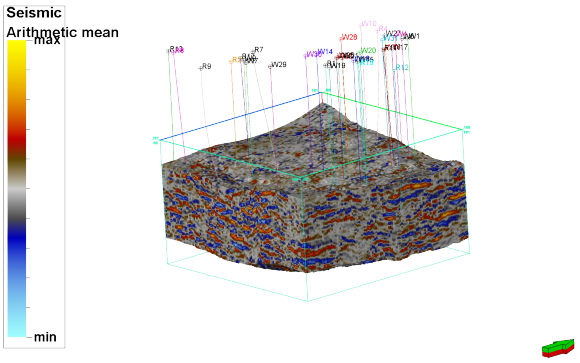
TÉCNICO LISBOA



Exploiting GPU Computing for Faster Oil&Gas Exploration

Tomás Novais Ferreirinha

Introdução à Investigação e ao Projecto em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Doutor Nuno Cavaco Gomes Horta
Orientador:	Doutor Nuno Filipe Valentim Roma
Vogais:	Doutor Gabriel Falcão Paiva Fernandes

Janeiro 2014

Resumo

Os algoritmos de inversão sísmica têm desempenhado um papel fundamental na caracterização de reservatórios de petróleo e gás, onde uma elevada precisão é necessária para ajudar a definir a localização óptima dos respectivos poços, maximizando assim a extracção destes recursos. Sendo que estes algoritmos se baseiam em simulações por computador que geram, processam e guardam grandes quantidades de informação, a utilização destes pela indústria encontra-se limitada devido aos elevados tempos de execução. Acelerar este tipo de algoritmos permite não só uma execução mais rápida destes, mas também o desenvolvimento de modelos maiores e mais precisos do subsolo. Nesta tese, técnicas de computação heterogénea são implementadas recorrendo à interface de programação OpenCL, para acelerar a execução de um algoritmo estocástico de inversão de amplitude versus deslocamento. Por forma a beneficiar do poder computacional disponibilizado por sistemas compostos tanto por CPUs como por GPUs, foi realizada uma divisão espacial da malha a ser simulada, permitindo que os processos de simulação, correspondentes a diferentes regiões do modelo geológico, sejam realizados em paralelo. Com esta abordagem, é esperada uma melhoria significativa no tempo de execução do algoritmo em estudo quando comparado com implementações actuais, sem que seja verificada uma perda substancial na precisão dos modelos obtidos.

Palavras-chave: Computação heterogénea, unidade de processamento gráfico, OpenCL, inversão sísmica.

Abstract

Seismic inversion algorithms have been playing a key role in the characterization of oil and gas reservoirs, where a high accuracy is required in order to support the decision about the optimal well locations, thus maximizing the extraction of such resources. Since these algorithms usually rely in computer simulations that generate, process and store huge amounts of data, their usage in the industry is limited by the huge execution times. Accelerating such algorithm allows not only for a faster execution, but also for the development of larger and more accurate models of the subsurface. In this thesis, heterogeneous computing techniques are implemented using the OpenCL programming framework, in order to accelerate the execution of a stochastic seismic amplitude versus offset inversion algorithm. In order to take advantage of the computational power made available by systems composed by both CPUs and GPUs, a spatial division of the simulation space is performed, enabling the parallel simulation of multiple regions of the geological model. With this approach it is expected to verify a significant reduction in the execution time of the algorithm in study when compared with the currently implemented solutions, without substantial accuracy losses in the obtained models.

Keywords: Heterogeneous computing, graphics processing unit, OpenCL, seismic inversion.

Contents

List of Tables

List of Figures

List of Acronyms

HPC	High Performance Computing
GPU	Graphics Processing Unit
CPU	Central Processing Unit
ALU	Arithmetic and Logic Unit
SM	Streaming Multiprocessor
SP	Stream Processors
SFU	Special Function Units
CU	Compute Unit
LDS	Local Data Share
CUDA	Compute Unified Device Architecture
NDRange	N-Dimensional Range space
API	Application Programming Interface
PAPI	Performance Application Programming Interface
AVO	Amplitude Versus Offset
DSS	Direct Sequential Simulation
SIS	Sequential Indicator Simulation
SGS	Sequential Gaussian Simulation
CRAVA	Conditioning Reservoir Variables to Amplitude vs Angle Data
SIMPAT	Stochastic Simulation with Patterns
Vp	P-wave velocity
Vs	S-wave velocity
lcdf	local cumulative distribution function

Chapter 1

Introduction

1.1 Motivation

Modern scientific and engineering applications have been demanding more and more computing power, in order to solve larger scale problems. This often happens because many scientific and technical problems rely in computer simulations that generate, process and store huge amounts of data. In order to efficiently solve these problems, High Performance Computing (HPC) solutions have been taking advantage of the heterogeneous architectures composed by multiple computing devices with different capabilities. The main purpose of heterogeneous computing is to exploit the benefits of the different architectures by combining them. Additionally, considerable improvements of the capabilities of general purpose Graphics Processing Units (GPUs) have been observed, enabling the usage of their high computational throughput architecture to efficiently perform parallel computations.

In the last few years, HPC platforms have been playing a key role in the Oil and Gas prospecting industry, by supporting the search for resources and thus maintaining the pace of oil and gas extraction. As an example, increased computing capabilities enabled the industry to develop computer models that help to predict where oil is located and how to maximize the extracted oil from drilling. Therefore, as the computing power is growing, HPC solutions provide an opportunity to increase the accuracy of the physics approximations used in the reservoir modelling.

In this thesis, heterogeneous computing platforms will be used in order to accelerate the execution of a stochastic seismic inversion algorithm. In particular, the stochastic seismic Amplitude Versus Offset (AVO) inversion algorithm [1] using Direct Sequential Simulation (DSS) [2], will be parallelized. This class of algorithms represents a commonly used methodology to solve geophysical inversion problems. Seismic inversion algorithms aim to model and characterize the properties of seismic reservoirs, by using a limited set of observed measurements. As a consequence, these algorithms play a key role in the characterization of oil and gas reservoirs, where accurate predictions are essential and the available information is often scarce and expensive. Nevertheless, the lack of available data is compensated by using complex geological interpretations and approximations in the computational models developed to simulate oil reservoirs.

However, most of the used approximations in stochastic inversion algorithms result in models with a high level of uncertainty, which may lead to a faulty understanding of the geological structure and consequently to drilling errors. Such errors can become rather expensive, not only in terms of investment costs, but also in terms of environmental impact. Improving the quality of such models will enable earth scientists and reservoir engineers to further enhance the understanding of the subsurface characteristics of the oil and gas fields. Therefore, HPC techniques applied to oil and gas applications not only allow for faster execution of the algorithms, but also to make it possible to develop larger and more accurate computational geology models, in a way that is still computationally unfeasible at this moment using conventional platforms.

1.2 Related Work

In order to deal with the complex large-scale Oil and Gas exploration, geoscience applications have been being improved in order to efficiently take advantage of the parallel capabilities of cluster systems composed by Central Processing Unit (CPU) and GPU devices. Thus, multiple parallel implementations of the algorithms that perform seismic data acquisition, processing and interpretation can be found in the literature. For example, a widely used method to process reflection based geophysical data is known as seismic migration. Even running on CPU clusters, the execution of seismic migration algorithms require a significant amount of time, compromising the production schedules. Such algorithms have been improved by taking advantage of the massive parallel nature of the GPUs. For instance, in [3] [4] [5] those algorithms have been implemented using both CPUs and GPUs, presenting significant performance gains when compared with the implementations using only CPU cores. Another example of the usage of heterogeneous computing in order to accelerate geoscience applications, is the parallelization of subsurface flow simulation algorithms [6] [7]. Once again, impressive improvements have been verified, with speedups greater than 30.

Despite the close relationship between the geoscience and HPC industries, the actual exploitation of parallel and concurrent computing techniques to accelerate geostatistics and sequential simulation algorithms is not yet widely developed. Regarding geostatistical seismic inversion algorithms, parallel implementations of the stochastic seismic AVO inversion algorithm using DSS were not found in the literature. However, algorithms with the same purposes have been already optimized and parallelized, as is the case of Conditioning Reservoir Variables to Amplitude vs Angle Data (CRAVA) in [8], [9]. In those works, straightforward multi-core parallelization approaches were implemented, being observed an almost linear speedup up to 16 cores.

In what concerns the stochastic simulation part of the algorithm in study, only one parallel implementation of the DSS algorithm was found [10], where a multi-core approach was implemented by considering a functional decomposition of the algorithm. Also, the same parallelization approach was implemented for other stochastic simulation algorithms, namely the Sequential Indicator Simulation (SIS) and the Sequential Gaussian Simulation (SGS), being the resulting speedups almost linear up to 4 cores

[1]. Another parallelization approach based on a spatial division of the problem was studied in [11], where it is shown that, as long as a certain distance is kept between the nodes being simulated in parallel, the obtained models result in the same properties as the ones generated by the original algorithms.

As for the usage of GPU architectures to parallelize stochastic simulation algorithms, a parallel implementation of the Stochastic Simulation with Patterns (SIMPAT) algorithm was developed in [12], being obtained some encouraging results, since the computational time was reduced by a factor of 26-85, depending on the problem size. Also, some mechanisms to manage the conflicts between the nodes being simulated in parallel are presented, which may be interesting in case it shows to be a significant problem during the development of this thesis.

1.3 Objectives

The main objective of this thesis is the conception of parallel and highly efficient implementations of a geoscience application used for prospecting, extracting and monitoring of energy sources like oil and gas. Accordingly, to achieve this objective the thesis should include:

- Profiling of the considered algorithm, in order to quantify its requirements in terms of computational resources, dependencies, arithmetic precision and memory usage;
- Functional-level partitioning of the algorithm's structure into several independent functional modules;
- Data-level partition of the most data intensive modules;
- Definition of appropriate scheduling, synchronization and load balancing policies;
- Selection of the target architectures, as well as the programming framework;
- Implementation of the defined parallelization model in a state of the art high performance computing platform, composed by a multi-core CPU and one or more GPUs;
- Experimental evaluation and characterization of the developed system.

Taking into account the aforementioned objectives, this report was divided in 5 chapters. In chapter 2, a brief introduction to the problem, as well as a description of the algorithm in study is presented. Then, an overview over the targeted architectures and also over the programming framework to be used is performed in chapter 3. In chapter 4, the considered algorithm is profiled and analysed, in order to support the definition of where the parallelization effort should be focused. Also, some parallelization approaches that will be further developed and optimized in the scope of this research are presented, as well as some intermediate results, being the report finished with chapter 5, where the main conclusions are drawn and future work is outlined.

[1] Results considering more than 4 cores were not presented.

Chapter 2

Seismic Inversion Algorithms

2.1 Problem Description

The main goal of inversion problems is to make inferences about physical objects or systems, given a limited set of observed measurements (figure 2.1). In the, geophysical application domain, inversion algorithms estimate a set of models that characterize the physical properties of the Earth's subsurface, by incorporating geophysical survey data. In particular, in seismic inversion algorithms, seismic reflection data is conveniently processed in order to produce the seismic models of the Lithosphere [1].

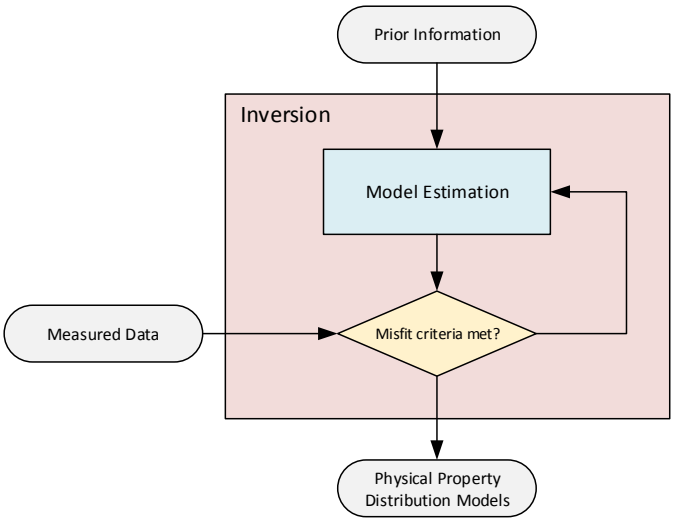


Figure 2.1: Simplified inversion workflow.

The seismic reflection data is usually obtained by using reflection seismology, which is the process of sending seismic waves into the Earth (using an energy source, such as a dynamite explosion or a specialized air gun), and measuring the respective reflections (figure 2.2). This class of waves propagate through the Earth with a velocity defined by the acoustic impedance of the medium supporting their propagation. Seismic waves can be divided in two different categories: body waves, which travel through

[1] Hard and rigid outer layer of the Earth

the interior of the Earth; surface waves, which travel across the Earth's surface. In what concerns the characterization of the Earth's subsurface, the body waves are commonly used to infer its structure. Body waves can be classified either as Primary waves (P-waves), which are longitudinal pressure waves that travel faster through the Earth than the other waves, or as Secondary waves (S-waves), which are transverse shear waves that can only travel through solid medium.

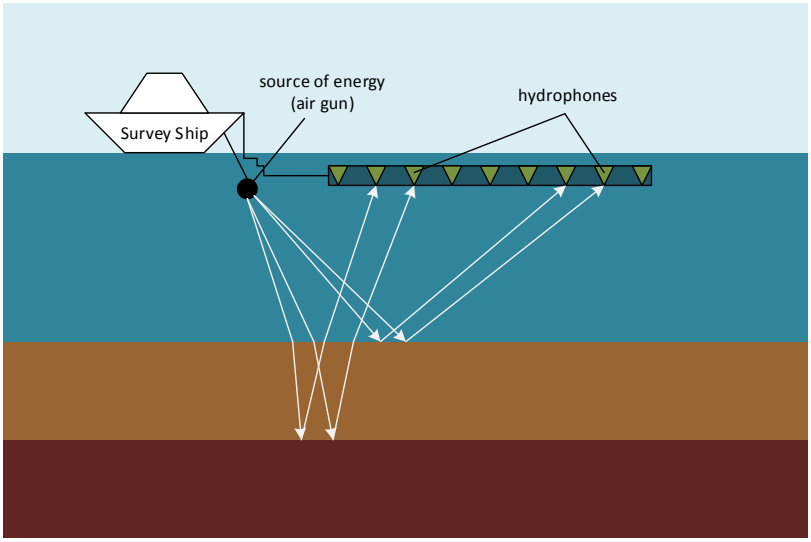


Figure 2.2: Illustration of marine seismic data acquisition.

When a seismic wave travels through the interface between two materials with different acoustic impedances, some of the wave energy is reflected, while the rest of the energy is refracted through the interface. Considering that a given source and corresponding receiver are not apart from each other (zero-offset), when a reflected wave is perceived in the receiving sensor (geophone or hydrophone), both the wave travel time and propagation velocity can be used to estimate the distance at which the reflection occurred, defining a half-sphere [2] of possible locations of the reflector. Higher detail about the structure of the subsoil can be obtained by using multiple receivers separated by a known distance, which also adds an extra complexity to the problem. Such complexity is reinforced if every reflection of the seismic waves is to be considered for the model computation. Usually, this type of problems is minimized with a technique known as *stacking*, where multiple similar weaker signals are stacked into a stronger one, thus reducing the amount of seismic data to be processed and increasing the signal to noise ratio.

Seismic inversion problems are usually solved either through a deterministic approach, where inverted Earth models simple enough to keep them mathematically manageable are produced, or by finding a set of equiprobable solutions using stochastic methods. The main disadvantage of the former methodology is that, contrary to the latter, it does not explore the existing data uncertainty. This is a significant disadvantage, since nearly all observed data is subject to some uncertainty, which will condition the solution found through deterministic approaches. On the other hand, finding stochastic solutions is

[2] Possibilities that occur above the surface are not considered to be reasonable.

computationally expensive, because the inverse problem is solved by an iterative minimization of the difference between simulated synthetic data and real observed seismic data.

In oil and gas applications, stochastic modelling normally makes use of well logs and core data. Such data is obtained by making a detailed record of the geologic formations penetrated by a borehole, either by analysis of samples brought to the surface or by lowering measurement equipments into the hole. Thus, core and log data is usually a reliable source of information, but scarce and expensive. This lack of information reinforces the high degree of uncertainty presented by the models. Nevertheless, the reservoir models that are generated via stochastic inversion algorithms can significantly be improved with the integration of different kinds of information [13], such as with the integration of both well log and seismic reflection data. The most commonly used methodology to incorporate seismic information in stochastic fine grid models is known as geostatistical inversion [14].

Geostatistical inversion methods consist in two steps:

1. Simulation of acoustic impedance values based on well data and spatial continuity pattern;
2. Transformation of the data in order to compare the simulated acoustic impedance values with the real seismic.

To accomplish the first step, stochastic simulation techniques constrained by real seismic data are used in order to model the reservoir characteristics. The stochastic realizations can be obtained either through sequential methods, where each realization is built step-by-step[3], or through iterative methods where a simulation is obtained by successive corrections of an initial image, leading to the minimization of an objective function. Regarding the second step, the transformation of the data can be performed either by using an inverse method that transforms the seismic data into acoustic impedance values, removing the effect of a known estimated wavelet [4], or by using a forward method that does the opposite by convolving the acoustic impedances with the estimated wavelet, obtaining synthetic seismic data that can be compared with the real seismic.

2.2 Stochastic Seismic Amplitude vs Offset Inversion

The particular algorithm that will be accelerated in the scope of this thesis is known as Stochastic Seismic AVO Inversion [1]. This algorithm represents an iterative geostatistical inversion method, based on the Global Stochastic Inversion [15] [13] approach, extended to pre-stack seismic data. This method directly inverts the density, the P-wave velocity (Vp) and the S-wave velocity (Vs) models, instead of inverting the acoustic impedance models as it usually happens in post-stack geostatistical inversion algorithms.

The stochastic seismic AVO inversion algorithm is based on two key procedures:

- Density, Vp and Vs models are perturbed towards an objective function.

[3] Each simulated element is conditioned by the previous simulated elements.

[4] One-dimensional wave-like oscillation pulse that represents how an impulse of energy propagates in the earth, providing a relation between the seismic and the well data.

- The best parts of each simulated model are iteratively selected with a genetic algorithm, computing the mismatch between the real and the synthetic seismic data, in order to evaluate the fitness of the model.

In the considered approach, the global transformation required to compare the simulated values with the real seismic is performed by applying a forward model to the simulated density, Vp and Vs images. Those images are obtained using the DSS with Joint Probability Distributions algorithm [2] [16]. Regarding the computation of the mismatch between synthetic and real seismic data, a simple co-located correlation coefficient is computed (figure 2.3). In this process, the cubes containing seismic amplitudes are vertically divided into layers, being the mismatch computed for each pair of layer-size columns of both seismic data cubes. The results are stored in a mismatch cube that provide the information about which locations in the reservoir are fitting the seismic and which need further improvement. Finally, by taking into account the locations with higher values of correlations, the best models of density, Vp and Vs are built and used as a secondary image, conditioning the next generation of simulations. Thus, the stochastic seismic AVO inversion method can be summarized in the following steps:

1. Stochastic simulation of the density/Vp/Vs data using DSS with joint probability distributions;
2. Calculation of the synthetic pre-stack seismic cube with the simulated density/Vp/Vs models, by using Shuey's approximation [17];
3. Comparison between the synthetic seismic cube and the real seismic data;
4. Creation of the seismic correlation cube;
5. Best density/Vp/Vs models are built by selecting areas of higher correlation using a genetic algorithm;
6. Creation of the correlation cubes regarding the best den/Vp/Vs models;
7. Repeat the step 1 using the built best models and respective correlation cubes as a secondary image, until a matching criteria is reached.

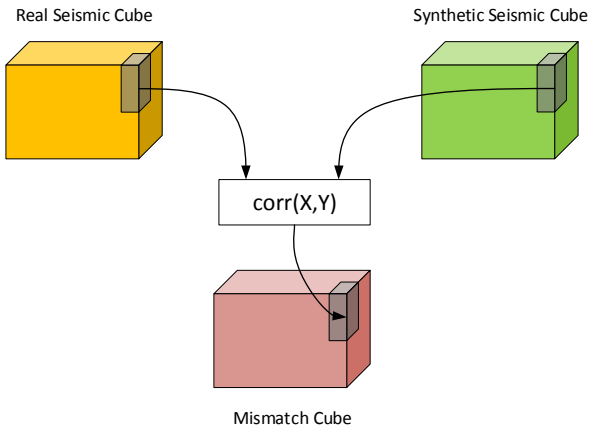


Figure 2.3: Construction of the mismatch cube

For each iteration of the algorithm, several density/Vp/Vs simulations can be performed. However, the used model to condition the variable being simulated is always the best model corresponding to the previous iteration. The only exception is the first iteration (when there is still no best models built), where DSS with joint probability distributions is only used for the simulation of Vp (conditioned by the previous simulated density model) and Vs (conditioned by the previous simulated Vp model). Accordingly, figure 2.4 illustrates the algorithm flowchart.

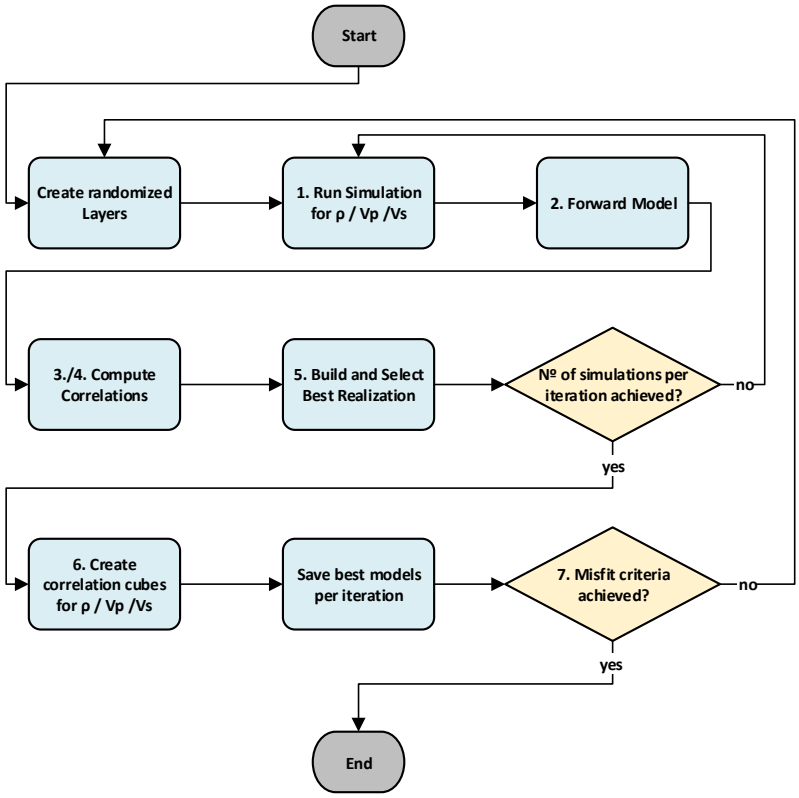


Figure 2.4: Stochastic seismic AVO inversion algorithm flowchart

As mentioned before, the stochastic simulation algorithm being used (step 1) is the Direct Sequential Simulation and Cosimulation [2] (see figure 2.5), which is a sequential stochastic simulation algorithm. As is typical of this class of stochastic simulation algorithms, it starts by defining a random path through the grid of nodes, that is followed during the simulation process. Afterwards, one node is simulated at a time, conditioned by the real data and all the previously simulated values. The simulation procedure consists in the estimation of a local cumulative distribution function (lcdf) by linear interpolating both the real and experimental data available within a neighbourhood around the node being simulated (kriging estimate). Finally, a simulated value is sampled from the defined conditional distribution function using a Monte Carlo simulation method.

Therefore, the DSS algorithm can be summarized in the following steps:

1. Randomly select a node from a regular grid;

2. Construction and solution of a kriging system for the selected node;

3. Estimate the lcdf at the selected node, conditioned to the original data and the previous simulated nodes;

4. Draw a value from the estimated lcdf (Monte Carlo simulation);

5. Return to the step 1 until all nodes have been visited by the random path.

The cosimulation variant of this algorithm enables the simulated variable to be conditioned to other previously simulated variables without any prior transformation, being one of the main advantages of the DSS algorithm when compared with the other traditional sequential simulation algorithms, as is the case of the SIS and SGS algorithms. In this case, it is only implemented the joint simulation using a single auxiliary variable.

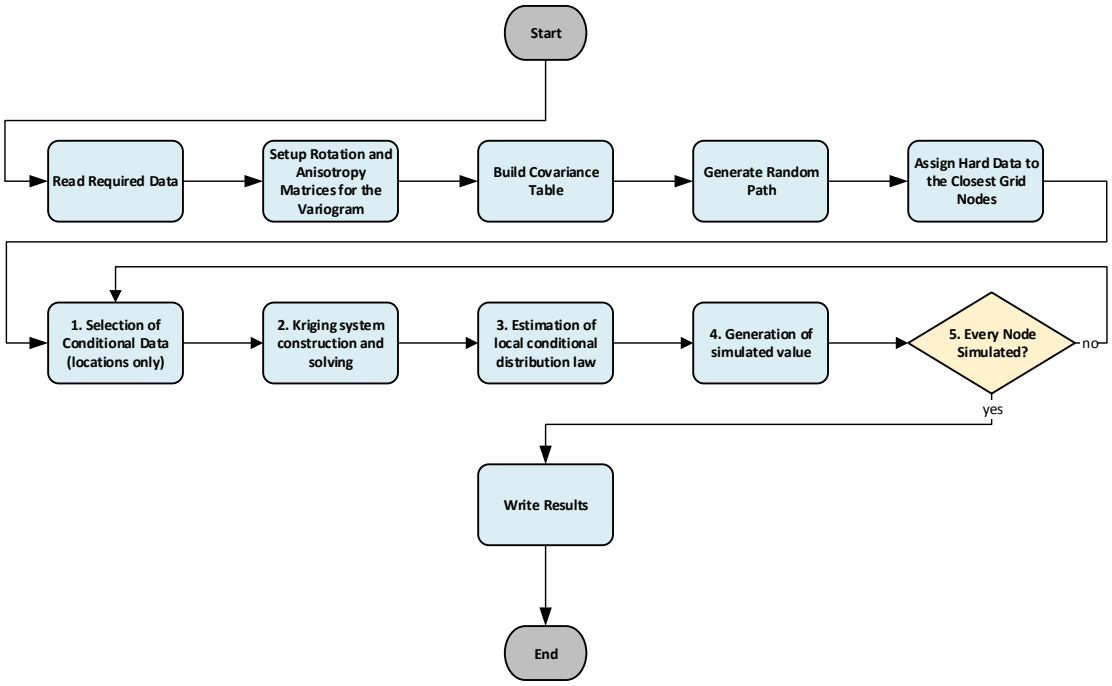


Figure 2.5: DSS algorithm flowchart

Chapter 3

Target Architectures

Heterogeneous computing systems are composed by a variety of different types of computational devices. Such systems can be composed by general-purpose processors (CPUs), special-purpose processors (GPUs, DSPs), and sometimes even by custom acceleration logic devices (FPGAs). In the scope of this thesis, heterogeneous systems are going to be used in order to accelerate the algorithm in study. In particular, the processing capabilities of systems composed by both CPUs and GPUs are pretended to be efficiently exploited.

3.1 CPU Architectures

The CPU, is the main component from a digital computer (see figure 3.1). It is responsible for executing every instruction that composes a computer program, by implementing a general purpose set of instructions. From the point of view of the hardware, a CPU is mainly composed by the datapath, where data-processing operations are performed, and the control unit where the flow of information between the multiple components of the CPU is managed.

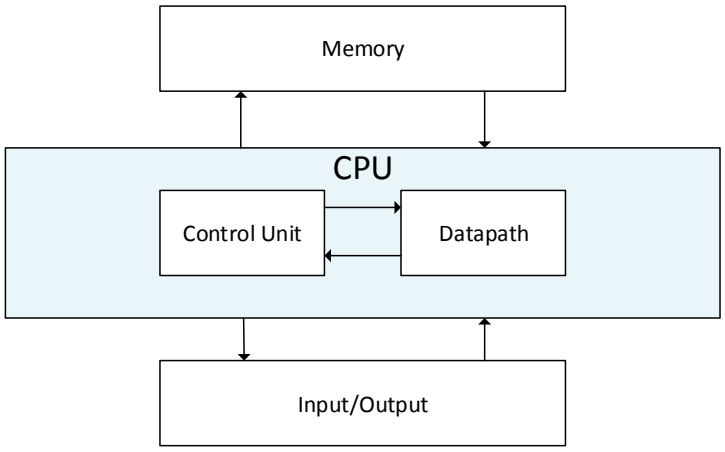


Figure 3.1: Block diagram of a CPU.

The datapath is mainly composed by an Arithmetic and Logic Unit (ALU), where the microoperations specified by the instructions are performed over the data stored in a set of registers, also located in the datapath. Regarding the control unit, it is mainly composed by an instruction decoder that transforms the bits that identify a given instruction into the corresponding control signals that control the other parts of the CPU in order to perform the operation specified by the instruction.

The execution of a given instruction can be summarized in the following steps:

1. Instruction Fetch: The instruction to be computed is read from the program memory.

2. Instruction Decode and Operand Fetch: The instruction is transformed into the set of control signals to be used in the subsequent phases and the operands required to perform the instruction are loaded from the registers.

3. Execute: The operation defined by the instruction is performed with the given operands.

4. Memory Access: Data memory reads and stores are performed if specified by the instruction.

5. Write Back: The result obtained from step 3 is stored into a register of the datapath.

The performance assessment of CPUs can be performed by using multiple metrics. Some commonly used metrics are: the speed at which it implements the operations, usually measured by the processor maximum clock frequency; the amount of instructions performed per clock cycle, also called the throughput; the amount of floating-point instructions that the processor is able to perform per second. Despite being widely used, none of this metrics is a good metric by itself. For example, none of this metrics consider that the implemented instruction set of the architectures being compared may vary. A difference in the instruction set may result in a faster execution of a specific operation/algorithm, although the processor itself may perform instructions slower. Therefore, when comparing multiple processors, the performance is frequently assessed by comparing the execution times of a given set of applications considered representative for the objective of the processors being evaluated. The evaluation of a processor may also depend on its purpose, which is the case of architectures optimized to be power efficient.

Depending on the design goals, the adopted techniques to improve the performance of a CPU may vary in terms of their complexity. A common optimization strategy results from the observation that in a single clock-cycle per instruction architecture, the execution path of an instruction leads into a long delay path, compromising the maximum frequency of execution. In order to solve this problem, this path is usually broken up using registers, leading into the so called pipeline architecture. This optimization enables the CPU to execute different parts of multiple instructions, at the same time, using an higher clock frequency. However, the throughput of the processor may become compromised due to dependencies between consecutive instructions. Such problem can be minimized by using other techniques, such as data-forwarding, speculative execution, branch prediction and simultaneous multithreading. Other commonly used architectural optimization tries to minimize the huge memory access latencies by using an hierarchical memory model. This scheme integrates several smaller and faster memories, also known

as caches, that take advantage of the principles of temporal locality [1] and spatial locality [2]. Despite the offered set of performance benefits, architectural optimizations usually come with an increase of complexity, area and consequently cost.

Further improvement on the CPU architectures continued by the development of increasingly faster single core CPUs, until they started facing several power efficiency limitations. Nowadays, more processing power is being achieved also by increasing the amount of processor cores within a single chip and by optimizing their interoperability when exploring parallel processing. This parallel processing approach revealed to have a good performance scalability and power efficiency. However, it was not transparent to the programmers, who faced a new programming paradigm and the algorithms had to be redesigned in order to take advantage of this kind of architectures. Regarding the new programming model, the following types of parallelism were identified:

- Task parallelism: Different computations can performed at the same time on the same or different sets of data.

- Data parallelism: The same instruction or set of instructions are applied to multiple sets of data at the same time.

- Instruction-level parallelism: Instructions are reordered and recombined into groups which can be executed in parallel. This method is also known as software pipelining.

Modern CPU architectures offer an efficient and complex set of architectural solutions to exploit the different kinds of parallelism (see figure 3.2).

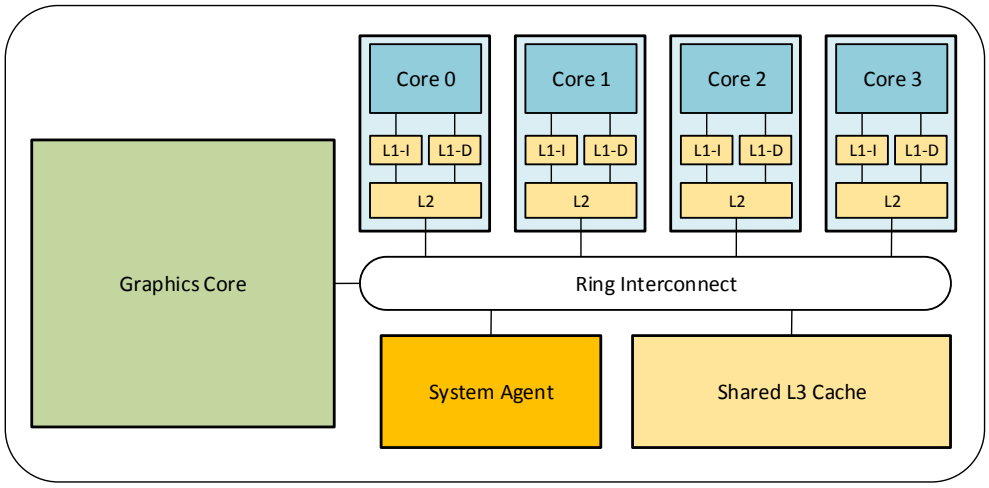


Figure 3.2: Modern CPU architecture.

[1] Temporal locality principle states that if a given memory position was accessed, it will probably be accessed again in the near future.

[2] Spatial locality principle states that if a given memory address is accessed, most likely nearby memory positions are going to be accessed in the near future.

3.2 GPU Architectures

A GPU is a specialized processor originally designed to render computer graphics. This kind of processor evolved in order to provide real-time high-definition 3D graphics rendering for the gaming industry. In graphics rendering applications, a set of operations are performed in order to transform coordinates from a 3D space into a 2D pixel space to be displayed on the screen. Computing each pixel in series using a CPU architecture was computationally unfeasible within the pretended interval of time. In order to solve this, the characteristics of this application led into a highly parallel many-core processor architecture, specifically designed to efficiently perform data-parallel computations. That way, in contrast with the CPU architecture, GPUs are composed by a massive number of smaller and simplified cores, optimized to provide higher throughput and memory bandwidth (no branch prediction, no out-of-order execution, no data-forwarding mechanisms). This difference is illustrated in figure 3.3.

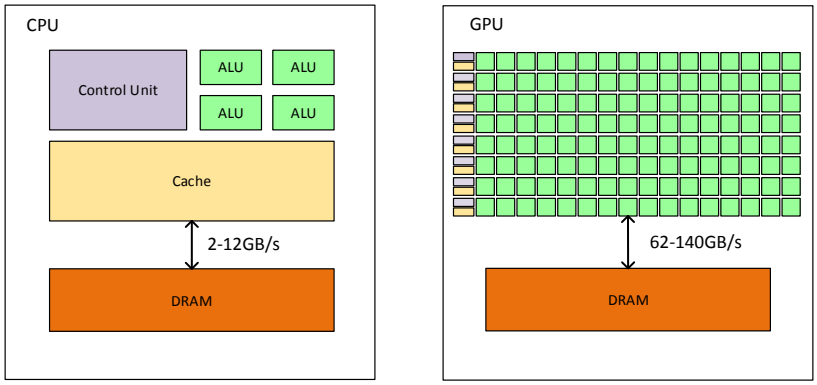


Figure 3.3: Relative occupation of resources comparing CPU and GPU architectures

Originally, the GPU architectures started by being a fixed function graphics pipeline devices (multiple vertex processors running vertex shader programs and multiple pixel fragment processors running pixel shader programs). In 2001, an important step in the evolution of GPU architectures was performed by the NVIDIA's GeForce 3 with the introduction of the first programmable pipeline architecture. The programmability of this chip was very limited and non-graphics applications had to be carefully crafted in order to fit into the graphics hardware pipeline. In 2006, with the GeForce 8 series, NVIDIA introduced the first architecture based on a fully programmable unified vertex and pixel processor, denoted as Streaming Multiprocessor. This new architecture design made the general-purpose GPU programming (GPGPU) easier, making GPUs an attractive resource to developers of HPC applications.

Nowadays there are two major GPU manufacturers: AMD and NVIDIA. Modern NVIDIA GPU architectures are mainly composed by multiple Streaming Multiprocessors (SMs), being each one of them composed by multiple Stream Processorss (SPs) as well as some memory load/store units and some Special Function Unitss (SFUs), as illustrated in figures 3.4 and 3.5.

Within the NVIDIA GPU architecture, memory is distributed over three hierarchical levels:

- Global memory: memory space shared between every thread. A subset of this memory is reserved for constant memory, from where values can be fully cached, allowing accesses as fast as the

ones performed to registers. Reads from global memory are cached but are still very slow when compared to the other memory classes.

- Shared memory: memory space shared between threads executing within a SM. It is located closer to the stream processors, allowing very fast accesses. It can be viewed as a user-programmable cache.
- Private memory: individual memory space for each thread. This class of memory reside in on-chip registers and is a very limited resource.

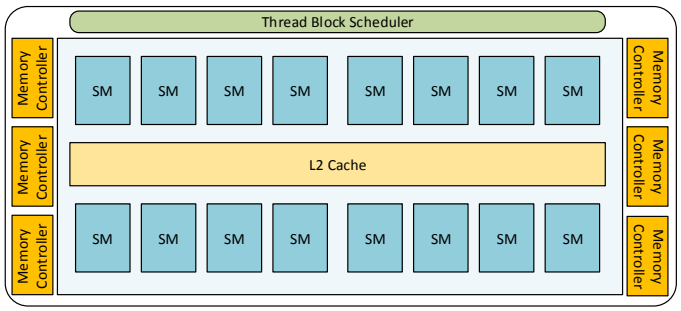


Figure 3.4: Modern NVIDIA GPU architecture

Synchronization is only possible between groups of threads executing within the same SM and not across the whole device. That happens because synchronization is only possible through the local memory and not through the global memory, due to cache consistency problems.

Each SM manages the scheduling and execution of instructions in groups of 32 parallel threads, also known as warps. Multiple warps may be assigned to the same SM and at each cycle the scheduler issues the next instruction corresponding to a warp that is ready to execute. Despite all the threads inside a warp have to execute the same instruction at the same time, each thread may have its own execution path. Therefore, maximum efficiency is obtained when all 32 threads of a given warp agree on their execution path. Whenever different threads of the same warp need to execute different instructions (usually a consequence of *if(condition)* statements), there is a significant loss of parallel efficiency, since all threads in the warp will require to pass through the instructions of both conditional branches. Threads that are not expected to execute a given conditional branch become inactive until execution paths converge again. This behaviour is called *warp divergence* and is a major concern when programming these architectures, since in the worst case it may lead into a performance loss factor of 32.

Another significant problem results from non consecutive accesses to the global memory, between threads of the same half-warp. In fact, the global memory access by threads of the same half-warp can be divided in at most two memory transactions [3] if the accessed words lie in the same memory segment [18]. When this condition is not fulfilled, separate memory transactions are issued for each thread, significantly compromising the throughput. This condition may be even more restrictive for older devices, that typically require accessed memory positions to be sequential for the threads within the same half-warp.

[3] Depending in the word size.

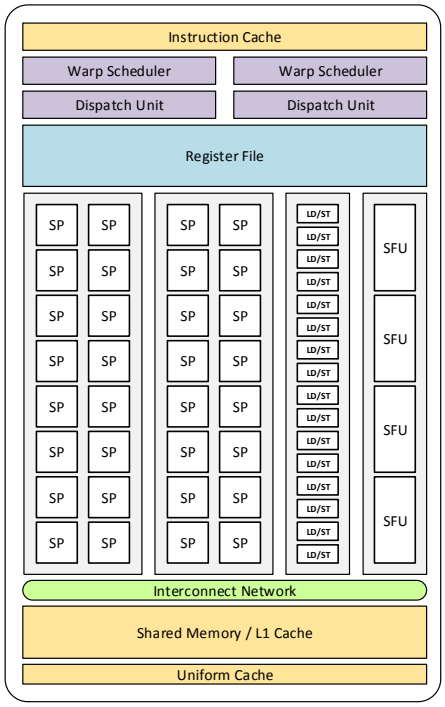


Figure 3.5: Streaming Multiprocessor architecture

Regardless the above mentioned problems, an important metric when analysing the GPU performance is called occupancy. Occupancy can be calculated using expression 3.1, and is an important metric for the choice of the number of threads to be executed in the same SM (thread block size), when programming GPUs.

$$\text{Occupancy} = \frac{\text{Number of active warps}}{\text{Maximum number of active warps}} \tag{3.1}$$

Thus, in order to obtain the maximum efficiency when programming GPUs, the occupancy must be maximized. However, occupancy is limited by several factors, such as the number of used registers per thread. In fact, since a SM has a limited number of available registers and the executing threads require a fixed number of registers, the lack of resources may limit the occupancy. In order to reach maximum occupancy, the number of registers per thread must agree with the condition expressed in the equation 3.2. Another limiting factor is the amount of shared memory used by the executing threads, because it is also a limited resource, and therefore it limits the number of active warps.

$$\text{Registers per Thread} \leqslant \frac{\text{Registers per SM}}{(\text{Max Warps per SM}) \cdot (\text{Threads per Warp})} \tag{3.2}$$

In what concerns modern AMD GPUs, their architectures are referred as a data-parallel processor array of multiple Compute Units (CUs), each of them composed by a Local Data Share (LDS) memory and by multiple vector and scalar units. The number of compute units in an AMD GPU, and the way they are structured, varies with the device family. Considering the *Southern Islands* devices architecture [19] [20], a CU is composed by 1 scalar unit and 4 vector units, being each vector unit composed by an array of 16 processing elements. Each of these arrays execute a single instruction over a block of 16 work-

items. An instruction takes four cycles to be executed, being performed over blocks of 64 work-items, also called a wavefront. Once again, lost of efficiency is verified whenever the execution path from work-items belonging to the same wavefront diverge. Each compute unit can work on multiple wavefronts in parallel, by simultaneously processing vector and scalar computations. Similarly to what happens with the NVIDIA GPUs, the memory hierarchy is divided in three memory spaces: global/constant memory, local memory and private memory. Synchronization is only possible between work-items executing within the same CU, since they can share memory through the LDS. The collection of work-items able to share data and synchronize with each other is called a work-group.

3.3 Framework Decision

Heterogeneous computing is increasingly becoming more important for the HPC market. The heterogeneity of hardware devices available in almost every desktop computer offers a great opportunity to efficiently exploit the different kinds of parallelism. Even modern CPUs, with the inclusion of on-chip graphic processing units, are already endowed with an heterogeneous architecture. Portable and efficient developing frameworks are thus required in order to map applications into this heterogeneous hardware.

There are two dominant frameworks for developing heterogeneous computing: NVIDIA's Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). The first one is proprietary and enables the programmers to use modern NVIDIA GPUs for general purpose programming, providing both a low level Application Programming Interface (API) and a high level API. The disadvantage of this framework, when compared to the previously explained OpenCL framework is that the latter is not restricted to a given vendor of GPUs neither to GPU architectures itself, offering more code portability and flexibility when compared to the CUDA programming language. However, that does not necessarily means that OpenCL code will run optimally on all of the distinct architectures, neither that the developed code is necessarily going to be portable between them all. It is up to the programmer to decide how efficient and how portable its implementation is going to be.

3.3.1 Description of the OpenCL Framework

The OpenCL is a project developed in order to facilitate the programming of applications for heterogeneous systems. Currently it is managed by the non-profit technology consortium Khronos Group [21]. OpenCL enables the programmer to exploit the advantages of different classes of architectures by efficiently combining them. For example, data parallel computations can be efficiently performed using GPU architectures, while CPU architectures perform sequential and task parallel computations. Thus, OpenCL tries to offer a uniform multi-platform development framework, providing an efficient way to map parallel applications into homogeneous or heterogeneous systems composed by different devices such as multi-core CPUs, GPUs, digital signal processors (DSPs) and field programmable gate arrays (FPGAs). At the same time, OpenCL tries to offer an implementation as close to the hardware as possible,

providing only enough abstraction in order to become device and vendor independent. The OpenCL specification is composed by four models:

- Platform model: Specifies the execution coordinating processor (the host) and the devices capable of executing the OpenCL functions (kernels).
- Execution model: Defines how a program is executed in the host and how kernels are grouped into programs and assigned to devices.
- Memory model: Defines the abstract memory hierarchy model that kernels use.
- Programming model: Defines how the adopted concurrency models are mapped into the hardware.

The platform model presents an abstract device architecture that programmers target when writing OpenCL C code. This model defines a device as an array of functionally independent compute units, which are further divided into processing elements. The mapping of the OpenCL abstract architecture into the physical hardware is provided by each platform vendor OpenCL API implementation. Thus, OpenCL platform targeted devices are limited to those who offer an OpenCL device driver.

Regarding the execution model, kernels are syntactically similar to C functions and correspond to the set of instructions that each work-item [4] will execute in parallel. When a kernel is launched, work-items are grouped into work-groups and mapped into a N-Dimensional Range space (NDRange). A NDRange is a one, two or three-dimensional space of work-items, whose dimensions are defined by the programmer when launching a kernel (figure 3.6). Inside a given kernel, work-items can identify themselves using OpenCL specific function calls that return the position of that specific work-item relative to the whole NDRange or relative to its work-group.

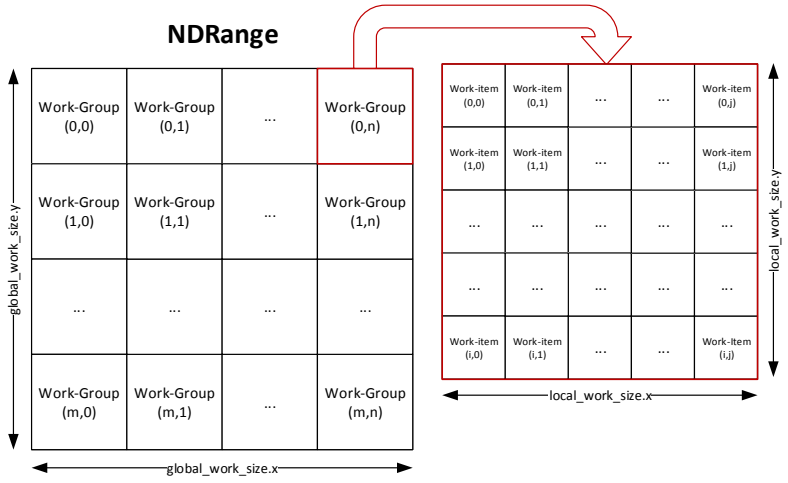


Figure 3.6: NDRange organization

In what concerns the memory model, the work-items within a work-group share a local memory space, in addition to the global memory space that is shared among every work-item in the device. This local memory space is usually smaller than the global memory space but provides faster access

[4] Similar to the concept of thread.

times. Additionally, every work-item also has a private memory space, which corresponds, in terms of hardware, to internal registers located inside the processor where the work-item is being executed. Both platform and memory models used by OpenCL closely correspond to the hardware architecture of the GPU.

Finally, in what concerns the programming model, OpenCL adopts both data parallel and task parallel models. Data parallel programming model corresponds to having several instances of the same kernel being executed at the same time by multiple work-groups, being each work-group composed by a sequence of data elements formed by multiple work-items. Task parallel programming model corresponds to several work-groups executing independently in a way that different kernels may be running in parallel. Every OpenCL compliant device must implement at least the data parallel model.

Chapter 4

Algorithm Paralellization

4.1 Problem Analysis

In order to efficiently parallelize an application, it is important to define how performance is going to be measured and to identify where and how the parallelization effort is going to be applied. Regarding the measurement of performance, when accelerating a given application, the most commonly used metric is known as speedup (ψ), which represents how much faster a parallel algorithm is being executed when compared with its sequential implementation. Thus, the speedup is defined as the ratio between the sequential execution time (T_s) and the parallel execution time (T_p) (see equation 4.1). Performance improvement is considered when the speedup is greater or equal to one. Therefore, the purpose of parallel programming is to reduce the execution time by evenly dividing the computational effort between the available processors (p) (see expression 4.2). As such, the efficiency (e) of a parallel implementation is given by the ratio between the speedup and the number of processors used (expression 4.3).

$$\psi = \frac{T_s}{T_p} \tag{4.1}$$

$$\psi = \frac{T_s}{\frac{T_s}{p}} = p \tag{4.2}$$

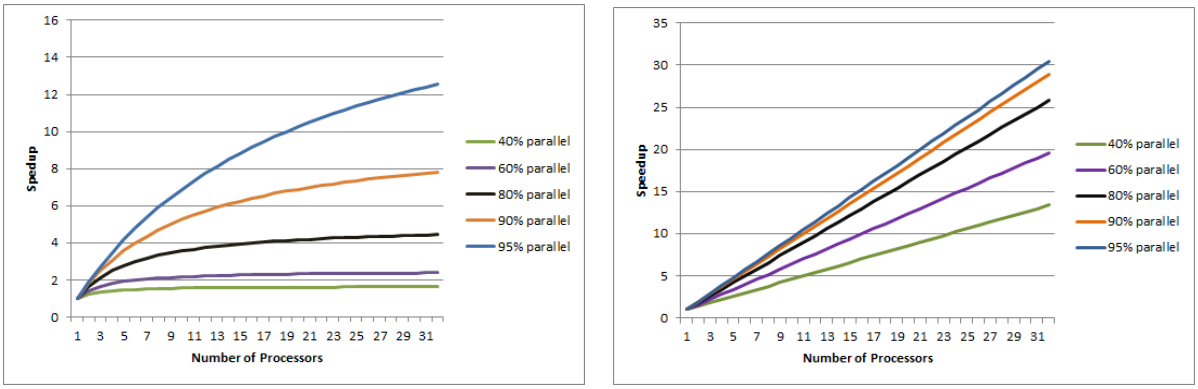
$$e = \frac{\psi}{p} \tag{4.3}$$

By analysing expression 4.2, where the optimal situation is considered, as the number of processors goes to infinity, the parallel execution time tends to zero, and consequently, the maximum achievable speedup when parallelizing a given application would also be infinite. However, real applications usually are not completely paralellizable, being composed by a sequential code fraction (s) that cannot be executed in parallel. This limits the maximum theoretical attainable speedup because the parallel execution time is never going to be lower than the time spent executing the sequential part of the application. Thus, the maximum speedup that can be achieved can be defined by expression 4.4, also known as the Amdahl's Law. This expression does not take into account the communication overheads inherent of a parallel algorithm implementation, which means that in most cases it represents an overestimation of

the maximum achievable speedup. Furthermore, Amdahl's law also does not consider the scalability of the problem being solved, by computing the speed-up assuming a fixed-size problem. Taking this problem into consideration, Gustafson introduced a fixed-time speedup model, considering that the parallel execution time is constant rather than the serial execution time, which means that as the number of processors increases, larger problems may be solved during the same time span. Thus, a *scaled speedup* can be calculated by using the expression 4.5. Figures 4.1(a) and 4.1(b) show the evolution of the maximum speedup estimation with the number of parallel processors both for Amdahl's and Gustafson's laws, respectively. Despite the observed differences, both Amdahl's and Gustafson-Barsis' Laws are overestimations of the achievable speedup. The difference lies in whether the problem is seen in the perspective of making an application to run faster with the same workload, or running the application in the same time span with a larger workload.

$$\psi = \frac{T_s}{T_p} \leqslant \frac{T_s}{s \times T_s + (1-s)\frac{T_s}{p}} = \frac{1}{s + \frac{1-s}{p}} \tag{4.4}$$

$$\psi = \frac{T_s}{T_p} \leqslant \frac{s + p \times (1-s)}{s + (1-s)} = s + p \times (1-s) = p + (1-p) \times s \tag{4.5}$$



(a) Amdahl's Law.

(b) Gustafson's Law.

Figure 4.1: Amdahl's and Gustafson's speedup laws.

By taking the Amdahl's law into account, one can observe that only the sections of an application where most of the execution time is spent are worth of being parallelized. By taking this observation into consideration, the algorithm in study was profiled both for a smaller dataset (composed by a grid of 101x101x90 nodes) and a bigger one (with 237x197x350 nodes), which better represents the dimension of data used in real problems, in order to find out where most of the application execution time is being spent.

For such purpose, the algorithm was conveniently profiled using the Performance Application Programming Interface (PAPI) [22], supported on the hardware performance counters found on most modern processors to measure the execution time of the different parts of the algorithm.

As it can be observed in table 4.1, more than 90% of the algorithm execution time is spent in the generation of the density/Vp/Vs models, by using the DSS algorithm. Consequently, this part was identified

as the block where the parallelization effort may result in a bigger speedup. In particular (for this case) the maximum speedup that can be achieved with the paralellization of this part of the algorithm, by considering that it represents 95% of the execution time (which is actually verified for the bigger dataset), is approximately 20. After observing that most of the time is spent in this simulation procedure, the same analysis can be performed in order to identify what is the part of the DSS algorithm that is worth to be parallelized. The obtained results are shown in table 4.2 .

	101x101x90		237x197x350	
	1 sim. 1 iter.	2 sim 2 iter	1 sim 1 iter	2 sim 2 iter
Run simulations	90.0%	93.8%	94.8%	96.8%
Forward model	0.9%	0.9%	0.5%	0.5%
Compute correlations	0.1%	0.1%	0.1%	0.1%
Build best realizations	1.1%	1.2%	0.5%	0.5%
Compute local correlations	3.1%	1.6%	1.6%	0.8%
Save best results per iter.	4.5%	2.2%	2.3%	1.2%

Table 4.1: Profiling results for the stochastic seismic AVO inversion algorithm.

	101x101x90			237x197x350		
	den.	Vp	Vs	den.	Vp	Vs
Read required data	0.1%	0.1%	0.2%	0.0%	0.1%	0.0%
Set rot. and anisotropy matrices	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Build covariance table	12.9%	9.7%	9.6%	0.6%	1.9%	0.3%
Generate random path	0%	0%	0%	0.1%	0.3%	0%
Assign data to closest nodes	0%	0%	0%	0%	0%	0%
Simulate Nodes	83.4%	85.5%	85.6%	97.6%	89.0%	98.0%
Write Results	3.5%	4.6%	4.5%	2.6%	8.6%	1.6%

Table 4.2: Profiling results for the DSS algorithm.

Once again, there is a well identified section of the code where most of the execution time is spent (see table 4.2). Thus, in a first parallelization approach, the node simulation procedure is where a reasonable amount of speedup can be obtained, by taking advantage of heterogeneous computing. Considering that the simulation of a single node is, on average, performed in approximately $25,5\mu s$, it is not worth going into a finer analysis in terms of identification of the part of the code to be parallelized, because the parallelization process of finer sections of the code would easily become memory and I/O bounded, due to the data transfers inherent to the algorithm that are required between different devices.

The next step in the parallelization process is to find out if there exists data dependencies between the simulation of different nodes, in order to support the decision about how the algorithm should be partitioned. As it was described in section 2.2, the DSS algorithm is a sequential stochastic simulation method, and thus, the simulated value of a given node is conditioned by all the previously simulated nodes in the random path, as well as by the nodes already assigned to real data. In particular, the simulation of a given node can be divided into the following five steps:

I - Selection of conditional data for the current node (locations only);

II - Construction of kriging system and respective solution;

III - Estimation of the local conditional distribution function;

IV - Generation of the simulated value given the local conditional distribution function;

V - Update of the information relative to the already simulated nodes.

As represented in figure 4.2, the estimation of the local conditional distribution function (step III) requires the values of the previously simulated nodes (step IV). Thus, although step I and II can be fully performed in parallel, the algorithm execution path is limited by the sequential random path through every node, composed by the steps III, IV and V. Moreover, as it can be verified in table 4.3, a significant part of the algorithm execution time is spent in the last three steps of the simulation procedure ($\simeq 80\%$), which means that with the parallelization of the first two steps a significant speedup is not expected to be achieved.

	101x101x90			237x197x350		
Part	den. [ns]	Vp [ns]	Vs [ns]	den.[ns]	Vp [ns]	Vs [ns]
I	855	863	864	1596	1644	1622
II	3314	3384	3384	3301	3297	3324
III	171	172	172	164	169	170
IV	6248	16108	17504	19851	22391	63183
V	66	132	199	66	129	194

Table 4.3: Discriminated average execution time per node for each part of the simulation process.

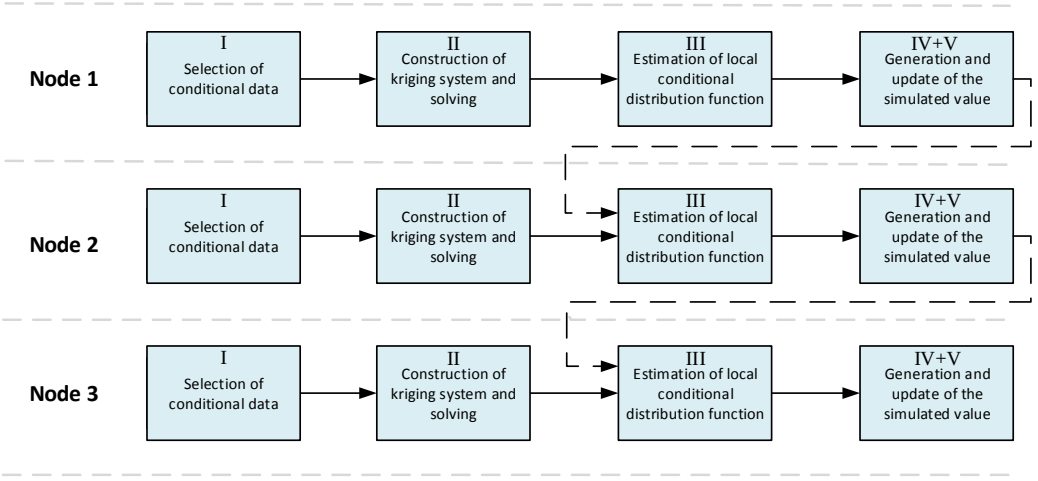


Figure 4.2: Data dependencies for an example with three nodes.

4.2 Proposed Parallelization Approach

As discussed in the previous section, the existing data dependencies between the simulation nodes represent a significant limitation to the speedup that can be obtained with a parallel implementation of the algorithm. Therefore, to circumvent this limitation, the proposed approach is based on a relaxation of this problem by allowing multiple nodes to be completely simulated in parallel. Nevertheless, since this approach implies the removal of some data dependencies, the choice of the nodes to be simulated in

parallel must be performed with some criteria, in order to still ensure the reproduction of the spatial distributions and inherent uncertainties of the physical properties being simulated. Since the kriging estimate is computed by only considering the available data in the neighbourhood of the node being simulated, a spatial division of the simulation grid will be performed. According to the proposed approach, the simulation procedure will be performed at the same time for one node from each sub-grid obtained from the division of the original simulation grid (see figure 4.3). At each step of the simulation process, the whole set of values being simulated (one from each sub-grid) is updated at once, conditioning the subsequent nodes to be simulated. Thus, before starting the simulation procedure, a random sub-path has to be defined through the nodes of a sub-grid. Every sub-grid selects nodes through the same relative sub-path, thus granting a constant distance between the nodes being simulated in parallel in different cells of the simulation grid.

Besides this data partitioning scheme based on the defined sub-grid, the anisotropy of the seismic data can also be taken into account. In fact, due to the nature of the seismic data being processed, there are significantly less dependencies in the vertical direction, which allows for a greater vertical division of the simulation grid. Hence, the main purpose of this complementary parallelization approach is to take advantage of the highly parallel architecture of the GPUs, in order to increase the number of nodes that are simulated at the same time.

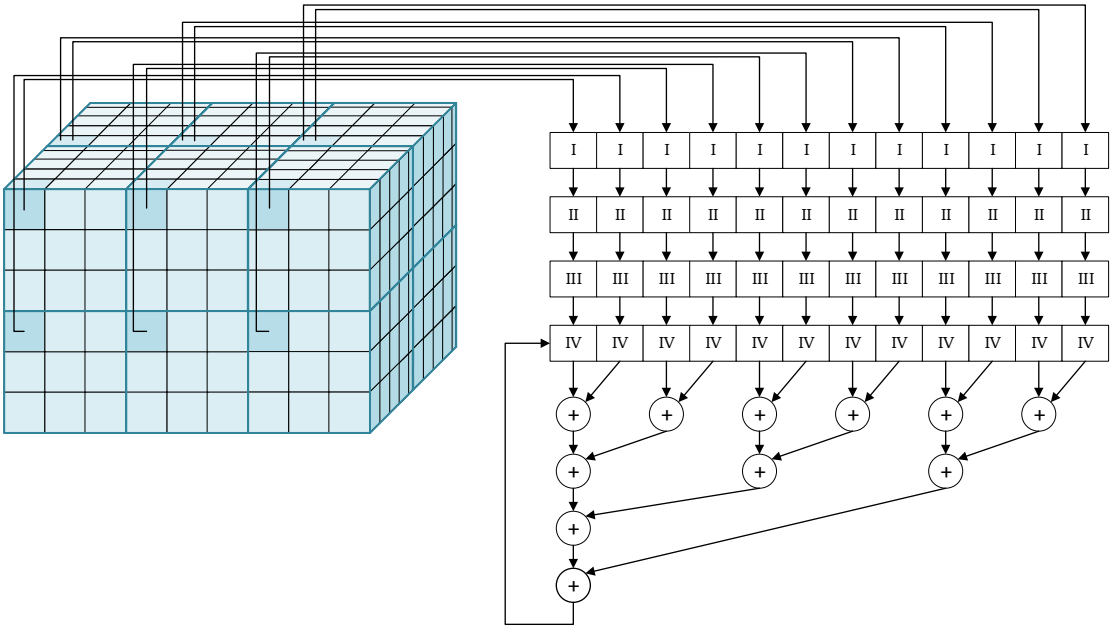


Figure 4.3: Spatial division of the original grid.

Considering the GPU architecture and the OpenCL programming framework discussed in chapter 3, there are at least two different approaches to map the problem, in order to try to maximize the occupancy of the GPU: i) consider that each sub-grid is to be simulated by a distinct OpenCL work-group, being the inherent parallelism of the algorithm explored by the OpenCL work-items within each work-group; ii) consider that each sub-grid is to be simulated by a distinct work-item. The main difference regarding both approaches is the number of nodes being simulated at the same time, since in the former approach

more resources are being assigned to the simulation of a single node, thus limiting the number of nodes that can actually be simulated at the same time. However, those resources are only being completely used when the code itself is parallelizable by a multiple of the warp/wavefront size, which is not the case for a significant part of the algorithm. On the other hand, regarding the second approach, a bigger amount of nodes can be simulated at the same time. However, by using this approach the problem becomes more memory demanding (intermediate buffers need to be replicated for every node under simulation), and performance losses may occur due to warp divergence and memory coalescence, since each thread is simulating its own node, which may lead to different execution paths. On the other hand, such parallelization approach provide a scalable way to improve the obtained speedup with the utilization of multipe GPU and CPU devices, which would provide more parallel computational capabilities. Hence, the efficient balance of the load between the multiple available devices is a challenge to be solved during the development of this thesis.

Finally, although the algorithm is being relaxed, with this parallelization approach it is not expected to observe a significant accuracy loss in the obtained results. In fact, as long as the nodes being simulated are sufficiently apart from each other, being kept outside of the search range, no data conflicts should be verified. However, this may become a limiting factor for smaller datasets, which provide less opportunities to split the grid in many pieces without avoiding such conflicts.

4.3 Preliminary Experimental Results

In order to test and evaluate the parallelization approaches discussed in the previous section, two different preliminary implementations were developed. The resulting speedup results were evaluated by considering the simulation grid with 237x197x350 nodes, executed in a machine composed by an Intel i7 3820 CPU, a NVIDIA GTX 660 Ti GPU and 16GB of memory RAM. The obtained results are merely indicative of the capabilities of both implementations, since not enough optimization time was dedicated to maximize the speedup of each implementation. In both implementations, the simulation procedure was divided in 7 different kernels with the following functionalities:

1. A search procedure is conducted for the available experimental data close to the node being simulated.
2. The kriging system is built, by considering the locations of the nodes selected in the previous step.
3. The kriging system is solved.
4. The lcdf is computed, producing the kriging estimate.
5. The distribution function is conditioned by the previously simulated values and, optionally, by the auxiliary variable.
6. A simulated value is generated by using a Monte Carlo method.
7. The information relative to the already simulated nodes is updated.

The results presented in tables 4.4 and 4.5 correspond to the average execution time for each implemented parallel kernel, and the average execution time that is spent in the corresponding section of the sequential version of the algorithm, during the density simulation. It must be noted that the execution of a given kernel corresponds to the simultaneous simulation of multiple nodes, which means that for the computation of the resulting speedup, the measured parallel execution time must be normalized taking into account the nodes that actually are being simulated. Furthermore, not every node in the simulation grid requires to be simulated since, before the simulation procedure, some nodes have already been assigned with the values of the real data given as input for the stochastic simulation algorithm. Therefore, the speedup computation must take into account the number of calls to the kernels (*numcalls*) that were required in order to perform the simulation, compared to the number of nodes (*numnodes*) that were required to be simulated (expression 4.6).

$$\psi = \frac{T_s \times numnodes}{T_{GPU} \times numcalls} \tag{4.6}$$

Table 4.4 presents the results corresponding to the implementation of the proposed approach, where the simulation of a single node is performed by a work-group. In this implementation, the grid was divided into 256 equal parts, with the addition of some smaller grids that had to be considered with the nodes that remained from the division. In this case, 16335188 nodes have been simulated, being required 60552 calls to the specified kernels. This implementation did not result in a significant improvement, due to the lack of parallelization opportunities within each kernel, which means that a significant amount of the time is spent with only a single work-item in the work-group effectively being executed. Since the GPU processing cores are simpler than the CPU ones, it is expected a slower execution when running sequential code in the GPU. Nevertheless, by executing the whole inversion algorithm, performing a single iteration composed by only one set of simulations (density, Vp and Vs), a speedup of 1,50 was obtained.

Part	Serial Execution Time	Parallel Execution Time	Speedup
1	1596 ns	432618 ns	0,99
2	2639 ns	130395 ns	5,46
3	662 ns	339389 ns	0,52
4	164 ns	55390 ns	0,79
5	79 ns	20427 ns	1,04
6	19772 ns	2166131 ns	2,46
7	66 ns	356936 ns	0,05
Full Simulation	415,4 s	213,7 s	1,94

Table 4.4: Obtained results for the density simulation execution by using a 237x197x350 dataset and by considering one work-group per node

In what concerns the alternative mapping approach, where each work-item is responsible for the simulation of a different node, the results were more encouraging (table 4.5). In this case, the simulation grid was divided in 3456 equal sub-grids, plus the remainder ones. Therefore, only 4654 calls to the kernels were required, in order to simulate the same 16335188 nodes. The obtained results reveal that some specific parts of the algorithm are actually not being accelerated, which means that this implementation

still must be analysed and optimized. Additionally, in the simulation of Vp and Vs, lower speedups were actually verified, mainly because in those cases the DSS with joint probability distributions algorithm is used, which introduces some additional computations that must be further optimized. In what regards the execution of the whole algorithm and considering a single iteration composed by only one set of simulations, a speedup of 3.19 was obtained.

Part	Serial Time Spent (avg)	Parallel Time Spent (avg)	Speedup
1	1596 ns	3021216 ns	1,85
2	3301 ns	1010185 ns	9,17
3	164 ns	3590327 ns	0,65
4	19917 ns	147161 ns	3,91
5	19917 ns	17944 ns	15,45
6	19917 ns	5065357 ns	13,70
7	19917 ns	33599 ns	6,89
Full Simulation	415 s	61 s	6,8

Table 4.5: Results for density simulation using the 237x197x350 dataset considering one work-item per node simulation

Taking into account the obtained results, further improvements are expected to be achieved by re-organizing and further optimizing the implemented kernels, taking into account the concepts of warp divergence and memory coalescence presented in section 3.2. Another optimization that must be taken into account is to balance the workload between both the GPU and the CPU processing cores, in order to fully exploit the processing power available. In both the developed preliminary implementations, during the simulation procedure, the whole CPU is only being used to schedule the kernels executed by the GPU, and to manage the data required by those kernels, which is not computationally demanding. This problem can be solved by using efficient static and dynamic task scheduling algorithms that promote the load balancing. Finally, even further improvement can be achieved by developing an implementation that supports multiple CPU and GPU devices, efficiently dividing the workload between every device available. After fully optimizing the simulation procedure, the whole algorithm must be profiled once again, in order to determine if most of the time is still being spent in the simulation procedure, or if a significant speedup can be achieved by accelerating another parts of the algorithm.

Chapter 5

Conclusions

In this work, heterogeneous computing is being used to accelerate the execution of a stochastic seismic inversion algorithm. The parallelization of such algorithms is important not only to allow for faster reservoir modelling, but also to make it possible to develop larger and more accurate computational models of the Earth's subsurface.

Within the scope of the thesis that is being developed, two different parallelization approaches were proposed, and preliminary implementations were developed and benchmarked, by taking into consideration those approaches. Both parallelization approaches, presented in section 4.2, consider a spatial division of the simulation grid, thus allowing for the parallel simulation of nodes corresponding to different regions of the model. Such division comes with no loss of accuracy in the results, as long as the nodes being simulated in parallel are spaced enough to prevent data dependencies.

Despite not being fully optimized, the developed implementations already revealed some performance improvements when compared to an optimized sequential implementation of the algorithm. Nevertheless, those implementations still require to be further optimized, in order to achieve speedups similar to those obtained in the literature, regarding the usage of GPUs in parallel implementations of similar algorithms.

5.1 Work Planning

In order to improve the preliminary implementations, the following objectives were defined:

- Further optimization of the implemented kernels, by taking into account the preliminary benchmarking results and the concepts of warp divergence and memory coalescence;
- Avoid wasting available processing power, by balancing the workload between the CPU and the GPU. For this purpose, efficient static and dynamic scheduling algorithms should be used to promote the load balancing;
- Take advantage of more than one GPU or CPU devices, when available;
- Parallelization of other parts of the algorithm, if deemed worthwhile.

Taking into consideration both the already developed and the aforementioned objectives, the Gantt chart illustrated in figure 5.1 presents the schedule to be followed during the development of the thesis.

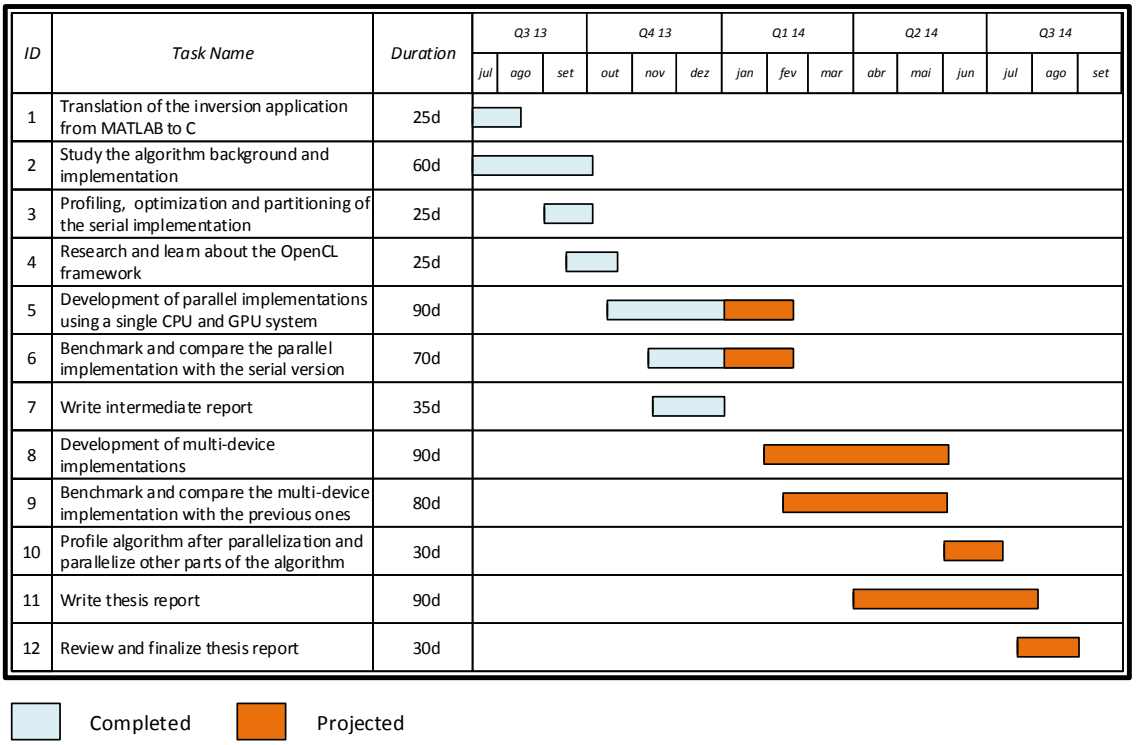


Figure 5.1: Gantt chart considering the already developed and the future work.

Bibliography

[1] L Azevedo, R Nunes, A Soares, and GS Neto. Stochastic seismic avo inversion. In *75th EAGE Conference & Exhibition incorporating SPE EUROPEC 2013*, 2013.

[2] Amilcar Soares. Direct sequential simulation and cosimulation. *Mathematical Geology*, 33(8):911–926, 2001.

[3] Rached Abdelkhalek, Henri Calandra, Olivier Coulaud, Jean Roman, and Guillaume Latu. Fast seismic modeling and reverse time migration on a gpu cluster. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 36–43. IEEE, 2009.

[4] Jairo Panetta, Thiago Teixeira, Paulo RP de Souza Filho, Carlos A da Cunha Finho, David Sotelo, F da Motta, Silvio Sinedino Pinheiro, I Pedrosa, Andre L Romanelli Rosa, Luiz R Monnerat, et al. Accelerating kirchhoff migration by cpu and gpu cooperation. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 26–32. IEEE, 2009.

[5] Bernard Deschizeaux and Jean-Yves Blanc. Imaging earth's subsurface using cuda. *GPU Gems*, 3:831–850, 2007.

[6] Eike Müller, Xu Guo, Robert Scheichl, and Sinan Shi. Matrix-free gpu implementation of a preconditioned conjugate gradient solver for anisotropic elliptic pdes. *arXiv preprint arXiv:1302.7193*, 2013.

[7] Stuart DC Walsh, Martin O Saar, Peter Bailey, and David J Lilja. Accelerating geoscience and engineering system simulations on graphics hardware. *Computers & Geosciences*, 35(12), 2009.

[8] Bent Ove Stinessen. *Profiling, Optimization and Parallelization of a Seismic Inversion Code*. PhD thesis, Norwegian University of Science and Technology, 2011.

[9] Andreas Dreyer Hysing. *Parallel Seismic Inversion for Shared Memory Systems*. PhD thesis, Norwegian University of Science and Technology, 2010.

[10] Ruben Nunes and José A Almeida. Parallelization of sequential gaussian, indicator and direct simulation algorithms. *Computers & Geosciences*, 36(8):1042–1052, 2010.

[11] Hs Vargas, H Caetano, and M Filipe. Parallelization of sequential simulation procedures. In *EAGE Petroleum Geostatistics*, 2007.

[12] Pejman Tahmasebi, Muhammad Sahimi, Grégoire Mariethoz, and Ardeshir Hezarkhani. Accelerating geostatistical simulations using graphics processing units (gpu). *Computers & Geosciences*, 46:51–59, 2012.

[13] Hugo Manuel Vieira Caetano. *Integration of seismic information in reservoir models: Global Stochastic Inversion*. PhD thesis, Universidade Técnica de Lisboa, 2009.

[14] Andrae Haas and Olivier Dubrule. Geostatistical inversion-a sequential method of stochastic reservoir modelling constrined by seismic data. *First break*, 12(11), 1994.

[15] Amílcar Soares, JD Diet, and L Guerreiro. Stochastic inversion with a global perturbation method. In *EAGE Petroleum Geostatistics*, 2007.

[16] Ana Horta and Amílcar Soares. Direct sequential co-simulation with joint probability distributions. *Mathematical Geosciences*, 42(3):269–292, 2010.

[17] RT Shuey. A simplification of the zoeppritz equations. *Geophysics*, 50(4):609–614, 1985.

[18] NVIDIA Corporation. *OpenCL Programming for the CUDA Architecture*, June 2009.

[19] AMD Advanced Micro Devices Inc. *Southern Islands Series Instruction Set Architecture Reference Guide*. AMD - Advanced Micro Devices Inc., 2012.

[20] AMD Advanced Micro Devices Inc. *AMD Accelerated Parallel Processing OpenCL Programming Guide*. AMD - Advanced Micro Devices Inc., 2013.

[21] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`. Accessed: December 2013.

[22] Innovative Computing Laboratory. Performance application programming interface. `http://icl.cs.utk.edu/papi/index.html`. Accessed: December 2013.

[23] JY Xu. Opencl-the open standard for parallel programming of heterogeneous systems. 2008.

[24] Manish Arora. The architecture and evolution of cpu-gpu systems for general purpose computing. *By University of California, San Diago*, 2012.

[25] NVIDIA Corporation. *OpenCL Programming Guide for the CUDA Architecture*. NVIDIA Corporation, 2009.

[26] NVIDIA Corporation. *NVIDIA GeForce GTX 680*. NVIDIA Corporation, 2012.