Accelerating a Stochastic Seismic Inversion Algorithm using OpenCL-based Heterogeneous Platforms

Tomás Novais Ferreirinha

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doctor Nuno Filipe Valentim Roma
Doctor Pedro Filipe Zeferino Tomás

Examination Committee
Chairperson: Doctor Nuno Cavaco Gomes Horta
Supervisor: Doctor Nuno Filipe Valentim Roma
Member of the Committee: Doctor Gabriel Falcão Paiva Fernandes

October 2014
Acknowledgments

First and foremost, I would like to express my sincere gratitude to the two people to whom I am the most indebted. They are the two professors who supervised the development of this master thesis, Prof. Nuno Roma and Prof. Pedro Tomás. Most specifically, I would like to thank for their immense help, guidance, encouragement and stimulating suggestions since the beginning of this research project. They always provided a motivating and enthusiastic atmosphere to work with, and therefore it is worth mentioning that it was a great pleasure to do this thesis under their supervision. I am also thankful to my external supervisor Frederico Pratas, for his guidance, generous contribution of knowledge, experience and valuable comments.

I also would like to thank Leonardo Azevedo, Rúben Nunes and Prof. Amílcar Soares from group CERENA of the department of Civil Engineering, Architecture and Georesources of the Instituto Superior Técnico, not only by providing all the required tools, information and data that were essential for the development of this project, but also by their unlimited availability to help me understanding the application and its theoretical background, as well as to validate the work that I developed.

I also wish to thank all my friends that helped me one way or another, and specially my co-workers and great friends Sebastião Miranda and David Nogueira, which not only supported me in the hardest times, but also exchanged their interesting ideas and thoughts, thus being of a great importance for the completion of this thesis.

Finally, my special thanks goes to my family and girlfriend for their unconditional love and support, always inspiring and encouraging me to go my own way, and consequently without whom I would be unable to achieve this stage of my work and my life. This academic achievement belongs as much to them as it does to me.
Abstract

Seismic inversion algorithms have been playing a key role in the characterization of oil and gas reservoirs, where the retrieved subsurface elastic models need to be reliable to improve decision making, for example decisions about 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 often limited by their long execution times. As a result, the demand for an efficient and accelerated execution of these algorithms is not only required to decrease the time to decision, but also to allow the development of larger and higher resolution models of the subsurface. In accordance, this thesis proposes a novel parallelization approach of a state of the art Stochastic Seismic Amplitude versus Offset Inversion algorithm, exploiting heterogeneous computing platforms based on a unified OpenCL programming framework, thus enabling the application to be executed in devices with different architectures and from different vendors. To take full advantage of the computational power made available by systems composed of multiple (and possibly different) 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, a performance speed-up as high as 27.65\times using two distinct GPUs was achieved, without compromising the accuracy of the obtained models.

Keywords: Heterogeneous Computing, Graphics Processing Unit, OpenCL, Seismic Inversion, Parallel Computing.
Resumo

Os algoritmos de inversão sísmica têm vindo a desempenhar um papel fundamental na caracterização de reservatórios de petróleo e gás, sendo que os modelos elásticos do subsolo por estes gerados necessitam de ser fidedignos por forma a melhorar a qualidade das decisões realizadas nesta indústria, como por exemplo decisões relativas à localização dos respetivos poços, procurando com isso maximizar a extração destes recursos. Sendo que estes algoritmos se baseiam em simulações por computador que geram, processam e guardam grandes quantidades de informação, a sua utilização pela indústria encontra-se limitada pelos seus elevados tempos de execução. Assim sendo, procura-se acelerar este tipo de algoritmos não só para permitir uma execução mais rápida, mas também para o desenvolvimento de modelos maiores e mais precisos do subsolo. Como tal, nesta tese é proposta uma nova abordagem de paralelização de um algoritmo estocástico de inversão sísmica de amplitude versus deslocamento do estado da arte, utilizando computação heterogénea baseada na interface unificada de programação OpenCL, o que permite que a aplicação seja executada em dispositivos com arquiteturas distintas ou de fabricantes diferentes. Por forma a tirar partido do poder computacional disponibilizado por sistemas compostos por vários (e possivelmente diferentes) CPUs e GPUs, foi realizada uma divisão espacial da região a ser simulada, permitindo que sejam simuladas em paralelo diversas regiões do modelo geológico. Com esta abordagem, uma melhoria no tempo de execução do algoritmo de 27.65× foi verificada ao utilizar duas GPUs distintas, sem que tenha sido comprometida a precisão dos modelos obtidos.

Palavras-chave: Computação Heterogénea, Unidade de Processamento Gráfico, OpenCL, Inversão Sísmica, Computação Paralela.
## CONTENTS

1 Introduction .......................... 1  
  1.1 Context ................................................. 2  
  1.2 Objectives ................................. 2  
  1.3 Contributions .......................... 3  
  1.4 Document Outline ..................... 4  

2 Parallel Programming Tools and Architectures .......... 5  
  2.1 Framework Decision ...................... 5  
  2.1.1 OpenCL Framework Description .......... 6  
  2.2 CPU Programming and Architecture Description ....... 8  
  2.3 GPU Programming and Architecture Description ...... 11  
  2.4 Summary .............................................. 15  

3 Seismic Inversion .................................... 17  
  3.1 Theoretical Background ................... 17  
  3.2 Stochastic Seismic Amplitude vs Offset Inversion ...... 20  
  3.2.1 Direct Sequential Simulation and Co-simulation .... 22  
  3.3 Acceleration of Stochastic Seismic Inversion Algorithms . . 25  
  3.4 Summary .............................................. 27  

4 Algorithm Parallelization .......................... 29  
  4.1 Problem Analysis ......................... 29  
  4.1.1 Algorithm Profiling ...................... 30  
  4.1.2 Algorithm Dependencies .................... 32  
  4.1.3 Parallelism in Stochastic Seismic Inversion Algorithms . . 33  
  4.1.4 Discussion ........................................ 36  
  4.2 Parallelization Approach .................. 37
4.2.1 Generation of the Random Path Criteria ........................................... 38
4.2.2 Load Balancing Mechanisms for Heterogeneous Platforms .................. 39
4.3 Summary ......................................................................................... 41

5 Exploiting GPU-based Heterogeneous Platforms ........................................ 43
  5.1 Single-Device Parallel Solution ......................................................... 43
     5.1.1 Mapping Methodology ............................................................. 45
  5.2 Multi-Device Single-Realization Parallel Solution ................................. 47
  5.3 Multi-Device Multi-Realization Parallel Solution .................................. 50
  5.4 Considered Optimizations .................................................................. 52
     5.4.1 Generic Optimizations ............................................................. 52
     5.4.2 Application Specific Optimizations .......................................... 54
     5.4.3 Other Algorithm Optimizations ................................................. 56
  5.5 Summary ......................................................................................... 57

6 Experimental Results ........................................................................... 59
  6.1 Experimental Setup ........................................................................... 59
  6.2 Performance Evaluation ......
LIST OF TABLES

4.1 Discriminated average execution time per node for each part of the simulation procedure. ........................................ 32
4.2 Performance results regarding the number of memory accesses and the number of floating-point operations during a single execution of the simulation procedure, by considering the dataset with 237x197x350 nodes. ........................................ 36
5.1 Obtained preliminary speed-up results for both mapping methodologies, considering a single density simulation execution over a dataset with 237x197x350 nodes in a NVIDIA GTX 660 Ti GPU. Speed-up results use as baseline the sequential execution time of an Intel i7-3820 CPU. ........................................ 47
6.1 Considered systems specification details. ........................................ 60
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>OpenCL platform model.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>NDRange organization.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Block diagram of a CPU.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Architecture of a 4-core Intel processor of the Sandy Bridge, Ivy Bridge or Haswell microarchitectures.</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Relative occupation of resources comparing CPU and GPU architectures.</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Modern NVIDIA GPU architecture.</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Streaming Multiprocessor architecture.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Simplified inversion workflow.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustration of marine seismic data acquisition.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Construction of the mismatch model.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Stochastic Seismic AVO Inversion algorithm flowchart.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>DSS algorithm flowchart.</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Selection of values from $Z_2(x)$, used in order to condition the simulation by a previously simulated variable.</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Profiling results for the Stochastic Seismic AVO Inversion algorithm, considering both the experimental datasets.</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Profiling results for the DSS algorithm, considering both the experimental datasets.</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Conditioning data locations during the simulation of a given node, considering k=6.</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Data dependencies on the stochastic simulation of the $\rho$, Vp and Vs models.</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Different granularities of parallelism that can be considered in Stochastic Seismic Inversion algorithms.</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>Functional parallelization of the DSS algorithm, by considering 3 parallel threads.</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>Roofline model of the sequential execution of the DSS algorithm.</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Spatial division of the original simulation grid in multiple sub-grids.</td>
<td>38</td>
</tr>
</tbody>
</table>
# LIST OF ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic and Logic Unit</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SP</td>
<td>Stream Processor</td>
</tr>
<tr>
<td>SFU</td>
<td>Special Function Unit</td>
</tr>
<tr>
<td>CU</td>
<td>Compute Unit</td>
</tr>
<tr>
<td>LDS</td>
<td>Local Data Share</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>NDRange</td>
<td>N-Dimensional Range space</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>PAPI</td>
<td>Performance Application Programming Interface</td>
</tr>
<tr>
<td>AVO</td>
<td>Amplitude Versus Offset</td>
</tr>
<tr>
<td>DSS</td>
<td>Direct Sequential Simulation</td>
</tr>
<tr>
<td>LCDF</td>
<td>Local Cumulative Distribution Function</td>
</tr>
</tbody>
</table>
LIST OF SYMBOLS

\( \rho \) Density

\( V_p \) Primary-wave (P-wave) velocity

\( V_s \) Secondary-wave (S-wave) velocity

\( N_s \) Number of realizations per algorithm iteration

\( \psi \) Speed-up

\( p \) Number of sub-grids that result from the spatial division of the simulation grid

\( N \) Number of nodes in the simulation grid
CHAPTER 1

INTRODUCTION

In the last few years, scientific and engineering applications have been demanding more and more computing power in order to solve larger scale problems in a reasonable amount of time, especially in cases where the technical problems rely in computer simulations that generate, process and store huge amounts of data. In many cases, the execution of such applications leads to long processing times, significantly limiting the use of algorithms with higher accuracies. To decrease the computation time, many solutions have been devised, including the development of specialized processing structures and the exploitation of modern computing devices both at the level of a single computing platform or at the level of a cluster or grid. In either case, to efficiently take advantage of the several computing devices on a single heterogeneous platform composed of one or more multi-core GPPs and multiple accelerators (e.g., NVIDIA/AMD GPUs, Intel Xeon phi or custom-made accelerators), efficient techniques must be employed such as to minimize computing overheads, hide memory latencies and distribute the computation across multiple heterogeneous processors. When such a care is taken, significant performance boosts can be achieved even at the level of a single machine.

Nowadays, High Performance Computing (HPC) platforms have been playing a key role in the Oil and Gas prospecting industry in order to satisfy the ever growing needs for oil and gas extraction. As a result of the increased computing capabilities that have become available to this prospecting industry, complex computational models of the Earth subsurface can now be applied to locate and estimate reserves and to diagnose and improve the performance of the producing fields. Among the multiple existing application domains (flow simulation, rock mechanics, etc), stochastic 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. This scarceness of data is mainly justified by the huge costs associated to deep water drilling (with depths greater than 500 meters, costing up to 1 million dollars per day and on average 100 million dollars per drill). The lack of available data is compensated by the usage of complex geological interpretations, together with several approximations in the computational models.
used to simulate the oil reservoirs. However, typical approximations result in models with high levels of uncertainty, leading to faulty understanding of the geological structure and consequently drilling errors. Such errors can become rather expensive, not only in terms of investment costs, but also in terms of the environmental impact.

Due to the huge amounts of data that is generated, processed and stored, the aforesaid stochastic algorithms are often characterized by huge execution times (up to months), limiting their usefulness in most cases. Nevertheless, as the computing power is growing, HPC solutions provide an opportunity to apply such algorithms in real-life situations, and even to increase the accuracy of the physical approximations used in reservoir modelling. The recently proposed Stochastic Seismic Amplitude Versus Offset (AVO) Inversion algorithm [1] using the Direct Sequential Simulation (DSS) [2] approach represents a promising methodology to solve geophysical inversion problems, improving the generated models at the cost of a significantly more complex processing of the gathered data. Namely, in order to incorporate seismic reflection data in inversion frameworks, multiple weak signals are usually stacked into a single (and stronger) one, thus making its processing less complex. However, the algorithm in study makes use of pre-stack seismic reflection data, consequently improving the quality of the generated seismic models. However, very few parallel implementations of such algorithms have been proposed, presenting several limitations in terms of scalability.

1.1 Context

This thesis arises from a cooperation between the groups CERENA from the Department of Civil Engineering, Architecture and Georesources of the Instituto Superior Técnico, and the group SiPS, which is part of the INESC-ID research facility and is connected with the department of Electrical and Computer Engineering from the Instituto Superior Técnico. The members of group CERENA were responsible for the development of the Stochastic Seismic AVO Inversion algorithm herein being accelerated, as well as for an initial implementation which was divided in two distinct parts: the inversion framework implemented in MATLAB; the stochastic simulation algorithm implemented in C/C++. However, such implementation presented high computational requirements, taking a long time to perform a sequential execution when considering datasets with realistic dimensions. In addition, the CERENA group also provided the datasets which were considered during the development of this thesis to evaluate the performance of the algorithm. The work presented herein was also partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects Threads (ref. PTDC/EEA-ELC/117329/2010) and P2HCS (ref. PTDC/EEI-ELC/3152/2012).

1.2 Objectives

The main goal of this thesis is the conception of a parallel and highly efficient strategy for accelerating a highly relevant application used for prospecting, extracting and monitoring energy sources like oil and gas. Accordingly, the following list of objectives are defined:
• Comprehensive study of the algorithm work-flow, supported by the corresponding theoretical background;

• Identification of the different parallelization opportunities such as to accelerate the Stochastic Seismic AVO Inversion algorithm;

• Selection of the target architectures, as well as the programming framework;

• Functional-level partitioning of the algorithm structure into several independent functional modules;

• Data-level partitioning of the most data intensive modules and corresponding mapping into the considered architectures;

• Definition of the appropriate scheduling, synchronization and load balancing policies;

• Implementation of the defined parallelization model in state of the art high performance computing platforms, composed by a multi-core Central Processing Unit (CPU) and one or more Graphics Processing Units (GPUs);

1.3 Contributions

After a comprehensive profiling and analysis of the Stochastic Seismic AVO Inversion algorithm, in order to quantify its requirements in terms of computational resources, dependencies and memory usage, it was verified that the most significant part of the algorithm under study corresponds to the stochastic simulation procedure, which is composed by millions of dependent iterations that individually are not computationally demanding. Although the parallelization of geostatistical simulation methods have already been studied, such solutions present several scalability limitations, since centralized processing mechanisms are often considered, and consequently the inherent required communications tend to rapidly become a performance bottleneck as the number of parallel threads significantly increases. Accordingly, taking into account the lack of data parallelism opportunities presented by the algorithm, in this thesis a fully distributed simulation approach is proposed by considering a relaxation of the algorithm in order to efficiently exploit the highly parallel GPU architectures. Additionally, the herein proposed multi-device implementations of the algorithm guarantee an efficient execution over heterogeneous platforms by using appropriate load-balancing techniques.

Such HPC techniques applied to seismic inversion applications allow not only for faster reservoir modelling, but also make it possible to develop larger and higher resolution computational geology models and to improve the physical approximations used in such algorithms, in a way that is still computationally unfeasible at this moment using conventional platforms. In accordance, the work herein developed provides significant performance improvements to the currently used reservoir simulation approaches. In fact, when compared with the original sequential execution, the developed parallel implementation of the Stochastic Seismic AVO Inversion algorithm provided reductions in the execution time of 97% when using two distinct GPUs.
The scientific contribution of the developed work had already been recognized by the scientific community, having the work lead to the communication of the following articles:


### 1.4 Document Outline

In order to cover the aforementioned objectives, this document was divided as follows. After this introductory chapter, the considered parallel programming framework as well as the target computational architectures to be used in order to accelerate the algorithm are described in Chapter 2. In the subsequent chapter, a brief contextualization to the topics being covered, as well as a detailed description of the algorithm under study are presented. In addition, the related work already developed on accelerating seismic inversion algorithms is also covered in Chapter 3. In Chapter 4 the algorithm is carefully profiled and analysed both in terms of the data dependencies and in terms of the types of parallelism that can be exploited, in order to support the decision of the parallelization approach, which is also comprehensively described in the same chapter. Chapter 5 covers the multiple considered implementations of the parallelization approach into the considered single-device and multi-device heterogeneous platforms, and corresponding optimizations. An experimental evaluation of the developed implementations both in terms of the attained performance and in terms of the quality of the inversion results is presented in Chapter 6. Finally, Chapter 7 concludes the paper, summarizing the main conclusions and contributions, and outlining the future work.
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 GPUs, are already endowed with a heterogeneous architecture. Portable and efficient developing frameworks are thus required in order to map applications into this heterogeneous hardware. Accordingly, this thesis targets the exploitation of systems composed by multiple and possibly different CPUs and GPUs to accelerate a stochastic seismic inversion application. The programmability of such devices is not straightforward, and consequently a decision has to be made regarding the programming framework to be used, and a detailed study of the architectures of such devices has to be conducted, in order to support an efficient partitioning and mapping of the algorithm.

2.1 Framework Decision

Nowadays, when considering the parallelization as a technique to significantly accelerate a given application, most implementations consider either the use of clusters of devices (usually composed by multiple multi-core CPUs) in order to independently execute multiple threads, or to use highly parallel architectures with efficient support for fine grained parallelism and with a greater proximity between the processing cores, facilitating inter-thread communications (e.g., modern GPUs). A commonly used cluster-driven methodology is the use of the so called MPI, which considers a message passing paradigm, where the execution is divided among the multiple available processing cores (from the same or different machines) by considering a distributed memory paradigm. Such memory has to be man-
aged by the programmer by passing messages between the multiple processing cores. Although being highly scalable, this mechanism is mainly limited by the overhead of communicating data among every processing core (sometimes being required communications between two distant nodes of the cluster), thus being more appropriate for a coarse-grained parallelism. On the other hand, GPU programming enables a more fine-grained parallelism, since multiple parallel cores are made available in a single architecture, sharing the same memory context.

Taking into account that this thesis considers the use of systems composed both by CPUs and GPUs, and since the most computational demanding part of the algorithm is composed by millions of dependent iterations that individually are not much time consuming (as further detailed and analysed in Chapter 4), GPU-driven programming frameworks were chosen in order to exploit the parallelism of such architectures, taking advantage of a fine grained parallelism and allowing for a lower inter-thread communication overhead, which otherwise would easily become a significant bottleneck. There are two dominant frameworks for developing GPU-based heterogeneous computing: NVIDIA’s Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). The first one is proprietary, enabling the programmers to use modern NVIDIA GPUs for general purpose programming. On the other hand, the OpenCL framework is not restricted to a given vendor of GPUs neither to GPU architectures, offering more code portability and flexibility when compared to the CUDA programming language. However, although correctness is guaranteed, the OpenCL code will not necessarily run optimally on all of the distinct architectures. Moreover, the developed code is not necessarily going to be portable between every device, since some OpenCL features are optional and may not be supported on all devices. Therefore, it is up to the programmer to decide how efficient and how portable its implementation is going to be.

Nevertheless, OpenCL remains as a more appropriate choice since it has a wider support for GPUs of different vendors. Accordingly, in this thesis the OpenCL framework was chosen in order to take full advantage of the computational power made available by systems composed by multiple (and possibly different) CPUs and GPUs, without imposing restrictions in terms of manufacturers.

2.1.1 OpenCL Framework Description

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 [3]. 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) [4]. 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 consists of an API for coordinating parallel computations across the multiple available devices, and a cross-platform programming language (based on C99) with a well-specified computation environment. The OpenCL specification is based in four hierarchical models:

- **Platform model**: Specifies the execution coordinating processor (the host), which is connected to one or more compute devices capable of executing the OpenCL cross-platform code.

- **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 code. As illustrated in Figure 2.1, this model defines a host connected to one or more OpenCL devices. An OpenCL device is composed by 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.

![Figure 2.1: OpenCL platform model.](image)

Regarding the execution model, an OpenCL program is divided in two parts: kernels, which are syntactically similar to C functions and correspond to the set of instructions that each work-item (similar to the concept of thread) executes in parallel; and a host program, that executes on the host and coordinates the computations being performed among the multiple devices. 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 2.2). 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.

The host program is responsible to define a context for the execution, which includes the collection of OpenCL devices to be used; the kernels to be executed on the OpenCL devices; and the memory objects, that contain values that can be operated by instances of a kernel. The OpenCL devices are
selected among a list of available devices obtained by OpenCL query calls, allowing the user to develop an automatic device selection according to its preferences (without prior knowledge of the available devices).

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 shorter access times. Additionally, each work-item also has a private memory space, which corresponds, in terms of hardware, to internal registers located inside the processing core 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, where each work-group is 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, although most GPU devices implement both.

### 2.2 CPU Programming and Architecture Description

The CPU, is the main component of a digital computer. 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 (see Figure 2.3).

![Block diagram of a CPU.](image)

The performance assessment of CPUs can be performed by using multiple metrics. Some commonly used metrics are: the time it takes to perform the operations, usually measured by operating the processor at the maximum clock frequency; the number of instructions performed per clock cycle, also called the instruction throughput; or the amount of floating-point instructions that the processor is able to perform per second. However, when comparing multiple processors, the performance is often assessed by comparing the execution times of a given set of applications considered representative for the objective of the processors being evaluated. The metrics used for 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 complexity. Such techniques usually aim to achieve either a higher operating frequency or a higher throughput or even lower average memory access time. In what concerns the minimization of the memory access time, CPU architectures make use of a hierarchical memory model. This scheme integrates several smaller and faster memories, also known as caches, that take advantage of the principles of temporal locality and spatial locality. Therefore, programmers must take into account those principles in order to optimize the applications performance. Despite the offered set of performance benefits, architectural optimizations usually come with an increase of complexity, area, power consumption and consequently cost.

---

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.
Further improvements 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 number 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 be 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.

Modern CPU architectures offer an efficient and complex set of architectural solutions to exploit the different kinds of parallelism (see Figure 2.4). As an example, in addition to the multi-core architecture itself, the processor capabilities were improved with simultaneous multithreading within the same core, being known in the Intel's architectures as Hyper-Threading Technology. CPU data parallelism capabilities have also been improved not only with the introduction of Single Instruction Multiple Data (SIMD) instruction set extensions, but also with the integration of on-chip GPUs, which can be used to accelerate general purpose data-parallel computations.

![Figure 2.4: Architecture of a 4-core Intel processor of the Sandy Bridge, Ivy Bridge or Haswell microarchitectures.](image-url)
2.3 GPU Programming and Architecture Description

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 for real-time applications. In order to solve this, the characteristics of this application led into a highly parallel many-core processor architecture, specifically designed to efficiently act as the CPU co-processor for graphics rendering. 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 2.5.

Originally, the GPU architectures were designed with a fixed function graphics pipeline (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 [5].

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 Processors (SPs) as well as some memory load/store units and some Special Function Units (SFUs), as illustrated in figures 2.6 and 2.7 [6, 7].
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.

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 shared 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 having to execute the same instruction at the same time, each thread may have its own execution path, by using predicative instructions. 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 when compared with the optimal execution. The Kepler GPU microarchitecture slightly minimized this issue by introducing two instruction dispatch units per warp scheduler, thus allowing the issue of two distinct instructions during the same clock cycle.

Another significant problem arises when the accesses to the global memory performed in parallel by the multiple threads of the same half-warp do not obey a specific set of memory access patterns supported by the GPU architecture. 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 fulfill the restrictions imposed by the architecture (e.g., threads accessing words in sequence that lie in the same memory segment) [8]. When this condition is not fulfilled, separate memory transactions are issued for each thread, significantly compromising the throughput. In older GPU devices, optimal memory access patterns were restricted to

\(^3\)Depending in the word size.
a fairly reduced set, in contrast with the most recent microarchitectures (e.g., Kepler microarchitectures), that offer optimal performance to a much more boarder set of address distributions across the threads within a half-warp [9].

Regardless the above mentioned problems, an important metric when analysing the GPU performance is called occupancy. Occupancy can be calculated using expression 2.1, 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.

\[
\text{Occupancy} = \frac{\text{Number of active warps}}{\text{Maximum number of active warps}}
\]  

(2.1)

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 an 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 2.2. This performance limiting factor can be optimized both by reusing the same registers as much as possible and by varying the granularity of the implemented kernels. Nevertheless, there are many other occupancy limiting factors, such as 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. Another limiting factor is concerned with the dependencies between multiple consecutive instructions, namely when the latency of a given instruction can not be hidden with the execution of other non-dependent subsequent instructions. In addition, the number of parallel threads must be a multiple of the warp size, and must be large enough in order to guarantee that the whole device architecture is occupied.

\[
\text{Registers per Thread} \leq \frac{\text{Registers per SM}}{\text{(Max Warps per SM)} \cdot \text{(Threads per Warp)}}
\]  

(2.2)

In what concerns modern AMD GPUs, their architectures are referred to 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 processing 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 [10, 11], 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, loss 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.
2.4 Summary

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 (e.g., GPUs, DSPs), and sometimes even by custom acceleration logic devices (e.g., FPGAs). In the scope of this thesis, heterogeneous systems are going to be used in order to accelerate an important algorithm for seismic inversion. Such a flexible solution is achieved by using the OpenCL API, allowing each part of the algorithm to be easily migrated among the several coexisting devices. Therefore, this chapter starts by presenting a comprehensive description of the OpenCL parallelization framework, being detailed its programming paradigm. In addition, a brief description of the target architectures is conducted (CPUs and GPUs), being highlighted the main programmability concerns, and corresponding benefits and limitations from a parallel programming perspective. Accordingly, CPU architectures are highly efficient when executing both general-purpose sequential and task-parallel applications, in contrast with modern GPUs, which provide a general-purpose highly parallel architecture, specially efficient for performing data-parallel computations, thus complementing the CPU capabilities.
 CHAPTER 3

SEISMIC INVERSION

A good understanding of the algorithm under study is essential to support an efficient parallelization approach. Accordingly, this chapter presents a comprehensive description of the algorithm being accelerated, and provides some insightful information into the theoretical concepts that serve as a background to the accelerated seismic inversion algorithm. In addition, a broad study of the state of the art in accelerating applications used in the Oil&Gas industry is also herein presented, highlighting the main benefits and limitations of the already developed implementations.

3.1 Theoretical Background

The main goal of inversion problems is to make inferences about physical objects or systems, given a limited set of observed measurements (see Figure 3.1). 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 \(^1\). Numerical three-dimensional subsurface Earth models, built from available well-log (obtained by drilling the Earth’s crust) and seismic reflection data, are usually considered essential tools for reliable decision making and risk analysis [12].

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 3.2). Seismic waves can be divided in two different categories: body waves, which travel through 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

\(^1\)Hard and rigid outer layer of the Earth
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.

Seismic reflection occurs at an interface between two materials with a contrast in P-wave velocity ($V_P$), S-wave velocity ($V_S$) or density ($\rho$), which are properties of the material, and can be related to other properties with higher interest for reservoir characterization, such as porosity. 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 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 computational model. Usually, this type of problems are simplified by using a technique known as stacking, where multiple similar weaker signals are combined (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 are devised, which are simple enough to keep them mathematically manageable, or a set of equiprobable solutions are conceived by using stochastic methods. The main disadvantage of the former methodology is that, contrary to the latter, it does not exploit the existing uncertainty in the acquired data. 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

---

2Possibilities that occur above the surface are not considered to be reasonable.
other hand, although being able to manage uncertainties, stochastic methods are usually much more
time-consuming, mainly because the iterative generation of subsurface models is computationally very
demanding.

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 a broad set of data with very different scale support [13]. Namely, a geostatis-
tical (or stochastic) framework allows the integration, within the same model, of well-log data, which is
sparsely located along the study area but has a very high vertical resolution; and seismic reflection data,
which covers a great spatial extent but has a low vertical resolution [14]. Unfortunately, the integration
of the seismic reflection data is not straightforward within the modeling workflow, since it is an indirect
measurement of the subsurface elastic properties of interest (e.g., Vp, Vs and $\rho$). Before this data can be
integrated, the so called seismic inversion problem needs to be solved [14]. However, due to measure-
ment errors and approximations in the physical models that are often used to solve the seismic inverse
problem, this is a ill-posed and highly nonlinear problem with a non-unique solution [14, 15].

The most commonly used methodology to incorporate seismic information in stochastic fine-grid
models is known as geostatistical inversion [16]. This inversion methodology consists in two steps:

1. Simulation of acoustic impedance values (property of the medium, defined by the product of the
   rock density and p-wave velocity) based on well data and spatial continuity pattern;
2. Transformation of the data in order to be able to compare the simulated acoustic impedance values with the real seismic data.

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, or through iterative methods, where a simulation is obtained by successive corrections of an initial image, leading to the minimization of an objective function [16]. In what concerns the second step of the geostatistical inversion framework, the transformation of the data can be performed either by using an inverse method, where the seismic data is transformed into acoustic impedance values by removing the effect of a known estimated wavelet, or by using a forward method, that does the opposite by convolving the acoustic impedances with the estimated wavelet, obtaining synthetic seismic model that can be compared with the real seismic data.

3.2 Stochastic Seismic Amplitude vs Offset Inversion

High quality pre-stack seismic data with high signal-to-noise ratio and with a considerably high fold number is becoming the standard in exploration and characterization projects, mainly due to the exploration of increasingly complex deep water reservoirs. For this reason, there has been an increase on the number of available inverse techniques for pre-stack seismic reflection data (e.g., [17, 18]). At the same time, due to the advantages on uncertainty assessment and data integration, geostatistical inverse procedures are increasing its popularity in seismic reservoir characterization studies. Accordingly, a state-of-the-art geostatistical pre-stack seismic inversion methodology [1, 19] that allows the inversion of pre-stack seismic data directly for density, P-wave and S-wave velocity models is herein adopted. Contrary to most inversion procedures that simply invert the seismic reflection data for acoustic and elastic impedance, this particular inversion methodology is able to individually invert each of the elastic properties of interest. For such purpose, the adopted methodology consists on an iterative geostatistical inversion method, based on the Global Stochastic Inversion approach [20]. This approach is based on two main concepts:

- $\rho$, $V_p$ and $V_s$ models are perturbed towards an objective function, according to a stochastic sequential simulation algorithm (Direct Sequential Simulation (DSS), co-DSS [2] and co-DSS with joint probabilities distribution [21]).

- A genetic algorithm, acting as a global optimizer, that ensures the convergence of the solution from iteration to iteration. This algorithm iteratively selects the best parts of each simulated model according to the mismatch between the real and the synthetic seismic data.

In the considered approach, the transformation required to compare the simulated with the real seismic data is performed by using a forward method. This forward method is computed by using each simulated element is conditioned by the previous simulated elements.

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 Shuey’s approximation [22], leading to a synthetic seismic model of the reservoir for each set of simulated models ($\rho$, $V_p$ and $V_s$). In order to compute the mismatch between the synthetic and real seismic data, a simple co-located correlation coefficient is computed, as shown in Figure 3.3. In this process, the seismic models are vertically divided into layers, and the mismatch for each pair of layer-size columns of both seismic data models is computed. The results are stored in a mismatch model, that provides the information about which locations in the reservoir are fitting the real seismic and which need further improvement. Finally, by taking into account the locations with higher values of correlation, the best models of $\rho$, $V_p$ and $V_s$ are built and used as a secondary image, conditioning the next generation of simulations.

At the end of this iterative and convergent methodology, synthetic pre-stack seismic data can be retrieved, with an accurate match in terms of reflectors’ position and amplitude variations with regards to recorded pre-stack seismic data. In addition, the corresponding density, P-wave and S-wave velocity models can also be retrieved. Therefore, the Stochastic Seismic AVO Inversion algorithm can be summarized in the following modules (see Figure 3.4):

- **Joint simulation of density, P-wave and S-wave velocities**
  1. Stochastic simulation of $N_s$ density models conditioned to the available well-log data, by using the DSS algorithm [2];
  2. Stochastic co-simulation of $N_s$ $V_p$ models given the $N_s$ previously simulated density models, by using the co-DSS with joint-distributions algorithm [21];
  3. Stochastic co-simulation of $N_s$ $V_s$ models given the $N_s$ previously simulated $V_p$ models, by using the co-DSS with joint-distributions algorithm [21];

- **Optimization for the iterative convergence**
  4. From the triplet of elastic models computed in the previous stage, calculation of $N_s$ angle-dependent synthetic pre-stack seismic volumes using Shuey’s linear approximation [22];
5. Comparison between each synthetic angle gather with the corresponding real angle gather on a trace-by-trace basis. Calculation of $N_s$ local correlation gathers for each location within the seismic grid;

6. Selection of the areas of the synthetic seismic models with the highest correlation coefficient in order to build the best $\rho$, Vp and Vs models, by using a genetic-based algorithm;

7. Compute the local correlation model regarding the best generated $\rho$, Vp and Vs models; iterate from 1), by using best models as secondary variables for the co-simulation of the elastic models generated during the next iteration, until the matching criteria (i.e., global correlation between the original and the synthetic pre-stack seismic) is reached.

---

**Figure 3.4:** Stochastic Seismic AVO Inversion algorithm flowchart.

### 3.2.1 Direct Sequential Simulation and Co-simulation

Due to its efficiency and simplicity, sequential simulation methods have been widely used in order to reproduce the spatial distribution and uncertainty of variables of different resources in Earth sciences.
Although being based on the same sequential procedure, the different versions of sequential simulation often require transformations of original variables and use different approaches to estimate local distribution functions.

As aforementioned, the stochastic sequential simulation algorithm adopted by the CERENA group, which provided the original algorithm implementation, is known as Direct Sequential Simulation (DSS). This algorithm allows the calculation of a set of subsurface Earth models, each called a realization, that honour the experimental data (available well-log data), the prior probability distribution (estimated from the well-log data) and a spatial continuity pattern as revealed by a covariance model (e.g., in two-point geostatistics a variogram). Contrary to most of the sequential simulation algorithms, the DSS algorithm does not require any prior or subsequent transformation of the original variables (in this case $\rho$, $V_p$ and $V_s$).

The main objective of the DSS algorithm is to assign a value $z_1(x_i)$ of the variable $Z_1(x)$ to each node of a regular three-dimensional simulation grid. This simulation grid corresponds to a spatial division (in multiple nodes) of the region to be modelled. The number of considered divisions depends on the pretended resolution of the generated models. Since this algorithm is a stochastic sequential method, after reading the experimental data values from an input file and assigning them to the corresponding positions of the simulation grid, the algorithm defines a random path through every node to be simulated. The whole reservoir grid is subsequently visited following this pre-defined random path, being simulated one node at a time. At each simulation step, a simulated value $z_1(x_0)$ is drawn for the selected node $x_0$, considering the original data and the previously simulated values. Since the node visiting path is random, different runs will produce different models and consequently the conditioning data at a specific grid node may differ [12]. Each step of the DSS algorithm (summarized with the flowchart presented in Figure 3.5) is mainly composed by the following processing blocks:

A. Given the next node to be simulated, search for k-nearest neighbouring nodes with available data (experimental or previously simulated values) that will condition the simulation (being the k defined as a simulation parameter). This is achieved by visiting the nodes from a pre-sorted list containing the nearest relative positions according to the non-euclidean distances between nodes (related with the anisotropy of the data);

B. Construction and solution of a kriging system for the selected node, by linear interpolating the previously selected conditioning data (simple kriging estimate and variance) [23];

C. Estimation of a Local Cumulative Distribution Function (LCDF) at the selected node (local mean and variance), by using the previously computed simple kriging estimate that considers both the original and the previously simulated data [2];

D. Draw of a value from the estimated LCDF, by using a Monte Carlo method;

E. Update the simulation grid and repeat from A, until all nodes have been visited by the random path.

The co-simulation variant of this algorithm enables the simulation of a variable $Z_2(x)$ to be conditioned by other previously simulated variable $Z_1(x)$, without any prior transformation of $Z_1(x)$. For such
purpose, before drawing a simulated value (processing block D), the conditional cumulative distribution function \( F[Z_2(x)|Z_1(x) = z_1(x_0)] \) is computed from the bi-distribution \( F[Z_1(x), Z_2(x)] \), by using moving classes composed by \( N_c \) elements of \( Z_2(x) \), being \( N_c \) a random value bounded by a maximum and a minimum values defined as a simulation parameters [21] (see Figure 3.6). This list of selected values from \( Z_2(x) \) is subsequently sorted and used as normal score in the back-transform routine of the Monte-Carlo method. Therefore, with this method not only the marginal histograms of \( Z_1(x) \) and \( Z_2(x) \) are being reproduced, but also the bi-histogram of the experimental values \((z_1(x), z_2(x))\) is respected.

In this case, the co-simulation variant of the DSS algorithm is only used for the simulation of Vp (conditioned by the previously simulated density model) and Vs (conditioned by the previously simulated Vp model). When compared with the other sequential simulation algorithms, such as the Sequential Indicator Simulation (SIS) and the Sequential Gaussian Simulation (SGS) algorithms, this is one of the main advantages of the DSS algorithm.

Altogether, the iterative procedure herein described ensures an accurate reproduction of the individual marginal distributions of each elastic property, as estimated from the well-log data, as well as the joint distributions between density and P-wave velocity and between P-wave and S-wave velocities. In addition, all the simulated and co-simulated models generated during this iterative procedure match the spatial continuity pattern, as revealed by a variogram model.
3.3 Acceleration of Stochastic Seismic Inversion Algorithms

In order to deal with the complexity of large-scale Oil and Gas wells, multiple geoscience applications have been being developed in order to efficiently take advantage of the parallel capabilities of cluster systems, where each node is composed of one or more CPUs and possibly multiple GPUs. Therefore, 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 architecture of the GPUs. For instance, in [24, 25, 26] 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 [27, 28]. Once again, impressive improvements have been verified, with speed-ups greater than $30 \times$.

Despite the close relationship between the geoscience and HPC industries, the actual exploitation of parallel and concurrent computing techniques to accelerate geostatistical inversion and sequential simulation algorithms is not yet widely developed. Although parallel implementations of the Stochastic Seismic AVO Inversion algorithm were not found in the literature, algorithms with the same purposes have already been optimized and parallelized, as is the case of Conditioning Reservoir Variables to
Amplitude vs Angle Data (CRAVA) [29, 30]. In those works, straightforward multi-core parallelization
approaches were implemented, being observed an almost linear speed-up when using up to 16 cores.

In what concerns the stochastic simulation part of the algorithm under study, one parallel implement-
ation of the DSS algorithm has been proposed [31], where a multi-core approach was proposed by
considering a straightforward functional decomposition of the algorithm. Nevertheless, this approach
presents a considerable limitation in terms of the offered scalability with the number of processing units,
being the speed-up results almost linear up to the limit of 4 cores. In fact, this same parallelization ap-
proach was also considered for other stochastic simulation algorithms, namely the Sequential Indicator
Simulation (SGS) and the Sequential Gaussian Simulation (SGS), being obtained similar results. An-
other parallelization approach based on a spatial division of the problem was studied in [32], where it is
shown that, as long as a certain distance is kept between the nodes being simulated in parallel (avoiding
data hazards), the obtained models present the same properties as the ones generated by the original
sequential algorithms. Note that this data hazards occur because the simulation of a given node makes
use of the k-nearest available data values as conditioning data. Therefore, a conflict occurs whenever a
node to be simulated belongs to the search range of another node being simulated in parallel. Such data
conflicts are not easily predictable not only due to the random nature of the algorithm but also because
conditioning nodes are selected from neighbourhoods that are not strictly defined.

Nevertheless, the generic parallelization of stochastic simulation methods have also been stud-
ied [33], where three distinct levels of parallelization were identified: realization level, path level and
node level. In such broader scope, some related stochastic algorithms have already been optimized
and parallelized using multi-core CPUs [33, 29, 30, 31] and GPUs [34, 35]. All these implementations
presented different approaches to maximize the parallelization of the algorithm, while trying to avoid as
much communication overhead as possible. However, whenever highly parallel environments are con-
sidered, non fully distributed master-slave approaches are still prominent. In these cases, the master
node often becomes a performance bottleneck as the number of parallel processing threads increases,
since it has to satisfy the requests of every slave node. Furthermore, the increasing communication over-
head between the multiple slaves and the master node can only be hidden by allowing asynchronous
communications, which often prevents an exact reproducibility of the results. Nonetheless, some meth-
ods to manage the conflicts between the nodes being simulated in parallel are presented in [34] (namely
blocking, waiting for a given threshold or postponing the simulation), which may be of interest depending
on the parallelization approach.

Finally, some specific parts that compose the DSS algorithm have already been accelerated by
using GPUs, such as the k-nearest neighbour search (part A) [36] or the construction and solution of a
kriging system (part B) [37]. However, both cases consider a different problem setting, which justifies
the acceleration using such devices. As an example, the proposed accelerated k-nearest neighbour
search only considers a single execution of the algorithm (which includes the distance computations).
Accordingly, the considered problem setting does not take into account that some data can be reused,
such as a list of relative positions sorted by distance to a given reference. As for the kriging estimate,
the proposed acceleration approach considers a more complex kriging methodology (universal kriging
3.4 Summary

In this chapter a comprehensive description and analysis of the algorithm being accelerated was conducted. Firstly, a contextualization of the application was made, by taking into account the main theoretical concepts required to understand the algorithm's behaviour. Namely, the main structure of an inversion problem was described, and the three physical properties of interest were briefly defined. The main composing modules of the algorithm were subsequently presented in detail, with a special focus in the stochastic simulation procedure, since it corresponds to the most computational demanding part of the inversion algorithm. Finally, a comprehensive study of the related work was conducted, highlighting most of the work that has been developed in the seismic inversion and stochastic simulation areas. To the best of the authors' knowledge, only one parallel implementation of the DSS algorithm was proposed, where a multi-core approach was implemented by considering a straightforward functional decomposition of the algorithm, presenting considerable limitations in terms of scalability. Nevertheless, other similar algorithms have already been optimized and accelerated, being obtained some promising results when considering the use of GPUs, although master-slave approaches are still prominent. Accordingly, in contrast with the state of the art solutions, the parallelization approach devised in the following chapters aims to efficiently exploit the target architectures by processing the algorithm in a more distributed manner, thus achieving a greater scalability.
CHAPTER 4

ALGORITHM PARALELLIZATION

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. For such purpose, the algorithm to be accelerated must be conveniently profiled and analysed, in order to define which parts of the algorithm are worth parallelizing by using the previously identified OpenCL enabled devices. In addition, the parallelization approach that presents less limitations in terms of scalability and that can be efficiently mapped into the target devices must be chosen. Accordingly, a comprehensive study of the multiple possible algorithm partitioning methodologies must be conducted, by taking into account the algorithm characteristics and dependencies. A good decision in terms of a partitioning of the algorithm is essential in order to enable the development of efficient and highly scalable implementations.

4.1 Problem Analysis

In what concerns the measurement of performance, when accelerating a given application, the most commonly used metric is performance $P$, which corresponds to the number of operations can be made per second. Since for a fixed algorithm and dataset the number of operations is fixed, the performance can be seen as the inversion of time, $P = \frac{1}{T}$.

A secondary relative metric is the speed-up ($\psi$), which represents how much faster an accelerated algorithm is being executed when compared with its original implementation. Thus, the speed-up is defined as the ratio between the original execution time ($T_s$) and the accelerated execution time ($T_p$) (see equation 4.1):

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

(4.1)

Performance improvement is considered when the speed-up is greater than one. Therefore, the main
The purpose of parallel programming is to reduce the execution time by evenly dividing the computational effort between the available processors ($n$), thus ideally achieving an execution improved by a factor of $n$ (see expression 4.2):

$$\psi = \frac{T_s}{T_p} = n$$  \hspace{1cm} (4.2)

As such, the efficiency ($e$) of a parallel implementation is given by the ratio between the speed-up and the number of processors used (expression 4.3):

$$e = \frac{\psi}{n}$$  \hspace{1cm} (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, which indicates that the maximum achievable speed-up when parallelizing a given application would also be infinite. However, real applications usually are not completely parallelizable, being composed by a sequential code fraction ($s$) that cannot be executed in parallel. Thus, a more precise estimation of the maximum attainable speed-up can be defined by expression 4.4, also known as the Amdahl’s Law [38]. Nevertheless, this expression does not take into account the communication and parallelization overheads inherent of a parallel algorithm implementation, which means that in most cases it represents an overestimation of the maximum achievable speed-up. 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.

$$\psi = \frac{T_s}{T_p} \leq \frac{T_s}{s \times T_s + (1 - s) \frac{T_r}{n}} = \frac{1}{s + \frac{1 - s}{n}}$$  \hspace{1cm} (4.4)

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. Therefore, in order to ensure an efficient partitioning and mapping of the algorithm into the considered parallelization platforms, a comprehensive analysis of its execution and of its implicit dependencies is of the utmost importance.

### 4.1.1 Algorithm Profiling

As stated by Amdahl’s law, only the most demanding and time-consuming sections of an application are worth parallelizing. Accordingly, the algorithm was profiled to find the most time consuming phases, considering two distinct datasets: a smaller one composed of a grid of 101x101x90 nodes; and a larger and more realistic one with 237x197x350 nodes. For such purpose, the execution time of the different parts of the algorithm was accurately measured using the Performance Application Programming Interface (PAPI) [39] to interface with the hardware performance counters.

As previously mentioned, the original implementation of the application was divided in two parts: an implementation of the stochastic simulation procedure implemented in C/C++; and an implementation of inversion framework (composed by modules 4, 5, 6 and 7) in MATLAB. Accordingly, in order to provide a more efficient execution, the algorithm was first unified by translating the parts implemented in MATLAB.
to C/C++ and improving their interoperability with the remaining code. Therefore, profiling results were obtained by considering the execution of an already optimized sequential implementation of the algorithm in an Intel i7-3820 processor (compiled using the -O3 compiler optimization flag), when executing 3 iterations composed by 10 realizations each.

From the obtained profiling results (see Figure 4.1), it can be observed that more than 95% of the algorithm execution time is spent in the generation of $\rho/V_p/V_s$ models. Consequently, this part of the algorithm was chosen as the prime focus for acceleration, since it is the part where the parallelization effort may result in a higher speed-up. In particular, by applying the expression 4.4 when considering that this part represents 98.1% of the execution time (larger dataset), it can be verified that the maximum speed-up that can be achieved with the parallelization of this part of the algorithm (considering an infinite number of parallel processors) is approximately $52.6 \times$. As for the smaller dataset, a maximum achievable speed-up of $29.4 \times$ is expected, since the simulation of $\rho/V_p/V_s$ represents 96.6% of the algorithm execution time. Once again, it must be noted that this values correspond to an overestimation of the attainable speed-up by only parallelizing this part of the algorithm, since it is being considered that this part of the algorithm is completely parallelizable, and that its parallelization has no communication and parallelization overheads.

![Figure 4.1: Profiling results for the Stochastic Seismic AVO Inversion algorithm, considering the sequential execution of the algorithm, composed of 3 iterations each with 10 realizations using both the experimental datasets.](image)

After observing that most of the execution time is spent in this stochastic 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. Once again, there is a well identified section of the code where most of the execution time is spent (see Figure 4.2). Thus, the node simulation procedure represents the first candidate for parallelization, by taking advantage of heterogeneous computing.

Finally, by analysing Table 4.1, it can be observed that the simulation of a single node (composed by parts A, B, C and D of the DSS algorithm) is performed on average in approximately $10,11 \mu s$ and $20,09 \mu s$ for the datasets composed by $101\times101\times90$ and $237\times197\times350$ nodes respectively. Therefore, it can be concluded that the simulation procedure is composed by millions of steps that individually are not computationally demanding. Consequently, 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 parallelization. In order to support the decision about the partitioning of the algorithm, we performed a comprehensive analysis of the several existing data dependencies in the generation of the $\rho/V_p/V_s$ models, in order to support the decision about the partitioning of the algorithm.

<table>
<thead>
<tr>
<th></th>
<th>$\rho$ [(\mu s)]</th>
<th>$V_p$ [(\mu s)]</th>
<th>$V_s$ [(\mu s)]</th>
<th>den. [(\mu s)]</th>
<th>$V_p$ [(\mu s)]</th>
<th>$V_s$ [(\mu s)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part A</td>
<td>0.808</td>
<td>0.814</td>
<td>0.814</td>
<td>1.551</td>
<td>1.584</td>
<td>1.562</td>
</tr>
<tr>
<td>Part B</td>
<td>1.968</td>
<td>1.991</td>
<td>1.990</td>
<td>1.972</td>
<td>1.973</td>
<td>1.980</td>
</tr>
<tr>
<td>Part C</td>
<td>0.150</td>
<td>0.155</td>
<td>0.155</td>
<td>0.152</td>
<td>0.155</td>
<td>0.156</td>
</tr>
<tr>
<td>Part D</td>
<td>2.064</td>
<td>9.627</td>
<td>9.789</td>
<td>4.884</td>
<td>2.662</td>
<td>41.633</td>
</tr>
<tr>
<td>Average single node simulation</td>
<td>10.109 [(\mu s)]</td>
<td>20.088 [(\mu s)]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Discriminated average execution time per node for each part of the simulation procedure.

### 4.1.2 Algorithm Dependencies

The subsequent step in the parallelization process considered a comprehensive analysis of the several existing data dependencies in the generation of the $\rho/V_p/V_s$ models, in order to support the decision about the partitioning of the algorithm.

As it was described in Section 3.2.1, the DSS algorithm is a sequential stochastic simulation method. Therefore, the simulated value of a given node is conditioned by the previously simulated nodes located in the initially defined random path, as well as by the nodes initially assigned with well-log data. In particular, at each step of the node simulation procedure (processing sequence from part A to D), conditioning data is selected by finding the k-nearest neighbouring nodes with available data (being the k value defined as a simulation parameter). A correct selection of conditioning data guarantees that the simulated values are coherent with the neighbouring data, thus ensuring a consistent spatial distribution of the variable being simulated. Therefore, since the simulation procedure follows a random-path, conditioning data is selected from a not strictly defined neighbourhood of the node being simulated (see Figure 4.3). This makes the data dependencies difficult to predict or to confine to a well defined spatial region, since it initially depends on the distribution of well-log data and subsequently depends on the positions of the nodes being simulated. However, it must be noted that the search radius tends to decrease.
as the number of simulated values increases, which means that the management of such dependencies is especially important during the first steps of the node simulation procedure.

Accordingly, as represented in Figure 4.4, the estimation of the local conditional distribution function (part C) at each node makes use of values obtained from previously simulated nodes (part D), imposing a strict commitment of a set of data dependencies within the processing grid.

**Figure 4.3:** Conditioning data locations during the simulation of a given node, considering $k=6$.

**Figure 4.4:** Data dependencies on the stochastic simulation of the $\rho$, $V_p$ and $V_s$ models.

### 4.1.3 Parallelism in Stochastic Seismic Inversion Algorithms

Taking into account the characteristics of a stochastic seismic inversion algorithm, it can be observed that in order to accelerate such algorithms there are multiple different levels of parallelism that can be considered, by assuming different granularities (as illustrated in Figure 4.5). By considering a coarse-grained perspective, given that each iteration is composed by several independent realizations (sets of $\rho$, $V_p$, and $V_s$)
Vp and Vs simulations), the parallel execution of multiple realizations can be considered. Nevertheless, the construction of the best model (in this case by using a genetic algorithm) still needs to be sequentially performed, since it has to select the best regions from each generated model by comparing with the current best model. As such, a realization-level parallelization is mainly limited by the execution of the inherently sequential part of the algorithm, and also by the number of realizations per iteration that are considered, directly limiting the number of parallel threads that can be considered.

Given such limitations of a coarse-grained parallelization perspective, a further level of scalability can be attained by individually accelerating each of the composing modules of the algorithm. In the class of algorithms herein being studied, the stochastic simulation is usually the most computational demanding part of the algorithm, consequently being the prime focus for acceleration. Considering a sequential simulation method (most commonly used stochastic simulation approach), three distinct levels of parallelization can be considered [33]:

- **path-level**, where multiple nodes from the random path are simulated at the same time by different parallel processing threads;

- **functional-level**, by simultaneously executing multiple independent parts of the stochastic simulation algorithm. Such can be achieved either by overlapping computations from different parts of the same node simulation step (if possible), or by executing in parallel independent parts from consecutive node simulation steps (e.g., in the DSS algorithm parallelize the execution of parts A
and B from different nodes, and compute the remaining steps sequentially);

- **data-level**, where each part of the stochastic simulation algorithm is individually accelerated by simultaneously processing independent data (e.g., individually accelerating parts A, B, C and D of the DSS algorithm).

In a sequential simulation method, a **path-level** parallelization is mainly limited by the spatial data dependencies that exist between the multiple nodes. As described in Section 4.1.2, such dependencies arise from the fact that the simulation of a given node may require the previously simulated values as conditioning data. Therefore, only the parallel simulation of sufficiently distant nodes (being outside of the search range) can be allowed in order to guarantee the same resulting models when comparing with the original algorithm. In what concerns a **functional-level** parallelization approach, it is mainly limited by the dependencies between the distinct processing modules that compose the algorithm and by their sequential execution times (since those modules are not being individually accelerated). Finally, a **data-level** parallelization approach is limited by the parallelization opportunities inherent to each part of the sequential simulation algorithm. Also, since this corresponds to a fine-grained parallelization approach, the acceleration gains must be greater than the parallelization overhead in order to result in significant performance improvements.

To accelerate the Stochastic Seismic AVO Inversion algorithm, the intrinsic characteristics of the DSS algorithm (the most computationally demanding part of the inversion procedure chosen as the prime focus of parallelization) must be taken into consideration. Accordingly, in order to guarantee the efficiently exploitation of heterogeneous computing platforms, the application of all the previously enumerated parallelization strategies must be analysed in detail. By observing the set of existing dependencies between multiple parts of the algorithm and the previously presented profiling results, it can be easily concluded that a **functional-level** parallelization would hardly provide good acceleration results (see Figure 4.6). In fact, although parts A and B can be fully performed in parallel, the algorithm acceleration is limited by the sequential execution of the random path through every node (parts C and D), which still represents most of the simulation execution time. As a matter of fact, taking into account that the last two parts represent approximately 78.8% of the simulation procedure execution time when considering the larger dataset, such parallelization scheme is limited by a maximum speed-up of 1.27×. Furthermore, an individual acceleration of each part of the algorithm (**data-level** parallelization) would hardly give good results too, not only because the multiple parts of the algorithm present few parallelization opportunities, but also because their sequential processing time is not enough to be worth the parallelization overhead. In fact, the complexity of the algorithm lies in the several millions of nodes that have to be simulated and not in the simulation of a single node. Finally, in order to guarantee the same results as the original sequential version of the algorithm, a **path-level** parallelization has to ensure that no conflicts are observed between the nodes being simulated in parallel (neighbourhoods must not overlap), which introduces serious limitations in terms of scalability. Such limitations are not only related with the overhead introduced by a conflict management mechanism, but also with the significant restriction that this mechanism would impose in the number of nodes that can actually be performed in parallel.
4.1.4 Discussion

In addition to the previously presented analysis, some complementary aspects of the algorithm were also evaluated. In particular, in addition to the performance counters used to measure time, other performance counters, available by using the PAPI, were used in order to evaluate the performance limiting factor of the algorithm. As it can be observed in Table 4.2, the DSS algorithm has a significant amount of memory instructions per floating point operation, which indicates that the application is mainly memory bounded. In fact, the same conclusion can be drawn by using SchedMon [40] to analyse the Cache-Aware Roofline model [41], where it is clearly observed that the execution samples are located in the memory bounded region of the model (see Figure 4.7).

Table 4.2: Performance results regarding the number of memory accesses and the number of floating-point operations during a single execution of the simulation procedure, by considering the dataset with 237x197x350 nodes.

<table>
<thead>
<tr>
<th></th>
<th>L1D Cache Accesses</th>
<th>Floating-Point Operations (flops)</th>
<th>Mem_accesses/flops ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ρ</td>
<td>279 ( G_{\text{accesses}} )</td>
<td>36.7 ( G_{\text{flops}} )</td>
<td>7.6</td>
</tr>
<tr>
<td>Vp</td>
<td>176 ( G_{\text{accesses}} )</td>
<td>52.8 ( G_{\text{flops}} )</td>
<td>3.3</td>
</tr>
<tr>
<td>Vs</td>
<td>580 ( G_{\text{accesses}} )</td>
<td>342 ( G_{\text{flops}} )</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Accordingly, it can be concluded that even by further optimizing the sequential implementation of the algorithm (optimizing the memory accesses and consequently vertically rising the samples in the model), the performance would rapidly become limited by the maximum attainable CPU bandwidth, which corresponds to the L1 Cache roof. Similarly, when migrating the application into a GPU, it is expected to obtain speed-ups mainly limited by the bandwidth increase from the CPU (considered as baseline) to the GPU being used to accelerate the application. Nevertheless, higher speed-ups may be verified due to code optimizations that take advantage of the higher computational parallelism provided by the GPU architectures (e.g., parallel reductions) or even by the differences between the CPU and GPU compiled code, which may vary due to the possibly different instruction sets made available on each device.
4.2 Parallelization Approach

As discussed in Section 4.1.3, the existing data dependencies between the simulation nodes represent a significant limitation to the attainable parallel performance. Therefore, to circumvent these limitations, the proposed approach is based on a relaxation of the algorithm definition in order to enable an efficient (but still accurate) path-level parallelization. Such a parallel implementation was achieved by: i) dividing the simulation grid in multiple sub-grids; and ii) randomly selecting a node within each individual sub-grid at each step of the simulation procedure. At the end of each simulation step, the whole set of nodes being simultaneously simulated is updated at once in the simulation grid, conditioning the subsequent simulation steps (see Figure 4.8). Therefore, the simulation of nodes from different sub-grids can be executed independently in different processing cores (from the same or from different devices), as long as the whole simulation grid is updated at the end of each simulation step.

The actual number of sub-grid divisions, is left as a simulation parameter, where different number of divisions result in different random paths, leading to unrelated node processing orders and data conditionings, which consequently result in distinct generated models. However, this number of divisions (with direct relation with the number of generated sub-grids) must be great enough in order to completely exploit the computing capabilities of the considered devices. This influence in the generated models restricts the possibility of using dynamic mechanisms to optimize the occupation of the considered devices, since the reproducibility of the resulting models would be lost. At this respect, it is worth noting that the considered sub-divisions of the simulation grid are not required to be cubic. In fact, the anisotropic nature of the well-log data being processed (sparsely located along the study area but with a very high vertical resolution) suggests that there are significantly less dependencies in the vertical direction, which allows for a greater vertical division of the simulation grid.

In accordance, after populating the simulation grid with the initially available well-log data, the sub-grid is divided and a random-path is generated, being the order by which the nodes are supposed to be
simulated stored in a array list (see Figure 4.9). Therefore, on each simulation step the positions of the next bundle of nodes to be simulated can be read from this array list by efficiently taking advantage of the coalesced memory accesses when a GPU device is considered (work-items will access consecutive memory positions from this data structure). While the OpenCL-enabled devices are responsible by accelerating the most computational demanding part of the algorithm (the node simulation procedure from the DSS algorithm), the host device must not only execute all the other not so time consuming parts of the algorithm, but it is also compelled to orchestrate the OpenCL execution, managing the contents of the device memory and launching the corresponding kernels.

4.2.1 Generation of the Random Path Criteria

Since this approach implies the removal of some data dependencies, the choice of the nodes to be simulated in parallel must be performed with criteria, in order to maximize the performance of the algorithm execution and still ensure the reproduction of the spatial distributions and inherent uncertainties of the physical properties being simulated ($\rho$, $V_p$ and $V_s$). Accordingly, regarding the generation of the random path, two distinct criteria can be considered to select the simulation nodes:

- Follow the same relative sub-path in each sub-grid;
- Allowing each sub-grid to have its own random sub-path.

Taking into account that as long as the nodes being simulated are sufficiently apart from each other (being kept outside of the search ranges), no data conflicts should occur [32], it can be observed that one of the main advantages of the first criterion is that it grants a constant and equidistant distance between the nodes being simulated in parallel, thus minimizing the number of conflicts. Moreover, if the
sub-grids are larger than the maximum search-range in all directions, data conflicts can be completely avoided. In addition, such regularity in the set of nodes being simulated in parallel can also present some advantages in terms of the GPU’s implementation efficiency, since the conditioning data search times would be similar among the multiple nodes being simulated in parallel (relative positions with available data would tend to be the same). On the other hand, the second alternative presents less restrictions to the considered random path, increasing the number of possible paths and consequently the number of possible solutions. In fact, considering that a simulation grid composed by $N$ nodes is divided into $p$ sub-grids, by following a distinct random sub-path on every sub-grid there are $\binom{N}{p}^p$ possible combinations of sub-paths, being significantly more than the $\frac{N}{p}$ possible sub-paths made available by using the first criterion. This aspect is significant, considering the stochastic exploration-based nature of the algorithm, thus taking advantage of a greater variety of solutions generated under the same circumstances (with different random seeds). Although in the rest of this thesis a preference has been given to the criterion that considers distinct random sub-paths on every sub-grid, this decision has been left as a simulation parameter, and a more detailed analysis regarding the performance and the quality of the resulting models when using both criteria is conducted in Section 6.2.1.

### 4.2.2 Load Balancing Mechanisms for Heterogeneous Platforms

To achieve the maximum performance on systems composed by multiple processors, it is important to have an even balance of the workload so that all processing units contribute equally to the task at hand.
The need to scale problems into a multi-device paradigm arises from the fact that a single architecture may not provide enough parallel computing capabilities for exploiting all the parallelism made available by an application. Accordingly, load balancing algorithms aim to efficiently distribute computations across systems composed by multiple devices, in such a way that all the devices will finish their computation within the same time. In particular, heterogeneous systems may present devices with different computing capabilities, thus introducing an additional level of complexity to the load balancing problem. There can be identified two types of heterogeneity: i) performance-level heterogeneity, corresponding to systems that combine devices with similar architectures (e.g., multiple CPUs) but with different computing capabilities; ii) architectural-level heterogeneity, corresponding to systems composed by devices with significantly different architectures, such as combining CPUs with GPUs or other type of co-processors. Nevertheless, both types of heterogeneity often correspond to significant differences in the computational performance (when executing a given task) and in communication performance.

Load balancing algorithms can be classified either as being static or dynamic. Static algorithms are characterized by using information available at compile-time in order to predict a good workload division. On the other hand, dynamic load balancing algorithms make use of run-time measurements (e.g., execution time, power consumption) in order to enable an efficient load distribution among the multiple devices. Although static load balancing algorithms are interesting from a theoretical point of view, since a perfect static algorithm would correspond to the most efficient execution since no run-time overheads are introduced in the algorithm with such scheme, their practical use is very limited due both to the difficulty in successfully predicting the devices performance, and to their lack of adaptive capabilities. Such adaptive capabilities enable not only an effective calibration of the work-load distribution taking into consideration the heterogeneity of the considered devices, but also are able to adjust the work-load distribution according to unpredictable environmental changes, for example when some devices are being partially used by other concurrent application.

Another aspect that can be classified regarding such algorithms concerns distinction between making the balancing decisions in a centralized manner, taking into account global load information, or in a distributed manner, being the load locally migrated among the multiple available processors, for example by using a technique called work-stealing, where after finishing its own tasks each processor tries to steal tasks from slower neighbouring processors. In addition, dynamic centralized algorithms can be further classified into task queue and predicting-the-future algorithms. Task queue algorithms, as the name indicates, consider a queue of independent tasks that are distributed as processors become available. As for predicting-the-future algorithms, tasks are distributed by predicting the performance of the multiple processors based on past information. Other more complex load balancing algorithms consider a functional performance model of the devices being used, thus representing with a greater accuracy the speed of processors at the cost of higher overheads in order to compute the performance models.

Taking into account that dynamic centralized predicting-the-future load balancing algorithms are suitable for data-intensive iterative routines, which is the case of the seismic inversion algorithm herein being accelerated, such methodology was chosen in order to improve the efficiency of a multi-device
approach (described in section 5.2). In particular, the algorithm being accelerated is characterized by having a relatively constant number of nodes being simulated in parallel on each simulation step, and by the difficulty to predict the overhead of processing each workload unit (simulation of a single node), since it varies both in time (last simulation steps are less computationally demanding than the first simulation steps) and on the spatial locations of the nodes being simulated in parallel (regions with different amounts of neighbouring data result in different processing times). Accordingly, the use of a traditional load balancing scheme was considered to be appropriate [45], being able to adapt to the circumstances imposed by the algorithm characteristics. This load balancing routine is mainly composed by the following steps (considering the execution over \( n \) devices):

1. Measure the computation execution times on each device \( t_{i0}, \ldots, t_{i(n-1)} \) for the simulation step \( i \).

2. If \( \max_{0 \leq j,k \leq n-1} \left| \frac{t_{ij}}{t_{ik}} - 1 \right| \leq \epsilon \) then a converged state has been achieved and no further weight redistribution is required.

3. Otherwise, updated weights \( w_{i+1}^1, \ldots, w_{i+1}^n \) are computed as \( w_{i+1}^j = \frac{s_j^i}{\sum_{k=0}^{n-1} s_k^i} \), where \( s_j^i \) stands for the speed of the processor \( j \) during the current simulation step, given by \( s_j^i = \frac{\text{nodes computed}}{t_{ij}} \).

4. The number of nodes to be computed in parallel during the next simulation step are distributed among the multiple devices according to the corresponding weights. Namely, the weight corresponds to the percentage of sub-grids being simulated by a given device.

### 4.3 Summary

Aiming an efficient parallelization of the algorithm, an extensive analysis of its processing modules was conducted and described in this chapter. Such analysis supports the subsequently proposed partitioning of the problem and the corresponding parallelization approach. In particular, the algorithm in study was conveniently profiled, being identified the stochastic simulation part of the algorithm as the most computational demanding. This part of the algorithm is composed by millions of dependent iterations that individually are not computationally demanding. Then, taking into consideration the algorithm characteristics, four distinct parallelization approaches (by considering different granularities) were identified: realization-level, by computing different sets of models in parallel; path-level, where multiple nodes from the random path are simulated at the same time by different parallel processing threads; functional-level, by simultaneously executing multiple independent parts of the algorithm; data-level, where each part of the algorithm is individually accelerated by simultaneously processing independent data. The main limitations for each approach were also identified. Therefore, due to limitations imposed by such dependencies and to the lack of data parallelism opportunities presented, a relaxation of the algorithm was considered in order to efficiently exploit the highly parallel architecture of the GPUs, significantly reducing the algorithm execution time without compromising the quality of the obtained models. This relaxation consists in a spatial division of the simulation grid, thus 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 ensure the reproduction of the spatial distributions and inherent uncertainties of the physical properties being simulated. Accordingly, two distinct criteria were identified regarding the generation of the random path, namely: i) by following the same relative sub-path in each sub-grid; or ii) by allowing each sub-grid to have its own random sub-path. Taking into account that the considered partitioning scheme allows for a great parallelism (as high as the number of divisions in the simulation grid), a study of the state of the art load balancing mechanisms was conducted, in order to efficiently take advantage of the increased parallel computing capabilities made available by heterogeneous platforms. From this study it was identified a traditional dynamic centralized predicting-the-future load balancing algorithm as the most appropriate mechanism to divide the workload among the available devices.
This thesis aims the conception of parallel and highly efficient implementations of the Stochastic Seismic AVO Inversion algorithm by considering heterogeneous platforms composed by multiple (and possibly different) CPUs and GPUs. Accordingly, such architectures were carefully described in Chapter 2, and a scalable partitioning scheme of the algorithm was devised in Chapter 4, thus enabling a coherent and well supported development of such highly efficient implementations. In order to circumvent the limitations imposed by the data dependencies, the proposed partitioning methodology considers a relaxation of the most computational demanding part of the seismic inversion algorithm being accelerated, allowing for a significant scaling in the degree of parallelism. This chapter discusses how such high degree of parallelism can be used in order to efficiently exploit the computational capabilities of the OpenCL enabled devices made available in heterogeneous platforms.

5.1 Single-Device Parallel Solution

As presented in the previous chapter, the proposed parallelization approach considers a spatial division of the simulation grid (used by the DSS algorithm), in order to allow the parallel simulation of multiple nodes. According to the OpenCL programming model (described in Section 2.1.1), a data parallel execution of multiple node simulations is performed by having several instances of the same kernels being executed at the same time by multiple work-groups. However, since GPUs are being considered as target devices, those kernels have to be carefully crafted in order to efficiently exploit the GPU architecture, by minimizing performance loss factors (warp-divergence, non-coalesced memory accesses) and maximizing the device occupancy. Therefore, when migrating the node simulation procedure of the
DSS algorithm into OpenCL code, the application has to be functionally partitioned into multiple kernels. An efficient functional partitioning is obtained by achieving a compromise between the granularity of the kernels (influencing the number of registers being used and consequently the occupation of the GPU) and the number of kernel calls (that have an inherent overhead). In addition, different parts of the algorithm were merged or separated into different kernels according to the number of intermediate global memory buffers that had to be created in order to transfer the required data between the multiple kernels. By merging two different parts of the algorithm into the same kernel, such information can be stored and accessed through the local memory or even by using local registers, significantly reducing the memory access times. The only limitation to such kernel management concerned the impossibility to merge the part where the simulation grid and other global variables are updated (see kernel IV) with other more significant parts of the algorithm, since in order to compute the mean and variance of the whole set of simulated values, some global synchronization mechanism was required (which in OpenCL is only possible by launching a new kernel instance). Accordingly, after implementing and optimizing the proposed parallelization approach, the resulting implementation considered 4 distinct kernels with the following functionalities:

I - Selection of conditioning data for the $\frac{N}{P}$ nodes being simulated in parallel (one from each sub-grid);

II - Construction of $\frac{N}{P}$ kriging systems, according to the locations of the selected conditioning data and subsequently solving them and to estimate the corresponding LCDFs;

III - Condition the distribution function by a previously simulated auxiliary variable\(^1\) and generation of the simulated values given the resulting LCDFs by using a Monte Carlo method;

IV - Store the simulated values into the simulation grid and update the information relative to the already simulated nodes (mean and variance).

It is worth noticing that, with the exception of a task that will be further detailed in Section 5.4, only the tasks related to the node simulation procedure are actually performed by the OpenCL enabled devices, being the host device responsible not only by the generation and transfer of the data required in the simulation procedure (overlapped with the computations), but also by the execution of all the other not so significant parts of the inversion algorithm (see Figure 5.1). The acceleration of other parts of the algorithm was not considered because they did not represent a significant part of the algorithm execution time. Therefore, by taking the Amdahl’s law into account, accelerating such parts was not deemed worthwhile, since it would not have a significant impact in the algorithm execution performance. In addition, parts corresponding to the inversion framework (modules 4 to 6 of the Stochastic Seismic AVO Inversion algorithm) required at least the use of all the generated models in the current realization, providing some scalability limitations not only in terms of the available device memory (due to the need to store multiple models that depend on the problem size), but also in terms of the time required to perform the corresponding communications, which may be significant when compared with the time spent performing the corresponding computations, and consequently become a bottleneck.

\(^1\)In this case it is only used for the simulation of $V_p$ (conditioned by the previously simulated density model) and $V_s$ (conditioned by the previously simulated $V_p$ model).
A mapping methodology defines how the multiple partitions that result from the algorithm partitioning scheme (see Section 4.2) are going to be distributed among the multiple parallel processing units made available by the target architectures. In this case the GPU architecture was chosen as the reference target architecture, not only because the OpenCL framework resembles its architecture, but also because an efficient parallelization over its highly parallel architecture is pretended. Therefore, considering the general constraints imposed by GPU architectures, allied with the adopted OpenCL programming framework, there are at least two different ways to map the proposed parallelization approach (see Figure 5.2):

1. each sub-grid is simulated by a distinct OpenCL work-group, being the inherent parallelism of the algorithm exploited by the OpenCL work-items within each work-group;
2. each sub-grid is simulated by a distinct work-item.

Figure 5.2: Mapping methodologies, considering n work-groups composed by 64 work-items each.

The main difference regarding both approaches is concerned with the number of nodes being simulated in parallel, 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. The use of a complete work-group in a single node simulation aims to achieve a faster execution by improving the node simulation, which is achieved through the exploitation of the data parallelism opportunities. However, those resources are only 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, where only a single work-item in the work-group would effectively be performing useful computations. Although the second approach allows for a greater amount of nodes to be simulated in parallel, it is more memory demanding since intermediate buffers need to be replicated for every node under simulation. Nevertheless, some performance losses may still occur due to warp divergence and non-coalesced memory accesses, since each thread is simulating its own node, which may lead to different execution paths or accesses to different regions of the memory.

As it can be observed by the preliminary results presented in Table 5.1, more encouraging performance results were obtained when considering the mapping approach where each work-item is responsible for the simulation of a different node. In particular, the second mapping methodology was able to outperform the first mapping methodology on every part of the stochastic simulation algorithm, achieving a $3.5\times$ faster simulation of density, when considering the larger dataset. As previously mentioned, this difference was verified mainly due to the fact that the simulation of a single node presents few parallelization opportunities. Therefore, the first approach is not able to efficiently exploit the whole GPU architecture, which conducted to the selection of the second approach as the implementation to be used for further optimization. It must be highlighted that the results used to draw this conclusion are merely indicative of the capabilities of both implementations, since similar non-optimized states of development were considered.
Table 5.1: Obtained preliminary speed-up results for both mapping methodologies, considering a single density simulation execution over a dataset with 237x197x350 nodes in a NVIDIA GTX 660 Ti GPU. Speed-up results use as baseline the sequential execution time of an Intel i7-3820 CPU.

Taking into account that the adopted mapping methodology implies additional intermediate buffers, in the event that the available device memory is not enough (depending on the problem size), the simulation procedure can still be performed by simulating the grid layer by layer, keeping a complete copy of the grid being simulated only in the host device. These layers must contemplate not only the specific sub-grids that will be simulated, but also the neighbouring sub-grids that condition the simulation. Another alternative would be firstly, by using the host device, to perform the selection of conditioning data for every node to be simulated in parallel (which requires the neighbouring sub-grids), and then try to maximize the number of nodes being simulated in parallel in the subsequent parts of the algorithm. Therefore, this methodology would not require to allocate space in the device memory for the resources used by the conditioning data selection procedure, which would allow for some increased scaling of the problem size. However, not only this alternative methodology would not correspond to a completely generic solution (since some limitations in terms of the attainable problem size would still exist), but also such a centralized mechanism to perform the conditioning data selection would rapidly become the bottleneck of such implementation, both because of the increased computational and communication overheads.

Nevertheless, as further detailed in Section 6.2.2 of the experimental results, it has been verified that even for the larger dataset provided by the members of group CERENA, which has been considered as being a realistic dataset in terms of problem size, the originally proposed mapping solution does not require much memory, allowing for the execution of such problems on commodity GPUs.

### 5.2 Multi-Device Single-Realization Parallel Solution

Another important aspect that should also be considered when accelerating a given application is the possibility of using more than a single device, in order to ensure another level of scalability to the algorithm execution. In fact, taking into account that the OpenCL framework is being used, multiple and possibly different accelerators can be used in cooperation for such purpose. Accordingly, the multi-device parallelization approach herein proposed takes advantage of the previously described spatial division of the simulation grid (see Section 4.2), and divides the obtained sub-grids between the several OpenCL enabled accelerators according to an efficient dynamic load balancing algorithm (as previously studied in Section 4.2.2). For such purpose, the original simulation grid is initially divided into sub-grids, being the number of divisions proportional to the number of available devices, in order to exploit the
increased computational capabilities of the system when compared with a single-device execution. The sub-grids are then distributed among the available accelerators according to its computational capabilities. The increased number of divisions that arise from using additional devices should preferably be applied to the vertical direction, since less dependencies are usually verified in this direction due to the high vertical resolution of the well-log data.

In order to efficiently take advantage of eventually different computational capabilities delivered by different devices, the load is balanced by dynamically distributing the nodes to be simulated between the multiple OpenCL enabled devices, according to real-time performance measurements obtained by OpenCL profiling events. Therefore, as presented in Algorithm 5.1, at each step of the simulation procedure the sub-grids are re-distributed according to a traditional iterative load balancing routine [45] (previously described in Section 4.2.2), until a converged load-balancing state has been achieved.

**Algorithm 5.1** Integration of the traditional iterative load balancing routine in the node simulation procedure.

1: **procedure** WORKER THREAD
2:   **while** exist nodes to simulate **do**
3:     sem.wait(& balanceResults);
4:     /*Launch kernels and measure performance*/
5:     sem.post(& executionComplete);
6:   **end if**
7: **end while**
8: **procedure** MAIN SIMULATION PROCEDURE
9:   /*Initialize variables and set load balancer initial state*/
10:   for all available devices **do**
11:     launch_workerThread();
12:   **end for**
13:   **while** exist nodes to simulate **do**
14:     sem.wait(& executionComplete);
15:     converged = TRUE;
16:     for each device i **do**
17:       if converged == TRUE then
18:         for each device j > i **do**
19:           if fabs(performance[i]-performance[j])/performance[i] > 0.01 then
20:             converged = FALSE;
21:             break;
22:         **end if**
23:     **end for**
24:     if converged == FALSE then
25:       for each device i **do**
26:         speed[i] = computed_nodes[i]/performance[i];
27:         sum_speeds += speed[i];
28:       for each device i **do**
29:         weight[i] = speed[i]/sum_speeds;
30:         number_of_toComputeNodes[i] = weight[i] * bundleSize[nextStep];
31:     sem.post(& balanceResults);

Another important aspect regards the memory management, which is required to ensure a coherent state of the memories of the multiple considered devices. Therefore, since the simulation grid is stored in the device memory (one copy per device), it must be updated with the information being computed in the other devices after the parallel simulation of every bundle of nodes. This implies an all-to-all communication scheme and a consequent synchronization point (see Figure 5.3). Since the current OpenCL API does not allow for inter-device communications (contrasting with recent CUDA implementations), the simulated values must be received in the host device and subsequently sent in parallel to every
accelerator. After receiving the whole set of new simulated values, each device updates (in parallel, by using multiple work-items) its simulation grid, together with the mean and variance values of the already simulated nodes, by using parallel reductions. In fact, in addition to the simulated values, the bi-histogram class in which those values are included has also to be transmitted, since whenever an auxiliary variable is being used (Vp and Vs co-simulations), the algorithm makes use of partial mean and variance values per bi-histogram class. The bi-histogram class is computed by using the value of the auxiliary variable in the position being simulated, which means that as an alternative to sending the class number, the auxiliary variable value could have been read and the bi-histogram class recomputed for every new simulated value. Nevertheless, this implies multiple redundant global memory accesses and computations, not being observed any benefit in terms of performance.

At the end of the simulation procedure, the resulting model may be read from any device, since every device has an updated copy of the simulation grid. After this data transfer, the used resources are subsequently released and the resulting simulation grid is enqueued in order to be written into a file by another CPU thread, thus immediately proceeding to the next simulation.

The scalability of this implementation is mainly limited by three distinct factors:

- The communication and synchronization overheads, that exist in order to ensure that an updated copy of the simulation grid is kept on every device. This problem becomes more significant as the
number of devices significantly increases, since the amount of data to communicate increases and
the complexity of the all-to-all communication scheme also increases.

- The unbalanced execution during the first steps of the node simulation procedure. This overhead
  arises from the scarce and not evenly distributed well-log data that is initially available, which
  imposes in the first simulation steps high conditioning data search times, that highly depend on
  the region of the nodes being simulated in parallel. Therefore, devices simulation nodes from sub-
  grids with few available data in the neighbourhood usually spend more time during this first steps
  of the simulation procedure, regardless of their computing capabilities. Note that, in this case, an
  efficient load balance is extremely important because of the synchronization requirement at every
  simulation step.

- After significantly accelerating the node simulation procedure, this part of the algorithm looses
  significance in the overall execution time. As a consequence, further accelerating this part of the
  algorithm by significantly increasing the number of used devices does not correspond to signifi-
  cantly reducing the seismic inversion algorithm execution time.

5.3 Multi-Device Multi-Realization Parallel Solution

Although significant improvements can be obtained by using multiple GPUs to accelerate a single real-
ization, the attained scalability of such approach is limited by the intrinsic demand to communicate the
simulated values upon each simulation step and by the fact that only the execution of the node simulation
procedure is being parallelized. Accordingly, greater performance levels are expected by considering a
parallelization scheme at the realization-level, where several independent simulations are performed at
the same time by multiple devices, thus minimizing the need for communications. In this approach, each
device computes a different set of simulations ($\rho, V_p, V_s$), together with the corresponding forward model
and the correlation related computations regarding those sets of simulations (see Figure 5.4).

Nevertheless, the construction of the best model still needs to be sequentially performed, since it
selects the best regions from each generated model by comparing with the current best model. Ac-
cordingly, after executing a given realization, the synthetic seismic model must be processed by a CPU
thread responsible for the best model construction. Hence, beyond the set of CPU threads dedicated
to orchestrate the execution over each device and to writing the output files, an extra host thread is
exclusively used to build and select the best realization, and to build the local correlation cubes (main
thread, in Figure 5.5).

It must be noted that, similarly to the previously described parallel solutions, the only procedure
that is effectively being accelerated by using OpenCL enabled devices is the simulation procedure.
However, this approach takes a greater advantage of the task-parallel capabilities of modern CPUs
(host device composed by multiple parallel processing cores) in order to enable the parallel execution
of multiple simulations. To maximize the performance, all devices iteratively simulate one realization
without any midpoint synchronization. The only difference between the realizations being executed in
the multiple devices is the random seeds that are used to generate the random path and the random values required by the Monte Carlo method, thus generating different models. However, whenever a set of \( N_s \) realizations of the stochastic simulation algorithm have been performed, the local correlation cubes regarding the best models that were built have to be created, in order to condition the next generation of simulations. Hence, as soon as the last simulation of the current iteration is completed, all devices still under execution are interrupted (indicated with a cross, in Figure 5.5), resuming the simulation of realizations once the best local correlation cubes are constructed. Accordingly, taking into consideration that no dependencies exist between the multiple realizations, this implementation considers a distributed load balancing mechanism, where each device individually executes more tasks (realizations) until more realizations are no longer required.

Accordingly, in contrast with the previously described multi-device approach, the multi-device multi-realization approach is limited by:

- The sequential execution time of the best model construction procedure \( t_{\text{best}} \) when compared with the time spent performing realizations \( t_{\text{sim}} \), which may only become a bottleneck if the number of devices \( n \) significantly increases, being \( n \times t_{\text{best}} \geq t_{\text{sim}} \).
Figure 5.5: Temporal diagram of the multi-device multi-realization approach, considering two devices with different computational capabilities and four sets of simulations per iteration.

- The number of realizations per iteration (defined as a inversion parameter), that limits the number of devices that can execute in parallel.

Nevertheless, the first problem can be minimized by accelerating the best model construction part of the algorithm if deemed worthwhile, and the latter by considering the use of both the multi-device approaches (which may coexist), in order to provide another level of scalability to the algorithm execution. In such hybrid approach, each realization execution is individually accelerated by using multiple OpenCL devices instead of only a single one.

5.4 Considered Optimizations

Several optimizations were considered in order to optimize the performance of the proposed parallel solutions. Such optimizations arise both from good GPU programming practices, and from a continuous profiling of the implemented solutions, in order to assess the main limitations and overheads. For such purpose, those optimizations were divided into three different categories: generic optimizations, which are mainly related with good GPU programming practices, and consequently can be similarly applied in other GPU programming projects; application specific optimizations, which are related with limitations intrinsic to the algorithm characteristics; and optimizations to the rest of the algorithm, which do not specifically concern the part being accelerated by using GPUs.

5.4.1 Generic Optimizations

A significant performance improvement to the developed kernels comes with the efficient usage of the GPU local memory. This memory has two characteristics that can be exploited when accelerating an application: i) has significantly lower access times and higher data transfer bandwidth when compared with the global memory; ii) is shared by the multiple work-items that belong to the same work-group. Accordingly, the proposed solution extensively uses these memories (whenever possible) to improve
overall execution time. In particular, the first aspect is used to improve the access times to global memory buffers that are frequently used, by copying the content of those buffers to the local memory when they are accessed for the first time, and subsequently updating them (if required) at the end of the computations. The second characteristic, is used, for example, in order to optimize parallel reductions, which in this case are required to compute the mean and variance of the previously simulated nodes, that are used as correcting factors to the probability distribution functions computed in the subsequent simulation steps. In parallel computing, parallel reductions stands for a widely used approach to accelerate computations that make use of every element of a given array and transform it into a single element (e.g., sum of every element). This mechanism takes advantage of the associativity of the operations involved in such computations, thus dividing the problem by computing in parallel partial results, and recursively combining them in a tree-like fashion until the final result is obtained. Therefore, as represented in Figure 5.6, this method can be applied by using GPUs, since the local memory is shared among multiple work-items. In order to combine partial results from different work-groups, a synchronization through the global memory has to be made, which in OpenCL implies another kernel call.

![Figure 5.6: Example of a GPU parallel reduction by considering the sum operator.](image)

Further improvements in terms of memory access efficiency are achieved by optimizing the used data structures and its indexing. The main goal of such optimization is to increase the coalescence of memory accesses, thus enabling the parallel access to multiple memory positions by the work-items of a given work-group. For such purpose, data buffers must be organized in a way such as to enable consecutive work-items to access consecutive memory positions. For example, considering a global memory buffer
that must store \texttt{b.size} elements per work-item, an access to \texttt{b[*num_workItems+workItem_id]} is more efficient when compared with an access to \texttt{b[workItem_id*b.size+i]}, when looping through the positions \textit{i} of such array. Note that the latter case usually corresponds to most efficient memory accesses when considering the use of CPUs, since the spatial locality principle is being exploited.

Although worth mentioning, data compaction techniques are not being used in order to optimize memory accesses. Data compaction considers the representation of data by using other formats using fewer bits (e.g., 8-bit values), in order to improve the memory accesses, for example by allowing to read multiple values in a single memory access. However, since this algorithm makes use of floating-point precision data, that use of a wide dynamic range of values (thus justifying the used precision), the use of such techniques was not considered. Similarly, the use of fixed-point notation to accelerate computations was also not considered, not only for the same reason but also because, as previously mentioned, the algorithm execution is mainly memory-bounded, which means that most of the execution time is consumed by performing memory accesses. Therefore, the overhead of transforming the data to make use of the fixed-point notation would probably not be worthwhile. Nonetheless, an experimental study of such transformations is left for future work.

Another aspect that in some cases may be significant is the overhead related with the OpenCL calls. In fact, considering the current AMD GPU drivers, those overheads are in the order of hundreds of microseconds, which may become significant if there are a considerable number of kernel calls or memory transfers (reads or writes from the GPU global memory to the host device) per simulation step. At this respect, the data transfers have to be performed using the minimum number of OpenCL enqueue buffer calls possible (different buffers with data of the same type are combined into a single buffer), and a compromise has to be achieved between the granularity of the kernels and the number of kernel calls. As described in Section 5.1, such compromise must consider not only the global memory requirements, that arise when separating different parts of the algorithm in order to transfer data between kernel calls, but also the limitations imposed to the occupancy of the GPU by the increase in the kernel granularity (sometimes requiring the use of an increased number of GPU registers, thus limiting the number of work-items that actually can be executed in parallel). On the other hand, OpenCL parallelization overheads in NVIDIA devices are significantly lower, which reduces the benefits that arise from such optimizations. Nevertheless, in both cases, memory transfers and other computations performed by the host device can be overlapped with the kernel executions, thus totally hiding (or at least significantly reducing) the penalties from such overheads.

5.4.2 Application Specific Optimizations

Given the characteristics of the algorithm under study, a significant part of the execution time lies in the first steps of the node simulation procedure, specifically in the conditioning data search. This happens because the conditioning data search procedure starts by looking for available data from the closest positions relative to the node being simulated, until a given number of conditioning nodes is found (k, defined as a simulation parameter). Therefore, since initially the available data is scarce and it is not
evenly distributed in the simulation grid (see Figure 5.7) the simulation of nodes in regions with few available data results in significantly larger execution times. As a matter of fact, the time spent selecting conditioning data in the first 3 steps of the node simulation procedure corresponds to approximately 90% of the time spent in this part during the whole simulation, and to approximately 57% of the whole simulation execution time.

This problem was minimized by postponing the simulation of nodes from sub-grids in which there is not enough available neighbouring data (both in the sub-grid from the node being simulated and in the neighbouring sub-grids). Note that this optimization can be performed during the definition of the random path, since it is only required to know the number of available data per sub-grid in each simulation step, which is possible to predict by using one counter per sub-grid (at each simulation step, one node is simulated from each of the active sub-grids, which corresponds incrementing the respective counter). As a consequence (see Figure 5.8), the execution time of the first steps is significantly reduced at the cost of some extra steps in the end of the simulation algorithm, when there is already a large amount of available data, thus reducing the global execution time of the simulation procedure.

Namely, in the considered example, after performing the referred optimization, the conditioning data search procedure during the first simulation step is executed in 0.16s instead of 16s. The use of such optimization came at the cost of 8 extra steps in the end of the node simulation procedure, which together are computed in approximately 1.3ms. As a result, an overall execution time reduction of 14s was observed from a simulation that originally required approximately 29s to be executed, corresponding to a performance improvement of 48%. In addition, benefits have also been verified when more that one GPU were used in order to accelerate the node simulation procedure (as presented in Section 5.2), since it enabled a more balanced execution regardless of the position of the nodes being simulated in...
parallel. A collateral benefit of this postponing method also arises in terms of the quality of the obtained results, since it avoids the simulation of nodes from regions with few data in a close neighbourhood (see Section 6.3).

Some other less significant application specific optimizations were also considered, such as the use of a bottom-up merge sort algorithm, instead of a quick-sort, in order to sort multiple arrays in parallel (required to be performed when a probability distribution function is being conditioned by a previously simulated variable, during the co-simulations of \(V_p\) and \(V_s\)). Although both algorithms have an average case complexity of \(O(n \log(n))\), the quick-sort algorithm usually presents the best sequential execution performance results, because it can be efficiently implemented when considering CPU architectures. However, the merge sort algorithm has a fixed execution path if the arrays being sorted have the same size, thus leading to significantly less warp divergence. Consequently, when executing the algorithm in GPUs, slightly better performance results were verified when the merge sort algorithm was considered. Notice that, in this case, since one sorting operation is required per work-item, the use of parallel sorting algorithms was not considered, given that the main goal was to optimize the performance of multiple sorting operations being executed in parallel and not a single sorting operation.

### 5.4.3 Other Algorithm Optimizations

Finally, some optimizations outside of the node simulation procedure were also performed, such as the use of the bitonic sort algorithm to optimize the sorting of the array that stores the nearest relative positions to a given reference, according to the non-euclidean distances between nodes (weighted distances taking into account the anisotropy of the data). Although this part of the algorithm represents slightly less than 13% of the sequential simulation execution time when considering the smaller dataset (most of the read and setup data part execution time, as presented in Section 4.1.1), this part becomes more representative after significantly accelerating the node simulation procedure. Therefore, despite being only significant when the smaller dataset is considered, a parallel sorting algorithm has been used in
order to accelerate this part of the simulation algorithm.

Moreover, the output files that have to be written during the execution of the algorithm are also written in parallel by different CPU threads, thus becoming overlapped with the computations, in order to avoid as much overhead as possible. There are two types of files that have to be written: data generated on every realization (e.g., \( \rho \), \( V_p \) and \( V_s \) models); and data generated on every iteration, as is the case of the best seismic models obtained on each iteration of the inversion framework, by combining information of every realization. Accordingly, one CPU thread per realization being simulated in parallel is responsible for writing the corresponding data (specially significant for the Multi-Device Multi-Realization parallelization approach discussed in Section 5.3), and another single CPU thread writes the data generated per iteration.

5.5 Summary

Taking into account the presented parallelization approach, the devised methodologies to efficiently exploit heterogeneous platforms composed both by CPUs and GPUs are described in this chapter. For such purpose, this chapter starts by describing the main components of the considered implementation (which is based in the GPU architecture), and how the computations are distributed between the host device and the OpenCL enabled accelerators. In addition, the mapping methodology used to efficiently take advantage of the highly parallel GPU architectures was discussed, being concluded that significantly higher benefits can be obtained by performing a distinct node simulation on every work-item. In what concerns the use of multiple devices in order to further accelerate the algorithm, two distinct solutions are proposed in this chapter: i) multi-device single-realization approach, that improves the node acceleration procedure by further dividing the simulation grid and distributing the nodes to be simulated in parallel among the multiple available devices according to a traditional dynamic load balancing scheme; ii) multi-device multi-realization approach, that implements a realization-level parallelization approach, using multiple devices to perform different realizations in parallel. Both multi-device approaches are complementary, thereby contributing together for a high scalability of the algorithm execution in heterogeneous clusters. Finally, the main considered optimizations to the proposed implementations are subsequently described in detail. Only a single optimization presents a further constraint to the original algorithm setting, which is the optimization that proposes the postponing of the simulation of nodes from sub-grids in which there is not enough available neighbouring data. By doing that, the execution time is significantly reduced and, as a collateral benefit, the quality of the obtained results is also ensured, since it avoids the simulation of nodes from regions with few data in a close neighbourhood.
CHAPTER 6

EXPERIMENTAL RESULTS

With the main objective of assessing the efficiency of the proposed parallelization approach, as well as the multiple implementations presented in the previous chapter, the algorithm execution is herein evaluated both in terms of the attained execution performance and in terms of the quality of the inversion results. In the first case, the algorithm is evaluated by computing the experimental speed-up regarding an optimized sequential implementation of the original algorithm. As for the analysis of the quality of the inversion results, the algorithm is herein evaluated not only in terms of its convergent behaviour, but also by comparing the obtained spatial distributions of the generated models. Such quality assessment is relevant considering the adopted relaxation of the original algorithm.

6.1 Experimental Setup

The execution of the Stochastic Seismic AVO Inversion algorithm is evaluated in the following sections by using two distinct datasets: one with 101x101x90 nodes and a larger one with 237x197x350 nodes, which size is consistent with typical experimental datasets of the CERENA research group. The inversion parameters were kept constant in both cases, and chosen to have realistic values according to typical executions of this algorithm (e.g., the k value that represents the number of neighbouring data to be used during the node simulation procedure was set by the CERENA researchers with a value of 12). In order to accurately evaluate the proposed parallelization approaches, the execution over multiple heterogeneous environments was considered (see Table 6.1). The obtained results (performance and quality) were always compared with the execution of an optimized serial implementation of the original algorithm. As such, the reference implementation considers the translation of the MATLAB code to C language, and further integration into the rest of the application, resulting in significant performance benefits. The serial version, corresponds to the execution in an Intel i7-3820 processor (System 1 in Table 6.1), by considering the -O3 compiler optimization flag, which enables automatic loop-level
vectorization into SSE and AVX instructions.

By analysing the three chosen prototyping systems presented in Table 6.1, it can be observed that they have significantly different characteristics in terms of their GPU devices. Namely, in addition to the CPU that serves as a performance baseline, System 1 provides two GPUs from different manufacturers, in order to assess the portability capabilities of the OpenCL API and the efficiency of the used load balancing mechanisms when dealing with different device architectures. On the other hand, System 2 has two similar NVIDIA GPUs, which can be used to provide a fair assessment of the scalability of the proposed multi-device solutions when the processing capabilities are theoretically increased by a factor of two. Finally, System 3 provides another type of heterogeneity, where two distinct devices of the same manufacturer (one with high and other with average computational capabilities) can be used in cooperation.

<table>
<thead>
<tr>
<th></th>
<th>System 1</th>
<th>System 2</th>
<th>System 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>i7 3820</td>
<td>Xeon E5-2609</td>
<td>i7 4770K</td>
</tr>
<tr>
<td>Number of cores</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Maximum clock speed</td>
<td>3.8 GHz</td>
<td>2.4 GHz</td>
<td>3.9 GHz</td>
</tr>
<tr>
<td>Maximum memory bandwidth</td>
<td>51.2 GB/s</td>
<td>34.1 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB</td>
<td>32 GB</td>
<td>32 GB</td>
</tr>
<tr>
<td>GPU 1</td>
<td>Hawaii R9 290X</td>
<td>GeForce GTX 680</td>
<td>GeForce GTX 780 Ti</td>
</tr>
<tr>
<td>Stream processing units</td>
<td>2816</td>
<td>1536</td>
<td>2880</td>
</tr>
<tr>
<td>Maximum clock speed</td>
<td>1000 MHz</td>
<td>1058 MHz</td>
<td>928 MHz</td>
</tr>
<tr>
<td>Maximum memory bandwidth</td>
<td>320 GB/s</td>
<td>192.2 GB/s</td>
<td>336 GB/s</td>
</tr>
<tr>
<td>GPU 2</td>
<td>GeForce GTX 560 Ti</td>
<td>GeForce GTX 680</td>
<td>GeForce GTX 660 Ti</td>
</tr>
<tr>
<td>Stream processing units</td>
<td>384</td>
<td>1536</td>
<td>1344</td>
</tr>
<tr>
<td>Maximum clock speed</td>
<td>822 MHz</td>
<td>1058 MHz</td>
<td>980 MHz</td>
</tr>
<tr>
<td>Maximum memory bandwidth</td>
<td>128 GB/s</td>
<td>192.2 GB/s</td>
<td>144.2 GB/s</td>
</tr>
</tbody>
</table>

Table 6.1: Considered systems specification details.

6.2 Performance Evaluation

In order to evaluate the performance of the proposed approach, the algorithm was executed in a realistic scenario by considering 3 iterations composed by 10 realizations each (similarly to what was previously performed in the profiling analysis). Single-device executions considered 32 divisions of the simulation grid in every direction, corresponding to a total of approximately $2^{15} = 32768$ sub-grids (the actual value is slightly higher since the remainder of the division of the datasets dimensions per 32 is not being considered). As a result, given the considered division of the simulation grid, the simulation of 16335188 nodes (when considering the larger dataset) is performed in less than 500 node simulation steps composed by 4 distinct kernel calls. Whenever more than a single device was considered to accelerate the simulation procedure (multi-device single-realization approach), the vertical direction has been further divided in proportion with the number of considered devices. When executing in the OpenCL enabled devices, node simulations were grouped into work-groups of 64 work-items. The execution time of the different parts of the algorithm was accurately measured by using the PAPI hardware counter API. The only exception concerned the measurement of the execution time of kernels, which has been accom-
6.2.1 Evaluation of the Random Path Generation Criteria

In what concerns the generation of the random path, the default results are obtained by using the criterion that considers that each sub-grid has its own independent random sub-path, thus increasing the number of possible paths and consequently the number of possible solutions, which is usually a major concern in real-life case studies. Nevertheless, the execution by using both path generation criteria has been evaluated (see Figure 6.1), being observed that slightly better performance results have been achieved when considering the use of the same relative sub-path in every sub-grid. Such already expected result arise from the regularity of the random-path, which leads to less warp-divergence. For example, in the conditioning data search procedure, neighbouring nodes will tend to be found in the same positions relative to the nodes being simulated. In fact, from further profiling analysis of the algorithm execution when considering both criteria, it was verified that more than 25% of the difference in the simulation execution time is justified by an improvement in the execution of the conditioning data search procedure; approximately 37% by the increasing complexity of the generation of the random path procedure (since a significantly higher amount of random values has to be generated when a distinct sub-path is considered on every sub-grid); and the remainder by less significant execution time improvements in the remaining kernels. The observed differences in the execution times represent approximately 9% of the full algorithm execution time when considering the larger dataset, which means that the subsequently presented performance results corresponding to this part of the algorithm are expected to achieve a further performance improvement of approximately $1.10 \times$ if the limitations imposed by having the same relative sub-paths in every sub-grid are considered acceptable.

It can also be observed that in the considered case no significant differences in the global correlation coefficients have been obtained when comparing both criteria. The main reason for this is that the considered case contemplates few realizations per iteration, thus not allowing for a significant exploitation of the increased exploration space offered by enabling the generation of a distinct sub-path for each sub-grid. It can also be observed that a significantly slower convergence rate is obtained when con-
considering the larger dataset, which indicates the demand to execute a great number of iterations (each composed with a high amount of realizations) in order to obtain acceptable models in real-life case studies, thus leading to very long sequential executions, and consequently justifying the need to significantly accelerate this type of algorithms. Despite not being observed significant differences in terms of the obtained global correlation coefficients, the CERENA researchers considered the random path generation criterion that uses a distinct sub-path for each sub-grid to be more appropriate for further evaluation.

6.2.2 Evaluation of the Single-Device Solution

Figure 6.2 shows the obtained performance results over several single-device mappings (programmed with the same OpenCL source code), when considering the larger dataset. It must be noted that the simulation of the different physical properties ($\rho$, $V_p$ and $V_s$) uses both different data and slightly different the simulation procedures, which naturally introduces some variations in the resulting execution times (approximately 15.3%, 12.4% and 72.3% of the sequential simulation execution time respectively), consequently leading to significantly different speed-up results. The significantly higher execution time of the $V_s$ simulations in this dataset is justified by the significantly larger size of the bi-distribution $F[Z_1(x), Z_2(x)]$, used to condition the $V_s$ simulation by the previously simulated $V_p$ model. It is also worth noticing that only the time during the node simulation procedure itself (part of the algorithm effectively being accelerated) was considered for the presented speed-up measurements, without considering the time required to setup the simulation data (less than 5% of the sequential simulation execution time, as presented in Section 4.1.1).

From the obtained results, it can be observed that the execution time was significantly reduced in all the considered mappings. In particular, it was obtained a speed-up of $18.06 \times$, when considering the execution of the whole algorithm using a GeForce GTX 780 Ti GPU, and a performance improvement of $5.90 \times$ when the algorithm was mapped into the CPU. In the latter case, the obtained speed-up is even greater than 4 (number of CPU cores), due to the exploitation of the hyper-threading technology. This results are in general coherent with the differences in the theoretical memory bandwidths made...
available by the multiple devices (as defined in the devices specifications), being obtained significantly higher speed-up results in the devices with the higher memory access bandwidth (Hawaii R9 290X and GeForce GTX 780 GPUs). This observation reinforces the conclusion drawn in Section 4.1.4, where it was suggested that the application was mainly memory bounded. It must be noted that the multiple considered systems use different CPU devices, which justifies some speed-up discrepancies between devices from different systems.

In what concerns the single-device performance results when the smaller dataset was considered (see Figure 6.3), although significant improvements have also been observed when comparing with the sequential execution, speed-ups are not as good as with the larger dataset. This is naturally expected since, as it was verified during the profiling analysis (see Section 4.1.1), the node simulation procedure only represents approximately 78% of the algorithm execution time (in contrast with the 94% when considering the larger dataset), significantly limiting the attainable speed-up to $4.55 \times$. However, since the most time consuming part of the data setup procedure was also optimized by using the bitonic sort and since the output files are being written in parallel with the computations, higher speed-up results than this theoretical limit value were obtained in some configurations. Nevertheless, this results offer some evidence in favour that the parallel execution of the algorithm scales well with the problem size.

![Graph: 101x101x90 Single-Device Performance Results](image)

Figure 6.3: Single-device execution speed-up results compared to the sequential CPU implementation (executed in System 1), when considering the dataset with 101x101x90 nodes.

In what concerns the variation of the speed-up with the number of considered grid divisions, it can be observed that the speed-up tends to significantly increase with the number of grid divisions until the device architectures are fully occupied (see Figure 6.4). It can also be verified that the amount of parallelism considered by default ($2^{15}$ grid divisions) is enough to fully occupy the considered devices, thus being a fair reference to compare their capabilities. Note that, after achieving a fully occupied state of the GPUs, the speed-up still tends to increase with the number of grid divisions, not only because the GPUs are able to use the exceeding work-items to hide the latencies of some instructions, but also because less kernel calls are actually being performed, thus reducing the parallelization overhead inherent to those OpenCL calls. Another interesting aspect that can be verified by analysing Figure 6.4 is the fact that, when few grid divisions are considered, the AMD GPU struggles to obtain a better
performance improvements when compared with the NVIDIA GPU, despite of its greater computing performance capabilities. The main reason for this is the fact that, considering the current OpenCL GPU drivers from both the considered manufacturers, significantly higher overheads related with the OpenCL calls are verified for the AMD GPUs. Therefore, since fewer parallelism implies a higher amount of kernel calls, worse performance results are verified for the AMD device.

![Graph showing speed-up results](image)

Figure 6.4: Single-device execution speed-up results when varying the number of grid divisions for both the GPUs from System 1, compared to the sequential CPU implementation (also executed in System 1). Results consider the full program execution over the dataset with 237x197x350 nodes.

In addition to the speed-up measurements, the proposed implementation has also been evaluated in terms of the amount of device memory required to perform the previously mentioned executions. Accordingly, despite being memory demanding to consider a distinct node simulation per work-item, such mapping approach resulted in implementations that occupied at most 460 MB of the device memory which, considering modern devices, still leaves room for a significant scaling of the problem size until a layer by layer simulation approach is required (which implies increased communication overheads).

### 6.2.3 Evaluation of the Multi-Device Single-Realization Solution

Regarding the acceleration of the node simulation procedure by considering multiple devices (see Figure 6.5), it can be observed that the algorithm execution time has been successfully further reduced. In fact, significant performance improvements have been verified even when considering the cooperative execution with multiple different devices, which demonstrates the scalability of the considered implementation in heterogeneous environments. Namely, by using GPUs from different manufacturers (System 1), a global speed-up of $18.6 \times$ was obtained. Such result corresponds to an improvement factor of approximately $1.26 \times$ and $1.87 \times$, when comparing with the single-device execution using the Hawaii R9 290X and the GeForce GTX 560 Ti GPUs respectively. Results have also been satisfactory when the use of both GPUs of System 3 has been considered, being obtained an even higher overall speed-up of $22.69 \times$. On the other hand, not so satisfactory results in terms of scalability were obtained when considering the use of the CPU in cooperation with both GPUs from System 1, in order to further accelerate
the algorithm execution. In fact, only the Vs simulation was slightly accelerated when considering such environment. The main reason for such lack of scalability is related with the fact that the OpenCL always assumes that the whole computational architecture from the OpenCL devices is available to perform the kernel computations. Consequently, the node simulation threads are executed concurrently with the other threads responsible for the orchestration of the OpenCL execution and writing the output files, slowing down this procedures. In addition, the increased computing capabilities made available by the CPU device (significantly limited by the previously referred concurrency, since the CPU is also the host device) were not worth the increased communication overheads imposed by the characteristics of this parallelization approach.

By analysing the execution on System 2 (composed by two NVIDIA GeForce GTX 680 GPUs), it can be observed that the obtained speed-up is slightly below the theoretical limit when considering a simplistic perspective that counts the number of used devices (a speed-up of $2 \times$ is theoretically expected when using two similar accelerators, compared to the use of a single one). In fact, performance improvements of up to $1.9 \times$ during the node simulation procedures have been verified (part effectively being accelerated), being the difference to the theoretical limit mainly justified by the communication overhead and by the unbalanced execution during the first steps of the node simulation procedure (due to the initially uneven distribution of data along the simulation grid). However, only a global speed-up of $1.23 \times$ was obtained because, as the performance is being improved, this procedure becomes less significant in the overall execution time, thus becoming limited by the sequential execution of other parts of the algorithm (see Figure 6.6).

Nevertheless, by evaluating the implementation from the point of view of the multiple device efficiency when accelerating the node simulation procedure, it was verified that the communication overhead corresponded to approximately 0.99%, 2.5% and 0.43% of the execution time for the $\rho$, Vp and Vs simulations, respectively. In addition, by analysing the efficiency of the traditional load balancing, by using expression 6.1, it was observed global efficiencies of approximately 87%, 93% and 97% for the three types of simulations ($\rho$, Vp and Vs respectively) when considering the execution over the two heterogeneous
GPUs of System 1. Similar load balancing efficiencies have also been verified when considering the remaining systems.

$$\epsilon_{LB} = \left(1 - \max_{1 \leq j, k \leq n} \frac{|T_k - T_j|}{\sum_{l=1}^{n} T_l}\right) \times 100\%$$

(6.1)

As an example of the adaptive capabilities of the considered traditional load balancing algorithm, in the particular case of the Vs simulation executed on System 1, the obtained weights after the load balancing algorithm converged indicated that approximately 27% of the load must be computed by the NVIDIA GPU, while the remaining 73% of the load should be computed using the AMD GPU. By comparing the use of this load balancing scheme with the simplistic static load balancing approach that evenly divides the workload among the multiple OpenCL devices, a global performance improvement of 1.28×, 1.01× and 1.28× has been verified when executing in System 1, System 2 and System 3. Such results indicate that the used load balancing scheme was not only able to successfully balance the load between the heterogeneous devices, but it was also able to efficiently adapt to the uneven spatial distribution of the available data during the first steps of the simulation procedure, thus obtaining slightly better performance results when the execution over two similar devices was considered (System 2), in spite of the overhead introduced by the dynamic load balancing mechanism. Nonetheless, it must be highlighted that the impact of the unbalanced execution during the first simulation steps was significantly minimized with the use of the node postponing optimization (avoiding the simulation of nodes from regions where the available data is scarce), thus equalizing the conditioning data search times on every parallel executing device. Therefore, this optimization also minimized the benefits of using a dynamic load balancing mechanisms instead of a static one.

### 6.2.4 Evaluation of the Multi-Device Multi-Realization Solution

When analysing the obtained results from the perspective of both the multi-device approaches (see Figure 6.5) it can be observed that significantly higher throughputs were effectively attained when considering the multi-device multi-realization approach. In particular, a speed-up of 27.65× was obtained when
executing the algorithm in System 3, by using the GPUs GeForce GTX780 Ti and GeForce GTX 660 Ti in cooperation. This performance gain corresponds to an improvement of approximately $1.22 \times$ when compared with the multi-device single-realization execution. As mentioned in Section 6.2.1, even higher speed-ups can be observed when considering the random path generation criterion which imposes that every sub-grid has to follow the same relative sub-path. Therefore, when applying such random path generation criterion to this implementation, a speed-up as high as $31.68 \times$ has been observed.

![237x197x350 Multi-Device Performance Results](image)

**Figure 6.7**: Comparison between the proposed multi-device approaches, by considering the dataset with 237x197x350 nodes.

Once again, the efficiency of this multi-device parallelization approach can be further assessed by analysing the obtained results on System 2. Accordingly, it has been observed that the multi-device multi-realization execution over the two GPUs of System 2 presents a speed-up of $2.08 \times$ when compared with the single-device execution. Such performance improvement is even higher than the simplistic theoretical limit mentioned in the previous subsection, mainly because most of the time computing the best model construction (computed in a host device dedicated thread) is being overlapped with the generation of new realizations. Such functional parallelization, that arises from the need to sequentially execute this part of the algorithm, is not considered in the original single-device parallelization scheme, which only aimed the acceleration of the most computational demanding part of the algorithm (the node simulation procedure). In fact, when considering an equivalent single-device execution, a speed-up of $1.89 \times$ has been observed. This speed-up value is coherent, since by profiling the algorithm execution it has been observed that in this system (which has a slightly slower CPU when compared with the others), the sequential overhead introduced at the end of each iteration only represents approximately 5% of the algorithm execution time, supporting the increased efficiency presented by a realization-level parallelization approach. Moreover, this sequential overhead tends to become less significant as the number of realizations per iteration increases, enabling even higher speed-ups.

Such already expected acceleration rate when comparing this implementation with the previously assessed multi-device single-realization arises not only because a greater part of the algorithm is actually being executed in parallel (as a result of the higher granularity parallelization), but also because there is no communication and synchronization overheads during the node simulation procedure. Nevertheless, it must be highlighted that although a significantly larger part of the algorithm is being executed in paral-
eler (by different CPU threads), only the simulation procedure is actually being accelerated by using the GPU devices.

In what concerns the scalability of the proposed implementation, after profiling the accelerated execution on System 2 (system with the slower CPU device), it has been verified that the time to execute the best model construction procedure \( t_{\text{best}} \) represents approximately 20% of the time to perform a single realization \( t_{\text{sim}} \). Therefore, considering that the sequential execution time of the best model construction becomes the bottleneck if the number of devices \( n \) becomes greater than \( \frac{t_{\text{sim}}}{t_{\text{best}}} \), in this system the implementation would efficiently scale up to the limit of 5 devices (considering devices with equivalent computational capabilities). Performing the same analysis in the other systems resulted in slightly better results, allowing for at least one additional device, despite being used GPUs with higher performances in such systems. Nevertheless, it must be noted that the best model construction procedure has not been accelerated (through parallelization or any other optimization techniques), which would allow for a greater scalability. In addition, an increased amount of devices can still be used to further accelerate the execution of the realizations, by using the methodology proposed in the multi-device single-realization implementation. Therefore, as an example, a system with 10 GPUs could be efficiently exploited by performing 5 realizations in parallel, each being accelerated by using 2 devices.

6.3 Quality of the Inversion Results

The quality of the retrieved inversion models was assessed both by analysing the attained convergence taking into account the global correlation coefficient of the generated models when compared with the real seismic data, but also by the visualization of the vertical sections of converged physical property models, thus evaluating the quality of the obtained models by comparing the spatial distributions of the physical properties being simulated. In both cases, in order to effectively assess the converging capabilities of the proposed parallelization approach, the execution of 6 iterations composed by 32 realizations each was considered both in the parallel and in the sequential implementations. By default, parallel executions consider 32 divisions of the simulation grid in every direction, using the multi-device multi-realization approach in System 1 (taking advantage of both GPUs). Although similar conclusions can be drawn by using the larger dataset, the results herein discussed only consider the smaller dataset with 101x101x90 nodes, since it does not only presents a considerably faster execution, but also has a significantly faster convergence ratio (see the convergence diagrams presented in Section 6.2.1), thus enabling an extensive study of both the parallel and sequential executions in a feasible amount of time. Once again, parallel results were obtained by using the random path generation criterion that considers that each sub-grid has its own independent random sub-path. Nevertheless, it must be noted that no significant differences were observed in terms of the quality of the obtained models for the considered dataset (given the considered setup) when comparing both the random-path generation criteria.

By analysing Figure 6.8, it can be verified that when comparing the best simulated seismic model with real seismic data at each iteration of the algorithm, the convergence is still verified, being the obtained global correlation coefficients similar both for the parallel and sequential implementations of
the algorithm. This demonstrates that the applied relaxations, toward an efficient parallelization, do not affect the convergent behaviour of the algorithm.

![Figure 6.8](image.png)

**Figure 6.8:** Evolution of the global correlation coefficient over the iteration number, comparing the parallel with the sequential executions of the algorithm, composed of 6 iterations each with 32 realizations over the dataset with 101x101x90 nodes. The represented curve corresponds to the median global correlation coefficient values obtained from 5 distinct executions.

However, by analysing the chart in Figure 6.9, it can be verified that despite similar global correlation coefficients have been obtained in both considered implementations, a significantly faster (in time) convergence rate of the parallel implementation was observed when considering a multi-device multi-realization execution of the algorithm. Therefore, although a relaxation of the original algorithm has been performed, similar converged models are still being obtained in substantially less time.

![Figure 6.9](image.png)

**Figure 6.9:** Evolution of the global correlation coefficient over time, comparing a single execution of the parallel and the sequential implementations of the algorithm, composed of 6 iterations each with 32 realizations over the dataset with 101x101x90 nodes.
Moreover, as it can be observed in Figure 6.10, where 10 independent runs are considered for each number of grid divisions (which corresponds to the nodes being simulated in parallel), satisfactory results in terms of convergence were obtained even when large amounts of grid divisions were considered, demonstrating that the adopted relaxation aiming at an efficient parallelization do not significantly affect the quality of the results, regardless of the considered degree of partitioning of the simulation grid. This insensitivity to the number of performed divisions can be justified with the considered postponing optimization that avoids the simulation of nodes that have few or no conditioning data in the neighbour sub-grids. As a result, when the sub-grid size becomes smaller (as a result of increasing the number of grid divisions), a consistent spatial distribution of the physical property being simulated is still granted, thus ensuring the algorithm convergence. Note that this aspect is crucial, since in the considered approach further parallelism (and consequently higher speed-up results) is achieved by further dividing the simulation grid, which is herein proven that does not correspond to worse results in terms of quality of the inverted models.

![Figure 6.10: Convergence analysis by considering 10 independent runs per amount of grid divisions, composed of 6 iterations each with 32 realizations using the dataset with 101x101x90 nodes.](image)

Finally, Figure 6.11 illustrates the spatial distribution of the mean model computed with all the models generated during the last iteration for the proposed parallelization approach, together with the corresponding sequential approach, for the considered physical properties ($\rho$, Vp and Vs). By interpreting these vertical sections, it can be observed that the original spatial distributions of the physical properties being simulated are guaranteed. In addition, both approaches are able to reproduce the main features as observed in the real elastic models. Similarly to what happens by using the sequential approach, the parallelized methodology is able to better reproduce Vp and density while struggles to reproduce correctly Vs. This effect is a consequence of the cascade approach for generating the set of elastic models. Nevertheless, it should be recalled that the inverted and the real models are not expected to be exactly the same, given the stochastic nature of the DSS algorithm.
6.4 Summary

In this chapter, a comprehensive evaluation of the proposed parallel implementations of the algorithm has been conducted. For such purpose, the proposed parallelization approach has been assessed both in terms of the achieved performance improvements and in terms of the quality of the inverted models. In both cases the parallel implementations have been compared with the original sequential implementation of the algorithm. Accordingly, a significant reduction in the algorithm execution time has been observed in all the considered single-device mappings, demonstrating that the proposed implementation was able to efficiently take advantage of every considered environment. Namely, speed-ups as high as $18.06 \times$ were obtained, when considering single-device executions (in this case by using a NVIDIA GeForce GTX 780 Ti GPU). Regarding the use of multiple devices in order to further improve the algorithm execution, both multi-device parallelization approaches have been successfully able to provide higher speed-up values. In fact, a speed-up as high as $31.68 \times$ was obtained when considering the cooperative execution of two distinct devices (GeForce GTX780 Ti and GeForce GTX 660 Ti) executing in a multi-device multi-realization fashion. Such results demonstrate the efficient and scalable execution of the proposed implementations over heterogeneous systems. In what concerns the quality of the obtained inversion models, it was observed that the parallel implementation of the algorithm was able to successfully reproduce the spatial distributions of the physical properties being simulated. In fact, similar conclusions have been drawn even when different number of divisions of the simulation grid were performed, indicating that the proposed relaxation towards an efficient implementation does not significantly influence the quality of the obtained models.
CHAPTER 7

CONCLUSIONS

State of the art stochastic seismic AVO inversion algorithms based on DSS approaches represent a promising methodology to solve complex geophysical inversion problems used in reservoir modelling. However, the involved computational cost has demanded the adoption of parallel and HPC techniques. In fact, the parallelization of such algorithms has been regarded as highly important, not only to allow for faster reservoir modelling, but also because it facilitates and provides the feasible means to develop larger and more accurate computational models of the Earth’s subsurface. For such purpose, the latest generation of heterogeneous platforms, based on multiple GPP and GPU devices, have already been adopted in several domains.

Accordingly, this thesis proposes an efficient parallelization of a state of the art Stochastic Seismic AVO Inversion algorithm in heterogeneous platforms. Such a flexible solution is achieved by using the OpenCL API, thus allowing each part of the algorithm to be easily migrated among the several coexisting CPUs and GPUs. To circumvent the strict data dependencies presented by this algorithm, being the most significant part of the algorithm composed by millions of dependent iterations that individually are not computationally demanding, the adopted approach considers a spatial division of the simulation grid, thus allowing the parallel simulation of multiple nodes corresponding to different regions of the model. Although this algorithm implies ignoring some data dependencies, such division show no apparent loss of accuracy in the inversion results, regardless of the considered degree of parallelism. With the purpose of exploiting the high degree of parallelism introduced with this approach, multiple parallel solutions have been carefully designed and optimized by considering heterogeneous platforms based on GPUs.

According to the obtained experimental results, the proposed acceleration provides a global speed-up of the whole simulation as high as $18.06	imes$, by considering the usage of a single GPU device (NVIDIA GeForce GTX 780 Ti). Nevertheless, the unified OpenCL source code is not restricted to a specific device architecture or manufacturer, being also verified significant performance improvements when the application was mapped into the CPU, or into GPUs with other performance capabilities from the
same or different vendors. In addition, the devised parallel implementation can also be executed in heterogeneous platforms with multiple (possibly different) CPUs and GPUs, thus allowing for even faster executions of this algorithm. In accordance, the herein proposed multi-device approaches were able to efficiently balance the work-load between multiple (possibly different) devices, being achieved speed-ups over $24 \times$ in the heterogeneous configuration with two different GPUs from different manufacturers, and a speed-up as high as $31.68 \times$ when considering the cooperative use of two distinct NVIDIA GPUs.

### 7.1 Future Work

Although significant performance improvements have already been observed by considering the developed techniques, several aspects of such an application can still be improved and further analysed. Namely, an hybrid implementation considering both the presented multi-device approaches can be considered in order to efficiently exploit the capabilities of heterogeneous clusters composed by multiple and possibly different CPUs and GPUs. Such an implementation would possibly require a mechanism to assess the computational capabilities of the available devices, in order to be able to efficiently distribute the multiple devices among the realizations being executed in parallel, thus maximizing the algorithm execution performance. The development of such mechanism is possible, since the characteristics of the OpenCL devices can be obtained by calling specific OpenCL queries.

In addition, since the most computational demanding part of the algorithm has already been significantly accelerated, the use of OpenCL enabled devices in order to accelerate other parts of the algorithm can now be deemed worthwhile (e.g., accelerate the forward model computation). In fact, it has been verified that one of the main scalability limitations (in terms of the number of devices that can be efficiently exploited in parallel) of the multi-device multi-realization implementation concerned the sequential execution time of a procedure being sequentially executed. Therefore, by accelerating this part of the algorithm, the efficient exploitation of a greater amount of parallel devices would be possible. Such an highly scalable multi-realization approach could also benefit from the use of the MPI API in order to enable the use of processors from different machines. Nevertheless, the study of advanced techniques to further optimize the OpenCL implementation of the stochastic simulation procedure can still be considered, such as the use of data compaction and compression techniques or the use of fixed-point notation.

Finally, since the OpenCL API is being used in order to allow for a greater portability of the implementation among several different devices, the use of FPGA’s or other architectures that support OpenCL can also be considered either to further accelerate the algorithm execution, or to provide a more energy efficient execution.


