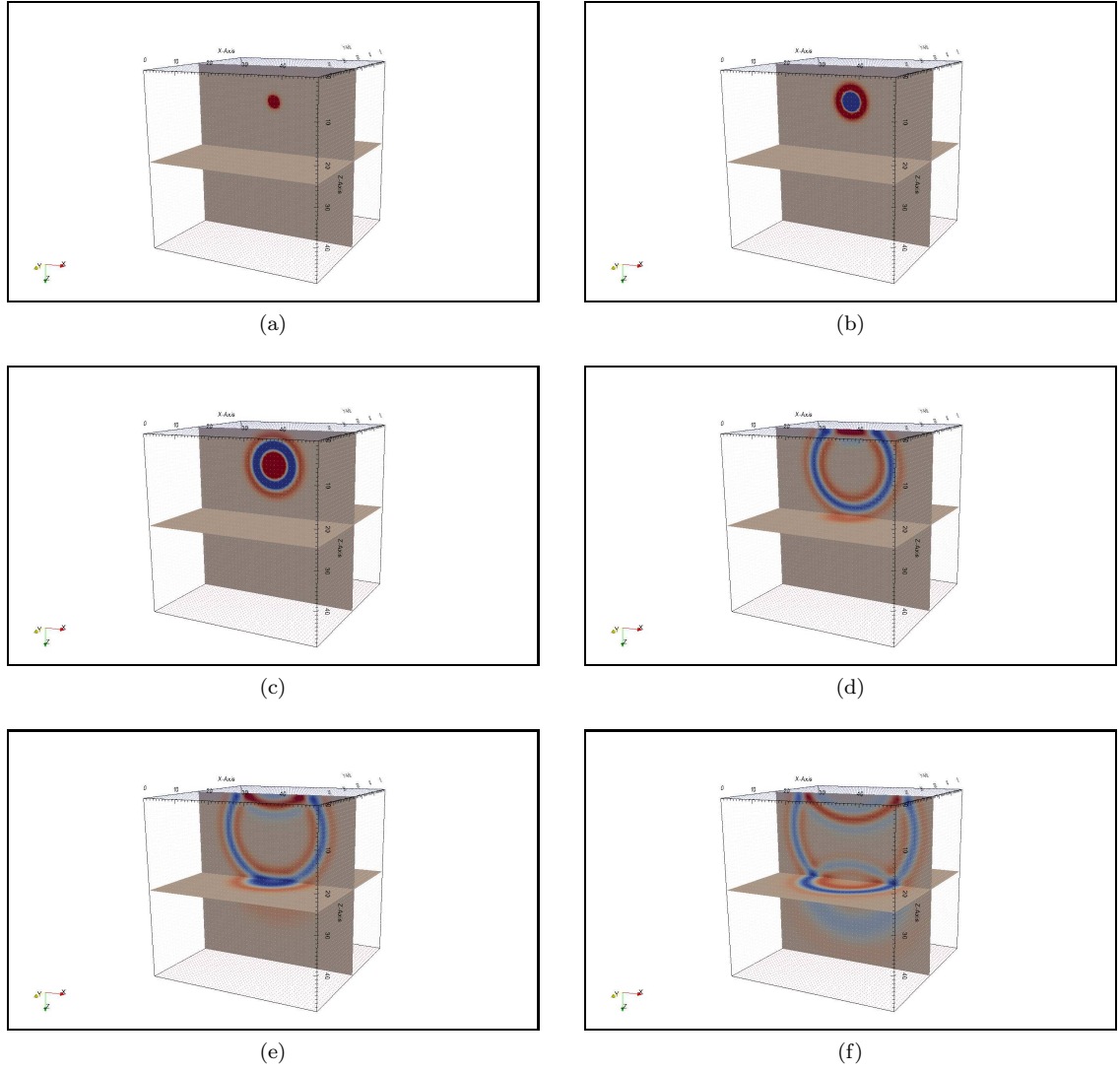


Figure 3.5: Seismic modelling implemented on CPU. All the wave propagation data is stored in “.vtk” format and visualized in Paraview. In a) and b) an Ricker energy source had been applied. c) Energy has reached the top boundary of the field. d) Energy is transmitted and reflected after entering in contact with the 2nd layer of the SVM. e) to f) Energy continue spreading through the medium.

3.9 GPU Implementation

Figure 3.6 shows the diagram flow for the GPU implementation, similar in some stages to the CPU implementation diagram flow. First a SVM and seismic modelling parameters are read. Specified parameters are:

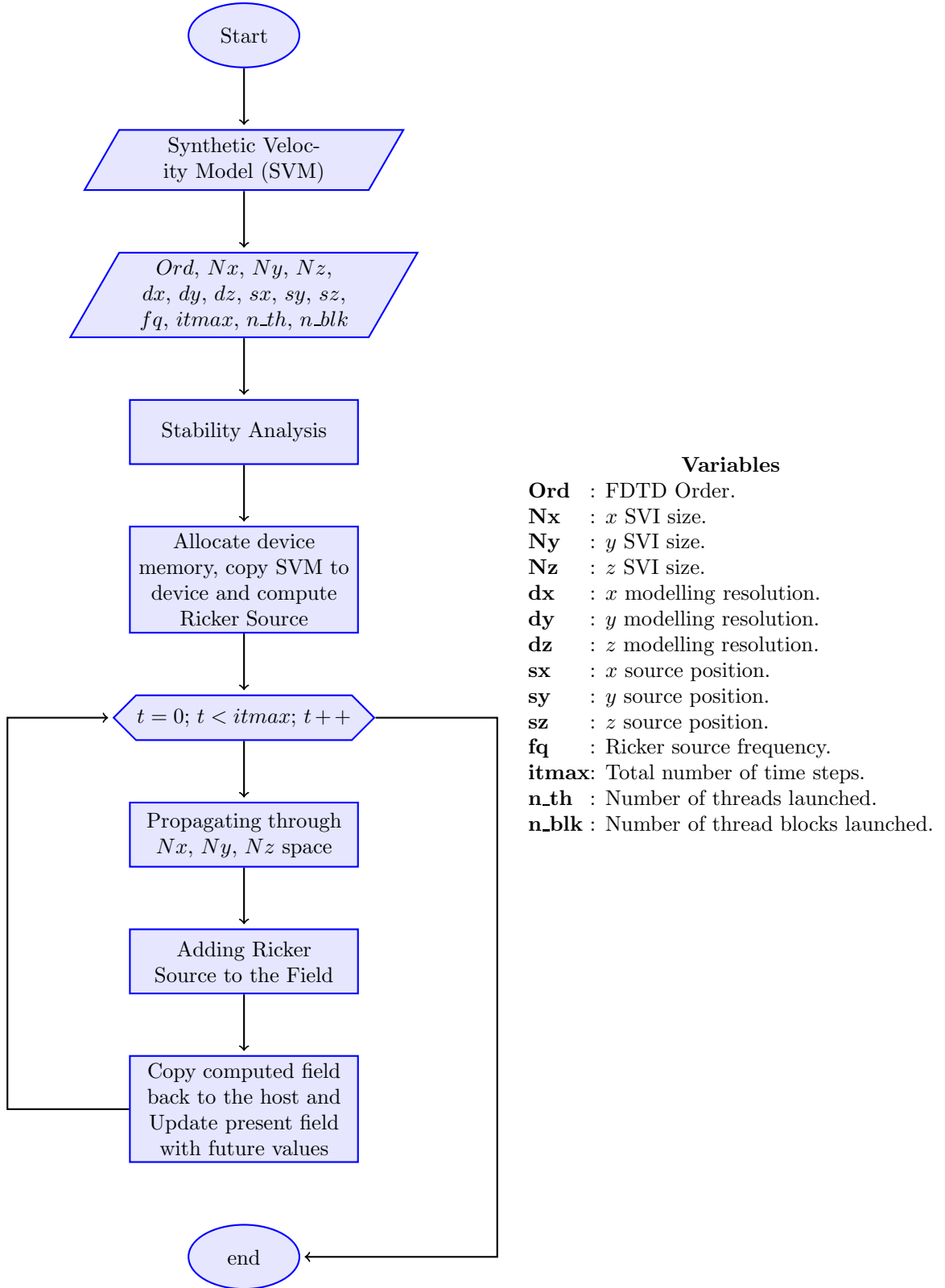


Figure 3.6: GPU implementation flow diagram.

the number of terms of the FDTD approximation (FDTD order) Ord ; x, y, z sizes of the modelling process Nx, Ny, Nz respectively; x, y, z spatial resolutions dx, dy, dz ; the Ricker source position, sx, sy, sz ; Ricker source frequency f_q , and the number of time steps $itmax$ and two parameters related to the number and distribution of parallel computing elements used; the number of threads launched (n_{th}), and the number of thread blocks used (n_{blk}).

After the stability analysis stage is performed, allocation of memory for the propagation fields and the SVM is made in the device. Following the SVM is copied from host memory to device memory. Then, the Ricker source is computed in device using the kernel shown in Listing 3.2.

Listing 3.2: Ricker Source Kernel.

```

1  __global__ void ricker_knl(float fq, float dt, int itmax,
2                               struct wavelet source) {
3
4  /* Thread index */
5  int it = threadIdx.x + blockIdx.x*blockDim.x;
6
7  /* Constraining threads */
8  if(it < itmax) {
9
10 /* Filling source data*/
11 t = ((float)it - 1.0f)*dt - (1.0f/fq);
12 arg = ((float)M_PI)*((float)M_PI)*fq*fq*t*t;
13 source.data[it]= (1.0f - 2.0f*arg)*(exp(-arg)); }

```

A time step is executed $itmax$ times. The time step is composed mainly of spatial propagation, energy source addition and the updating of the computed field. In Listing 3.3, the kernel for the spatial propagation is shown. In the GPU, each spatial output point is computed by a thread, using the FDTD wave equation (see equation 3.9). This means that, according to the number of available threads and field points the spatial output field can be computed at once.

Listing 3.3: Stencil Kernel.

```

1  __global__ void stencil_eval_gpu(int  Nx, int Ny, int Nz, /* Model size */\
2      float* p,                      /* Past field */\
3      float* q,                      /* Present field */\
4      float* r,                      /* Future field */\
5      float* c,                      /* Velocity field */\
6      float dt,                      /* Time step */\
7      float dx,                      /* Model resolution */\
8      float dy,                      /*      (dx,dy,dz) */\
9      float dz,                      /* */\
10     int  Ord,                      /* FD aprox. order */\
11     float* coef,                  /* 2 deriv. FD coef */\
12
13     /* 3D Thread index */
14     int idx = threadIdx.x + blockIdx.x*blockDim.x;
15     int idy = threadIdx.y + blockIdx.y*blockDim.y;
16     int idz = threadIdx.z + blockIdx.z*blockDim.z;
17
18     /* Constraining threads */
19     if(idx < Nx && idy < Ny && idz < Nz) {
20
21         FDTD wave equation (stencil)
22
23     }
24 }
```

After that, the energy from the Ricker source for that step is added using the kernel shown in Listing 3.4. Finally, the data is copied back to the host to be stored, and the present field is updated.

Although GPU implementation looks similar to the CPU implementation there is a little difference between data. In the scope of this work that amount of error is not relevant as far as the results are similar.

Listing 3.4: Addition Kernel.

```
1  __global__ void add_source_gpu(int addsubflag, int Nx, int Ny, int Nz, int its,
2                                float *r, struct wavelet source){
3
4      /* Declaring variables*/
5      int i = threadIdx.x + blockIdx.x*blockDim.x;
6
7      /* Constraining threads */
8      if (i < source.ntr){
9          if(addsubflag==1)
10             r(source.x[i], source.y[i], source.z[i]) += source.data[i*source.ns+its];
11         else
12             r(source.x[i], source.y[i], source.z[i]) -= source.data[i*source.ns+its];
13     }
14 }
```

Modelling of the Implementation

This chapter presents the methodology used in this work, some of the analytical models studied, the selected model and the process of adapting the model. Also, the identification and extraction of the parameters needed to validate the model. Finally, the discussion and conclusions of this work are presented at the end of the chapter.

4.1 Methodology

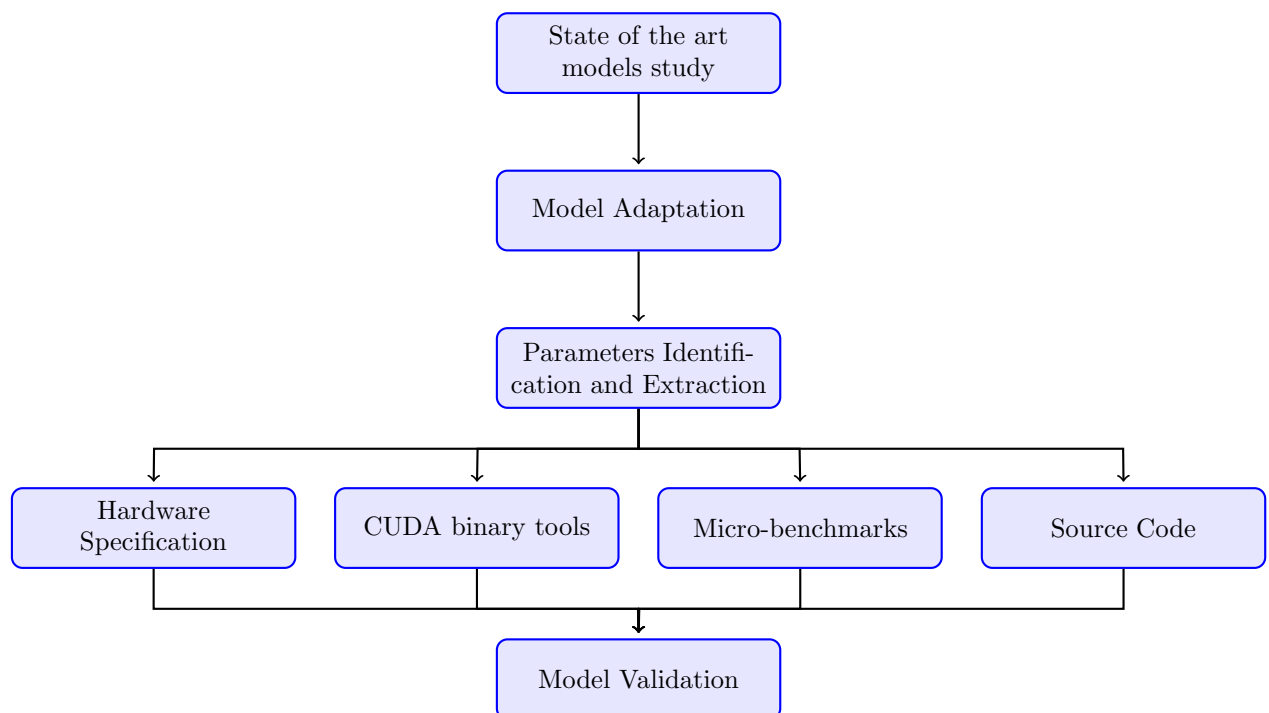


Figure 4.1: Methodology.

The methodology utilized in this work is shown in Figure 4.1. First, several analytical models from the literature are studied to determine the characteristics of the GPUs analytical models. Secondly, analytical models with similar characteristics to those found in the seismic modelling implementation, presented in chapter 3, are selected. Then, the best-fit model is adapted to consider the implementation features (see sections 4.3 and 4.4). Hence, the identification and extraction of the parameters needed by the model is performed. In this stage, hardware specifications, micro-benchmarks and CUDA binary tools are taken into account to obtain those parameters (see section 4.5). Once all the data had been gathered, the validation process is made as it is shown in section 4.6.

4.2 Related Work

A GPU Analytical Model is a set of equations that involve parameters to estimate performance measures, such as the execution time, of a GPU implementation. According to the number of different GPU implementations that the analytical model can estimate, they are considered as *General Analytical Models* or *Specific Analytical Models*. Usually, specific models have less error than general models due to their inner detail. However, these models are only suitable for few implementations. There are some examples of specific models in the literature: in [43] the authors proposed a tool named CuMAPz that models memory effects as data reuse; global memory access coalescing; shared memory bank conflict; etc. Another example is the performance model of a sparse matrix-vector multiply (SpMV) operation presented in [41]. Unfortunately, there aren't specific analytical models for applications with similar characteristics to those of the seismic modelling implementation presented in chapter 3.

On the other hand, general models have a greater percentage error than the specific models because they cover common characteristics of many implementations. Among the general models, the MAX(SUM) model that merges classical models such as the Bulk Synchronous Parallel (BSP) model, the PRAM model and Queue-Read-Queue-Write (QRQW) PRAM model is proposed in [39]. This model calculates the execution time of a program that consists of several kernels by summing the time spent by each kernel. This model is tested with Matrix multiplication, List Ranking, and histogram generation, although the model was proposed in 2009 for past GPU architectures, it gives some ideas about computational models. In [40] a model that is based on Control Flow Graphs (CFG), Work Flow Graphs (WFG) and the Instruction-Level Parallelism (ILP) of GPU kernels to estimate their execution time is presented. This model was tested in matrix multiplication, prefix sum scan, FFT, and sparse matrix-vector benchmarks. After studying a set of state of the art models and according to [47] one the most accepted analytical models in the literature is the MWP-CWP model proposed in [44] and adjusted to be used in a Performance Analysis Framework in [45]. This model has been validated for the GPU implementations of tiled matrix multiplication algorithms, and the Fast Multipole Method (FMM) algorithm and will be adapted in this work to the seismic modelling implementation.

Furthermore, a different model for a 2.5D stencil implementation in a GPU was proposed in [46], but due to the large difference between the seismic modelling implementation this model was not used.

4.3 Memory Warp Parallelism-Compute Warp Parallelism model

The Memory Warp Parallelism-Compute Warp Parallelism (MWP-CWP) model [44] [45] uses the number of instructions, level of parallelism, and GPU architecture parameters to estimate the execution time of a given kernel. Its two main concepts are described below.

Memory Warp Parallelism (MWP):

The MWP measures the memory parallelism warp level per SM (*i.e.*, how many warps can access to the memory simultaneously).

Compute Warp Parallelism (CWP):

The CWP represents the warps that can executed a computing period while during a memory waiting period plus one.

According to [44] and [45], an implementation can be characterized in terms of memory and computation, resulting in 3 cases: when $MWP < CWP$, or $MWP \geq CWP$, or *Not enough warps*.

1. $MWP < CWP$: The cost of computation is less than the cost of accessing memory, therefore the execution time is given by the memory operations.
2. $MWP \geq CWP$: The cost of memory operations is less or equal than the cost of computing operations, thus, the execution time is given by the compute operations.
3. *Not enough warps*: The number of warps is not enough to guarantee parallelism, so the memory and compute cost fully affect the execution time.

The first equation of the MWP-CWP analytical model establishes the execution time (T_{exec}) as the sum of the compute operations time (T_{comp}) and the memory operations time (T_{mem}), minus the overlap time ($T_{overlap}$) between (T_{comp}) and (T_{mem}) as it is shown in equation 4.1.

$$T_{exec} = T_{comp} + T_{mem} - T_{overlap}. \quad (4.1)$$

Hence, T_{comp} considers the parallel computation cost ($W_{parallel}$) and the serial computation cost (W_{serial}) (see equation 4.2).

$$T_{comp} = W_{parallel} + W_{serial}. \quad (4.2)$$

The $W_{parallel}$ equation (eq. 4.3) includes several parameters such as: the number of kernel compute instructions ($\#inst$), the number of total warps launched ($\#total_warps$), the number of SMs running the kernel ($\#active_SMs$), the average floating point instruction latency (avg_inst_lat) and the amount of parallelism among threads represented by the inter-thread instruction-level parallelism ($ITILP$).

$$W_{parallel} = \frac{\#inst \times \#total_warps}{\#active_SMs} \times \frac{avg_inst_lat}{ITILP} \quad (4.3)$$

Equations 4.4 to 4.6 are used to calculate the $ITILP$, where the ILP represents the instruction level parallelism among warps (see equation 4.5). To compute the ILP , the instructions are grouped in blocks known as Basic Blocks (BBs).

$$ITILP = \min(ILP \times N, ITILP_{max}) \quad (4.4)$$

$$ILP(MLP)_{AVG} = \sum_{K=1}^{\#BBs} \frac{ILP(MLP)_K \times \#accesses_to_BB_k}{\#basic_blocks} \quad (4.5)$$

$$ITILP_{max} = \frac{avg_inst_lat}{warp_size/SIMD_width} \quad (4.6)$$

On the other hand, the W_{serial} equation (eq. 4.7) considers the computational cost due to: synchronization O_{sync} , use of special function units (O_{SFU}), execution of additional instruction in divergent branches O_{CFdiv} and the cost due to the shared memory bank conflicts (O_{bank}).

$$W_{serial} = O_{sync} + O_{SFU} + O_{CFdiv} + O_{bank} \quad (4.7)$$

The memory operations time (T_{mem}) is calculated as shown in equation 4.8. As can be seen, the $\#total_warps$, $\#active_SMs$ parameters are the same for T_{comp} and T_{mem} . Other parameters include the number of memory instructions ($\#mem_inst$), the average memory access time $AMAT$, and the inter-thread memory-level parallelism ($ITMLP$).

$$T_{mem} = \frac{\#mem_inst \times \#total_warps}{\#active_SMs \times ITMLP} \times AMAT \quad (4.8)$$

The $AMAT$ equation considers the cache hit latency hit_lat , the cache miss ratio $miss_ratio$ and the average DRAM access latency $DRAM_latency$. $DRAM_latency$ is calculated as it is shown in equation 4.10, by using the minimum departure distance between two consecutive memory warps (Departure Delay (Δ)) and the average number of transactions per memory request in a warp (avg_trans_warp).

$$AMAT = avg_DRAM_lat \times miss_ratio + hit_lat \quad (4.9)$$

$$avg_DRAM_lat = DRAM_lat + (avg_trans_warp - 1) \times \Delta \quad (4.10)$$

The $ITMLP$ is given by equation 4.11, where the MLP represents the a intra-warp memory-level parallelism.

$$ITMLP = \min(MLP \times MWP_{cp}, MWP_{peak_bw}) \quad (4.11)$$

Then, equations from 4.12 to 4.19 are used to calculate the $ITMLP$.

$$MWP_{cp} = \min(\max(1, CWP - 1), MWP) \quad (4.12)$$

$$CWP = \min(CWP_full, N) \quad (4.13)$$

$$CWP_full = \frac{mem_cycles + comp_cycles}{comp_cycles} \quad (4.14)$$

$$comp_cycles = \frac{\#insts \times avg_inst_lat}{ITILP} \quad (4.15)$$

$$mem_cycles = \frac{\#mem_inst \times AMAT}{MLP} \quad (4.16)$$

$$MWP = \min \left(\frac{avg_DRAM_lat}{\Delta}, MWP_peak_bw, N \right) \quad (4.17)$$

$$MWP_peak_bw = \frac{mem_peak_bandwidth}{BW_per_warp \times \#active_SMs} \quad (4.18)$$

$$BW_per_warp = \frac{freq \times transaction_size}{avg_DRAM_lat} \quad (4.19)$$

$$transaction_size = Data_size \times \#threads_per_warp \quad (4.20)$$

The $T_{overlap}$ corresponds to the time overlap between the computational and the memory operations. $T_{overlap}$ is calculated using equation 4.21. The overlap function (see equation 4.22) depends on the MWP , and CWP values, as well as the number of concurrently running warps N .

$$T_{overlap} = \min(T_{comp} \times F_{overlap}, T_{mem}) \quad (4.21)$$

$$F_{overlap} = \frac{N - \zeta}{N}, \zeta = \begin{cases} 1 & \text{if } (CWP \leq MWP) \\ 0 & \text{if } (CWP > MWP) \end{cases} \quad (4.22)$$

4.4 Adapted Analytical Model and Parameter Identification for the 3D-Stencil Case

In this section, the adaptation of the MWP-CWP general model to a 3D stencil-based kernel implementation is presented. Note that due to the parallel nature of the GPU implementation exposed in chapter 3, knowing the execution time of one stencil would be enough to estimate the execution time of a 3D spatial volume. The adaptation consists of modifications to the MWP-CWP model equations considering serial factors (e.g. synchronization, divergent branches, etc), GPU cache levels, instruction level parallelism, memory level parallelism and departure delay.

4.4.1 W_{serial} equation

The W_{serial} equation (eq. 4.7) take into consideration the costs of using synchronization barriers O_{sync} , bank conflicts in shared memory (O_{bank}), employing special function units (O_{SFU}) and the cost of divergent branches O_{CFdiv} . As shown in chapter 3, the implementation does not have synchronization barriers meaning that the O_{sync} is zero. Also, it does not use shared memory neither SFU, for these reason the cost associated to that memory (O_{bank}) and those special units (O_{SFU}) can be neglected. Besides, the Control Flow Graph (CFG) of stencil-based application (see section 4.5) was analyzed, observing that they were not significant branches created annulling the O_{CFdiv} term. Therefore the W_{serial} term is equal to zero, making the computational cost equal to the parallel work W_{parallel} , this is shown in equation 4.23.

$$T_{\text{comp}} = W_{\text{parallel}}. \quad (4.23)$$

4.4.2 AMAT equation

According to [44], [45], $AMAT$ equation (eq. 4.9) can be modified to involve other cache levels. Kepler' GPUs have 2 cache levels: the $L1$ cache level inside the SM and the $L2$ cache level outside the SM. Hence, the $AMAT$ equation was modified as it is expressed in equation 4.24.

$$AMAT = avg_DRAM_lat \times miss_ratio_l2 + hit_lat_l2 \times miss_ratio_l1 + hit_lat_l1. \quad (4.24)$$

4.4.3 Departure Delay (Δ)

By using the definition presented in [44], equation 4.25 is proposed to compute the departure delay. The input data are the global memory latencies using 1 warp, 2 warps, and so on, until 32 warps. Latencies are obtained through the micro-benchmarks presented in section 4.5.

$$\Delta = \frac{\sum_{i=1}^{\#warps-1} (\delta_i)}{\#warps - 1} = \frac{\sum_{i=1}^{\#warps-1} (warps_{i+1} - warps_i)}{\#warps - 1} \quad (4.25)$$

4.4.4 ILP

The description provided in [44] to calculate the *ILP* lacks of detail in the process for extracting the basic blocks. For this reason, according to the CFG of the stencil-based implementation the equation 4.5 was replaced for an expression that follows the *ILP* definition involving the number of total instructions and the number of basic blocks (see equation 4.26). From here, each basic block corresponds to the execution independent instructions that can be grouped.

$$ILP = \frac{\#total_instructions}{\#basic_blocks} \quad (4.26)$$

4.5 Parameters Identification and Extraction

The list of the parameters needed by the adapted model to estimate the execution time of a 3D-stencil and a brief description of them is given in Table 4.1.

Table 4.1: Parameters used for the adapted model. Adapted from [45].

<i>Model Parameter</i>	<i>Definition</i>
<i>#inst</i>	# of total instructions per warp.
<i>#mem_inst</i>	# of memory instructions per warp.
<i>#FP_inst</i>	# of floating point instructions per warp.
<i>#total_warps</i>	Total number warps in kernel.
<i>#active_SMs</i>	# of active SMs.
<i>N</i>	# of concurrently running warps on one SM.
<i>AMAT</i>	Average memory access latency.
<i>avg_trans_warp</i>	Average memory transactions per memory request.
<i>avg_inst_lat</i>	Average instruction latency.
<i>miss_ratio</i>	Cache miss ratio.
<i>size_of_data</i>	The size of input data.
<i>ILP</i>	Instructions-level parallelism in one warp.
<i>MLP</i>	Memory-level parallelism in one warp.
<i>MWP (per SM)</i>	Max #warps that can concurrently access memory.
<i>CWP (per SM)</i>	# of warps executed during one memory period plus one.
<i>MWP_{peak_bw} (per SM)</i>	MWP under peak memory band width.
<i>warp_size</i>	# of threads per warp.
Δ	Transaction departure delay.
<i>DRAM_lat</i>	Baseline DRAM access latency.
<i>FP_lat</i>	FP instruction latency.
<i>hit_lat</i>	Cache hit latency.
<i>SIMD_width</i>	# of scalar processors (SPs) per SM.

The process of extracting the parameters listed in Table 4.1 is described here. The set of parameters required to estimate the execution time of the 3D-stencil, are extracted from four different sources:

1. The GPU hardware specifications [54].
2. The source code of the stencil based implementation.
3. CUDA binaries tools [48], [55], [57], [53].
4. Specific pseudo assembly routines known as micro-benchmarks (ubmk).

The correct extraction of the parameters is of major importance since the validation of the analytical model with the measured execution times relies on it.

4.5.1 Hardware Specifications

The GPU hardware specifications include the number of cores per SM, the number of SMs, cores clock frequency, among others. These specifications allow knowing in advantage the available resources in the GPU and the physical chip limitations. Some of these specifications are parameters of the analytical model. In Table 4.2, the hardware characteristic and its correspondent value for the NVIDIA K40 Kepler GPU, are shown.

In addition, the last column of Table 4.2 shows the name of the parameter that is directly computed with the hardware characteristic. Also, those hardware characteristics represented by — mean that the characteristic is indirectly used in the computation of one or many parameters. The specifications given in the table are extracted via the execution of the CUDA samples, which is a set of NVIDIA CUDA applications samples.

Table 4.2: Hardware specifications used directly or indirectly to compute the set of parameters.

<i>Hardware Specification</i>	<i>K40</i>	<i>Parameters</i>
Number of multiprocessors (SMs)	15	—
Number of cores	2880	—
Number of cores/SM	192	<i>SIMD_width</i>
GPU clock rate	745 [MHz]	freq
Memory bus width	384-bit	—
L2 Cache Size	1572864 bytes	—
Warp size	32	<i>warp_size</i>
Bandwidth	183644.7 [MB/s]	<i>mem_peak_bw</i>

¹ — indirectly used.

4.5.2 Parameters from CUDA Binary Tools

There are many tools to obtain the information of the resources used by the GPU. Some of those tools allow extracting information while executing the kernel (metrics and events [53]). As for the rest of tools, they analyze the binary files created in the source code compilation process (e.g. CUDA binary file (CUBIN)) [48], [55], [57]. The parameters obtained using this set of tools are:

a. Number of instructions per warp (*#inst*):

This parameter can be found by counting the number of instructions in the Control Flow Graph (CFG) generated from the cubin file. Reading the metric “*inst_per_warp*” while running the executable can give an idea of the number of instructions. As the executable file could differ from the cubin, differences can be found between the two measures.

In Figure 4.2 the CFG of a kernel that sums an input data element with a constant is shown. The CFG represents the GPU instructions using a pseudo assembly language known as Parallel Thread Execution (PTX) [57]. Some of the instructions include; load from global memory (*LD.E*), float addition (*FADD*) and storage to global memory (*ST.E*).

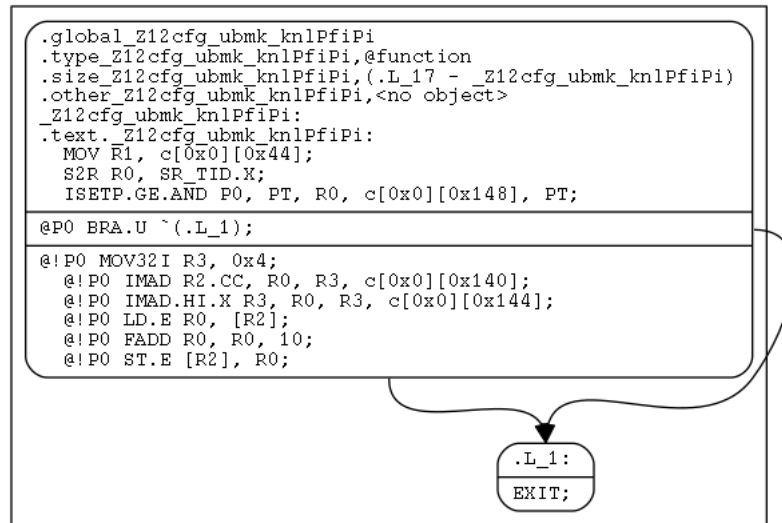


Figure 4.2: Control Flow Graph.

b. Instruction level parallelism (*ILP*):

To calculate the *ILP* parameter, the kernel instructions are counted using the CFG generated. Then, instructions with not execution dependences are grouped into basic blocks. Knowing the number of instructions and the number of blocks, the *ILP* is computed using the equation 4.26.

c. Average memory transactions per memory request (*avg_trans_warp*):

This parameter represents the average memory transactions generated per warp memory request. It can be obtained from the metric “*gld_transactions_per_request*”.

d. Memory level parallelism (*MLP*):

Represents the number of memory requests between a load request and the first instruction that sources the memory request [44]. *MLP* is equal to the number of instructions counted before the first load instruction.

e. *Miss_ratio*:

The miss ratio between *L1* cache and global memory is a metric that depends on the implementation. It can be obtained subtracting the CUDA metric *l1_cache_global_hit_rate* from the total number of cache requests, represented here by 100%. A description of the metrics and events supported for profiling can be found in [53].

f. *Miss_ratio_l1_l2*:

The miss ratio between *L2* and *L1* cache memory can be computed by subtracting the CUDA metric *l2_l1_read_hit_rate* from the total number of cache requests (100%). Similar to the *l1_cache_global_hit_rate* metric, the value of this metric depends on the implementation.

4.5.3 Parameters from Micro-benchmarks

The parameters presented in this section can not be obtained by using the previously described tools, they are obtained via micro-benchmarks that employ the GPU special registers. A GPU counts of several registers inside each SM. One of these registers, is the special register *%clock* in charge of counting the number of clock cycles passed inside the SM. The micro-benchmarks presented here were designed to read the *%clock* register at the beginning and the end of an assembly instruction. The difference between the recorded time values is calculated and the cost of reading the *%clock* register is subtracted obtaining the number of clock cycles taken to execute the instruction. Each micro-benchmark was executed 50 times launching from 1 warp (32 threads) to 32 warps (1024 threads).

The output latency was calculated taking the average among threads and iterations. The micro-benchmark results show in this section were implemented in an NVIDIA K40 Kepler GPU.

a. Latency of the special register `%clock`:

The code used for this micro-benchmark is shown in Listing 4.1. It consists of reading the `%clock` register twice. The difference between the (*start*) and the (*stop*) time values is the cost of using the `%clock` register.

Listing 4.1: `%clock` register micro-benchmark assembly.

```

1  __asm__("mov.u32 %00, %%clock; \n\t"// Start cycle
2      "mov.u32 %01, %%clock;      "// Stop cycle
3      : "=r" (start), "=r" (stop));
4
5  d_start[idx] = start; d_stop[idx] = stop;
6  clk_cycles[idx] = (int)(stop-start);// Difference

```

Figure 4.3 shows the number of clock cycles spent on accessing the `%clock` register. From 1 warp to 15 warps it has had a constant latency of 16 [*clock cycles*]. At 16 warps the SM is saturated and the latency start to increase exponentially until ≈ 20 [*clock cycles*] for 32 warps.

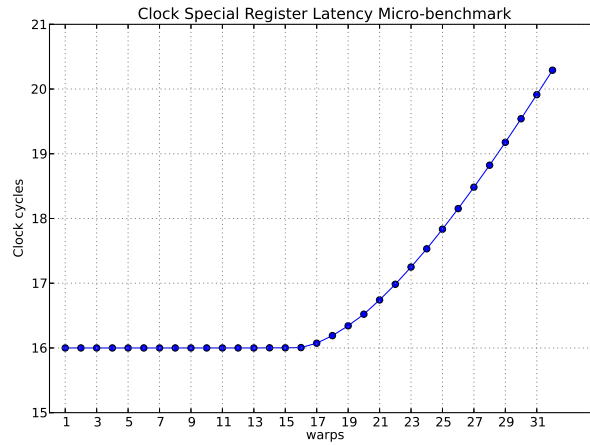


Figure 4.3: Clock special register latency micro-benchmark.

b. Function *clock()* micro-benchmark:

Similar to the assembly instruction to read the *%clock* register, the *clock()* function, provided by the CUDA API, can also be used to measure the number of clock cycles taken to execute an instruction. In Listing 4.2 the function *clock()* is used twice without any line between them to get the cost of applying that function to the kernel.

Listing 4.2: Function *clock()* micro-benchmark assembly.

```
1  /*Initializing variables*/
2  clock_t start;
3  clock_t stop;
4
5  /* Timing*/
6  start = clock();
7
8  stop  = clock();
9
10 /* Clock cycles difference */
11 d_start[idx] = start; d_stop[idx] = stop;
12 clk_cycles[idx] = (int)(stop-start);
```

Figure 4.4 shows the number of clock cycles employed in using the function *clock()*. As expected, the behaviour is similar to the one displayed figure 4.4 since both micro-benchmarks read the same special register.

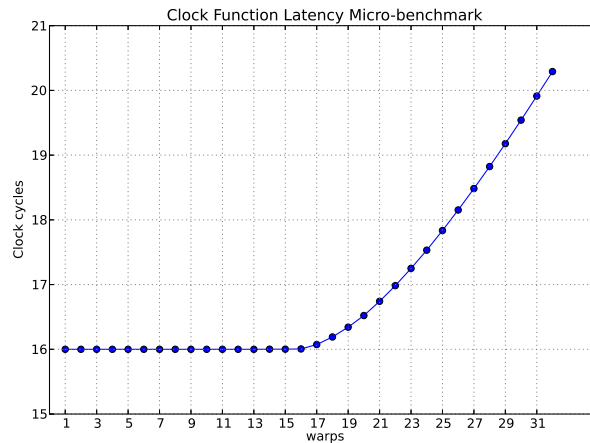


Figure 4.4: Clock function latency micro-benchmark.

c. Register latency micro-benchmark:

To obtain the number of clock cycles spent on using a register the micro-benchmark in Listing 4.3 was designed. First, two registers are declared. Then, the data is loaded from global memory and copied to one register. After that, the data is moved from one register to the other. At the end, the data is copied back to global memory.

Listing 4.3: Register latency micro-benchmark assembly.

```
1  /* Extended assembly inline: 1 thread, 1 data */
2  __asm__ (".reg      .f32  reg_c0, reg_c1; \n\t"// f32 registers
3          "ld.global.f32  reg_c0, [%03]; \n\t"// reg_c0 = d_a[0]
4          "mov.u32        %00, %%clock; \n\t"// Writing start cycle
5          "mov.f32        reg_c1, reg_c0; \n\t"// reg_c0=reg_c1
6          "mov.u32        %01, %%clock; \n\t"// Writing stop cycle
7          "mov.f32        %02,  reg_c1;      " // c0 = reg_c0
8          : "=r"(start), "=r"(stop), "=f"(c0)
9          : "l"(&d_a[0]));
10
11 /* Clock cycles difference */
12 d_start[idx] = start; d_stop[idx] = stop;
13 clk_cycles[idx] = (int)(stop-start);
```

Observation to the control flow graph generated for the register latency benchmark kernels (see Figure 4.5) showed that the use of the `%clock` register always creates an additional `mov` instruction.

```
MOV R9, R8;
S2R R8, SR_CLOCKLO;
MOV R8, R8;
MOV R12, R9;
S2R R9, SR_CLOCKLO;
MOV R9, R9;
```

Figure 4.5: Control flow graph detail of the register latency micro-benchmark kernel.

Figure 4.6 displays the clock cycles captured for the register latency micro-benchmark. 41 [*clock cycles*] for 1 warp and ≈ 46 [*clock cycles*] for 32 warps.

These results include the cost of using the function `clock()` or `%clock` register to measure the latency and the extra `mov` instruction created. For this reason, subtractions between the cycles show in Figure 4.6 and the cycles show in Figure 4.3 per each warp must be made to know the cost of two `mov` instructions. Then, this result can be divided by 2 to obtain the actual latency for one `mov` operation.

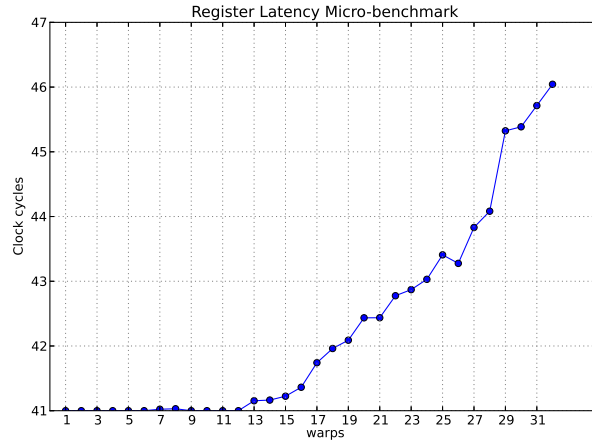


Figure 4.6: Register benchmark.

d. Float point operations latency micro-benchmark:

The float point (FP) operations latency depends on the operation being executed. The benchmarked operations were 3D stencil operations (e.g. addition, subtraction, multiplication, division). Then latency of the stencil operation with the larger value was chosen as the "*avg_inst_lat*" parameter.

d.1 Addition floating point operation latency micro-benchmark: In Listing 4.4 the code used in the addition floating point operation is shown. Previously, two elements have been loaded from global memory. The assembly addition instruction for float data of 32-bits is `add.f32`. This instruction adds the values of two registers and stores the result in a third register. The addition operation is put between two `%clock` registers instructions to capture the addition instruction latency.

Figure 4.7 displays the clock cycles captured for the addition operation latency micro-benchmark. 41 [*clock cycles*] for 1 warp and ≈ 43 [*clock cycles*] for 32 warps. As well as the register latency micro-benchmark, these results include the cost of using the function `clock()` or the `%clock` register to measure the latency and the extra `mov` instruction created.

Listing 4.4: FP addition operation latency micro-benchmark assembly.

```

1  /* Extended assembly inline for add operation*/
2  __asm__("mov.u32 %0, %%clock;\n\t" // Start cycle
3         "add.f32 %2, %3, %4; \n\t" // c = a + b
4         "mov.u32 %1, %%clock;"      // Stop cycle
5         : "=r" (start), "=r" (stop), "=f" (c0)
6         : "f" (a0), "f" (b0) )
7
8  /* Clock cycles difference */
9  d_start[idx] = start; d_stop[idx] = stop;
10 clk_cycles[idx] = (int) (stop-start); // Difference

```

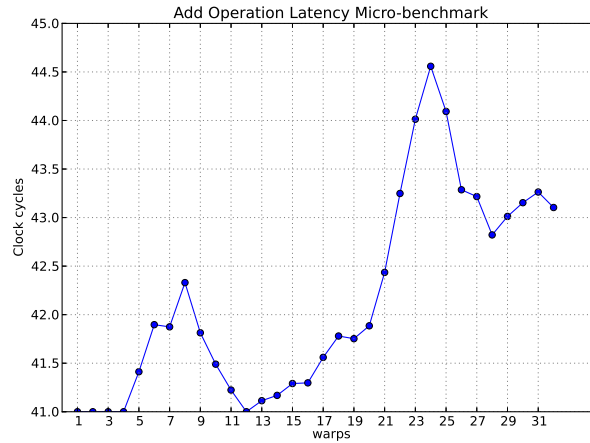


Figure 4.7: Addition floating point operation latency micro-benchmark.

d.2 Subtraction floating point operation latency micro-benchmark: Listing 4.5 presents the code used in the subtraction floating point operation. Previously, two elements have been loaded from global memory. The assembly subtraction instruction for floats of 32-bits is *sub.f32*. This instruction subtracts the values of two registers and stores the result in a third register. The subtraction operation is put between two *%clock* register instructions to capture the instruction latency.

Listing 4.5: FP subtraction latency operation micro-benchmark assembly.

```

1  /* Extended assembly inline for sub operation*/
2  __asm__("mov.u32 %0, %%clock;\n\t" // Start cycle
3         "sub.f32 %2, %3, %4; \n\t" // c = a - b
4         "mov.u32 %1, %%clock;"    // Stop cycle
5         : "=r" (start), "=r" (stop), "=f" (c0)
6         : "f" (a0), "f" (b0));
7
8  /* Clock cycles difference */
9  d_start[idx] = start; d_stop[idx] = stop;
10 clk_cycles[idx] = (int) (stop-start); // Difference

```

Figure 4.8 displays the clock cycles captured for the addition operation latency micro-benchmark. 41 [clock cycles] for 1 warp and ≈ 44 [clock cycles] for 32 warps. As well as the register latency micro-benchmark, these results include the cost of using the function *clock()* or the *%clock* register to measure the latency and the extra *mov* instruction created.

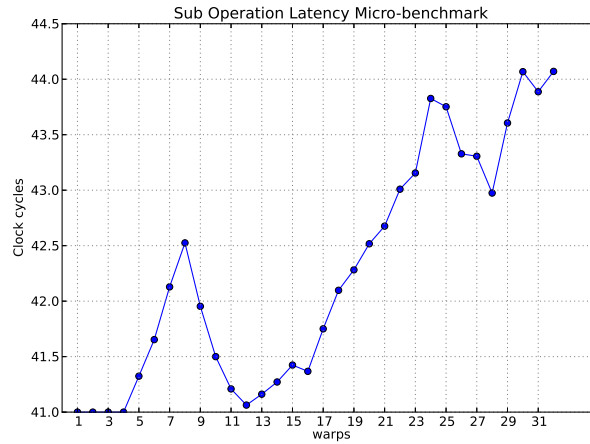


Figure 4.8: Subtraction floating point operation latency micro-benchmark.

d.3 Multiplication floating point operation latency micro-benchmark: Listing 4.6 presents the code used in the multiplication floating point operation. Previously, two elements have been loaded from global memory. The assembly multiplication instruction for floats of 32-bits is *mul.f32*.

This instruction multiplies the values of two registers and stores the result in a third register. The multiplication operation is put between two `%clock` registers to capture the instruction latency.

Listing 4.6: FP multiplication latency operation micro-benchmark assembly.

```

1  /* Extended assembly inline for mul operation */
2  __asm__("mov.u32 %0, %%clock;\n\t" // Start cycle
3         "mul.f32 %2, %3, %4; \n\t" // c = a * b
4         "mov.u32 %1, %%clock;"    // Stop cycle
5         : "=r" (start), "=r" (stop), "=f" (c0)
6         : "f" (a0), "f" (b0));
7
8  /* Clock cycles difference */
9  d_start[idx] = start; d_stop[idx] = stop;
10 clk_cycles[idx] = (int) (stop-start); // Difference

```

Figure 4.9 displays the clock cycles captured for the multiplication operation latency micro-benchmark. 41 [clock cycles] for 1 warp and ≈ 44 [clock cycles] for 32 warps. As well as the register latency micro-benchmark, these results include the cost of using the function `clock()` or `%clock` register to measure the latency and the extra `mov` instruction created.

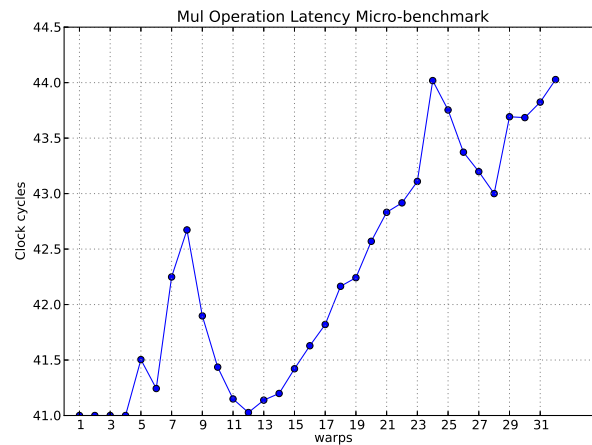


Figure 4.9: Multiplication floating point operation latency micro-benchmark.

d.4 Division floating point operation latency micro-benchmark: Listing 4.7 presents the code used in the division floating point operation. Previously, two elements have been loaded from global memory. The assembly division instruction for floats of 32-bits is *div.f32*. This instruction divides the values of two registers and stores the result in a third register. The division operation is put between two *%clock* register instructions to capture the instruction latency.

Listing 4.7: FP division latency operation micro-benchmark assembly.

```

1  /* Extended assembly inline for div operation*/
2  __asm__("mov.u32 %0, %%clock;  \n\t"// Start cycle
3         "div.full.f32 %2, %3, %4;\n\t"// c = a / b
4         "mov.u32 %1, %%clock;"      // Stop cycle
5         : "=r" (start), "=r" (stop), "=f" (c0)
6         : "f" (a0), "f" (b0));
7
8  /* Clock cycles difference */
9  d_start[idx] = start; d_stop[idx] = stop;
10 clk_cycles[idx] = (int)(stop-start); // Difference

```

Figure 4.10 displays the clock cycles captured for the division operation latency micro-benchmark. 235 [clock cycles] for 1 warp and ≈ 255 [clock cycles] for 32 warps. As well as the register latency micro-benchmark, these results include the cost of using the function *clock()* or *%clock* register to measure the latency and the extra *mov* instruction created.

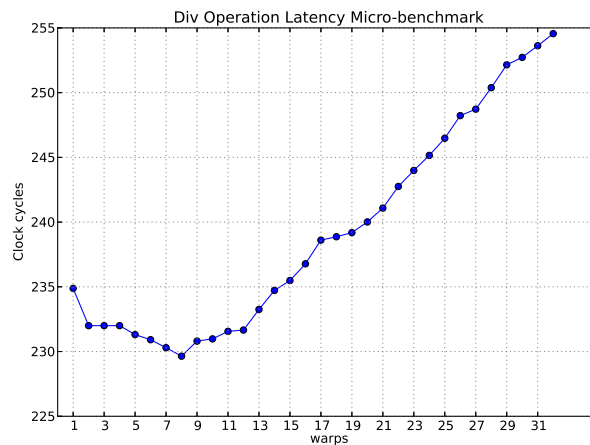


Figure 4.10: Division floating point operation latency micro-benchmark.

e. Global memory access latency micro-benchmark:

The *DRAM_{lat}* parameter is obtained via benchmarking the latency of accessing the global memory. It consist in several threads loading and storing data from/in that memory. In Listing 4.8 the micro-benchmark code for 1 input element is shown. First data is loaded from the *d_a* array and copied to the register *reg_c0*. The instruction to load from global memory 32-bit float point data is *ld.global.f32*. This instruction is put between two *%clock* register instruction to capture the instruction latency. Then, data in the *reg_c0* register is copied to the *c0* variable. Instruction *st.global.f32* can be used to store the floating point data in global memory, however when using this instruction the control flow graph show undesired instructions. For this reason, the storage instruction was implemented as *d_c[0] = c0* and the control flow graph was checked to verify that the desired *st.global.f32* instruction had been generated.

Listing 4.8: Global memory access latency micro-benchmark assembly

```
1  /* Extended assembly inline for load: 1 thread, 1 data*/
2  __asm__(".reg .f32 reg_c0;    \n\t" // f32 register
3         "mov.u32 %00, %%clock;\n\t" // Start reading
4         "ld.global.f32 reg_c0, [%03];\n\t" // reg_c0 = d_a[0]
5         "mov.u32 %01, %%clock;\n\t" // Stop reading
6         "mov.f32 %02,  reg_c0;    " // c0 = reg_c0
7         : "=r" (start_r), "=r" (stop_r), "=f" (c0)
8         : "l" (&d_a[0]));
9
10 /* Copying data from registers to global memory */
11 __asm__("mov.u32 %0, %%clock;":"=r" (start_w)); // Start writing
12         d_c[0] = c0;
13 __asm__("mov.u32 %0, %%clock;":"=r" (stop_w)); // Stop writing
14
15 /* Clock cycles difference */
16 d_start_r[idx] = start_r; d_stop_r[idx] = stop_r;
17 d_start_w[idx] = start_w; d_stop_w[idx] = stop_w;
18 clk_cycles_r[idx] = (int) (stop_r-start_r);
19 clk_cycles_w[idx] = (int) (stop_w-start_w);
```

Figure 4.11 displays the latency obtained through the micro-benchmark for reading and writing the global memory. Approximately 431 [clock cycles] with 1 warp, and \approx 428 [clock cycles] with 32

warps for the reading instruction. For the writing instruction; 50 [*clock cycles*] with 1 warp, and ≈ 53 [*clock cycles*] with 32 warps. It is important to observe that the storage cost is less than the load cost because of the searching that has to be made when loading an input data. As well as the register latency micro-benchmark, these results include the cost of using the function *clock()* or *%clock* register to measure the latency and the extra *mov* instruction created.

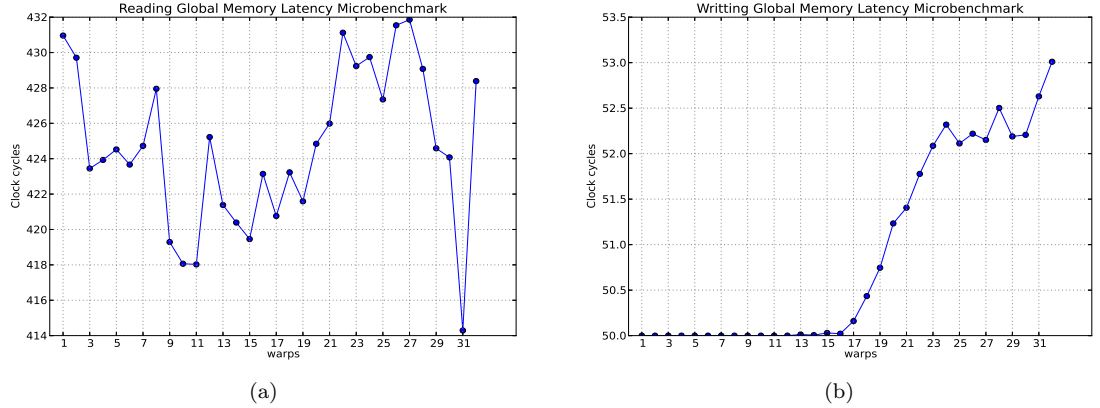


Figure 4.11: a) Reading global memory latency. b) Writing global memory latency. Global memory access latency micro-benchmark.

f. Shared memory latency micro-benchmark:

To benchmark the latency of accessing to shared memory several threads loading and storing data from/in that memory were executed. In Listing 4.9 the micro-benchmark code for 1 input element is shown. First data is loaded from the *d_a* array and copied to the register *reg_c0*. Then, data in the register is stored in shared memory. Instruction to store 32-bit floating point data in shared memory is *st.shared.f32*. The *lt.shared.f32* instruction is put between two *%clock* register instructions to capture the instruction latency. Then, data in the *reg_c0* register is copied to the *c0* variable. After storing data in shared memory, the load operation was made using the instruction *ld.shared.f32*. This instruction is also put between two *%clock* register instructions.

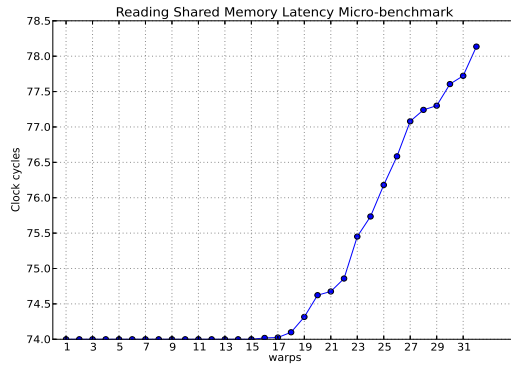
Figure 4.12 displays the latency obtained through the micro-benchmark for reading and writing the shared memory. Approximately 74 [*clock cycles*] with 1 warp, and ≈ 78 [*clock cycles*] with 32 warps for the reading instruction. For the writing instruction; 41 [*clock cycles*] with 1 warp, and ≈ 47 [*clock cycles*] with 32 warps. Note that similar to the global memory, the storage cost is less than the load cost because of the searching that has to be made when loading an input data. As well as the register latency micro-benchmark, these results include the cost of using the function *clock()* or the *%clock* register to measure the latency and the extra *mov* instruction created.

Listing 4.9: Shared memory latency micro-benchmark assembly.

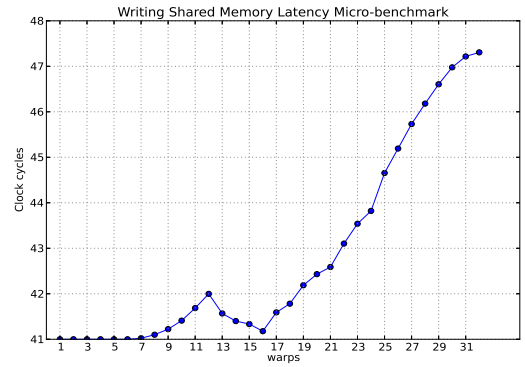
```

1  /* Extended assembly inline: 1 thread, 1 data*/
2  __asm__(".reg      .f32  reg_c0;          \n\t"// f32 registers
3         ".shared   .f32  shd_c0;          \n\t"// Shared memory variable
4         "ld.global.f32 reg_c0, [%05];    \n\t"// reg_c0 = d_a[0]
5         "mov.u32     %00,  %%clock;      \n\t"// Writing start cycle
6         "st.shared.f32 [shd_c0], reg_c0;\n\t"// sh_c0=reg_c0
7         "mov.u32     %01,  %%clock;      \n\t"// Writing stop cycle
8         "mov.u32     %02,  %%clock;      \n\t"// Reading start cycle
9         "ld.shared.f32 %04, [shd_c0];    \n\t"// c0=sh_c0
10        "mov.u32     %03, %%clock;        \n\t"// Reading stop cycle
11        : "=r" (start_w), "=r" (stop_w), "=r" (start_r), "=r" (stop_r), "=f" (c0)
12        : "l" (&d_a[0]));
13
14 /* Clock cycles difference */
15 d_start_r[idx] = start_r; d_stop_r[idx] = stop_r;
16 d_start_w[idx] = start_w; d_stop_w[idx] = stop_w;
17 clk_cycles_r[idx] = (int) (stop_r-start_r);
18 clk_cycles_w[idx] = (int) (stop_w-start_w);

```



(a)



(b)

Figure 4.12: a) Reading shared memory latency. b) Writing shared memory latency. Shared memory access latency micro-benchmark.

g. *hit_lat_l1*, *hit_lat_l2* parameters:

These parameters represent the latency of finding data in cache levels *L1* and *L2*. Due to the lack of information about the cache memory found in the documentation provided by the builder, the latency of the cache memory is computed base on the benchmarks presented above.

*g.1 L1 cache memory latency (*hit_lat_l1*):* *L1* cache memory is placed between the registers bank and the shared memory level in the GPU memory hierarchy. Furthermore, taking into account that each level in the memory hierarchy aims to reduce by half the latency of accessing to the next level, the latency of *L1* cache can be computed using equation 4.27. As can be seen, this equation involves the latencies of the GPU registers and the shared memory.

$$hit_lat_l1 = \frac{shared_memory_latency + register_latency}{2}. \quad (4.27)$$

Figure 4.13 shows the latencies obtained for *L1* cache memory by using equation 4.27. These results were calculated using the latencies for reading these memory units leading to ≈ 57 [clock cycles] with 1 warp, and ≈ 62 [clock cycles] with 32 warps for reading *L1* cache memory. As the input latencies for equation 4.27 include the cost of using the *%clock* register and the extra *mov* instruction, these results also include them.

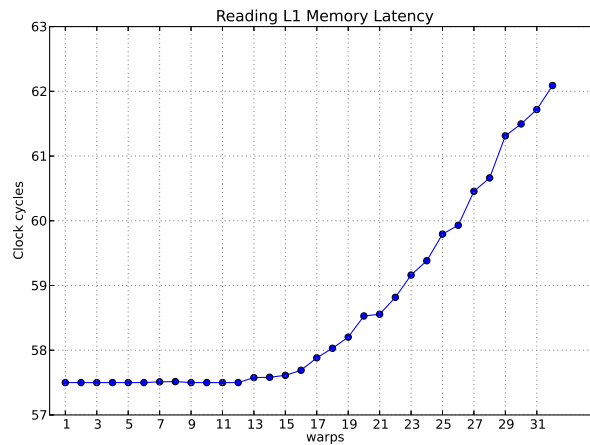


Figure 4.13: Reading L1 Memory Latency.

g.2 L2 cache memory latency (hitLatJ2): L2 cache memory is placed between the global memory and the L1/shared memory. Equation 4.28 relates the latencies of global memory and L1 cache memory to compute the latency of L2 cache memory.

$$hitLatJ2 = \frac{global_memory_latency + hitLatJ1}{2}. \quad (4.28)$$

In Figure 4.14 the latencies obtained for L2 cache memory are shown. These results were calculated using the latencies for reading these memory units leading to ≈ 244 [clock cycles] with 1 warp, and ≈ 245 [clock cycles] with 32 warps for reading L2 cache memory. As the input latencies for equation 4.28 include the cost of using the %clock register and the extra *mov* instruction, these results also include them.

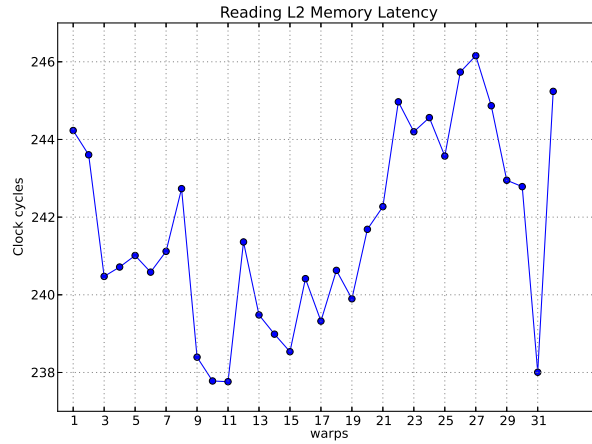


Figure 4.14: Reading L2 Memory Latency.

h. Departure delay (Δ) micro-benchmark:

To obtain this parameter, equation 4.25 was evaluated using data from the global memory micro-benchmark. First the difference between the clock cycles spent by 2 warps and 1 warp was calculated, then between 3 and 2 warps, and so on for 32 warps ($\#warps$). At last, results for each difference were summed and divided by the number of operations made ($\#warps/2$). The value for this parameter was $\Delta = 3$ [clock cycles].

In Table 4.3 the results of the micro-benchmarks implemented in a NVIDIA K40 Kepler GPU are shown. To these results the cost of using the clock register and the extra *mov* instruction has been subtracted.

Table 4.3: Micro-benchmarks results.

<i>Micro-benchmark</i>	1 warp [clock cycles]	32 warps [clock cycles]
Special register <i>%clock</i>	16	20
Function <i>clock()</i>	16	20
Register	13	13
Addition FP operation	12	10
Subtraction FP operation	12	11
Multiplication FP operation	12	11
Division FP operation	206	222
Global memory Read	402	395
Global memory Write	21	20
Shared memory Read	45	45
Shared memory Write	12	14
L1 cache memory	29	29
L2 cache memory	215.5	212
Departure delay Δ	3	3

To calculate the micro-benchmarks uncertainty guidelines in [60] were employed. Some of these results are shown in Table 4.4. Hence, it is important to note that the uncertainty depends on the scale and dispersion of the measured data.

Table 4.4: Micro-benchmarks uncertainty.

<i>Micro-benchmark</i>	1 warp [clock cycles]	32 warps [clock cycles]
Special register <i>%clock</i>	16 ± 0.57	20 ± 0.57
Function <i>clock()</i>	16 ± 0.57	20 ± 0.57
Register	13 ± 0.57	13 ± 0.63
FP operation	12 ± 0.57	10 ± 0.61
Global memory Read	402 ± 5.88	395 ± 6.17
Shared memory Read	45 ± 0.57	45 ± 0.60

4.5.4 Parameters from the source code

The user has the task of selecting the number of threads and blocks to be launched in a GPU program. This selection depends on the input data, the strategy for reading the data, and the hardware resources to be used. The parameters of the model that are computed via static analysis of the code are: the total number of warps (*#total_warps*), the number of active SMs (*#active_SMs*) and active warps on one SM (*N*).

a. Total number of warps (*#total_warps*):

A warp is a group of 32 threads, this parameter is calculated dividing the total number of launched threads over 32.

$$\#total_warps = \frac{\#total_thread_launched}{32} \quad (4.29)$$

b. Number of active SMs (*#active_SMs*):

The threads blocks are uniformly distributed between all the available SMs in the GPU. According to [57] each SM can run as many as eight blocks. For a GPU with 5 SMs and with 10 blocks being launched there would be 2 blocks for each SM.

c. Active warps per SM (*N*):

The number of warps per SM is calculated as follows

$$N = \frac{\#blocks_per_SM * \#threads_per_block}{32} \quad (4.30)$$

d. Transactions size (*transaction_size*):

Is the number of bytes for each warp. The transaction size can be calculated as:

$$transaction_size = Data\ size * \#threads_per_warp, \quad (4.31)$$

where the data size is 4 [B] per float data and the number of threads per warp is 32.

4.6 Validating the Adapted Model

In this section the adapted model is used to estimate the execution time of the implementation presented in chapter 3. The validation process uses parameters extracted in 4.5 to compute the model equations described in 4.4. Recall that the implementation was made for the propagation of the acoustic wave equation over a 3D synthetic velocity model of two layers, with size of $Nx = 50$ [points], $Ny = 50$ [points], $Nz = 50$ [points]. In the following, the implementation is modeled for a 2nd order stencil, and launching $(8 \times 8 \times 4)$ thread blocks in a $(8 \times 8 \times 8)$ blocks grid.

To calculate T_{comp} , first $ITLP_{max}$ is computed as it is shown in equation 4.32. The *avg_inst_lat* parameter represents the average instruction latency and its value is taken from the cost of the floating operation (i.e. division). The warp size is 32 threads and the *SIMD_width* is the number of cores per SM.

$$ITLP_{max} = \frac{avg_inst_lat}{warp_size/SIMD_width} = \frac{201}{32/192} = 1206 \text{ [clock cycles]}. \quad (4.32)$$

Then, to obtain the instruction level parallelism *ILP* the control flow graph of the kernel is analyzed. The compute instructions and the memory instructions are counted and then grouped into basic execution blocks. Hence, the total number of instruction is divided by the number of basic blocks as shown in equation 4.33.

$$ILP = \frac{\#total_instructions}{\#basic_blocks} = \frac{86}{18} \approx 5. \quad (4.33)$$

So, *ITILP* can be calculated with equation 4.34, where N represents the number of active warps per SM.

$$ITILP = \min(ILP \times N, ITLP_{max}) = \min(5 \times 8, 192) = 40. \quad (4.34)$$

Next, T_{comp} (see equation 4.23) is calculated as follows:

$$T_{comp} = W_{parallel} = \frac{75 \times 4096}{15} \times \frac{201}{40} = 102912 \text{ [clock cycles]}. \quad (4.35)$$

To calculate T_{mem} several terms must be computed before. First, the avg_DRAM_lat (see equation 4.10) is calculated using equation 4.36. The avg_trans_warp can be obtained from the metric $gld_transactions_per_request = 21$.

$$avg_DRAM_lat = 399 + (21 - 1) \times 3 = 459 [clock\ cycles]. \quad (4.36)$$

Then, CUDA metrics $l1_cache_global_miss_rate$ and $l2l1_read_hit_rate$ can be used to calculate the $miss_ratio_l2$ and $miss_ratio_l1$. Hence, the $AMAT$ term is now given by equation 4.24

$$AMAT = 459 \times 100\% + 214 \times 5\% + 29 = 498.7 [clock\ cycles]. \quad (4.37)$$

With the clock frequency from hardware specifications $freq = 1,1 [GHz]$ and the transactions size from equation 4.38

$$transaction_size = Data_size \times \#threads_per_warp = 4 [B] \times 32 = 128 [B] \quad (4.38)$$

the BW_per_warp is

$$BW_per_warp = \frac{1,11 [GHz] \times 128 [B]}{459 [clock\ cycles]} = 0,207 [GB/s]. \quad (4.39)$$

Having the $mem_peak_bandwidth$ from the CUDA properties $memoryClockRate$ and $memoryBusWidth$, the MWP_{peak_bw} computed is

$$MWP_{peak_bw} = \frac{mem_peak_bandwidth}{BW_per_warp \times \#active_SMs} = \frac{288.384 [GB/s]}{0.207 \times 15} = 94.243 [GB/s]. \quad (4.40)$$

from here the MWP can be calculated as

$$MWP = \min \left(\frac{avg_DRAM_lat}{\Delta}, MWP_{peak_bw}, N \right) = \min \left(\frac{459}{3}, 94.243, 8 \right) = 8. \quad (4.41)$$

The number of compute clock cycles $comp_cycles$ depends on the number of instructions $\#inst$, the instructions average latency and the $ITILP$ as shown in equation 4.42

$$comp_cycles = \frac{\#insts \times avg_inst_lat}{ITILP} = \frac{75 \times 201}{40} = 376.875 [clock\ cycles]. \quad (4.42)$$

In a similar fashion mem_cycles is calculated from equation 4.43, where according to the control flow the $MLP = 17$.

$$mem_cycles = \frac{\#mem_inst \times AMAT}{MLP} = \frac{11 \times 498.7}{17} = 322.68 [clock\ cycles]. \quad (4.43)$$

With the number of $comp_cycles$ and mem_cycles the CWP_full is

$$CWP_full = \frac{mem_cycles + comp_cycles}{comp_cycles} = \frac{376.875 + 414.375}{414.375} = 1.909 [clock\ cycles]. \quad (4.44)$$

Following, the CWP is

$$CWP = \min(CWP_full, N) = \min(1.909, 8) = 1.909 \quad (4.45)$$

Thereby, the MWP_{cp} is

$$MWP_{cp} = \min(\max(1, CWP - 1), MWP) = \min(\max(1, 1.909 - 1), 8) = 1. \quad (4.46)$$

Then, the $ITMLP$

$$ITMLP = \min(MLP \times MWP_{cp}, MWP_peak_bw) = \min(17 \times 1, 94.243) = 17. \quad (4.47)$$

and T_{mem} is computed using the terms obtained above as

$$T_{mem} = \frac{\#mem_insts \times \#total_warps}{\#active_SMs \times ITMLP} \times AMAT = \frac{11 \times 4096}{15 \times 17} \times 498.7 = 88115.40 [clock\ cycles]. \quad (4.48)$$

The last term of equation 4.1 corresponds to the overlap time between compute and memory operations, as mentioned above. Overlap function must be first calculated as shown in 4.22. As $CWP \leq MWP$ then $\zeta = 1$ and $F_{overlap}$ is

$$F_{overlap} = \frac{N - \zeta}{N} = \frac{8 - 1}{8} = 0.875 \quad (4.49)$$

Using $F_{overlap}$ $T_{overlap}$ is computed as

$$T_{overlap} = \min(T_{comp} \times F_{overlap}, T_{mem}) = (102912 \times 0.875, 88115.4) = 88115.40 [clock\ cycles] \quad (4.50)$$

Finally, the execution time T_{exec} is

$$T_{exec} = T_{comp} + T_{mem} - T_{overlap} = 102912 + 88115.40 - 88115.40 = 102912 [clock\ cycles] \quad (4.51)$$

The validation process showed above was repeated for different stencil orders (i.e. 2nd, 4th and 8th) using 8, 16, 24 and 32 warps. These estimations are summarized in the Table 4.5. Also, experimental latencies were obtained using a NVIDIA K40 Kepler GPU. These result are shown in Table 4.6.

Stencil order	8 warps [clock cycles]	16 warps [clock cycles]	24 warps [clock cycles]	32 warps [clock cycles]
2	102912	75877.38	113706.33	29331.58
4	119280.13	97053.7	61939.463	46942.44
8	165837.10	108656.87	69355.692	43488.105

Table 4.5: Theoretical latencies of the implementation of the wave equation propagation for different stencil orders and different number of warps per SM.

<i>Stencil order</i>	8 warps [clock cycles]	16 warps [clock cycles]	24 warps [clock cycles]	32 warps [clock cycles]
2	34520.3	35065.2	26689.2	44319.4
4	61090.7	61002.9	52243.3	65586
8	101598	95727.9	71233	104026

Table 4.6: Experimental latencies of the implementation of the wave equation propagation for different stencil orders and different number of warps per SM.

The experimental results from Table 4.6 and the theoretical results of Table 4.5 are shown in Figure 4.15. According to the figure the adapted model doesn't follow the behaviour of the GPU, one of the reasons is that the MWP-CWP model was designed for a previous GPU architecture known as Fermi. Hence, is important to remind that the aim of the adaptation was to adapt the general model to a specific application, not to a GPU architecture. Also, the `#total_warps` parameter considers the total number of warps launched. However, the number of warps launched is different from the number of warps actually working in the SM.

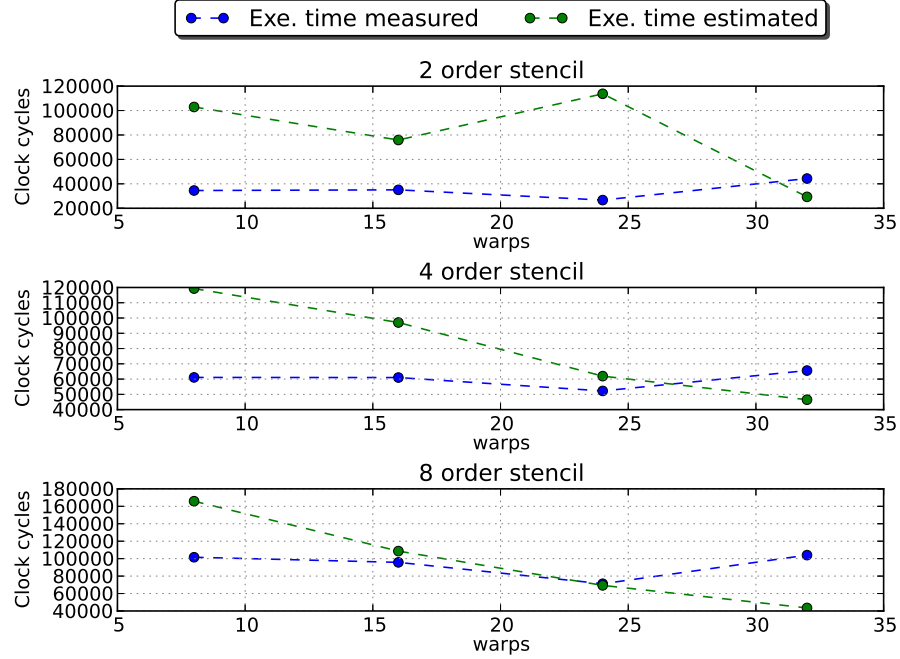


Figure 4.15: Theoretical latencies versus experimental latencies.

After adjusting the adjusting the `#total_warps` parameter the results shown in Table 4.7 were obtained. Figure 4.16 presents the experimental results from 4.6 and the theoretical results of Table 4.7. Observe that despite adjusting the `#total_warps` the model maintains its behaviour.

<i>Stencil order</i>	8 warps [clock cycles]	16 warps [clock cycles]	24 warps [clock cycles]	32 warps [clock cycles]
2	98138.25	54004.57	85674.57	28642.29
4	129447.0	69120.94	47041.26	38853.37
8	158144.46	77880.28	52658.0	42804.16

Table 4.7: Theoretical latencies for different stencil orders and different number of warps per SM considering the *#total_warps*.

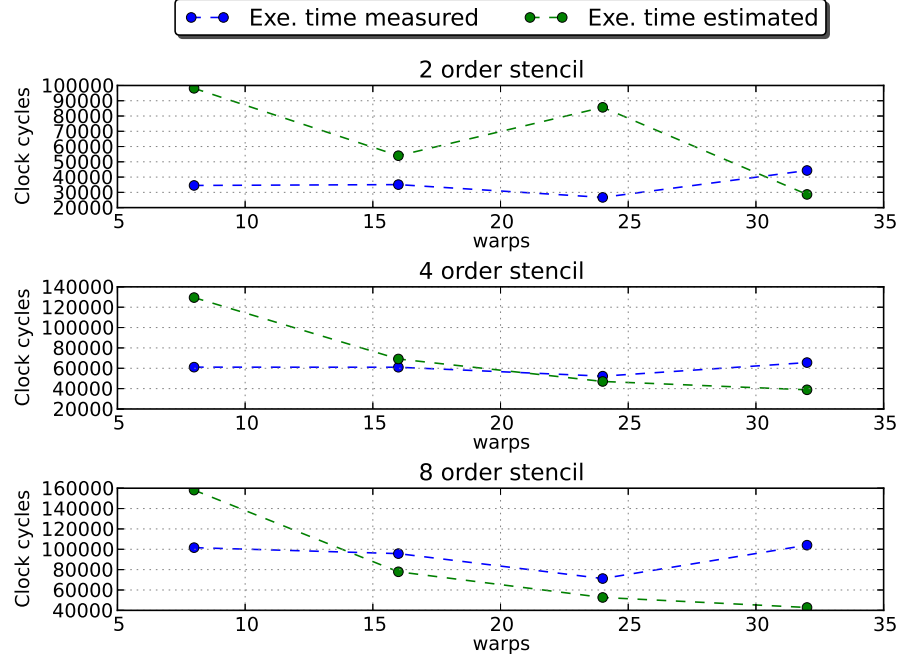


Figure 4.16: Theoretical latencies versus experimental latencies.

4.7 Discussion

One of the work drawbacks, is that the model adaptation was only aim to stencil-based kernels. However, it doesn't mean that it can't be used for other applications.

Figures 4.15 and 4.16 show the implementation execution time versus the execution time estimated by the adapted model. When the number of GPU resources used (i.e. number of warps being executed) is increased, the number of clock cycles spent is also increased. Note that there is some drop in the execution time for 24 warps, this drop can be a consequence of task scheduling. However, the experimental behaviour is not followed for the analytical model spite of the adaptations proposed to represent the architecture. Hence, any minor changes to the GPU architecture will affect the model.

For that reason, a deep understanding of the GPU is needed to fit the model. In addition, the parameters extraction employs tools provided by the GPU builder, but those tools lack of detailed descriptions of the metrics, and how they are measured. As future work, other parameters should be studied to guarantee that the analytical model follows the behaviour of the implementation.

4.8 Conclusions

The conclusions obtained from the development of this work are presented below:

Implementations of the acoustic wave equation solution in both CPU and GPU were made, allowing to analyze the instructions created for a stencil-based kernel. Stencil-based kernels are the basis for complex algorithms, such as the Reverse-Time Migration (RTM) and the Full-Wave Inversion (FWI) among others. For this reason, knowing the execution time of an stencil is fundamental to estimate the execution time of those complex algorithms.

GPU analytical models can be used to estimate the execution time of GPU implementations. In the literature, several models have been proposed, nevertheless, there is not an standard approach to that problem. Furthermore, as the model represents the behaviour of the GPU, any minor change in the GPU architecture can lead to misrepresentations. Those matters could be sort out in the best-case scenario, by fitting the model to the architecture changes.

The MWP-CWP model calculates the execution time taking into account the compute operations cost, the memory operations cost and the overlap between compute and memory operations. An adaptation of the MWP-CWP model directed towards stencil-based implementations is presented. This adaptation includes changes in the model equations to consider cache levels, compute the *ILP* and to neglect effects of bank conflicts, branch divergences, use of special function units among others.

Parameters needed by the model were identified, these parameters could be extracted from hardware specifications, source code, CUDA binaries tools and the use of micro-benchmarks. For this work several micro-benchmarks were designed to obtain the latency of using resources, such as global memory, floating point operations, etc. In addition, the uncertainty associated to the micro-benchmarks was calculated. Unfortunately, the uncertainty of the hardware specifications and the parameters obtained via CUDA binary tools is unknown, and it can not be computed using the information provided by the builder.

The adapted model has been validated comparing the execution time of the GPU implementation with the estimated execution time given by the adapted model. As can be seen for the results, the theoretical time is greater than the experimental time, which might be a consequence of using a model that was originally designed for a former GPU architecture. On the other hand, the `#total_warps` parameter was adjusted to consider the real number of warps being used for the GPU architecture,

and although the difference between the theoretical and experimental results was decreased, deeper model adjustments must to be made to consider the newer GPU architectures.

Finally, this document had presented detailed descriptions of all the stages involved in the development process (e.g. background information, study of the model, adaptation process, parameters extraction, model validation, etc.) of this work. This work can be used as a tool for anyone interested in the GPU architecture modelling field.

References

- [1] Lecture 3, How do rocks deform ? [online]. Available:
http://geoscience.wisc.edu/~chuck/Classes/Mtn_and_Plates/rock_deformation.html.
[Visited: dec-2015].
- [2] Introduction to Structural Geology [online]. Available:
http://www.geosci.usyd.edu.au/users/prey/Patrice_Intro_to_SG.pdf.
[Visited: dec-2015].
- [3] Earth Structure An Introduction to Structural Geology and Tectonics [online]. Available:
<http://www.globalchange.umich.edu/ben/ES/>. [Visited: dec-2015].
- [4] Crustal Faults [online]. Available:
<http://www.sfu.ca/~acalvert/WebSite/Research/CascadiaShallow/CascadiaShallow.htm>.
[Visited: dec-2015].
- [5] EnergyindustryPhotos.com [online]. Available:
http://www.energyindustryphotos.com/photos_of_seismic_surveying_equi.htm.
[Visited: dec-2015].
- [6] Queensland Government [online]. Available:
<http://www.qld.gov.au/>. [Visited: dec-2015].
- [7] Geoscience Overview, Passion for Geoscience [online]. Available:
<http://www.cgg.com/en/Who-We-Are/Geoscience-Overview>. [Visited: dec-2015].
- [8] Air gun / seismic survey [online]. Available:
http://www.nauticexpo.com/prod/sercel/product-40158-430203.html#product-item_326392.
[Visited: dec-2015].

- [9] Industrial Vehicules International, HEMI 60 A 61,000 Pound Vibrator [online]. Available: <http://www.indvehicles.com/index.cfm?id=3&catID=2&prodID=2>.
[Visited: dec-2015].
- [10] Ricker wavelet [online]. Available: http://wiki.seg.org/wiki/Dictionary:Ricker_wavelet. [Visited: dec-2015].
- [11] File:MexicanHatMathematica.svg [online]. Available: <https://en.wikipedia.org/wiki/File:MexicanHatMathematica.svg>. [Visited: dec-2015].
- [12] USGS Seismometer, Seismic instruments at the Rolnick Observatory [online]. Available: <https://www.was-ct.org/resources/the-rolnick-seismometer/>. [Visited: dec-2015].
- [13] Martin, G. S., Wiley, R., & Marfurt, K. J. (2006). Marmousi2: An elastic upgrade for Marmousi. The Leading Edge, 25(2), 156.
- [14] Marmousi Model, Trevor Irons [online]. Available: <http://www.reproducibility.org/RSF/book/data/marmousi/paper.pdf>.
Available: Online, Visited 2015.
- [15] VTK, Welcome VTK [online]. Available: <http://www.vtk.org/>. [Visited: dec-2015].
- [16] Paraview [online]. Available: <http://www.paraview.org/>. [Visited: dec-2015].
- [17] Graphics Processing Unit [online]. Available: https://en.wikipedia.org/wiki/Graphics_processing_unit.
[Visited: dec-2015].
- [18] Video Display Controller [online]. Available: https://en.wikipedia.org/wiki/Video_display_controller.
[Visited: dec-2015].
- [19] The History of the Modern Graphics Processor [online]. Available: <http://www.techspot.com/article/650-history-of-the-gpu>.
[Visited: dec-2015].
- [20] From Voodoo to GeForce: The Awesome History of 3D Graphics [online]. Available: <http://www.maximumpc.com/from-voodoo-to-geforce-the-awesome-history-of-3d-graphics>.
[Visited: dec-2015].
- [21] Gun Fight [online]. Available: <http://www.giantbomb.com/gun-fight/3030-32574>.
[Visited: dec-2015].
-

- [22] ATI Wonder Series [online]. Available:
https://en.wikipedia.org/wiki/ATI_Wonder_series.
 [Visited: dec-2015].

- [23] Advanced Micro Devices [online]. Available:
<http://www.amd.com>.
 [Visited: dec-2015].

- [24] NVIDIA [online]. Available:
<http://www.nvidia.com>.
 [Visited: dec-2015].

- [25] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110" [online]. Available:
<http://www.nvidia.com>.
 [Visited: dec-2015].

- [26] NVIDIA, "CUDA C Programming GUIDE v5.5" [online]. Available:
<http://www.nvidia.com>.
 [Visited: dec-2015].

- [27] NVIDIA, "CUDA C Best Practices GUIDE v5.5" [online]. Available:
<http://www.nvidia.com>.
 [Visited: dec-2015].

- [28] Yee, K. (1966). Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions on*, 14(3), 302-307.

- [29] Taflov, A. (1980). Application of the Finite-Difference Time-Domain Method to Sinusoidal Steady-State Electromagnetic-Penetration Problems. *IEEE Transactions on Electromagnetic Compatibility, EMC-22(3)*, 191-202. IEEE.

- [30] Hrennikoff, A. (1941). Solution of Problems of Elasticity by the Frame-Work Method. *Applied Scientific Research*, A8, 169-175.

- [31] Understanding the Finite-Difference Time-Domain Method 2010, John B. Schneider [online]. Available:
<http://www.eecs.wsu.edu/~schneidj/ufdttd>.
 [Visited: dec-2015].

- [32] Lines, L., Slawinski, R., and Bording, R. (1999). Short Note: A recipe for stability of finite-difference wave-equation computations. *Geophysics*, vol. 64, 967-969. SEG.
- [33] Numerical stability [online]. Available:
https://en.wikipedia.org/wiki/Numerical_stability.
 [Visited: dec-2015].
- [34] Von Neumann stability analysis [online]. Available:
https://en.wikipedia.org/wiki/Von_Neumann_stability_analysis.
 [Visited: dec-2015].
- [35] Numerical dispersion [online]. Available:
<http://www.ness-music.eu/target-systems/virtual-room-acoustics/numerical-dispersion>
 [Visited: dec-2015].
- [36] Analytical Models,
<http://serc.carleton.edu/introgeo/mathstatmodels/Analytical.html>
 Available: Online, Visited 2015.
- [37] Random-Access machine,
https://en.wikipedia.org/wiki/Random-access_machine
 Available: Online, Visited 2015.
- [38] Counter machine,
https://en.wikipedia.org/wiki/Counter_machine Available: Online, Visited 2015.
- [39] Kothapalli, K., Mukherjee, R., Rehman, M. S., Patidar, S., Narayanan, P. J., & Srinathan, K. (2009).
 A performance prediction model for the CUDA GPGPU platform.
 2009 International Conference on High Performance Computing (HiPC) (pp. 463-472). IEEE.
- [40] Bagsorkhi, S., Delahaye, M., Patel, S., Gropp, W., Hwu, W.-mei: An adaptive performance modeling tool for GPU architectures. In: *Principles and Practice of Parallel Programming*, Bangalore (2010)
- [41] Choi, J. W., Singh, A., & Vuduc, R. W. (2010).
 Model-driven autotuning of sparse matrix-vector multiply on GPUs.
ACM SIGPLAN Notices, 45(5), 115.

- [42] Zhang, Y., & Owens, J. D. (2011).
A quantitative performance analysis model for GPU architectures.
Proceedings - International Symposium on High-Performance Computer Architecture, 382-393.
- [43] Kim, Y., & Shrivastava, A. (2011).
CuMAPz: A tool to analyze memory access patterns in CUDA.
2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 128-133. IEEE.
- [44] Hong, S. and Kim, H.: An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In: International Chinese Statistical Association, San Francisco (2009)
- [45] Sim, J., Dasgupta, A., Kim, H., Vuduc, R.: A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In: Symposium on Principles and Practice of Parallel Programming , San Francisco (2012)
- [46] Tang, W., Tan, W., Krishnamoorthy, R., Wong, Y., Kuo, S., Goh, R., Turner, S., et al.: Optimizing and Auto-Tuning Iterative Stencil Loops for GPUs with the In-Plane Method. In: International Symposium on Parallel & Distributed Processing, Boston (2013)
- [47] Lopez-Novoa, U., Mendiburu, A., & Miguel-Alonso, J.: A survey of performance modeling and simulation techniques for accelerator-based computing. In: IEEE Transactions on Parallel and Distributed Systems, 26(1), 272-281. IEEE Computer Society (2015).
- [48] CUDA Binary Utilities. Nvidia (2015)
- [49] CUDA C Best Practices Guides. Nvidia (2013)
- [50] CUDA Compiler Driver NVCC. Nvidia (2015)
- [51] CUDA C Programming Guide. Nvidia (2013)
- [52] CUDA Memcheck. Nvidia (2015)
- [53] CUDA Profiler User's Guide. Nvidia (2015)
- [54] CUDA Samples. Nvidia (2015)
- [55] cuobjdump. Nvidia (2012)
- [56] Parallel Thread Execution ISA. Nvidia (2015)

- [57] Using Inline PTX Assembly in CUDA. Nvidia (2011)
- [58] Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA (2009)
- [59] Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. NVIDIA (2013)
- [60] Centro Español de Metrología. Evaluación de datos de medición: guía para la expresión de la incertidumbre de medida. España: El instituto, 2008. 142 p.

Bibliography

Adavanced Micro Devices [online]. Available:

<http://www.amd.com>.

[Visited: dec-2015].

Air gun / seismic survey [online]. Available:

http://www.nauticexpo.com/prod/sercel/product-40158-430203.html#product-item_326392.

[Visited: dec-2015].

Analytical Models,

<http://serc.carleton.edu/introgeo/mathstatmodels/Analytical.html>

Available: Online, Visited 2015.

ATI Wonder Series [online]. Available:

https://en.wikipedia.org/wiki/ATI_Wonder_series.

[Visited: dec-2015].

Baghsorkhi, S., Delahaye, M., Patel, S., Gropp, W., Hwu, W.-mei: An adaptive performance modeling tool for GPU architectures. In: Principles and Practice of Parallel Programming, Bangalore (2010).

Centro Español de Metrología. Evaluación de datos de medición: guía para la expresión de la incertidumbre de medida. España: El instituto, 2008. 142 p.

Choi, J. W., Singh, A., & Vuduc, R. W. (2010).

Model-driven autotuning of sparse matrix-vector multiply on GPUs.

ACM SIGPLAN Notices, 45(5), 115.

Counter machine,
https://en.wikipedia.org/wiki/Counter_machine Available: Online, Visited 2015.

Crustal Faults [online]. Available:
<http://www.sfu.ca/~acalvert/WebSite/Research/CascadiaShallow/CascadiaShallow.htm>.
[Visited: dec-2015].

CUDA Binary Utilities. Nvidia (2015).

CUDA Compiler Driver NVCC. Nvidia (2015).

CUDA Memcheck. Nvidia (2015).

CUDA Profiler User's Guide. Nvidia (2015).

CUDA Samples. Nvidia (2015).

cuobjdump. Nvidia (2012).

Earth Structure An Introduction to Structural Geology and Tectonics [online]. Available:
<http://www.globalchange.umich.edu/ben/ES/>. [Visited: dec-2015].

EnergyindustryPhotos.com [online]. Available:
http://www.energyindustryphotos.com/photos_of_seismic_surveying_equi.htm.
[Visited: dec-2015].

File:MexicanHatMathematica.svg [online]. Available:
<https://en.wikipedia.org/wiki/File:MexicanHatMathematica.svg>. [Visited: dec-2015].

From Voodoo to GeForce: The Awesome History of 3D Graphics [online]. Available:
<http://www.maximumpc.com/from-voodoo-to-geforce-the-awesome-history-of-3d-graphics>.
[Visited: dec-2015].

Geoscience Overview, Passion for Geoscience [online]. Available:
<http://www.cgg.com/en/Who-We-Are/Geoscience-Overview>. [Visited: dec-2015].

Graphics Processing Unit [online]. Available:
https://en.wikipedia.org/wiki/Graphics_processing_unit.
[Visited: dec-2015].

Gun Fight [online]. Available:
<http://www.giantbomb.com/gun-fight/3030-32574>.
[Visited: dec-2015].

Hong, S. and Kim, H.: An Analytical Model for a GPU Architeture with Memory-level and Thread-level Parellelism Awareness. In: International Chinese Statistical Association, San Francisco (2009).

Hrennikoff, A. (1941). Solution of Problems of Elasticity by the Frame-Work Method. Applied Scientific Research, A8, 169-175.

Industrial Vehicules International, HEMI 60 A 61,000 Pound Vibrator [online]. Available:
<http://www.indvehicles.com/index.cfm?id=3&catID=2&prodID=2>.
[Visited: dec-2015].

Introduction to Structural Geology [online]. Available:
http://www.geosci.usyd.edu.au/users/prey/Patrice_Intro_to_SG.pdf.
[Visited: dec-2015].

Kim, Y., & Shrivastava, A. (2011).
CuMAPz: A tool to analyze memory access patterns in CUDA.
2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 128-133. IEEE.

Kothapalli, K., Mukherjee, R., Rehman, M. S., Patidar, S., Narayanan, P. J., & Srinathan, K. (2009).
A performance prediction model for the CUDA GPGPU platform.
2009 International Conference on High Performance Computing (HiPC) (pp. 463-472). IEEE.

Lecture 3, How do rocks deform ? [online]. Available:
http://geoscience.wisc.edu/~chuck/Classes/Mtn_and_Plates/rock_deformation.html.
[Visited: dec-2015].

Lines, L., Slawinski, R., and Bording, R. (1999). Short Note: A recipe for stability of finite-difference wave-equation computations. *Geophysics*, vol. 64, 967-969. SEG.

Lopez-Novoa, U., Mendiburu, A., & Miguel-Alonso, J.: A survey of performance modeling and simulation techniques for accelerator-based computing. In: *IEEE Transactions on Parallel and Distributed Systems*, 26(1), 272-281. IEEE Computer Society (2015).

Martin, G. S., Wiley, R., & Marfurt, K. J. (2006). Marmousi2: An elastic upgrade for Marmousi. *The Leading Edge*, 25(2), 156.

Marmousi Model, Trevor Irons [online]. Available:
<http://www.reproducibility.org/RSF/book/data/marmousi/paper.pdf>.
Available: Online, Visited 2015.

Numerical dispersion [online]. Available:
<http://www.ness-music.eu/target-systems/virtual-room-acoustics/numerical-dispersion>
[Visited: dec-2015].

Numerical stability [online]. Available:
https://en.wikipedia.org/wiki/Numerical_stability.
[Visited: dec-2015].

NVIDIA [online]. Available:
<http://www.nvidia.com>.
[Visited: dec-2015].

NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110” [online]. Available: <http://www.nvidia.com>.
[Visited: dec-2015].

NVIDIA, “CUDA C Programming GUIDE v5.5” [online]. Available:
<http://www.nvidia.com>.
[Visited: dec-2015].

NVIDIA, “CUDA C Best Practices GUIDE v5.5” [online]. Available:
<http://www.nvidia.com>.
[Visited: dec-2015].

Parallel Thread Execution ISA. Nvidia (2015).

Paraview [online]. Available: <http://www.paraview.org/>. [Visited: dec-2015].

Queensland Government [online]. Available:
<http://www.qld.gov.au/>. [Visited: dec-2015].

Random-Access machine,
https://en.wikipedia.org/wiki/Random-access_machine
Available: Online, Visited 2015.

Ricker wavelet [online]. Available:
http://wiki.seg.org/wiki/Dictionary:Ricker_wavelet. [Visited: dec-2015].

Sim, J., Dasgupta, A., Kim, H., Vuduc, R.: A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In: Symposium on Principles and Practice of Parallel Programming, San Francisco (2012).

Taflove, A. (1980). Application of the Finite-Difference Time-Domain Method to Sinusoidal Steady-State Electromagnetic-Penetration Problems. IEEE Transactions on Electromagnetic Compatibility, EMC-22(3), 191-202. IEEE.

Tang, W., Tan, W., Krishnamoorthy, R., Wong, Y., Kuo, S., Goh, R., Turner, S., et al.: Optimizing and Auto-Tuning Iterative Stencil Loops for GPUs with the In-Plane Method. In: International Symposium on Parallel & Distributed Processing, Boston (2013).

The History of the Modern Graphics Processor [online]. Available:
<http://www.techspot.com/article/650-history-of-the-gpu>.
[Visited: dec-2015].

Understanding the Finite-Difference Time-Domain Method 2010, John B. Schneider [online]. Available: <http://www.eecs.wsu.edu/~schneidj/ufdtd>. [Visited: dec-2015].

USGS Seismometer, Seismic instruments at the Rolnick Observatory [online]. Available: <https://www.was-ct.org/resources/the-rolnick-seismometer/>. [Visited: dec-2015].

Using Inline PTX Assembly in CUDA. Nvidia (2011).

Video Display Controller [online]. Available: https://en.wikipedia.org/wiki/Video_display_controller. [Visited: dec-2015].

Von Neumann stability analysis [online]. Available: https://en.wikipedia.org/wiki/Von_Neumann_stability_analysis. [Visited: dec-2015].

VTK, Welcome VTK [online]. Available: <http://www.vtk.org/>. [Visited: dec-2015].

Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA (2009).

Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. NVIDIA (2013).

Yee, K. (1966). Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. Antennas and Propagation, IEEE Transactions on, 14(3), 302-307.

Zhang, Y., & Owens, J. D. (2011).

A quantitative performance analysis model for GPU architectures.

Proceedings - International Symposium on High-Performance Computer Architecture, 382-393.